Table of Contents

[ABSTRACT 4](#_Toc137930432)

[TOKENISER 5](#_Toc137930433)

[SYMBOL TABLE 11](#_Toc137930434)

[ENGINE 24](#_Toc137930435)

[GENERATOR 37](#_Toc137930436)

[COMPILER 41](#_Toc137930437)

# ABSTRACT

The JACK compiler is a powerful tool in the field of programming languages and software development. It plays a crucial role in the compilation process of JACK, a high-level object-oriented language designed for the development of software for the Nand2Tetris computer architecture.

The JACK compiler serves as an essential component in the transformation of human-readable JACK code into low-level machine code that can be executed by the Nand2Tetris hardware platform. It performs various crucial tasks such as lexical analysis, syntax parsing, semantic analysis, and code generation.

At the lexical analysis stage, the JACK compiler breaks down the input JACK source code into tokens, identifying and categorizing elements such as keywords, identifiers, operators, and literals. The syntax parsing phase follows, where the compiler verifies the syntactic correctness of the code based on predefined grammar rules. This process ensures that the code adheres to the specified language structure and syntax.

The next step involves semantic analysis, where the compiler examines the meaning and context of the code. It checks for semantic errors, type mismatches, and enforces language-specific rules. This analysis ensures the integrity and coherence of the code.

Finally, the JACK compiler generates target code, translating the high-level JACK code into low-level machine code or intermediate representations like assembly language or bytecode. This code generation phase optimizes the output to enhance efficiency and performance.

In this project we have implemented this jack code in python and has been referred to from GitHub.

# TOKENISER

import re

class Tokenizer:

    """A tokenizer to tokenize a Jack source file.

    """

    def \_\_init\_\_(self, raw\_code):

        """Turns input raw code to a list of tokens

        Args:

            raw\_code (list): Input from file.

        """

        self.current\_token\_index = 0

        self.tokens = []

        clean\_code = Tokenizer.clean\_code(raw\_code)

        for line in clean\_code:

            self.tokens.extend(Tokenizer.handle\_line(line))

        self.total\_tokens = len(self.tokens)

    def advance(self):

        """Advance the token pointer by one. Throws error if no more tokens."""

        if self.has\_more\_tokens():

            self.current\_token\_index += 1

        else:

            raise IndexError('No more tokens.')

    def has\_more\_tokens(self):

        """Check if there are more tokens available."""

        return self.current\_token\_index < (self.total\_tokens - 1)

    def token\_type(self):

        """Returns the token type.

        Returns:

            str: KEYWORD, SYMBOL, IDENTIFIER, INT\_CONST, STRING\_CONST

        """

        symbol\_type = None

        token = self.curr\_token

        if token in ('class', 'constructor', 'function', 'method',

                     'field', 'static', 'var', 'int', 'char', 'if',

                     'boolean', 'void', 'true', 'false', 'null',

                     'this', 'let', 'do', 'return', 'else', 'while'):

            symbol\_type = 'KEYWORD'

        elif token in '{}()[].,;+-\*/&|<>=~':

            symbol\_type = 'SYMBOL'

        elif token.isdigit():

            symbol\_type = 'INT\_CONST'

        elif token.startswith('"'):

            symbol\_type = 'STRING\_CONST'

        elif (not token[0].isdigit()):

            symbol\_type = 'IDENTIFIER'

        else:

            raise SyntaxError('Invalid token : {}'.format(token))

        return symbol\_type

    @staticmethod

    def handle\_line(line):

        """Converts a line of clean code to a list of tokens.

        Required so that I can tokenize string constants without

        using re.Scanner.

        May need to later add token type here instead of parser.

        Args:

            line (string): Line of clean Jack code.

        Returns:

            list: a list of valid Jack tokens.

        """

        line = line.strip()

        ret = []

        if '"' in line:

            match = re.search(r"(\".\*?\")", line)

            ret.extend(Tokenizer.handle\_line(match.string[:match.start()]))

            ret.append(match.string[match.start():match.end() - 1])

            ret.extend(Tokenizer.handle\_line(match.string[match.end():]))

        else:

            for candidate in line.split():

                ret.extend(Tokenizer.handle\_token\_candidate(candidate))

        print(ret)

        return ret

    @staticmethod

    def handle\_token\_candidate(candidate):

        """Cleans and handles a possible token

        Args:

            candidate (string): A candidate for token (which

            can consist of multiple tokens)

        Returns:

            list: a list of tokens

        """

        if not candidate:

            return []

        ret = []

        match = re.search(

            r"([\&\|\(\)<=\+\-\\*>\\/.;,\[\]}{~])", candidate.strip()

        )

        if match is not None:

            ret.extend(Tokenizer.handle\_token\_candidate(

                match.string[:match.start()]

            ))

            ret.append(match.string[match.start()])

            ret.extend(Tokenizer.handle\_token\_candidate(

                match.string[match.end():]

            ))

        else:

            ret.append(candidate)

        return ret

    @staticmethod

    def clean\_code(raw\_code):

        """ Removes comments and newlines from the input raw code.

        Args:

            raw\_code (list): A list (str) of unclean code from the file.

        Returns:

            list: A list (str) of clean code.

        """

        lines = []

        comment\_on = False

        for line in raw\_code:

            line = line.strip()

            if line.startswith('/\*') and (not line.endswith('\*/')):

                comment\_on = True

            if not comment\_on:

                lines.append(line)

            if line.startswith('\*/') or line.endswith('\*/'):

                comment\_on = False

        lines = [line.split('//')[0].strip() for line in lines

                 if Tokenizer.is\_valid(line)]

        return lines

    @staticmethod

    def is\_valid(line):

        """Is it a valid Jack line?

        Args:

            line (str): A line from Jack file.

        Returns:

            bool: Is it a valid Jack line?

        """

        return line and (not line.startswith('//')) and (

            not line.startswith('/\*'))

    @property

    def curr\_token(self):

        """Return the current token.

        Returns:

            str: Current token

        """

        return self.tokens[self.current\_token\_index]

    @property

    def next\_token(self):

        """Returns next token if there is one.

        """

        if self.has\_more\_tokens():

            return self.tokens[self.current\_token\_index + 1]

    @property

    def prev\_token(self):

        """Returns the previous token, if there is one.

        """

        if self.current\_token\_index > 0:

            return self.tokens[self.current\_token\_index - 1]

if \_\_name\_\_ == "\_\_main\_\_":

    with open('SEM\_2/EOC2/Main.jack', 'r') as f:

        TEST\_LINES = f.readlines()

    TOKENIZER = Tokenizer(TEST\_LINES)

    #print(TOKENIZER.tokens)

    for i, tk in enumerate(TOKENIZER.tokens):

        print(i, tk)

    print('-----------------')

    print(TOKENIZER.handle\_token\_candidate('a/=2;b+=3;'))

