Laporan Tugas Struktur Data

Muhammad Haekal Muhyidin Al-Araby
5024221030

1 Source Code

1.1 Interpreter

```python
from LogicOperation import logicPostfix, calculateLogic, isString
import os
import sys

DEBUG = 0


def debug_log(*args, sep=" ", end="\n", file=sys.stdout, flush=False):
    print(*args, sep=sep, end=end, file=file, flush=flush) if DEBUG else None


class var:
    def __init__(self, name: str, value):
        self.name = name
        self.value = value


class label:
    def __init__(self, name: str, line: int):
        self.name = name
        self.line = line


class funct:
    def __init__(self, name: str, content: str, arg: list[str]):
        self.name = name
        self.content = content
        self.arg = arg

    def exec(self, arg, parent=None, *args):
        local_interpreter = interpreter(parent)
        for a, b in zip(self.arg, arg):
            local_interpreter.var_list.append(var(a, b))
        for e in args:
            local_interpreter.var_list.append(e)
        ok, ret = local_interpreter.execstring(self.content)
        if ok:
            return ret
        return None


class class_def:
    def __init__(self, name: str, content: str):
        self.name = name
        self.content = content


class class_var:
    def __init__(self, base: class_def, arg=[], parent=None):
        self.class_interpreter = interpreter(parent)
        self.class_interpreter.execstring(base.content)
        self.class_list = self.class_interpreter.class_list
        self.var_list = self.class_interpreter.var_list
        self.funct_list = self.class_interpreter.funct_list
        self.parent = parent
        self.exec_funct("init", arg)
        debug_log("Init Self = ", self)
        debug_log("Init Self.parent = ", self.parent)

    def exec_funct(self, name, arg):
        for e in self.funct_list:
            if e.name == name:
                return e.exec(arg, self.class_interpreter, var("this", self))
        return None

    def get_var(self, name):
        for e in self.var_list:
            if e.name == name:
                return e
        return None


class interpreter:
    def __init__(self, parent=None):
        self.parent = parent
        self.var_list = []
        self.label_list = []
        self.funct_list = []
        self.class_list = []
        self.line_done = []
        self.in_goto = False

    def find_var(self, var_name, recursive=False):
        for e in self.var_list:
            if e.name == var_name:
                return e
        if recursive and self.parent != None:
            return self.parent.find_var(var_name, True)
        return -1

    def find_class(self, class_name):
        debug_log("class_name \t\t-->", class_name)
        debug_log("self.class_list \t-->", [i.name for i in self.class_list])
        debug_log("self.parent \t\t-->", self.parent)
        for e in self.class_list:
            if e.name == class_name:
                debug_log("return e", e.name)
                return e
        if self.parent != None:
            return self.parent.find_class(class_name)
        return -1

    def find_funct(self, funct_name):
        debug_log("funct_name \t\t-->", funct_name)
        debug_log("self.funct_list \t-->", [i.name for i in self.funct_list])
        debug_log("self.parent \t\t-->", self.parent)
        for e in self.funct_list:
            if e.name == funct_name:
                return e
        if self.parent != None:
            return self.parent.find_funct(funct_name)
        return -1

    def check_operation(self, input: str):
        debug_log("input operation \t-->", input)
        if len(input) == 0:
            return None
        if input[0] == "[" and input[-1] == "]":
            tmp = input[input.find("[") + 1:input.rfind("]")
                        ].strip().split(",")
            if len(tmp) == 1 and tmp[0] == "":
                debug_log("Returning []")
                return []
            tmp = [self.check_operation(e.strip()) for e in tmp]
            return tmp

        cprOp = ["!=", "==", "<=", ">=",
                 "<", ">", "+", "/", "*", "-", "^", "%", "(", ")"]
        word_op = ["and", "or", "not"]

        tmp_2 = input.strip()

        str_tmp = ""
        list_str = []
        in_quote = False
        for c in tmp_2:
            if c == "\"":
                if in_quote:
                    list_str.append(str_tmp)
                    str_tmp = ""
                    in_quote = False
                else:
                    in_quote = True
                continue
            if in_quote:
                str_tmp += c
                continue

        debug_log("list_str \t\t-->", list_str)
        for i, e in enumerate(list_str):
            while tmp_2.count(f"\"{e}\""):
                tmp_2 = tmp_2.replace(f"\"{e}\"", f"$__str__{i}$")

        debug_log("tmp_2 \t\t\t-->", tmp_2)
```

```python
for i, e in enumerate(cprOp):
    while tmp_2.count(e):
        tmp_2 = tmp_2.replace(e, f"$__op__{i}$")

for i, e in enumerate(word_op):
    prob_case = [f" {e} ",
                 f"){e}(",
                 f"){e} ",
                 f" {e}("]
    for a in prob_case:
        while tmp_2.count(a):
            tmp_2 = tmp_2.replace(a, f"$__w_op__{i}$")

if tmp_2.find("not ") == 0:
    tmp_2 = tmp_2.replace("not ", "$__w_op__2$")

tmp_ls = tmp_2.split("$")

# Check for list/array
ls_tmp = []
str_tmp = ""
bracket_level = 0
for e in tmp_ls:
    if len(e) == 0:
        continue
    if e[:7] == "__str__":
        str_in = int(e[7:])
        e = f"\"{list_str[str_in]}\""
    elif e[:8] == "__w_op__":
        w_op_in = int(e[8:])
        e = word_op[w_op_in]
    elif e[:6] == "__op__":
        op_in = int(e[6:])
        e = cprOp[op_in]
    if e.count("[") != e.count("]"):
        bracket_level += e.count("[") - e.count("]")
    if bracket_level == 0:
        ls_tmp.append(str_tmp + e)
        str_tmp = ""
    else:
        str_tmp += e
tmp_ls = ls_tmp

# Check for function
ls_tmp = []
in_funct = 0
func_tmp = ""
funct_index = False
debug_log("tmp_ls operation \t-->", tmp_ls)
for i, e in enumerate(tmp_ls):
    e = e.strip()
    if e in ["", " "]:
        continue
    if i == 0:
        ls_tmp.append(e)
        continue
    if e == "(":
        debug_log("ls_tmp[- 1] \t\t-->", ls_tmp[-1])
        if ls_tmp[-1][0].isalpha() and not ls_tmp[-1] in cprOp:
            funct_index = True
            in_funct += 1
        elif in_funct > 0:
            in_funct += 1
    if e == ")" and in_funct:
        in_funct -= 1
    if in_funct > 0:
        func_tmp += e
    if in_funct == 0:
        if funct_index:
            ls_tmp[-1] += func_tmp + e
            funct_index = False
        else:
            ls_tmp.append(e)
        func_tmp = ""
tmp_ls = ls_tmp

# Operation after formatting
debug_log("tmp_ls after \t\t-->", tmp_ls)
for i, e in enumerate(tmp_ls):
    e = e.strip()
    if e in ["", " "]:
        continue
    if e in cprOp or e in word_op:
        continue
    try:
        tmp_ls[i] = float(e)
        continue
    except:
        pass

    ok, res = self.checkkeyword(e)
    if ok:
        debug_log("res from keyword -->", res)
        tmp_ls[i] = res
    elif e == "NULL":
        tmp_ls[i] = None
    elif e.isdigit() or isString(e):
        tmp_ls[i] = e
    elif e.find(".") != -1:
        obj = e[:e.find(".")]
        e = e[e.find(".") + 1:]
        found_var = self.find_var(obj, True)
        if found_var != -1:
            if type(found_var.value) != class_var:
                print("error:", obj, "is not a class")
                return None
            var_val = found_var.value.get_var(e)
            if var_val == None:
                print("error:", e, "is not defined")
                return None
            tmp_ls[i] = var_val.value
        else:
```

```
                    print("error:", obj, "is not defined")
                    return
            elif e.find("[") != -1:
                arr_index = e[e.find("[") + 1:e.rfind("]")].strip()
                arr_index = self.check_operation(arr_index)
                e = e[:e.find("[")]
                found_var = self.find_var(e, True)
                if found_var != -1:
                    if type(found_var.value) != list:
                        print("error:", e, "is not an array")
                        return None
                    if arr_index + 1 > len(found_var.value):