* The code begins with importing the regular expression module, **re**, which will be used for pattern matching and string manipulation.
* A class named **Tokenizer** is defined. It represents a tokenizer for the Jack programming language and will be responsible for converting Jack source code into tokens.
* The **\_\_init\_\_** method is the constructor of the **Tokenizer** class. It takes in the **raw\_code** parameter, which is expected to be a list representing the lines of Jack source code.
* The constructor initializes two instance variables: **current\_token\_index** to keep track of the current token index during tokenization, and **tokens** as an empty list to store the generated tokens.
* The constructor calls the **clean\_code** method, which removes comments and newlines from the **raw\_code**. It then iterates over each line of the cleaned code and calls the **handle\_line** method to tokenize each line. The generated tokens are added to the **tokens** list.
* The total number of tokens generated is stored in the **total\_tokens** instance variable.
* The **advance** method is responsible for moving the token pointer to the next token. It increments the **current\_token\_index** by one.
* If there are more tokens available (checked using the **has\_more\_tokens** method), the **current\_token\_index** is incremented. Otherwise, an **IndexError** is raised, indicating that there are no more tokens to process.
* The **has\_more\_tokens** method checks if there are more tokens to be processed by comparing the **current\_token\_index** with the total number of tokens.
* It returns **True** if the **current\_token\_index** is less than the total number of tokens minus 1, indicating that there are more tokens available. Otherwise, it returns **False**.
* The **token\_type** method is responsible for determining the type of the current token.
* A variable **symbol\_type** is initialized as **None**, and the current token is obtained using the **curr\_token** property.
* The token is checked against a list of keywords in Jack. If it matches any of these keywords, the **symbol\_type** is set to **'KEYWORD'**.
* If the token matches any of the symbols in **{}()[].,;+-\*/&|<>=~**, the **symbol\_type** is set to **'SYMBOL'**.
* If the token consists of digits only, the **symbol\_type** is set to **'INT\_CONST'**.
* If the token starts with a double quote (indicating a string constant), the **symbol\_type** is set to **'STRING\_CONST'**.
* If the token does not start with a digit, it is considered an identifier, and the **symbol\_type** is set to **'IDENTIFIER'**.
* If none of the above conditions are met, it means that the token is invalid, and a **SyntaxError** is raised.
* The determined **symbol\_type** is returned.
* The **handle\_line** method takes a line of clean Jack code as input and converts it into a list of tokens.
* The input line is stripped of leading and trailing whitespace, and an empty list **ret** is initialized to store the tokens.
* If the line contains a double quote (indicating the presence of a string constant), the line is divided into three parts: the part before the string constant, the string constant itself, and the part after the string constant. The **handle\_line** method is recursively called on the parts before and after the string constant, and the extracted string constant is appended to the list of tokens.
* If the line does not contain a string constant, it is split into individual tokens using whitespace as the delimiter. Each token is processed by calling the **handle\_token\_candidate** method, and the resulting tokens are added to the **ret** list.
* The list of tokens is printed for debugging purposes, and then it is returned
* The **handle\_token\_candidate** method is responsible for cleaning and handling a possible token.
* If the candidate is an empty string, an empty list is returned. Otherwise, an empty list **ret** is initialized to store the tokens.
* A regular expression pattern is used to search for special symbols within the candidate token. The symbols include **&**, **|**, **(**, **)**, **<=**, **+**, **-**, **\***, **/**, **.**, **;**, **,**, **[**, **]**, **}**, **{**, and **~**. The pattern is applied to the stripped candidate token.
* If a match is found, the token is split into three parts: the part before the symbol, the symbol itself, and the part after the symbol. The **handle\_token\_candidate** method is recursively called on the parts before and after the symbol, and the symbol itself is added to the list of tokens.
* The list of tokens is returned.
* The **clean\_code** method is responsible for removing comments and newlines from the input raw code.
* An empty list **lines** is initialized to store the cleaned code. A boolean variable **comment\_on** is used to keep track of whether a multiline comment is in progress. The code iterates over each line of the raw code and performs the cleansing actions.
* The cleaned lines are further processed to remove single-line comments. Each line is split using the **//** delimiter, and only the part before the delimiter is retained. Leading and trailing whitespace are stripped from each line. The lines are filtered using the **is\_valid** method, which checks if a line is not empty or a pure comment.

# SYMBOL TABLE

class SymbolTable(dict):

    """A hashtable to perform variable lookup.

    Subclass of dict with a few special methods.

    """

    def \_\_init\_\_(self, \*args):

        """Creates a new SymbolTable.

        """

        dict.\_\_init\_\_(self, \*args)

        self.\_field\_scope = {}

        self.\_static\_scope = {}

        self.\_subroutine\_scope = {}

        self.\_count = {

            'STATIC': 0,

            'ARG': 0,

            'FIELD': 0,

            'VAR': 0

        }

    def \_\_getitem\_\_(self, key):

            if key in self.\_subroutine\_scope:

                return self.\_subroutine\_scope[key]

            elif key in self.\_field\_scope:

                return self.\_field\_scope[key]

            elif key in self.\_static\_scope:

                return self.\_static\_scope[key]

            else:

                raise KeyError("{} not in any scope.")

    def get(self, key, default=(None, None, -1)):

            try:

                ret = self[key]

            except KeyError:

                ret = default

            finally:

                return ret

    def define(self, name, \_type, kind):

        """Add a new element.

        Args:

            name (str): The name of the variable

            \_type (str): The type of the variable

            kind (str): STATIC, ARG, FIELD or VAR

        Raises:

            TypeError: The kind of given variable is invalid

        """

        try:

            i = self.\_count[kind]

        except KeyError:

            raise TypeError('{} is not a supported kind.'.format(kind))

        if kind in ('ARG', 'VAR'):

            self.\_subroutine\_scope[name] = (\_type, kind, i)

        elif kind == 'STATIC':

            self.\_static\_scope[name] = (\_type, kind, i)

        else:  # == 'FIELD

            self.\_field\_scope[name] = (\_type, kind, i)

        self.\_count[kind] += 1

        return i

    def reset(self):

        """Clears the \_subroutine\_scope.

        """

        self.\_subroutine\_scope.clear()

        self.\_count['ARG'] = 0

        self.\_count['VAR'] = 0

    def var\_count(self, kind):

        return self.\_count[kind]

if \_\_name\_\_ == '\_\_main\_\_':

    d = SymbolTable()

    d.define('this', 'Point', 'ARG')

    print(d['this'])

    d.reset()

    print(d.get('this'))

    print(d)

SYMBOL TABLE  
from generator import VMWriter

from symbolTable import SymbolTable

class CompilationEngine:

    """Creates an AST of the input file.

    """

    def \_\_init\_\_(self, input\_stream, output\_file):

        self.tokenizer = input\_stream

        self.outfile = output\_file

        self.class\_name = None

        self.out\_stream = []

        self.buffer = []

        self.if\_count = 0

        self.while\_count = 0

        self.generator = VMWriter(self.out\_stream)

        self.symbol\_table = SymbolTable()

        self.op\_table = {

            '+': 'ADD', '-': 'SUB', '&': 'AND', '|': 'OR',

            '<': 'LT', '>': 'GT', '=': 'EQ'

        }

        self.convert\_kind = {

            'ARG': 'ARG',

            'STATIC': 'STATIC',

            'VAR': 'LOCAL',

            'FIELD': 'THIS'

        }

    def compile\_class(self):

        """Compiles a Jack class to VM file.

        Raises:

            SyntaxError: If the current token is not expected, a SyntaxError \

             is raised.

        Returns:

            list: Output stream containing the commands

        """

        tk = self.tokenizer

        tk.advance()  # "class"

        self.class\_name = tk.curr\_token

        tk.advance()

        tk.advance()  # "{"

        while tk.curr\_token in ('static', 'field'):

            self.compile\_class\_var\_dec()

        while tk.curr\_token in ('constructor', 'function', 'method'):

            self.compile\_subroutine()

        if tk.curr\_token != '}':

            raise SyntaxError('} expected at end.')

        with open(self.outfile, 'w') as f:

                f.write('\n'.join(self.out\_stream))

    def compile\_class\_var\_dec(self):

        """Compiles the Jack class variable declaration(s).

        Raises:

            SyntaxError: When the programmer is idiot.

        """

        tk = self.tokenizer

        cat = tk.curr\_token.upper()

        tk.advance()  # "static" or "field"

        #  variable type

        \_type = tk.curr\_token

        tk.advance()

        # Check if variable name is a valid identifier

        if tk.token\_type() != 'IDENTIFIER':

            raise SyntaxError('{} is not a valid Jack identifier'

                              .format(tk.curr\_token))

        self.symbol\_table.define(tk.curr\_token, \_type, cat)

        tk.advance()

        while tk.curr\_token != ';':

            tk.advance()  # ","

            if tk.token\_type() != 'IDENTIFIER':

                raise SyntaxError('{} is not a valid Jack identifer.'

                                  .format(tk.curr\_token))

            self.symbol\_table.define(tk.curr\_token, \_type, cat)

            tk.advance()

        tk.advance()  # ";"

    def compile\_subroutine(self):

        """Compiles a Jack subroutine.

        Raises:

            SyntaxError: When unexpected input is given.

        """

        tk = self.tokenizer

        self.symbol\_table.reset()

        subroutine\_type = tk.curr\_token

        if subroutine\_type == 'method':

            self.symbol\_table.define('this', self.class\_name, 'ARG')

        tk.advance()

        tk.advance()  # ("void" | type)

        if tk.token\_type() != 'IDENTIFIER':

            raise SyntaxError("Subroutine name ({}) not a valid identifier"

                              .format(tk.curr\_token))

        func\_name = "{}.{}".format(self.class\_name, tk.curr\_token)

        tk.advance()

        tk.advance()  # "("

        self.compile\_parameter\_list()

        tk.advance()  # ")"

        tk.advance()  # "{"

        while 'var' == tk.curr\_token:

            self.compile\_var\_dec()

        n\_args = self.symbol\_table.var\_count('VAR')

        self.generator.write\_function(func\_name, n\_args)

        if subroutine\_type == 'constructor':

            n\_fields = self.symbol\_table.var\_count('FIELD')

            self.generator.write\_push\_pop('push', 'CONST', n\_fields)

            self.generator.write\_call('Memory.alloc', 1)

            self.generator.write\_push\_pop('pop', 'POINTER', 0)

        elif subroutine\_type == 'method':

            self.generator.write\_push\_pop('push', 'ARG', 0)

            self.generator.write\_push\_pop('pop', 'POINTER', 0)

        self.compile\_statements()

        tk.advance()  # "}"

    def compile\_parameter\_list(self):

        """Compiles parameter list for a Jack subroutine.

        Raises:

            SyntaxError: When unexpected input is given.

        """

        tk = self.tokenizer

        cat = 'ARG'

        if tk.curr\_token == ')':

            return

        \_type = tk.curr\_token

        tk.advance()

        if tk.token\_type() != 'IDENTIFIER':

            raise SyntaxError('{} is not a valid Jack identifier'

                              .format(tk.curr\_token))

        self.symbol\_table.define(tk.curr\_token, \_type, cat)

        tk.advance()

        while tk.curr\_token != ')':

            tk.advance()  # ","

            \_type = tk.curr\_token

            tk.advance()

            if tk.token\_type() != 'IDENTIFIER':

                raise SyntaxError('{} is not a valid Jack identifer.'

                                  .format(tk.curr\_token))

            self.symbol\_table.define(tk.curr\_token, \_type, cat)

            tk.advance()

    def compile\_var\_dec(self):

        """Compiles Jack variable declaration(s).

        Raises:

            SyntaxError: When unexpected input is provided.

        """

        tk = self.tokenizer

        tk.advance()

        cat = 'VAR'

        \_type = tk.curr\_token

        tk.advance()

        if tk.token\_type() != 'IDENTIFIER':

            raise SyntaxError('{} is not a valid Jack identifer.'

                              .format(tk.curr\_token))

        self.symbol\_table.define(tk.curr\_token, \_type, cat)

        tk.advance()

        while tk.curr\_token != ';':

            tk.advance()  # ","

            if tk.token\_type() != 'IDENTIFIER':

                raise SyntaxError('{} is not a valid Jack identifer.'

                                  .format(tk.curr\_token))

            self.symbol\_table.define(tk.curr\_token, \_type, cat)

            tk.advance()

        tk.advance()  # ";"

    def compile\_statements(self):

        """Compiles a Jack if/while/do/let/return statement.

        """

        tk = self.tokenizer

        func\_to\_call = {

            'if': self.compile\_if\_statement,

            'let': self.compile\_let\_statement,

            'do': self.compile\_do\_statement,

            'while': self.compile\_while\_statement,

            'return': self.compile\_return\_statement

        }

        while tk.curr\_token in ('if', 'while', 'let', 'do', 'return'):

            f = func\_to\_call.get(tk.curr\_token)

            f()

    def compile\_let\_statement(self):

        """Compiles a Jack "let" statement.

        Raises:

            SyntaxError: Unexpected input

        """

        tk = self.tokenizer

        tk.advance()  # "let"

        if tk.token\_type() != 'IDENTIFIER':

                raise SyntaxError('{} is not a valid Jack identifer.'

                                  .format(tk.curr\_token))

        \_type, cat, i = self.symbol\_table.get(tk.curr\_token)

        cat = self.convert\_kind[cat]

        tk.advance()

        if tk.curr\_token == '[':  # array assignment

            tk.advance()  # [

            self.compile\_expression()

            tk.advance()  # ]

            self.generator.write\_push\_pop('push', cat, i)

            self.generator.write\_arithmetic('ADD')

            self.generator.write\_push\_pop('pop', 'TEMP', 0)

            tk.advance()  # =

            self.compile\_expression()

            self.generator.write\_push\_pop('push', 'TEMP', 0)

            self.generator.write\_push\_pop('pop', 'POINTER', 1)

            self.generator.write\_push\_pop('pop', 'THAT', 0)

        else:

            tk.advance()  # =

            self.compile\_expression()

            self.generator.write\_push\_pop('pop', cat, i)

        tk.advance()  # ";"

    def compile\_if\_statement(self):

        """Compiles a Jack "if" statement.

        """

        tk = self.tokenizer

        tk.advance()  # "if"

        tk.advance()  # "("

        self.compile\_expression()

        tk.advance()  # ")"

        l1 = "IF\_TRUE{}".format(self.if\_count)

        l2 = "IF\_FALSE{}".format(self.if\_count)

        l3 = "IF\_END{}".format(self.if\_count)

        self.generator.write\_ifgoto(l1)

        self.generator.write\_goto(l2)

        self.generator.write\_label(l1)

        self.if\_count += 1

        tk.advance()  # "{"

        self.compile\_statements()

        self.generator.write\_goto(l3)

        tk.advance()  # "}"

        self.generator.write\_label(l2)

        if tk.curr\_token == 'else':

            tk.advance()  # "else"

            tk.advance()  # "{"

            self.compile\_statements()

            tk.advance()  # "}"

        self.generator.write\_label(l3)

    def compile\_while\_statement(self):

        """Compiles a Jack "while" statement.

        """

        tk = self.tokenizer

        tk.advance()  # "while"

        l1 = "WHILE\_EXP{}".format(self.while\_count)

        l2 = "WHILE\_END{}".format(self.while\_count)

        self.while\_count += 1

        self.generator.write\_label(l1)

        tk.advance()  # "("

        self.compile\_expression()

        self.generator.write\_arithmetic("NOT")

        tk.advance()  # ")"

        tk.advance()  # "{"

        self.generator.write\_ifgoto(l2)

        self.compile\_statements()

        self.generator.write\_goto(l1)

        self.generator.write\_label(l2)

        tk.advance()  # "}"

    def compile\_do\_statement(self):

        """Compiles a Jack "do" statement.

        Raises:

            SyntaxError: Unexpected input

        """

        tk = self.tokenizer

        tk.advance()  # "do"

        if tk.token\_type() != 'IDENTIFIER':

            raise SyntaxError('{} is not a proper identifier.'

                              .format(tk.curr\_token))

        var\_name = tk.curr\_token

        tk.advance()

        self.compile\_subroutine\_call(var\_name)

        self.generator.write\_push\_pop('pop', 'TEMP', 0)  # void method

        tk.advance()  # ";"

    def compile\_return\_statement(self):

        """Compiles a Jack "return" statement.

        """

        tk = self.tokenizer

        tk.advance()  # "return"

        if tk.curr\_token != ';':

            self.compile\_expression()

        else:

            # if no val to return, push 0 to stack

            self.generator.write\_push\_pop('push', 'CONST', 0)

        self.generator.write\_return()

        tk.advance()  # ";"

    def compile\_expression\_list(self):

        """Compiles a Jack expression list.

        Returns:

            n\_args (int): Number of arguments for subroutine call

        Raises:

            SyntaxError: Unexpected input

        """

        tk = self.tokenizer

        n\_args = 0

        if tk.curr\_token == ')':

            return n\_args

        self.compile\_expression()

        n\_args += 1

        while tk.curr\_token != ')':

            tk.advance()  # ","

            self.compile\_expression()

            n\_args += 1

        return n\_args

    def compile\_expression(self):

        """Compiles a Jack expression.

        """

        tk = self.tokenizer

        self.compile\_term()

        while tk.curr\_token in (

            '+', '-', '\*', '/', '&', '|', '<', '>', '='

        ):

            op = tk.curr\_token

            tk.advance()

            self.compile\_term()

            if op in self.op\_table:

                self.generator.write\_arithmetic(self.op\_table.get(op))

            elif op == '\*':

                self.generator.write\_call('Math.multiply', 2)

            elif op == '/':

                self.generator.write\_call('Math.divide', 2)

            else:

                raise ValueError("{} not supported op.".format(op))

    def compile\_term(self):

        """Compiles a Jack term.