                        print("error: index out of range")
                        return None
                    else:
                        tmp_ls[i] = found_var.value[arr_index]
                else:
                    print("error:", e, "is not defined")
                    return None
            else:
                found_var = self.find_var(e, True)
                if found_var != -1:
                    if type(found_var.value) == list:
                        val = found_var.value
                        return [e_arr for e_arr in val]
                    else:
                        tmp_ls[i] = found_var.value
                else:
                    print("error:", e, "is not defined")
                    return None
        debug_log("tmp_ls operation \t-->", tmp_ls)
        try:
            ls = calculateLogic(logicPostfix(tmp_ls))
            return ls
        except:
            debug_log("error in calculateLogic")
            return tmp_ls[0] if len(tmp_ls) == 1 else None

    def check_assigment(self, input: str):
        debug_log("input assign \t\t-->", input)
        bracketIndex = input.find('(')
        asgnIndex = input.find('=')
        if asgnIndex == -1 or (bracketIndex != -1 and bracketIndex < asgnIndex):
            return self.checkkeyword(input.strip())
        tmpVar_name = input[0:asgnIndex].strip()
        tmpVar_val = input[asgnIndex+1:].strip()
        tmpVar_scope = "local"

        if tmpVar_name.find(" ") != -1:
            splt = tmpVar_name.split(" ")
            tmpVar_name = splt[-1]
            for e in splt[:-1]:
                if e == "global":
                    tmpVar_scope = "global"

        try:
            tmpVar_val = float(tmpVar_val)
        except:
            debug_log("Masuk check Operation")
            tmpVar_val = self.check_operation(tmpVar_val)
            debug_log("tmpVar_val \t\t-->", tmpVar_val)
        debug_log("val \t\t\t-->", tmpVar_val)
        if tmpVar_val == None:
            return
        arr_index = -1
        if tmpVar_name.find("[") != -1:
            arr_index = tmpVar_name[tmpVar_name.find(
                "[") + 1:tmpVar_name.rfind("]")].strip()
            arr_index = self.check_operation(arr_index)
            tmpVar_name = tmpVar_name[:tmpVar_name.find("[")]

        obj = None
        if "." in tmpVar_name:
            obj = tmpVar_name[tmpVar_name.find(".") + 1:]
            tmpVar_name = tmpVar_name[:tmpVar_name.find(".")]

        old_var = self.find_var(
            tmpVar_name, True if tmpVar_scope == "global" else False)
        if old_var == -1:
            self.var_list.append(var(tmpVar_name, tmpVar_val))
        else:
            if arr_index != -1:
                while len(old_var.value) < arr_index + 1:
                    old_var.value.append(None)
                old_var.value[arr_index] = tmpVar_val
            elif obj != None:
                if type(old_var.value) != class_var:
                    print("error:", tmpVar_name, "is not a class")
                    return
                var_val = old_var.value.get_var(obj)
                if var_val == None:
                    old_var.value.var_list.append(var(obj, tmpVar_val))
                else:
                    var_val.value = tmpVar_val
            else:
                old_var.value = tmpVar_val

    def checkkeyword(self, inp: str) -> tuple[bool, any]:
        debug_log("input keyword \t-->", inp)
        if inp.find("(") == -1:
            return (False, None)

        obj = None
        key = None
        arg = None

        def splitting(ch, str):
            bracket_level = 0
            # print("str",str)
            for i, c in enumerate(reversed(str)):
                # print("c",c)
                # print("bracket_level",bracket_level)
                if c == "(":
                    bracket_level += 1
                if c == ")":
                    bracket_level -= 1
                if bracket_level == 0 and c == ch:
                    return (str[:len(str) - i - 1], str[len(str) - i:])
            # print("return None")
            return None

        if "." in inp:
            if splitting(".", inp) != None:
                obj, key = splitting(".", inp)
                arg = key[key.find("(") + 1:key.rfind(")")].strip().split(",")
                key = key[:key.find("(")]

        key = inp[:inp.find("(")] if key == None else key
        arg = inp[inp.find("(") + 1:inp.rfind(")")
                  ].strip().split(",") if arg == None else arg

        debug_log("inp_key \t\t-->", key)
        debug_log("key \t\t\t-->", key)
        debug_log("obj \t\t\t-->", obj)
        if obj != None:
            old_var = None
            if obj.find("(") != -1:
                ok, obj = self.checkkeyword(obj)
                if not ok:
                    print("error:", obj, "is not defined")
                    return (False, None)
            else:
                old_var = self.find_var(obj, True)
                if old_var == -1:
                    print("error:", obj, "is not defined")
                    return (True, None)
                else:
                    obj = old_var.value
                if type(obj) != class_var:
                    print("error:", obj, "is not a class")
                    return (True, None)

            match key:
                case "var_list":
                    tmp = []
                    for e in obj.var_list:
                        tmp.append((e.name, e.value))
                    return (True, tmp)
                case "funct_list":
                    tmp = []
                    for e in obj.funct_list:
                        tmp.append((e.name, e.arg))
                    return (True, tmp)

            return (True, obj.exec_funct(key, arg))

        arg_tmp = []
        str_tmp = ""
```

```
bracket_level = 0

for e in arg:
    if len(e) == 0:
        continue
    if e.count("(") != e.count(")"):
        bracket_level += e.count("(") - e.count(")")
    if bracket_level == 0:
        arg_tmp.append(str_tmp + e)
        str_tmp = ""
    else:
        str_tmp += e + ","
arg = arg_tmp

arg_tmp = []
str_tmp = ""
total_quote = 0

for e in arg:
    if len(e) == 0:
        continue
    if e.count("\"") != 0:
        total_quote += e.count("\"")
    if total_quote % 2 == 0:
        arg_tmp.append(str_tmp + e)
        str_tmp = ""
    else:
        str_tmp += e + ","
arg = arg_tmp

debug_log("key \t\t\t-->", key)
debug_log("arg \t\t\t-->", arg)

if key == "out":
    str_to_print = ""
    break_line = True
    for e in arg:
        e = e.strip()
        if e == "no_break":
            break_line = False
            continue
        if len(e) == 0:
            continue
        if not isString(e):
            e = self.check_operation(e)
        if type(e) == str and isString(e):
            e = e[1:-1]
        str_to_print += str(e)
    str_to_print = str_to_print.replace("\\n", "\n").replace(
        "\\t", "\t").replace("None", "NULL")
    print(str_to_print, end="")
    if break_line:
        print()
    return (True, None)
if key == "in":
    if len(arg) == 0:
```

```
                input()
                return (True, None)
            val = input()
            if val.isdigit():
                val = float(val)
            else:
                val = "\"" + val + "\""
            for e in arg:
                e.strip()
                if len(e) == 0:
                    continue
                old_var = self.find_var(e, True)
                if old_var == -1:
                    self.var_list.append(var(e, val))
                else:
                    old_var.value = val
            return (True, None)
        if key == "sizeof":
            if len(arg) == 0:
                return (True, None)
            old_var = self.find_var(arg[0], True)
            if old_var == -1:
                print("error:", arg[0], "is not defined")
                return (True, None)
            if type(old_var.value) == list:
                return (True, len(old_var.value))
            else:
                print("error:", arg[0], "is not an array")
                return (True, None)
        func = self.find_funct(key)
        if func != -1:
            for i, e in enumerate(arg):
                e = e.strip()
                e = self.check_operation(e)
                arg[i] = e
            return (True, func.exec(arg, self))
        cls = self.find_class(key)