        Raises:

            SyntaxError: Unexpected input

        """

        tk = self.tokenizer

        if tk.token\_type() == 'STRING\_CONST':

            self.compile\_string()

        elif tk.token\_type() == 'INT\_CONST':

            self.generator.write\_push\_pop('push', 'CONST', int(tk.curr\_token))

            tk.advance()

        elif tk.curr\_token in ('true', 'false', 'null'):

            self.generator.write\_push\_pop('push', 'CONST', 0)

            if tk.curr\_token == 'true':

                self.generator.write\_arithmetic("NOT")

            tk.advance()

        elif tk.curr\_token == 'this':

            # "this" is the 0th argument

            self.generator.write\_push\_pop('push', 'POINTER', 0)

            tk.advance()

        elif tk.curr\_token in ('-', '~'):

            op = tk.curr\_token

            tk.advance()

            self.compile\_term()

            if op == '-':

                self.generator.write\_arithmetic('NEG')

            else:

                self.generator.write\_arithmetic('NOT')

        elif tk.curr\_token == '(':

            tk.advance()  # "("

            self.compile\_expression()

            tk.advance()  # ")"

        else:

            if tk.token\_type() != 'IDENTIFIER':

                    raise SyntaxError('{} is not a valid identifier.'

                                      .format(tk.curr\_token))

            var\_name = tk.curr\_token

            tk.advance()

            if tk.curr\_token == '[':

                tk.advance()  # "["

                self.compile\_expression()

                tk.advance()  # "]"

                \_type, cat, i = self.symbol\_table.get(var\_name)

                cat = self.convert\_kind[cat]

                self.generator.write\_push\_pop('push', cat, i)

                self.generator.write\_arithmetic('ADD')

                self.generator.write\_push\_pop('pop', 'POINTER', 1)

                self.generator.write\_push\_pop('push', 'THAT', 0)

            elif tk.curr\_token in ('.', '('):

                self.compile\_subroutine\_call(var\_name)

            else:

                \_type, cat, i = self.symbol\_table.get(var\_name)

                cat = self.convert\_kind[cat]

                self.generator.write\_push\_pop('push', cat, i)

    def compile\_subroutine\_call(self, var\_name):

        tk = self.tokenizer

        func\_name = var\_name

        n\_args = 0

        if tk.curr\_token == '.':

            tk.advance()  # "."

            sub\_name = tk.curr\_token  # subroutine name

            tk.advance()

            \_type, cat, i = self.symbol\_table.get(var\_name)

            if \_type != None:  # it's an instance

                cat = self.convert\_kind[cat]

                self.generator.write\_push\_pop('push', cat, i)

                func\_name = "{}.{}".format(\_type, sub\_name)

                n\_args += 1

            else:  # it's a class

                func\_name = "{}.{}".format(var\_name, sub\_name)

        elif tk.curr\_token == '(':

            sub\_name = var\_name

            func\_name = "{}.{}".format(self.class\_name, sub\_name)

            n\_args += 1

            self.generator.write\_push\_pop('push', 'POINTER', 0)

        tk.advance()  # "("

        n\_args += self.compile\_expression\_list()

        tk.advance()  # ")"

        self.generator.write\_call(func\_name, n\_args)

    def compile\_string(self):

        tk = self.tokenizer

        string = tk.curr\_token[1:]

        self.generator.write\_push\_pop('push', 'CONST', len(string))

        self.generator.write\_call('String.new', 1)

        for char in string:

            self.generator.write\_push\_pop('push', 'CONST', ord(char))

            self.generator.write\_call('String.appendChar', 2)

        tk.advance()

if \_\_name\_\_ == '\_\_main\_\_':

    engine = CompilationEngine('10/Square/SquareGame.jack',

                               '10/Square/SquareGame\_local.xml',

                               ['Main', 'Square', 'SquareGame'])

    if engine.tokenizer.curr\_token == 'class':

        xml\_output = engine.compile\_class()

        with open(engine.outfile, 'w') as f:

            f.write(xml\_output)

    else:

        raise SyntaxError('The {} file should begin with class declaration.'

                          .format(engine.tokenizer.curr\_token))

The code defines a class **SymbolTable** that represents a symbol table for variable lookup. It is a subclass of the built-in **dict** class.

The **\_\_init\_\_** method initializes a new **SymbolTable** object. It sets up three dictionaries (**\_field\_scope**, **\_static\_scope**, **\_subroutine\_scope**) to store variables based on their kind (field, static, or subroutine). It also initializes a **\_count** dictionary to keep track of the count of each kind of variable.

The **\_\_getitem\_\_** method overrides the behavior of accessing items using the square bracket notation (**[]**). It first checks if the key exists in the **\_subroutine\_scope**, **\_field\_scope**, or **\_static\_scope** dictionaries and returns the corresponding value if found. If the key is not found in any scope, a **KeyError** is raised.

The **get** method provides a way to retrieve the value associated with a key from the symbol table. It is similar to the **\_\_getitem\_\_** method but returns a default value (a tuple **(None, None, -1)** by default) if the key is not found in any scope.

The **define** method adds a new variable to the symbol table. It takes three arguments: **name** (the name of the variable), **\_type** (the type of the variable), and **kind** (the kind of the variable: STATIC, ARG, FIELD, or VAR). It first checks if the **kind** is valid by looking it up in the **\_count** dictionary. If it's not a valid kind, a **TypeError** is raised.

Depending on the **kind** of the variable, it is added to the corresponding scope dictionary (**\_subroutine\_scope**, **\_field\_scope**, or **\_static\_scope**). The variable is stored as a tuple containing its type, kind, and index (**i**). The index is obtained from the **\_count** dictionary and then incremented.

The **reset** method clears the **\_subroutine\_scope** dictionary and resets the counts for ARG and VAR variables.

The **var\_count** method returns the count of variables of a given kind.

The code checks if the file is being run directly (not imported) and creates a **SymbolTable** object **d**. It then defines a variable **'this'** of type **'Point'** and kind **'ARG'**. It prints the value associated with the key **'this'** using the square bracket notation. It then resets the symbol table and prints the value associated with **'this'** using the **get** method. Finally, it prints the symbol table itself.

The code defines a basic symbol table that can store variables based on their kind (field, static, argument, or local) and provides methods for defining variables, retrieving their values, and resetting the table.

# ENGINE

from generator import VMWriter

from symbolTable import SymbolTable

class CompilationEngine:

    """Creates an AST of the input file.

    """

    def \_\_init\_\_(self, input\_stream, output\_file):

        self.tokenizer = input\_stream

        self.outfile = output\_file

        self.class\_name = None

        self.out\_stream = []

        self.buffer = []

        self.if\_count = 0

        self.while\_count = 0

        self.generator = VMWriter(self.out\_stream)

        self.symbol\_table = SymbolTable()

        self.op\_table = {

            '+': 'ADD', '-': 'SUB', '&': 'AND', '|': 'OR',

            '<': 'LT', '>': 'GT', '=': 'EQ'

        }

        self.convert\_kind = {

            'ARG': 'ARG',

            'STATIC': 'STATIC',

            'VAR': 'LOCAL',

            'FIELD': 'THIS'

        }

    def compile\_class(self):

        """Compiles a Jack class to VM file.

        Raises:

            SyntaxError: If the current token is not expected, a SyntaxError \

             is raised.

        Returns:

            list: Output stream containing the commands

        """

        tk = self.tokenizer

        tk.advance()  # "class"

        self.class\_name = tk.curr\_token

        tk.advance()

        tk.advance()  # "{"

        while tk.curr\_token in ('static', 'field'):

            self.compile\_class\_var\_dec()

        while tk.curr\_token in ('constructor', 'function', 'method'):

            self.compile\_subroutine()

        if tk.curr\_token != '}':

            raise SyntaxError('} expected at end.')

        with open(self.outfile, 'w') as f:

                f.write('\n'.join(self.out\_stream))

    def compile\_class\_var\_dec(self):

        """Compiles the Jack class variable declaration(s).

        Raises:

            SyntaxError: When the programmer is idiot.

        """

        tk = self.tokenizer

        cat = tk.curr\_token.upper()

        tk.advance()  # "static" or "field"

        #  variable type

        \_type = tk.curr\_token

        tk.advance()

        # Check if variable name is a valid identifier

        if tk.token\_type() != 'IDENTIFIER':

            raise SyntaxError('{} is not a valid Jack identifier'

                              .format(tk.curr\_token))

        self.symbol\_table.define(tk.curr\_token, \_type, cat)

        tk.advance()

        while tk.curr\_token != ';':

            tk.advance()  # ","

            if tk.token\_type() != 'IDENTIFIER':

                raise SyntaxError('{} is not a valid Jack identifer.'

                                  .format(tk.curr\_token))

            self.symbol\_table.define(tk.curr\_token, \_type, cat)

            tk.advance()

        tk.advance()  # ";"

    def compile\_subroutine(self):

        """Compiles a Jack subroutine.

        Raises:

            SyntaxError: When unexpected input is given.

        """

        tk = self.tokenizer

        self.symbol\_table.reset()

        subroutine\_type = tk.curr\_token

        if subroutine\_type == 'method':

            self.symbol\_table.define('this', self.class\_name, 'ARG')

        tk.advance()

        tk.advance()  # ("void" | type)

        if tk.token\_type() != 'IDENTIFIER':

            raise SyntaxError("Subroutine name ({}) not a valid identifier"

                              .format(tk.curr\_token))

        func\_name = "{}.{}".format(self.class\_name, tk.curr\_token)

        tk.advance()

        tk.advance()  # "("

        self.compile\_parameter\_list()

        tk.advance()  # ")"

        tk.advance()  # "{"

        while 'var' == tk.curr\_token:

            self.compile\_var\_dec()

        n\_args = self.symbol\_table.var\_count('VAR')

        self.generator.write\_function(func\_name, n\_args)

        if subroutine\_type == 'constructor':

            n\_fields = self.symbol\_table.var\_count('FIELD')

            self.generator.write\_push\_pop('push', 'CONST', n\_fields)

            self.generator.write\_call('Memory.alloc', 1)

            self.generator.write\_push\_pop('pop', 'POINTER', 0)

        elif subroutine\_type == 'method':

            self.generator.write\_push\_pop('push', 'ARG', 0)

            self.generator.write\_push\_pop('pop', 'POINTER', 0)

        self.compile\_statements()

        tk.advance()  # "}"

    def compile\_parameter\_list(self):

        """Compiles parameter list for a Jack subroutine.

        Raises:

            SyntaxError: When unexpected input is given.

        """

        tk = self.tokenizer

        cat = 'ARG'

        if tk.curr\_token == ')':

            return

        \_type = tk.curr\_token

        tk.advance()

        if tk.token\_type() != 'IDENTIFIER':

            raise SyntaxError('{} is not a valid Jack identifier'

                              .format(tk.curr\_token))

        self.symbol\_table.define(tk.curr\_token, \_type, cat)

        tk.advance()

        while tk.curr\_token != ')':

            tk.advance()  # ","

            \_type = tk.curr\_token

            tk.advance()

            if tk.token\_type() != 'IDENTIFIER':

                raise SyntaxError('{} is not a valid Jack identifer.'

                                  .format(tk.curr\_token))

            self.symbol\_table.define(tk.curr\_token, \_type, cat)

            tk.advance()

    def compile\_var\_dec(self):

        """Compiles Jack variable declaration(s).

        Raises:

            SyntaxError: When unexpected input is provided.

        """

        tk = self.tokenizer

        tk.advance()

        cat = 'VAR'

        \_type = tk.curr\_token

        tk.advance()

        if tk.token\_type() != 'IDENTIFIER':

            raise SyntaxError('{} is not a valid Jack identifer.'

                              .format(tk.curr\_token))

        self.symbol\_table.define(tk.curr\_token, \_type, cat)

        tk.advance()

        while tk.curr\_token != ';':

            tk.advance()  # ","

            if tk.token\_type() != 'IDENTIFIER':

                raise SyntaxError('{} is not a valid Jack identifer.'

                                  .format(tk.curr\_token))

            self.symbol\_table.define(tk.curr\_token, \_type, cat)

            tk.advance()

        tk.advance()  # ";"

    def compile\_statements(self):

        """Compiles a Jack if/while/do/let/return statement.

        """

        tk = self.tokenizer

        func\_to\_call = {

            'if': self.compile\_if\_statement,

            'let': self.compile\_let\_statement,

            'do': self.compile\_do\_statement,

            'while': self.compile\_while\_statement,

            'return': self.compile\_return\_statement

        }

        while tk.curr\_token in ('if', 'while', 'let', 'do', 'return'):

            f = func\_to\_call.get(tk.curr\_token)

            f()

    def compile\_let\_statement(self):

        """Compiles a Jack "let" statement.

        Raises:

            SyntaxError: Unexpected input

        """

        tk = self.tokenizer

        tk.advance()  # "let"

        if tk.token\_type() != 'IDENTIFIER':

                raise SyntaxError('{} is not a valid Jack identifer.'

                                  .format(tk.curr\_token))

        \_type, cat, i = self.symbol\_table.get(tk.curr\_token)

        cat = self.convert\_kind[cat]

        tk.advance()

        if tk.curr\_token == '[':  # array assignment

            tk.advance()  # [

            self.compile\_expression()

            tk.advance()  # ]

            self.generator.write\_push\_pop('push', cat, i)

            self.generator.write\_arithmetic('ADD')

            self.generator.write\_push\_pop('pop', 'TEMP', 0)

            tk.advance()  # =

            self.compile\_expression()

            self.generator.write\_push\_pop('push', 'TEMP', 0)

            self.generator.write\_push\_pop('pop', 'POINTER', 1)

            self.generator.write\_push\_pop('pop', 'THAT', 0)

        else:

            tk.advance()  # =

            self.compile\_expression()

            self.generator.write\_push\_pop('pop', cat, i)

        tk.advance()  # ";"

    def compile\_if\_statement(self):

        """Compiles a Jack "if" statement.

        """

        tk = self.tokenizer

        tk.advance()  # "if"

        tk.advance()  # "("

        self.compile\_expression()

        tk.advance()  # ")"

        l1 = "IF\_TRUE{}".format(self.if\_count)

        l2 = "IF\_FALSE{}".format(self.if\_count)

        l3 = "IF\_END{}".format(self.if\_count)

        self.generator.write\_ifgoto(l1)

        self.generator.write\_goto(l2)

        self.generator.write\_label(l1)

        self.if\_count += 1

        tk.advance()  # "{"

        self.compile\_statements()

        self.generator.write\_goto(l3)

        tk.advance()  # "}"

        self.generator.write\_label(l2)

        if tk.curr\_token == 'else':

            tk.advance()  # "else"

            tk.advance()  # "{"

            self.compile\_statements()

            tk.advance()  # "}"

        self.generator.write\_label(l3)

    def compile\_while\_statement(self):

        """Compiles a Jack "while" statement.

        """

        tk = self.tokenizer

        tk.advance()  # "while"

        l1 = "WHILE\_EXP{}".format(self.while\_count)

        l2 = "WHILE\_END{}".format(self.while\_count)

        self.while\_count += 1

        self.generator.write\_label(l1)

        tk.advance()  # "("

        self.compile\_expression()

        self.generator.write\_arithmetic("NOT")

        tk.advance()  # ")"

        tk.advance()  # "{"

        self.generator.write\_ifgoto(l2)

        self.compile\_statements()

        self.generator.write\_goto(l1)

        self.generator.write\_label(l2)

        tk.advance()  # "}"

    def compile\_do\_statement(self):

        """Compiles a Jack "do" statement.

        Raises:

            SyntaxError: Unexpected input

        """

        tk = self.tokenizer

        tk.advance()  # "do"

        if tk.token\_type() != 'IDENTIFIER':

            raise SyntaxError('{} is not a proper identifier.'

                              .format(tk.curr\_token))

        var\_name = tk.curr\_token

        tk.advance()

        self.compile\_subroutine\_call(var\_name)

        self.generator.write\_push\_pop('pop', 'TEMP', 0)  # void method

        tk.advance()  # ";"

    def compile\_return\_statement(self):

        """Compiles a Jack "return" statement.