        if cls != -1:
            for i, e in enumerate(arg):
                e = e.strip()
                e = self.check_operation(e)
                arg[i] = e
            # print("arg",arg)
            return (True, class_var(cls, arg, self))
        return (False, None)

    def execline(self, input: str):
        input = input.strip()
        if len(input) == 0:
            return
        cmt_index = input.find("#")
        if cmt_index != -1:
            input = input[:cmt_index]
        if input[:4] == "goto":
            self.exec_goto(input)
            return
        if input[:len('import')] == "import":
            mod = input[len('import'):].strip()
            if not os.path.exists(mod):
                print("error :", mod, "does not exist")
                return
            self.execfile(mod)
            return
        res = self.check_assigment(input)
        return res

    def exec_if(self, inp: list[str]):
        block_k = ["if", "while", "fn"]

        ls_cond = []
        ls_task = []

        tmp = ""
        nested = 0

        for e in inp:
            es = e.strip()
            for k in block_k:
                if es[:len(k)] == k:
                    nested += 1
                    break
            if e == "end":
                nested -= 1
            if nested > 1:
                tmp += es + '\n'
                continue

            cond_key = es if es == "else" else es[:es.find(" ")]
            # debug_log("es \t\t\t-->",es)
            # debug_log("conditional key \t-->",cond_key)

            if cond_key == 'if':
                ls_cond.append(e[2:].strip())
            elif cond_key == 'else_if':
                ls_cond.append(e[7:].strip())
                ls_task.append(tmp)
                tmp = ""
            elif cond_key == 'else':
                ls_cond.append('1')
                ls_task.append(tmp)
                tmp = ""
            else:
                tmp += es + '\n'
        ls_task.append(tmp)
        debug_log("condition \t\t-->", ls_cond)

        for condition, task in zip(ls_cond, ls_task):
            if self.check_operation(condition):
                return self.execstring(task)
                # return interpreter(self).execstring(task)
        return (False, None)
```

```python
def exec_while(self, inp):
    condition = inp[0][inp[0].find("while") + len('while'):]
    condition.strip()
    str = ""
    for e in inp[1:]:
        str += e + "\n"
    while self.check_operation(condition):
        ok, ret = self.execstring(str)
        # ok,ret = interpreter(self).execstring(str)
        if ok:
            return (True, ret)
    return (False, None)

def exec_goto(self, inp):
    label_name = inp[4:].strip()
    for e in self.label_list:
        if e.name == label_name:
            in_goto = True
            goto_line = e.line
            str_to_exec = ""
            for lin in self.line_done[goto_line:]:
                str_to_exec += lin + "\n"
            ok, ret = self.execstring(str_to_exec)
            in_goto = False
            if ok:
                return (True, ret)
            return (False, None)
    return (False, None)

def def_fn(self, inp):
    name = inp[0][inp[0].find("fn") + len('fn'):inp[0].find("(")].strip()
    arg = inp[0][inp[0].find("(") + 1:inp[0].rfind(")")].strip().split(",")
    arg = [e.strip() for e in arg]
    content = ""
    for e in inp[1:]:
        content += e + "\n"
    fn_tmp = funct(name, content, arg)
    self.funct_list.append(fn_tmp)

def def_class(self, inp):
    name = inp[0][inp[0].find("class") + len('class'):].strip()
    content = ""
    for e in inp[1:]:
        content += e + "\n"
    self.class_list.append(class_def(name, content))

def execstring(self, input, is_root=False):
    block_k = ["if", "while", "fn", "execPy", "class"]
    input = input.split("\n")
    in_block = 0
    block_type = ""
    block_content = []
    for i, l in enumerate(input):
        l = l.strip()
        if is_root and not self.in_goto:
            self.line_done.append(l)
            if len(l) == 0:
                continue
            if l[-1] == ':':
                label_name = l[:-1].strip()
                self.label_list.append(label(label_name, i))
                continue

            key = l[:l.find(" ")] if l.find(" ") != -1 else l
            if key in block_k:
                if not in_block:
                    block_type = key
                in_block += 1
            if in_block > 0 and key == "end":
                in_block -= 1
                if in_block == 0:
                    match block_type:
                        case "if":
                            ok, ret = self.exec_if(block_content)
                            if ok:
                                return (True, ret)
                        case "while":
                            ok, ret = self.exec_while(block_content)
                            if ok:
                                return (True, ret)
                        case "fn":
                            self.def_fn(block_content)
                        case "execPy":
                            str_to_exec = ""
                            for lin in block_content[1:]:
                                str_to_exec += lin + "\n"
                            exec(str_to_exec)
                        case "class":
                            self.def_class(block_content)
                    block_content.clear()
                    continue
            if in_block:
                block_content.append(l)
            else:
                if key == "return":
                    ret = self.check_operation(l[6:].strip())
                    return (True, ret)
                self.execline(l)
        return (False, None)

    def execfile(self, path, chdir=False):
        os.chdir(os.path.dirname(path)) if chdir else None
        file_in = open(os.path.basename(path) if chdir else path, "r")
        self.execstring(file_in.read(), True)
        file_in.close()


def main():
    it = interpreter()
    line_input = ""
    input_tmp = ""
```

```
key_block = ["if", "while", "fn", "execPy", "class"]
in_block = 0

if len(sys.argv) == 1 or sys.argv[1] == "-i":
    if len(sys.argv) == 3:
        print(">> import", sys.argv[2])
        it.execfile(sys.argv[2], True)
        print()
    no_print = False
    while not line_input in ["/q", "/quit", "/exit"]:
        if not no_print:
            if in_block:
                line_input = input(".. ").strip()
            else:
                line_input = input(">> ").strip()
        else:
            no_print = False
            line_input = input().strip()
        input_tmp += "\n" + line_input

        key = line_input[:line_input.find(" ")] if line_input.find(
            " ") != -1 else line_input
        arg = line_input[line_input.find(" "):]

        debug_log("key \t\t\t-->", key)
        if key in key_block:
            in_block += 1
        if key == "cd":
            arg = arg.strip()
            if os.getcwd() == arg:
                return
            # print('cd',arg)
            no_print = True
            debug_log(f"is {arg} path ?",
                      "Yes" if os.path.isdir(arg) else "No")
            if os.path.isdir(arg):
                os.chdir(arg)
            continue
        if line_input == "lab":
            ls = [e.name for e in it.label_list]
            print(ls)
            continue
        if line_input == "cls":
            os.system("cls")
            continue
        if line_input == "wd":
            print(os.getcwd())
            continue
        if line_input == "var":
            ls = [(e.name, e.value) for e in it.var_list]
            for e in ls:
                print(e, sep="\n")
            continue
        if line_input == "funct":
            ls = [(e.name, e.arg) for e in it.funct_list]
            print(ls)
```

```
                continue
            if line_input == "it":
                print("interpreter", it)
                continue
            if line_input == "end":
                in_block -= 1
            if in_block == 0:
                ok, ret = it.execstring(input_tmp)
                print(ret) if ret != None else None
                input_tmp = ""

    else:
        it.execfile(sys.argv[1], True)
        input("\nPress Enter to continue...")


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

1.2 Text Editor