        """

        tk = self.tokenizer

        tk.advance()  # "return"

        if tk.curr\_token != ';':

            self.compile\_expression()

        else:

            # if no val to return, push 0 to stack

            self.generator.write\_push\_pop('push', 'CONST', 0)

        self.generator.write\_return()

        tk.advance()  # ";"

    def compile\_expression\_list(self):

        """Compiles a Jack expression list.

        Returns:

            n\_args (int): Number of arguments for subroutine call

        Raises:

            SyntaxError: Unexpected input

        """

        tk = self.tokenizer

        n\_args = 0

        if tk.curr\_token == ')':

            return n\_args

        self.compile\_expression()

        n\_args += 1

        while tk.curr\_token != ')':

            tk.advance()  # ","

            self.compile\_expression()

            n\_args += 1

        return n\_args

    def compile\_expression(self):

        """Compiles a Jack expression.

        """

        tk = self.tokenizer

        self.compile\_term()

        while tk.curr\_token in (

            '+', '-', '\*', '/', '&', '|', '<', '>', '='

        ):

            op = tk.curr\_token

            tk.advance()

            self.compile\_term()

            if op in self.op\_table:

                self.generator.write\_arithmetic(self.op\_table.get(op))

            elif op == '\*':

                self.generator.write\_call('Math.multiply', 2)

            elif op == '/':

                self.generator.write\_call('Math.divide', 2)

            else:

                raise ValueError("{} not supported op.".format(op))

    def compile\_term(self):

        """Compiles a Jack term.

        Raises:

            SyntaxError: Unexpected input

        """

        tk = self.tokenizer

        if tk.token\_type() == 'STRING\_CONST':

            self.compile\_string()

        elif tk.token\_type() == 'INT\_CONST':

            self.generator.write\_push\_pop('push', 'CONST', int(tk.curr\_token))

            tk.advance()

        elif tk.curr\_token in ('true', 'false', 'null'):

            self.generator.write\_push\_pop('push', 'CONST', 0)

            if tk.curr\_token == 'true':

                self.generator.write\_arithmetic("NOT")

            tk.advance()

        elif tk.curr\_token == 'this':

            # "this" is the 0th argument

            self.generator.write\_push\_pop('push', 'POINTER', 0)

            tk.advance()

        elif tk.curr\_token in ('-', '~'):

            op = tk.curr\_token

            tk.advance()

            self.compile\_term()

            if op == '-':

                self.generator.write\_arithmetic('NEG')

            else:

                self.generator.write\_arithmetic('NOT')

        elif tk.curr\_token == '(':

            tk.advance()  # "("

            self.compile\_expression()

            tk.advance()  # ")"

        else:

            if tk.token\_type() != 'IDENTIFIER':

                    raise SyntaxError('{} is not a valid identifier.'

                                      .format(tk.curr\_token))

            var\_name = tk.curr\_token

            tk.advance()

            if tk.curr\_token == '[':

                tk.advance()  # "["

                self.compile\_expression()

                tk.advance()  # "]"

                \_type, cat, i = self.symbol\_table.get(var\_name)

                cat = self.convert\_kind[cat]

                self.generator.write\_push\_pop('push', cat, i)

                self.generator.write\_arithmetic('ADD')

                self.generator.write\_push\_pop('pop', 'POINTER', 1)

                self.generator.write\_push\_pop('push', 'THAT', 0)

            elif tk.curr\_token in ('.', '('):

                self.compile\_subroutine\_call(var\_name)

            else:

                \_type, cat, i = self.symbol\_table.get(var\_name)

                cat = self.convert\_kind[cat]

                self.generator.write\_push\_pop('push', cat, i)

    def compile\_subroutine\_call(self, var\_name):

        tk = self.tokenizer

        func\_name = var\_name

        n\_args = 0

        if tk.curr\_token == '.':

            tk.advance()  # "."

            sub\_name = tk.curr\_token  # subroutine name

            tk.advance()

            \_type, cat, i = self.symbol\_table.get(var\_name)

            if \_type != None:  # it's an instance

                cat = self.convert\_kind[cat]

                self.generator.write\_push\_pop('push', cat, i)

                func\_name = "{}.{}".format(\_type, sub\_name)

                n\_args += 1

            else:  # it's a class

                func\_name = "{}.{}".format(var\_name, sub\_name)

        elif tk.curr\_token == '(':

            sub\_name = var\_name

            func\_name = "{}.{}".format(self.class\_name, sub\_name)

            n\_args += 1

            self.generator.write\_push\_pop('push', 'POINTER', 0)

        tk.advance()  # "("

        n\_args += self.compile\_expression\_list()

        tk.advance()  # ")"

        self.generator.write\_call(func\_name, n\_args)

    def compile\_string(self):

        tk = self.tokenizer

        string = tk.curr\_token[1:]

        self.generator.write\_push\_pop('push', 'CONST', len(string))

        self.generator.write\_call('String.new', 1)

        for char in string:

            self.generator.write\_push\_pop('push', 'CONST', ord(char))

            self.generator.write\_call('String.appendChar', 2)

        tk.advance()

if \_\_name\_\_ == '\_\_main\_\_':

    engine = CompilationEngine('10/Square/SquareGame.jack',

                               '10/Square/SquareGame\_local.xml',

                               ['Main', 'Square', 'SquareGame'])

    if engine.tokenizer.curr\_token == 'class':

        xml\_output = engine.compile\_class()

        with open(engine.outfile, 'w') as f:

            f.write(xml\_output)

    else:

        raise SyntaxError('The {} file should begin with class declaration.'

                          .format(engine.tokenizer.curr\_token))

* `from generator import VMWriter`: This line imports the `VMWriter` class from the `generator` module. The `VMWriter` class is responsible for generating VM (virtual machine) code.
* `from symbolTable import SymbolTable`: This line imports the `SymbolTable` class from the `symbolTable` module. The `SymbolTable` class is used for managing the symbol table during the compilation process.
* `class CompilationEngine:`: This line begins the definition of the `CompilationEngine` class, which serves as the main engine for the compilation process.
* 6. `def \_\_init\_\_(self, input\_stream, output\_file):`: This line defines the constructor method for the `CompilationEngine` class. The constructor takes two parameters: `input\_stream` (tokenizer) and `output\_file` (the file to write the compiled VM code to).
* 8. `self.tokenizer = input\_stream`: This line assigns the `input\_stream` parameter to the `tokenizer` attribute of the object. The tokenizer is responsible for tokenizing the input Jack code.
* 9. `self.outfile = output\_file`: This line assigns the `output\_file` parameter to the `outfile` attribute of the object. It represents the file where the compiled VM code will be written.
* 10. `self.class\_name = None`: This line initializes the `class\_name` attribute to `None`. It will store the name of the current class being compiled.
* 11. `self.out\_stream = []`: This line initializes an empty list called `out\_stream`. It will store the generated VM code.
* 12. `self.buffer = []`: This line initializes an empty list called `buffer`. It will be used to store temporary data during the compilation process.
* 13. `self.if\_count = 0`: This line initializes the `if\_count` attribute to 0. It is a counter for "if" statements encountered during compilation.
* 14. `self.while\_count = 0`: This line initializes the `while\_count` attribute to 0. It is a counter for "while" statements encountered during compilation.
* 15. `self.generator = VMWriter(self.out\_stream)`: This line creates an instance of the `VMWriter` class and assigns it to the `generator` attribute of the object. The `VMWriter` instance is responsible for writing VM code to the `out\_stream` list.
* 16. `self.symbol\_table = SymbolTable()`: This line creates an instance of the `SymbolTable` class and assigns it to the `symbol\_table` attribute of the object. The `SymbolTable` instance is used to manage the symbol table during compilation.
* 17. `self.op\_table = {...}`: This line initializes the `op\_table` attribute with a dictionary. The dictionary maps arithmetic operators to their corresponding VM commands.
* 18. `self.convert\_kind = {...}`: This line initializes the `convert\_kind` attribute with a dictionary. The dictionary maps Jack variable kinds to their corresponding VM segment names.
* 21. `def compile\_class(self):`: This line defines the `compile\_class` method. It is responsible for compiling a Jack class to VM code.
* 23. `tk = self.tokenizer`: This line creates a local variable `tk` and assigns it the value of the `tokenizer` attribute. It is used to refer to the tokenizer object conveniently.
* 25-27. `tk.advance()`: These lines advance the tokenizer to the next token. In this case, it skips the "class" keyword (line 25), advances to the class name (line 26), and then advances to the opening curly brace (line 27).

# GENERATOR

class VMWriter:

    """Generates the VM code.

    """

    def \_\_init\_\_(self, \_stream):

        """Initialize a new VMWriter object.

        Args:

            \_stream (list): The output stream to which to write.

        """

        self.\_segment\_mapping = {key: key.lower() for key in ['POINTER',

                                 'LOCAL', 'STATIC', 'THIS', 'THAT', 'TEMP']}

        self.\_segment\_mapping.update({

            'ARG': 'argument', 'CONST': 'constant', 'FIELD': 'this'

        })

        self.stream = \_stream

    def write\_push\_pop(self, command, segment, index):

        """Writes a push VM command.

        Args:

            command (str): 'push' or 'pop'

            segment (str): One of the virtual memory segments on Hack computer

            index (int): The index on the virtual segment to push from/pop to

        """

        seg = self.\_segment\_mapping.get(segment)

        if seg is None:

            raise TypeError("{} segment is not available.".format(segment))

        self.stream.append("{} {} {}".format(command, seg, index))

    def write\_arithmetic(self, command):

        """Writes a VM supported arithmetic command.

        Args:

            command (str): One of the supported VM arithmetic commands.

        """

        if command not in ('ADD', 'SUB', 'NEG', 'EQ', 'GT',

                           'LT', 'AND', 'OR', 'NOT'):

            raise TypeError('{} not supported.'.format(command))

        self.stream.append(command.lower())

    def write\_label(self, label):

        self.stream.append('label {}'.format(label))

    def write\_goto(self, label):

        self.stream.append('goto {}'.format(label))

    def write\_ifgoto(self, label):

        self.stream.append('if-goto {}'.format(label))

    def write\_call(self, name, n\_args):

        self.stream.append('call {} {}'.format(name, n\_args))

    def write\_function(self, name, n\_local):

        self.stream.append('function {} {}'.format(name, n\_local))

    def write\_return(self):

        self.stream.append('return')

if \_\_name\_\_ == '\_\_main\_\_':

    from engine import CompilationEngine

    from tokenizer import Tokenizer

    engine = CompilationEngine(Tokenizer([]), 'Main', 'out.xml', [])

    writer = VMWriter(engine.out\_stream)

    writer.write\_arithmetic('ADD')

    writer.write\_call('abc', 3)

    writer.write\_function('abc', 2)

    writer.write\_goto('gotolabel')

    writer.write\_ifgoto('ifgotolabel')

    writer.write\_label('LABEL')

    writer.write\_push\_pop('push', 'ARG', 2)

    writer.write\_push\_pop('pop', 'LOCAL', 1)

    writer.write\_return()

    print(engine.out\_stream)

* `class VMWriter:`: This line begins the definition of the `VMWriter` class, which is responsible for generating VM code.
* `def \_\_init\_\_(self, \_stream):`: This line defines the constructor method for the `VMWriter` class. The constructor takes one parameter `\_stream`, which represents the output stream to which the VM code will be written.
* 5. `self.\_segment\_mapping = {key: key.lower() for key in ['POINTER', 'LOCAL', 'STATIC', 'THIS', 'THAT', 'TEMP']}`: This line initializes a dictionary called `\_segment\_mapping` that maps the virtual memory segment names used in Jack code to their corresponding lowercase representation used in VM code. It includes the segments: "POINTER", "LOCAL", "STATIC", "THIS", "THAT", and "TEMP".
* 6. `self.\_segment\_mapping.update({'ARG': 'argument', 'CONST': 'constant', 'FIELD': 'this'})`: This line updates the `\_segment\_mapping` dictionary by adding mappings for additional segments: "ARG" is mapped to "argument", "CONST" is mapped to "constant", and "FIELD" is mapped to "this".
* 7. `self.stream = \_stream`: This line assigns the `\_stream` parameter (the output stream) to the `stream` attribute of the object. It represents the list where the generated VM code will be stored.
* 10. `def write\_push\_pop(self, command, segment, index):`: This line defines the `write\_push\_pop` method. It is responsible for writing VM code for the "push" and "pop" commands.
* 13. `seg = self.\_segment\_mapping.get(segment)`: This line retrieves the lowercase representation of the `segment` argument using the `\_segment\_mapping` dictionary. If the segment is not found in the dictionary, it raises a `TypeError`.
* 16. `self.stream.append("{} {} {}".format(command, seg, index))`: This line generates a string representing the VM code for the "push" or "pop" command with the given `command`, `seg`, and `index` values. The generated string is appended to the `stream` list.
* 19. `def write\_arithmetic(self, command):`: This line defines the `write\_arithmetic` method. It is responsible for writing VM code for supported arithmetic commands.
* 22. `if command not in ('ADD', 'SUB', 'NEG', 'EQ', 'GT', 'LT', 'AND', 'OR', 'NOT'):`: This line checks if the `command` argument is not one of the supported arithmetic commands. If it is not supported, it raises a `TypeError`.
* 25. `self.stream.append(command.lower())`: This line appends the lowercase representation of the `command` to the `stream` list, representing the corresponding VM code.
* 28. `def write\_label(self, label):`: This line defines the `write\_label` method. It writes VM code for a label command.
* 30. `self.stream.append('label {}'.format(label))`: This line appends the label command with the given `label` to the `stream` list.
* 33. `def write\_goto(self, label):`: This line defines the `write\_goto` method. It writes VM code for a goto command.
* 35. `self.stream.append('goto {}'.format(label))`: This line appends the goto command with the given `label` to the `stream` list.
* 38. `def write\_ifgoto(self, label):`: This line defines the `write\_ifgoto` method. It writes VM code for an if-goto command.
* 40. `self.stream.append('if
* -goto {}'.format(label))`: This line appends the if-goto command with the given `label` to the `stream` list.
* 43. `def write\_call(self, name, n\_args):`: This line defines the `write\_call` method. It writes VM code for a call command.
* 45. `self.stream.append('call {} {}'.format(name, n\_args))`: This line appends the call command with the given `name` and `n\_args` to the `stream` list.
* 48. `def write\_function(self, name, n\_local):`: This line defines the `write\_function` method. It writes VM code for a function command.
* 50. `self.stream.append('function {} {}'.format(name, n\_local))`: This line appends the function command with the given `name` and `n\_local` to the `stream` list.
* 53. `def write\_return(self):`: This line defines the `write\_return` method. It writes VM code for a return command.
* 55. `self.stream.append('return')`: This line appends the return command to the `stream` list.
* 59-76: This block of code is outside the class definition and demonstrates the usage of the `VMWriter` class. It creates an instance of `VMWriter`, calls various methods to generate VM code, and prints the resulting VM code stored in the `out\_stream` list.
* Overall, the `VMWriter` class provides methods to generate VM code for different commands and stores the generated code in a list.

# COMPILER

import os

import sys

import argparse

from engine import CompilationEngine

from tokenizer import Tokenizer

from tkinter import \*

import tkinter.messagebox as messagebox

from tkinter.filedialog import askdirectory

from tkinter import filedialog

def get\_names(path):

    """Returns the names and paths of Jack classes.

    Args:

        path (str): Input paths (Jack file or dir of jack files)

    Returns:

        tuple: A tuple of lists of class names and their paths.