```
import customtkinter as ctk
import tkinter as tk
import tkinter.messagebox as tkmessagebox

import subprocess
import os
from syntax_identifier import syntax_identifier
from LogicOperation import isOpLgc
import threading

app = ctk.CTk()
app.title("PyHaekal IDE")
app.geometry("1280x720")

main_frame = ctk.CTkFrame(app)

frame1 = ctk.CTkFrame(main_frame)
frame1.grid(row=2, column=0, padx=(10, 0), pady=(0, 10), sticky="nsew")
frame1.grid_remove()

frame2 = ctk.CTkFrame(main_frame, fg_color="transparent")
frame2.grid(row=2, column=1, padx=(10, 5), pady=(0, 10), sticky="nsew")

folder_frame = ctk.CTkScrollableFrame(frame1)
folder_frame.grid(row=10, column=0, padx=5, pady=5, sticky="nsew")
folder_frame.grid_remove()
frame1.grid_rowconfigure(10, weight=1)

frame1.grid_columnconfigure(0, weight=1)

# app.grid_columnconfigure(0,weight=1)
main_frame.grid_columnconfigure(1, weight=3)
main_frame.grid_rowconfigure(2, weight=1)

text_box = ctk.CTkTextbox(frame2, activate_scrollbars=True)
```

```
frame2.grid_columnconfigure(0, weight=1)
frame2.grid_rowconfigure(0, weight=3)
text_box.grid(row=0, column=0, padx=0, pady=0, sticky="nsew")

text_box.cget("font").configure(size=18)
text_box.cget("font").configure(family="fira code")
text_box.configure(undo=True)


def tab_pressed(event: tk.Event) -> str:
    text_box.insert(tk.INSERT, "    ")
    return "break"


text_box.bind("<Tab>", tab_pressed)

current_file = ""
current_folder = ""


class index_counter:
    def __init__(self):
        self.index = 0

    def reset(self):
        self.index = 0

    def get(self):
        a = self.index
        self.index += 1
        return a


current_file_selected = None


class file_button:
    def __init__(self, name, path, depth, index: index_counter):
        self.label = os.path.basename(path)
        self.path = path

        self.button = ctk.CTkButton(
            folder_frame, text="  " * depth + "  " + name, command=lambda: (self.select(), openFil

        self.button.grid(row=index.get(), column=0,
                         padx=2, pady=2, sticky="ew")
        self.button.grid_remove()
        self.button.configure(anchor="w")
        self.button.cget("font").configure(size=14)
        self.button.cget("font").configure(family="fira code")
        self.button.configure(text_color=("black", "white"))
        self.button.configure(fg_color="transparent")

    def select(self):
        global current_file_selected
        if current_file_selected != None:
            current_file_selected.unselect()
        current_file_selected = self
        self.button.configure(fg_color="gray")

    def unselect(self):
        self.button.configure(fg_color="transparent")

    def destroy(self):
        self.button.destroy()


class folder_button:
    def __init__(self, name, path, depth, index: index_counter):
        self.index = index
        self.label = os.path.basename(path)
        self.path = path
        self.showed = False
        self.depth = depth
        self.files = []
        self.button = ctk.CTkButton(
            folder_frame, text="  " * depth + "→ " + name, command=self.toggle)
        self.button.grid(row=self.index.get(), column=0,
                         padx=2, pady=2, sticky="ew")
        self.button.grid_remove()
        self.depth = depth
        self.listfiles()
        self.button.cget("font").configure(size=14)
        self.button.cget("font").configure(family="fira code")
        self.button.configure(anchor="w")
        self.button.configure(text_color=("black", "white"))
        self.button.configure(fg_color="transparent")
        if self.depth == 0:
            self.button.grid()

    def toggle(self):
        if self.showed:
            self.hide()
        else:
            self.show()

    def show(self):
        self.button.configure(text="  " * self.depth + "↓ " + self.label)
        self.showed = True
        for e in self.files:
            e.button.grid()

    def hide(self):
        self.button.configure(text="  " * self.depth + "→ " + self.label)
        for e in self.files:
            if isinstance(e, folder_button):
                e.hide()
            e.button.grid_remove()
        self.showed = False

    def destroy(self):
        self.button.destroy()
        for e in self.files:
            e.destroy()

    def listfiles(self):
        def list_f():
            tmp_files = []
            for e in os.listdir(self.path):
                try:
                    if os.path.isdir(os.path.join(self.path, e)):
                        self.files.append(folder_button(
                            e, os.path.join(self.path, e), self.depth + 1, self.index))
                    else:
                        tmp_files.append(e)
                except:
                    continue
            for e in tmp_files:
                self.files.append(file_button(e, os.path.join(
                    self.path, e), self.depth + 1, self.index))
            if self.depth == 0:
                self.show()
        if self.depth == 0:
            thread = threading.Thread(target=list_f)
            thread.start()
        else:
            list_f()


current_folder_button = None


def openFolder():
    global current_folder, current_folder_button, current_file_selected

    current_file_selected = None

    directory = ctk.filedialog.askdirectory()
    if directory == "":
        return
    if current_folder_button != None:
        current_folder_button.destroy()
    current_folder = directory
    proc.stdin.write("cd " + current_folder + '\n')
    proc.stdin.flush()

    current_folder_button = folder_button(
        os.path.basename(directory), directory, 0, index_counter())
    frame1.grid()
    folder_frame.grid()


def refreshFolder():
    global current_folder, current_folder_button
    if current_folder == "":
        return
    if current_folder_button != None:
        current_folder_button.destroy()
    folder_frame.grid()
    current_folder_button = folder_button(
        os.path.basename(current_folder), current_folder, 0, index_counter())
    current_folder_button.show()


def refreshCode():
    current_code = text_box.get('1.0', 'end')
    if current_code.strip() == '':
        return
    insert_cursor = text_box.index(tk.INSERT)
    scroll_pos = text_box.yview()
    text_box.delete("1.0", "end")
    word_color = {}
    block = ['fn', 'if', 'while', 'else',
             'else_if', 'end', 'return', 'class', ':', 'import', 'goto']
    for e in block:
        word_color.update({e: 'block'})
    # print('current_file ',current_file)
    # print('current_folder ',os.path.dirname(current_file))
    identifier = syntax_identifier(os.path.dirname(current_file))
    identifier.identify_string(current_code)
    for e in identifier.var_list:
        word_color.update({e: 'var'}) if e not in word_color else None
    for e in identifier.funct_list:
        word_color.update({e: 'funct'}) if e not in word_color else None
    for e in identifier.class_list:
        word_color.update({e: 'class'}) if e not in word_color else None
    for e in identifier.label_list:
        word_color.update({e: 'label'}) if e not in word_color else None

    for l, line in enumerate(current_code.split('\n')):
        # print('line',line)
        # stripped = line.lstrip()
        # text_box.insert('end',' ' * (len(line) - len(stripped)))
        quoted = 0
        tmp = ''
        in_comment = False
        for i, char in enumerate(line):
            separator = [' ', '\t', ',', '(', ')', '[', ']', ':',
                         '.', '=', '+', '/', '*', '-', '^', '%', '<', '>']
            if char == '"':
                text_box.insert('end', char, 'quote')
                quoted += 1
                continue
            if quoted % 2 != 0:
                text_box.insert('end', char, 'string')
                continue
            if char == '#':
                in_comment = True
            if in_comment:
                text_box.insert('end', char, 'comment')
                continue
            if char in separator or i == len(line) - 1:
                char_in_tmp = False
```

```python
                if i == len(line) - 1 and char not in separator:
                    char_in_tmp = True
                    tmp += char
                if tmp in word_color:
                    text_box.insert('end', tmp, word_color[tmp])
                elif tmp.isdigit() or tmp == 'NULL':
                    text_box.insert('end', tmp, 'digit')
                elif isOpLgc(tmp):
                    text_box.insert('end', tmp, 'operator')
                else:
                    text_box.insert('end', tmp)
                tmp = ''

                if char_in_tmp:
                    continue

                if char in word_color:
                    text_box.insert('end', char, word_color[char])
                elif isOpLgc(char):
                    text_box.insert('end', char, 'operator')
                elif char in ['[', ']', '=', '.']:
                    text_box.insert('end', char, 'operator')
                else:
                    text_box.insert('end', char)
            else:
                tmp += char
        # print('l',l)
        # print('len',len(current_code.split('\n')) - 2)
        if l < len(current_code.split('\n')) - 2:
            text_box.insert('end', '\n')
    text_box.mark_set(tk.INSERT, insert_cursor)
    text_box.yview_moveto(scroll_pos[0])


def openFilePath(path):
    save()
    global current_file
    current_file = path
    file = open(path, "r")
    text_box.delete("1.0", "end")
    text_box.insert('1.0', file.read())
    refreshCode()


def openFile():
    global current_file
    directory = ctk.filedialog.askopenfiles()
    if len(directory) == 0:
        return
    current_file = directory[0].name
    openFilePath(current_file)


def saveFile():
    global current_file
    directory = ctk.filedialog.asksaveasfilename()
    if directory == "":
        return
    current_file = directory
    file = open(directory, "w")
    file.write(text_box.get("1.0", "end"))
    file.close()
    refreshFolder()


def save(confirmed=False):
    current_doc = text_box.get("1.0", "end")
    current_doc = current_doc.strip()
    if current_doc == '' and not confirmed:
        return
    global current_file
    if current_file == "" or not os.path.exists(current_file):
        saveFile() if tkmessagebox.askyesno(
            "Save", "Do you want to save your code?") else None
        return
    if open(current_file, "r").read().strip() == current_doc:
        return
    if not confirmed and not tkmessagebox.askyesno("Save", "Do you want to save your code?"):
        return
    file = open(current_file, "w")
    file.write(text_box.get("1.0", "end"))
    file.close()


def runCurrent():
    global current_file
    if current_file == "":
        openFile()
    if current_file == "":
        return
    save()
    # print('current working directory:',os.getcwd())
    # print('current file:',current_file)

    proc.stdin.write("cd " + os.path.dirname(current_file) + '\n')
    proc.stdin.flush()

    proc.stdin.write("import " + current_file + '\n')
    proc.stdin.flush()


def run_interactive_separate():
    global current_file
    if current_file == "":
        openFile()
    if current_file == "":
        return
    save()
    subprocess.call('start ipython -- ./interpreter.py -i ' +
                    current_file, shell=True)
```

```
def run_separate():
    global current_file
    if current_file == "":
        openFile()
    if current_file == "":
        return
    save()
    subprocess.call('start ipython -- ./interpreter.py ' +
                    current_file, shell=True)


def new_file():
    save()
    global current_file
    current_file = ""
    text_box.delete("1.0", "end")
    refreshCode()


def new_folder():
    if current_folder == "":
        return
    input_dialog = ctk.CTkInputDialog(title="New Folder", text="Folder name:")
    new_dir = input_dialog.get_input()
    if new_dir == "":
        return
    os.mkdir(os.path.join(current_folder, new_dir))
    refreshFolder()


def tes():
    refreshCode()


def menu_action(action):
    match action:
        case "Open File":
            openFile()
        case "Open Folder":
            openFolder()
        case "Save":
            save(True)
        case "Save as":
            saveFile()
        case "Run":
            runCurrent()
        case "Tes":
            tes()
        case "New File":
            new_file()
        case "New Folder":
            new_folder()
    menubar.set("Menu")


menu_frame = ctk.CTkFrame(main_frame, fg_color="transparent")
```

```python
menu_frame.grid(row=0, column=0, padx=0, pady=0, sticky="nsew", columnspan=2)


def run_on_choice(choice):
    global run_menu
    match choice:
        case 'Run':
            runCurrent()
        case 'Run in external terminal':
            run_separate()
        case 'Run interactive in external terminal':
            run_interactive_separate()
        case 'Restart':
            restart()
    run_menu.set('Run')


run_menu = ctk.CTkOptionMenu(menu_frame, values=['Run', 'Run in external terminal', 'Run interacti
                             command=run_on_choice, width=100)
run_menu.configure(font=("fira code", 14))
run_menu.configure(dropdown_font=("fira code", 14))
run_menu.grid(row=0, column=1, padx=0, pady=10, sticky="nsew")


# runButton = ctk.CTkButton(menu_frame,text="Run",command=runCurrent,width=10)
# runButton.grid(row=0,column=1,padx=0,pady=10,sticky="w")


menubar = ctk.CTkOptionMenu(menu_frame, bg_color="transparent", values=[
                            'New File', "Open File", 'New Folder', "Open Folder", "Save", "Save as
menubar.set("Menu")
menubar.configure(font=("fira code", 14))
menubar.configure(dropdown_font=("fira code", 14))
# menubar._dropdown_menu.configure()

menubar.grid(row=0, column=0, padx=10, pady=10, sticky="nsew")

main_frame.pack(fill=ctk.BOTH, expand=1)
app.after(0, lambda: app.state('zoomed'))

console_frame = ctk.CTkFrame(main_frame)
console_frame.grid(row=2, column=2, padx=0, pady=(0, 10), sticky="nsew")
main_frame.grid_columnconfigure(2, weight=2)

text_box_console = ctk.CTkTextbox(console_frame, activate_scrollbars=True)
text_box_console.pack(fill=ctk.BOTH, expand=8, padx=5,
                      pady=(5, 0))
text_box_console.configure(font=("fira code", 14))

console_input = ctk.CTkEntry(
    console_frame, border_width=0, height=40, fg_color=("white", "#1e1e1e"))
console_input.pack(fill=ctk.X, pady=(10, 5), padx=5)
console_input.configure(font=("fira code", 14))


def console_on_enter(event):
    data = console_input.get() + '\n'

    text_box_console.configure(state='normal')
    text_box_console.insert('end', data)
    text_box_console.configure(state='disabled')

    if proc.poll():
        restart()
    proc.stdin.write(data)
    proc.stdin.flush()
    console_input.delete(0, 'end')


console_input.bind('<Return>', console_on_enter)
proc = subprocess.Popen('python ./interpreter.py', text=True,
                        stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