    """

    paths = []

    out\_names = []

    if os.path.isfile(path):

        paths.append(path)

        path, tmp\_name = os.path.split(path)

        name, ext = os.path.splitext(tmp\_name)

        out\_names.append(os.path.join(path, name + '.vm'))

        if ext != '.jack':

            print("Provided file is not a jack file.")

            sys.exit(1)

    elif os.path.isdir(path):

        paths = [x for x in os.listdir(path)

                 if os.path.splitext(x)[1] == '.jack']

        names = [os.path.splitext(x)[0] for x in paths]

        paths = [os.path.join(path, x) for x in paths]

        out\_names = [os.path.join(path, x + '.vm') for x in names]

    else:

        print('{} doesn\'t exist.'.format(path))

        sys.exit(1)

    return paths, out\_names

def main():

    global filename

    file\_paths, outnames = get\_names(filename)

    for pth, out\_pth in zip(file\_paths, outnames):

        with open(pth, 'r') as f:

            tk = Tokenizer(f.readlines())

        engine = CompilationEngine(tk, out\_pth)

        engine.compile\_class()

    print("Finished compilation...")

def browseFiles():

    global filename

    filename = filedialog.askopenfilename(initialdir = "/",title = "Select a File",filetypes = (("Text files","\*.txt\*"),("all files","\*.\*")))

    main()

def browseDirectory():

    global filename

    file\_path=[]

    directory\_path = askdirectory()

    if directory\_path:

        files = os.listdir(directory\_path)

        for file\_name in files:

            if file\_name.endswith('jack'):

                file\_path.append( os.path.join(directory\_path, file\_name))

        for i in file\_path:

            filename=i

            main()

root=Tk()

root.title("Compiler")

root.geometry("500x300")

root.configure(bg="#FFFFFF")

frame1 = Frame(root,background="#FFFFFF",height=40,width=500)

lbl1= Label(frame1, text = "Compiler", font=('Arial',25,'bold'),bg="#FFFFFF",fg='#000FFF')

lbl1.pack()

frame1.place(x=170,y=0)

frame2=Frame(root,background="#FFFFFF",height=200,width=500)

btn1 = Button(frame2,command=browseFiles, text="open File",relief='flat',font=('Ariel',8,'bold'),height=1,bg="#000FFF",fg="#FFFFFF",activebackground="#FFFFFF",activeforeground="#000FFF")

btn1.place(x=125,y=20)

btn2 = Button(frame2,command=browseDirectory, text="open directory",relief='flat',font=('Ariel',8,'bold'),height=1,bg="#000FFF",fg="#FFFFFF",activebackground="#FFFFFF",activeforeground="#000FFF")

btn2.place(x=250,y=20)

frame2.place(x=0,y=40)

root.mainloop()

* `import os`: This line imports the `os` module, which provides functions for interacting with the operating system.
* `import sys`: This line imports the `sys` module, which provides access to some variables used or maintained by the interpreter and to functions that interact with the interpreter.
* `import argparse`: This line imports the `argparse` module, which provides a way to create command-line interfaces and parse command-line arguments.
* `from engine import CompilationEngine`: This line imports the `CompilationEngine` class from the `engine` module.
* `from tokenizer import Tokenizer`: This line imports the `Tokenizer` class from the `tokenizer` module.
* `from tkinter import \*`: This line imports all classes, functions, and constants from the `tkinter` module.
* `import tkinter.messagebox as messagebox`: This line imports the `messagebox` module from `tkinter` and assigns it an alias `messagebox`.
* `from tkinter.filedialog import askdirectory`: This line imports the `askdirectory` function from the `filedialog` module in `tkinter`.
* `from tkinter import filedialog`: This line imports the `filedialog` module from `tkinter`.
* `def get\_names(path):`: This line starts the definition of the `get\_names` function. It takes a `path` argument and returns a tuple of lists containing class names and their paths.
* `paths = []`: This line initializes an empty list called `paths` to store the paths of Jack files.
* `out\_names = []`: This line initializes an empty list called `out\_names` to store the output file names.
* `if os.path.isfile(path):`: This line checks if the `path` argument corresponds to a file.
* `paths.append(path)`: This line appends the `path` to the `paths` list.
* `path, tmp\_name = os.path.split(path)`: This line splits the `path` into the directory and the file name.
* `name, ext = os.path.splitext(tmp\_name)`: This line splits the temporary file name into the name and the extension.
* `out\_names.append(os.path.join(path, name + '.vm'))`: This line creates the output file name by joining the directory path, the base name (without extension), and the ".vm" extension.
* `if ext != '.jack':`: This line checks if the extension of the input file is not ".jack".
* `print("Provided file is not a jack file.")`: This line prints a message indicating that the provided file is not a Jack file.
* `sys.exit(1)`: This line exits the program with a non-zero status code.
* `elif os.path.isdir(path):`: This line checks if the `path` argument corresponds to a directory.
* `paths = [x for x in os.listdir(path) if os.path.splitext(x)[1] == '.jack']`: This line retrieves a list of file names within the directory that have the extension ".jack".
* `names = [os.path.splitext(x)[0] for x in paths]`: This line extracts the base names (without extension) from the file names in `paths`.
* `paths = [os.path.join(path, x) for x in paths]`: This line creates the full paths by joining the directory path with each file name in `paths`.
* `out\_names = [os.path.join(path, x + '.vm') for x in names]`: This line creates the output file names by joining the
* directory path with each base name in `names` and adding the ".vm" extension.
* `else:`: This line is the `else` branch of the condition in line 17, indicating that the provided `path` is neither a file nor a directory.
* `print('{} doesn\'t exist.'.format(path))`: This line prints a message indicating that the provided `path` doesn't exist.
* `sys.exit(1)`: This line exits the program with a non-zero status code.
* `return paths, out\_names`: This line returns a tuple containing the `paths` and `out\_names` lists.
* `def main():`: This line starts the definition of the `main` function.
* `global filename`: This line declares the variable `filename` as global.
* `file\_paths, outnames = get\_names(filename)`: This line calls the `get\_names` function with `filename` as an argument and assigns the returned values to `file\_paths` and `outnames`.
* `for pth, out\_pth in zip(file\_paths, outnames):`: This line starts a loop that iterates over `file\_paths` and `outnames` simultaneously using the `zip` function.
* `with open(pth, 'r') as f:': This line opens the file at path `pth` in read mode and assigns the file object to `f`. The `with` statement ensures that the file is properly closed after use.
* `tk = Tokenizer(f.readlines())`: This line reads all lines from the file object `f` and passes them as an argument to the `Tokenizer` class constructor, creating a `Tokenizer` object assigned to `tk`.
* `engine = CompilationEngine(tk, out\_pth)`: This line creates a `CompilationEngine` object with `tk` as the tokenizer and `out\_pth` as the output file path.
* `engine.compile\_class()`: This line calls the `compile\_class` method on the `engine` object to start the compilation process.
* `print("Finished compilation...")`: This line prints a message indicating that the compilation process has finished.
* `def browseFiles():`: This line starts the definition of the `browseFiles` function.
* `global filename`: This line declares the variable `filename` as global.
* `filename = filedialog.askopenfilename(initialdir = "/",title = "Select a File",filetypes = (("Text files","\*.txt\*"),("all files","\*.\*")))`: This line opens a file dialog window and allows the user to select a file. The selected file path is assigned to `filename`.
* `main()`: This line calls the `main` function.
* `def browseDirectory():`: This line starts the definition of the `browseDirectory` function.
* `global filename`: This line declares the variable `filename` as global.
* `file\_path=[]`: This line initializes an empty list called `file\_path` to store the paths of Jack files within a directory.
* `directory\_path = askdirectory()`: This line opens a directory selection dialog and assigns the selected directory path to `directory\_path`.
* `if directory\_path:`: This line checks if `directory\_path` is not an empty string.
* `files = os.listdir(directory\_path)`: This line retrieves a list of all files and directories within `directory\_path`.
* `for file\_name in files:`: This line starts a loop that iterates over the `files` list.
* `if file\_name.endswith('jack'):`: This line checks if the current `file\_name` ends with the string `'jack'`.
* `file\_path.append(os.path.join(directory\_path, file\_name))`: This line creates the full path of the Jack file by joining `directory\_path` with the current `file\_name` and appends it to the `file\_path` list.
* `for i in file\_path:`: This line starts a loop that iterates over the `file\_path` list.
* `filename = i`: This line assigns the current `i` (file path) to the `filename` variable.
* `main()`: This line calls the `main` function.
* These lines define the graphical user interface (GUI) using Tkinter. The code sets up a window titled "Compiler" with a size of 500x300 pixels. It includes a label, two buttons for file and directory selection, and handles the button click events to call the corresponding functions (`browseFiles` and `browseDirectory`).
* `root.mainloop()`: This line starts the Tkinter event loop, which waits for user interactions and responds accordingly. It keeps the GUI window displayed until it is closed by the user.
* It uses the `Tokenizer` and `CompilationEngine` classes to compile the Jack code. The output is saved to corresponding `.vm` files.