# proc = subprocess.Popen('pwsh',text=True,
# stdin=subprocess.PIPE,stdout=subprocess.PIPE,stderr=subprocess.PIPE

bef = ''


def read_proc():
    while proc.poll() is None:
        data = os.read(proc.stdout.fileno(), 1 << 20)
        # data = proc.stdout.readline()
        data = data.replace(b"\r\n", b"\n")
        decoded = data.decode()
        if data:
            if data == b'\x0c':
                text_box_console.configure(state='normal')
                text_box_console.delete('1.0', 'end')
                text_box_console.configure(state='disabled')
                continue
            global bef
            if decoded.strip()[-2:] == '>>':
                proc.stdin.write("cd " + current_folder + '\n')
                proc.stdin.flush()

            text_box_console.configure(state='normal')
            text_box_console.insert(
                'end', decoded)
            text_box_console.configure(state='disabled')
            text_box_console.see('end')
            bef = decoded
        else:
            return None


def read_err():
    while proc.poll() is None:
        data = os.read(proc.stderr.fileno(), 1 << 20)
        data = data.replace(b"\r\n", b"\n")
        decoded = data.decode()
        if data:
            if data == b'\x0c':
                text_box_console.configure(state='normal')
                text_box_console.delete('1.0', 'end')
                text_box_console.configure(state='disabled')
                continue
            text_box_console.configure(state='normal')
            text_box_console.insert('end', decoded)
            text_box_console.configure(state='disabled')
            text_box_console.see('end')
        else:
            return None


thread = threading.Thread(target=read_proc)
thread.start()

thread_err = threading.Thread(target=read_err)
thread_err.start()


def restart():
    global proc, thread, thread_err
    proc = subprocess.Popen('python ./interpreter.py', text=True,
                            stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE
    thread = threading.Thread(target=read_proc)
    thread.start()

    thread_err = threading.Thread(target=read_err)
    thread_err.start()


def on_closing():
    proc.terminate()
    save()
    if tkmessagebox.askyesno("Quit", "Do you want to quit?"):
        app.destroy()


app.protocol("WM_DELETE_WINDOW", on_closing)

refresh_cycle_delay = 1000


def set_refresh_cycle_delay(x):
    global refresh_cycle_delay
    x = x.replace(' ms', '')
    refresh_cycle_delay = int(x)


delay_label = ctk.CTkLabel(menu_frame, text="Refresh delay :")
delay_label.grid(row=0, column=2, padx=(10, 0), pady=10, sticky="nsew")
delay_label.configure(font=("fira code", 14))

delay_input = ctk.CTkOptionMenu(
    menu_frame, values=['100 ms', '500 ms', '1000 ms', '2000 ms', '3000 ms', '4000 ms', '5000 ms'
delay_input.grid(row=0, column=3, padx=10, pady=10, sticky="nsew")
delay_input.configure(font=("fira code", 14))
delay_input.set('1000 ms')

is_auto_save = False
auto_save_delay = 1000


def set_auto_save_delay(x):
    global auto_save_delay
    x = x.replace(' ms', '')
    auto_save_delay = int(x)


auto_save_delay_input = ctk.CTkOptionMenu(menu_frame, values=[
                                          '100 ms', '500 ms', '1000 ms', '2000 ms', '3000 ms', '40
auto_save_delay_input.grid(row=0, column=6, padx=10, pady=10, sticky="nsew")
auto_save_delay_input.configure(font=("fira code", 14))
auto_save_delay_input.set('1000 ms')
auto_save_delay_input.grid_remove() if not is_auto_save else None


def set_auto_save():
    global is_auto_save
    is_auto_save = auto_save_input.get()
    auto_save_delay_input.grid() if is_auto_save else auto_save_delay_input.grid_remove()
    # print(is_auto_save)


auto_save_label = ctk.CTkLabel(menu_frame, text="Auto save")
auto_save_label.grid(row=0, column=4, padx=(10, 0), pady=10, sticky="nsew")
auto_save_label.configure(font=("fira code", 14))

auto_save_input = ctk.CTkSwitch(
    menu_frame, text='', command=set_auto_save, width=20)
auto_save_input.grid(row=0, column=5, padx=(10, 0), pady=10,
                     sticky="nsew")
auto_save_input.configure(font=("fira code", 14))


def refresh_cycle():
    # print('refresh',refresh_cycle_delay)
    refreshCode()
    app.after(refresh_cycle_delay, refresh_cycle)


def auto_save_cycle():
    # print('auto save cycle')
    # print('auto save',is_auto_save)
    if is_auto_save and current_file != "" and text_box.get("1.0", "end").strip() != '':
        print('auto save')
        save(True)
    app.after(auto_save_delay, auto_save_cycle)


def switch_mode():
    ctk.set_appearance_mode('dark' if mode_switch.get() else 'light')
    tag_color = {
        'block': ('brown1', 'brown4'),
        'class': ('yellow', 'yellow4'),
        'funct': ('aqua', 'blue4'),
        'var': ('orange', 'orange4'),
        'digit': ('yellow', 'yellow4'),
        'operator': ('cadetblue', 'cadetblue4'),
        'label': ('brown1', 'brown4'),
        'comment': ('gray', 'gray20'),
        'string': ('white', 'black'),
        'quote': ('orange', 'orange4')
    }

    if mode_switch.get():
        for e in tag_color:
            text_box.tag_config(e, foreground=tag_color[e][0])
    else:
        for e in tag_color:
            text_box.tag_config(e, foreground=tag_color[e][1])


mode_switch = ctk.CTkSwitch(menu_frame, text='Dark mode', width=20)
mode_switch.grid(row=0, column=7, padx=(10, 0), pady=10, sticky="nsew")
mode_switch.configure(font=("fira code", 14))
mode_switch.configure(command=switch_mode)
mode_switch.toggle()

switch_mode()

app.after(auto_save_delay, auto_save_cycle)
app.after(refresh_cycle_delay, refresh_cycle)

os.chdir(os.path.dirname(os.path.realpath(__file__)))
app.iconbitmap('Icon.ico')
# print('current working directory:',os.getcwd())
app.mainloop()
```

1.3 Logic Operation

```
def isString(n: str):
    return (n[0] == "\"" and n[-1] == "\"" and n.count('"') == 2)


def cprVal(n):
    if n in ["not"]:
        return 2
    if n in ["and", "or"]:
        return 1
    if n in ["!=", "==", "<=", ">=", "<", ">"]:
        return 3
    match n:
        case "+" | "-": return 4
        case "*" | "/": return 5
        case "^": return 6
```

```python
def checkBracketLs(n: list):
    openBracket = n.count("(")
    closeBracket = n.count(")")
    if openBracket > closeBracket:
        for i in range(openBracket - closeBracket):
            n.append(")")
    else:
        for i in range(closeBracket - openBracket):
            n.insert(0, "(")
    return n


def isOpLgc(c):
    cprOp = ["and", "or", "not", "!=", "==", "<=", ">=",
             "<", ">", "+", "/", "*", "-", "^", "%", "(", ")"]
    return c in cprOp


def logicPostfix(ls: list):
    n = checkBracketLs(ls)
    result = []
    aux = []

    for c in n:
        if c in [" ", ""]:
            continue
        if not isOpLgc(c):
            result.append(c)
        elif c == '(':
            aux.append(c)
        elif c == ')':
            while len(aux) > 0 and aux[-1] != '(':
                result.append(aux.pop())
            aux.pop()
        else:
            while len(aux) > 0 and aux[-1] != "(" and cprVal(aux[-1]) >= cprVal(c):
                result.append(aux.pop())
            aux.append(c)

    while len(aux) > 0:
        result.append(aux.pop())
    return result


def calculateLogic(n: list):
    # print("calculateLogic",n)
    OPERATION = {
        "and": lambda a, b: int(a and b),
        "or": lambda a, b: int(a or b),
        "not": lambda a: int(not a),
        "!=": lambda a, b: int(a != b),
        "==": lambda a, b: int(a == b),
        "<=": lambda a, b: int(a <= b),
        ">=": lambda a, b: int(a >= b),
        "<": lambda a, b: int(a < b),
        ">": lambda a, b: int(a > b),
        "+": lambda a, b: a + b,
        "/": lambda a, b: a / b,
        "*": lambda a, b: a * b,
        "-": lambda a, b: a - b,
        "^": lambda a, b: a ** b,
        "%": lambda a, b: a % b
    }
    result = []
    for c in n:
        # print("c",c)
        # print("result",result)
        try:
            result.append(float(c))
            continue
        except:
            pass

        if isString(c):
            result.append(c[1:-1])
            continue
        a = 0
        b = result.pop()
        if c != "not":
            if not (len(result) == 0 or isOpLgc(result[-1])):
                a = result.pop()
        # print("a",a)
        # print("b",b)
        a = int(a) if type(a) == float and a % 1 == 0 else a
        b = int(b) if type(b) == float and b % 1 == 0 else b
        # print("1 a",a)
        # print("1 b",b)
        try:
            if c == "not":
                tmp = OPERATION[c](b)
            else:
                tmp = OPERATION[c](a, b)
            result.append(tmp)
        except Exception as e:
            print("error:", e)
            return None

    # print("result after op",result)
    res = result.pop()
    if type(res) == float:
        if res.is_integer():
            return int(res)
    if type(res) == str:
        return "\"" + res + "\""
    return res
```

1.4 Syntax Identifier

```
import os
```

```python
class syntax_identifier:
    def __init__(self, work_dir: str = None):
        if work_dir is not None and os.path.exists(work_dir):
            self.work_dir = work_dir
        self.var_list = []
        self.funct_list = ['out', 'in', 'sizeof', 'funct_list', 'var_list']
        self.class_list = ['this']
        self.label_list = []

    def identify_string(self, str):
        lines = str.split('\n')
        for line in lines:
            self.identify_line(line)

    def identify_file(self, file):
        self.identify_string(file.read())

    def identify_line(self, line: str):
        line = line.strip()
        if line == '':
            return
        if line[-1] == ':':
            self.label_list.append(line[:-1])
            return
        line = line.replace('(', ' ( ')
        line = line.replace(')', ' ) ')
        line = line.replace('=', ' = ')
        line = line.replace('.', ' . ')
        line = line.replace(',', ' ,')
        line = line.replace('[', ' [ ')
        line = line.replace(']', ' ] ')

        tokens = line.split(' ')
        tokens = list(filter(lambda x: x != '', tokens))
        tokens = list(filter(lambda x: x != ',', tokens))
        # tokens = list(filter(lambda x: x != '(',tokens))
        # tokens = list(filter(lambda x: x != ')',tokens))
        for i in range(len(tokens)):
            tokens[i] = tokens[i].strip()
        key = tokens[0]
        match key:
            case 'import':
                if len(tokens) < 2:
                    return
                path = ''
                for i in tokens[1:]:
                    path += i
                # print('import',path)
                try:
                    # print('current working directory:',os.getcwd())
                    # print('import',path)
                    file = open(os.path.join(self.work_dir, path), 'r')
                    self.identify_file(file)
                    file.close()
                except:
                    pass
            case 'fn':
                if len(tokens) < 2:
                    return
                self.funct_list.append(
                    tokens[1]) if tokens[1] not in self.funct_list else None
                for e in tokens[3:tokens.index(')') if ')' in tokens else None]:
                    self.var_list.append(
                        e) if e not in self.var_list else None
            case 'class':
                if len(tokens) < 2:
                    return
                self.class_list.append(
                    tokens[1]) if tokens[1] not in self.class_list else None
        if len(tokens) < 2:
            return
        if tokens[1] == '=' or tokens[1] == '.' or tokens[1] == '[':
            self.var_list.append(
                tokens[0]) if tokens[0] not in self.var_list else None
            if tokens[1] == '.' and tokens[3] != '(':
                self.var_list.append(
                    tokens[2]) if tokens[2] not in self.var_list else None


if __name__ == '__main__':
    file = open('Script/input.pyhk', 'r')
    identifier = syntax_identifier()
    identifier.identify_string(file.read())
    print('var', identifier.var_list)
    print('fn', identifier.funct_list)
    print('class', identifier.class_list)
    print('label', identifier.label_list)

    file.close()
```

2 Alur Program

2.1 Interpreter

Interpreter adalah program console yang akan membaca suatu perintah lalu menjalankan perintah tersebut. Terdapat dua mode yaitu interactive mode dan non-interactive mode. Non-interactive mode dapat dijalankan dengan command py interpreter.py [namafile]. Sedangkan interactive mode dapat dijalankan dengan command py interpreter.py atau py interpreter.py -i [namafile]. Non-interactive akan membaca dan mengeksekusi file lalu selesai. Sedangkan interactive akan membaca dan mengeksekusi file bila diberikan lalu membaca input dari user. Bila diberikan input file maka interpreter akan membaca file tersebut lalu mengeksekusi string yang didapat dengan fungsi execstring dari class Interpreter. Pada fungsi execstring, string akan dipecah menjadi per baris. Lalu diiterasikan dan diperiksa apakah string atau bagian dari string itu adalah block atau tidak. Bila tidak maka akan langsung dimasukkan pada fungsi `execline`. Bila iya maka akan memanggil fungsi `exec_if`, `exec_while`, `def_fn`, `def_class` tergantung dari jenis block tersebut.

2.1.1 Fungsi `execline`

Fungsi execline akan menghapus seluruh line yang ada di belakang "#". Selanjutnya akan diperiksa apakah ada goto atau tidak. Jika iya maka fungsi `exec_goto` akan dijalankan. Dimana fungsi tersebut akan menjalankan lagi command pada label yang dituju. Selanjutnya diperiksa import. Dimana akan dijalankan `exec_file` untuk file pada import. Selanjutnya akan masuk `check_assignment` dimana pada fungsi akan dicari index dari tanda "=". Bila tidak ada maka akan di jalankan fungsi `check_keyword`. Bila ada maka akan dipisah menjadi dua bagian yaitu variabel dan value. Lalu akan dijalankan `check_operation` untuk value. dan dimasukkan ke dalam variabel. Yang dimana variabel tersebut bisa berupa variable biasa, array, atau variabel pada class. Bila variabel tidak ditemukan maka akan dibuat variabel baru dan diappend pada list variabel pada interpreter. Bila ditemukan maka akan diubah nilainya.

2.1.2 Fungsi `check_operation`

Pada `check_operation` akan diperiksa apakah array atau tidak. Bila iya maka akan mereturn array yang dimana tiap elemennya akan dijalankan `check_operation` lagi. Bila tidak maka akan akan dibagi tiap elemennya dengan operator arithmetic maupun logic. Sebelumnya tiap string pada input akan disembunyikan agar tidak terpisah. Setelah semua operasi untuk memecah strin akan didapat list yang terdiri dari Operator dan Operand. dimana operand sendiri akan diperiksa kembali apakah memanggil fungsi atau tidak. variable atau bukan. yang dimana akan cari nilai dari operand tersebut. Bila berhasil maka list tersebut akan dihitung menggunakan diolah menjadi postfix lalu dihitung. Bila gagal akan dikembalikan elemen pertama atau None

2.1.3 Fungsi `check_keyword`

Pada fungsi ini akan diperiksa apakah fungsi tersebut adalah built-in function, fungsi atau class yang sudah define. Yang akan menjalankan fungsi yang tersimpan lalu mengembalikan hasilnya. Bila tidak maka akan mengembalikan None.

2.1.4 Fungsi `exec_if`

Fungsi ini akan mengambil kondisi dan block code dibawahnya lalu dimasukkan list kondisi dan list task yang akan dijalankan. Dimana kondisi akan diperiksa nilainya dengan fungsi `check_operation` bila nilainya 1 maka block code dibawahnya dijalankan dengan fungsi `exec_string` bila tidak maka akan memeriksa kondisi setelahnya. Dan seterusnya dimana untuk else diberikan nilai 1 langsung.

2.1.5 Fungsi `exec_while`

Fungsi ini akan menggunakan while loop dimana kondisi yang diberikan adalah nilai return `check_operation`. Dimana dimana pada while dijalankan fungsi `exec_string` untuk block code yang ada di dalam while.

2.1.6 Fungsi `def_fn`

Fungsi ini akan menyimpan nama fungsi, argumen, dan block code yang akan dijalankan. Dimana bila fungsi dipanggil maka akan dibuat interpreter baru sebagai scope variabel lalu argument akan dimasukkan pada sebagai variabel pada interpreter baru tersebut.

2.1.7 Fungsi `def_class`

Fungsi ini akan menyimpan nama class, argumen, fungsi pada class. Dimana saat class akan diinstantiate akan dibuat interpreter baru untuk scope class tersebut. Dimana seluruh block code pada class akan dijalankan untuk memasukkan fungsi dan variable pada class tersebut. Selanjuta fungsi pada class dapat dipanggil dengan menggunakan `nama_class.nama_fungsi` pada fungsi dalam class untuk mereferensi class tersebut dapat menggunakan `this`. Class adalah mutable object sehingga bila dimasukkan pada fungsi, elemen pada class dapat dirubah.

2.2 Text Editor

Text Editor dapat membuka folder atau file. Lalu ditampilkan pada Aplikasi. Program ini menggunakan library customtkinter dan beberapa dari tkinter untuk tampilannya. Program ini mampu syntax highlighting untuk bahasa pemrograman yang saya buat dimana data didapat dari syntax identifier. Lalu program dapat menjalankan code pada textbox. dengan menggunakan subprocess untuk menjalankan program interpreter. Program ini juga dilengkapi dengan interactive terminal dimana cara kerja dari interactive terminal adalah menjalankan subprocess untuk menjalankan program interpreter. Lalu terdapat input yang dapat tiap kali ditekan enter maka isinya akan dikirim ke proc.stdin Lalu dijalankan oleh interpreter. Lalu output dari interpreter akan ditampilkan pada terminal. Untuk menampilkan output dari terminal saya menggunakan thread baru agar tidak mengganggu main thread. Dimana thread tersebut akan menjalankan fungsi `read_proc` lalu dioutputkan pada textbox yang dibuat seolah olah seperti terminal.

2.3 Syntax Identifier

Syntax Identifier adalah program yang akan mengidentifikasi syntax dari suatu file. Program ini akan membaca file lalu mendeteksi setiap keyword yang sudah ditentukan ia juga dapat mendeteksi variable, fungsi, dan class yang dibuat user. hasilnya akan dibaca text editor untuk syntax highlighting.

3 Contoh Input & Output

3.1 Input

3.1.1 `code_example.pyhk`

```
# Variable Assignment
out("Variable Assignment")
num1 = 10
num2 = num1
num3 = num2

out("num1 : ", num1)
out("num2 : ", num2)
out("num3 : ", num3)

out()

# input
out("Input A: ", no_break)
in(A)
out("Input B: ", no_break)
in(B)

out("A : ", A)
out("B : ", B)

out()

# Arithmethic
out("Arithmethic")
num1 = num1 + 10
num2 = num2 * num1 / 3
num3 = num1 + num2 - num3
num4 = (num1 % 3) / num2 * (num1 + 100)

out("num1 : ", num1)
out("num2 : ", num2)
out("num3 : ", num3)
out("num4 : ", num4)
out()

# String
hello = "Hello World"
gbye = "Good Bye World"
c = hello + " and " + gbye
d = (c + ", ") * 3

out("hello : ", hello)
out("gbye : ", gbye)
out("c : ", c)
out("d : ", d)

out()

#Comparison
out("num1 : ", num1)
out("num2 : ", num2)
out("num3 : ", num3)
out("num4 : ", num4)

out()

out("num1 > num2 : ",num1 > num2)
out("num2 > num3 : ",num2 > num3)
out("num3 > num4 : ",num3 > num4)

out()

# Conditional Statement

out("Conditional Statement")
if num1 > num2 and num2 > num3
    out("This statement is True: num1 > num2 and num2 > num3")
else_if num1 < num2 and num2 > num3
    out("This statement is True: num1 < num2 and num2 > num3")
else_if num1 < num2 and num2 < num3
    out("This statement is True: num1 < num2 and num2 < num3")
else_if num1 > num2 and num2 < num3
    out("This statement is True: num1 > num2 and num2 < num3")
else
    out("None of the statement above is True")
end

out()

# Loop

out("Loop")
i = 0
while i < 10
    out(i, " ", no_break)
    i = i + 1
end
out()
out()

# List
out("List")

ls_A = [-3, -2, -99, 3,1,7,6,4]

ls_B = []
ls_B[10] = NULL

out("ls_A : ", ls_A)
out("ls_B : ", ls_B)
out("sizeof ls_A : ", sizeof(ls_A))
out("sizeof ls_B : ", sizeof(ls_B))

out("\nSorting")
n = sizeof(ls_A)
i = 0
while i < n
    j = 0
    while j < n - i - 1
        if ls_A[j] > ls_A[j+1]
            tmp = ls_A[j]
            ls_A[j] = ls_A[j+1]
            ls_A[j+1] = tmp
        end
        j = j + 1
    end
    i = i + 1
end

out("ls_A : ", ls_A)

out()

# Function
out("Function")

fn sort(list)
    n = sizeof(list)
    i = 0
    while i < n
        swapped = 0
        j = 0
        while j < n - i - 1
            if list[j] > list[j+1]
                tmp = list[j]
                list[j] = list[j+1]
                list[j+1] = tmp
                swapped = 1
            end
            j = j + 1
        end
        if not swapped
            out("sorted in ", i, " iterations")
            return list
        end
        i = i + 1
    end
    out("sorted in ", i, " iterations")
    return list
end

list_example = [23, 56, -99, -3, 0.1, 45.6, 100, 23, 45, 45, 45]
sorted_list = sort(list_example)

out("list_example : ", list_example)
out("sorted_list : ", sorted_list)

# Function Support Recursion

back_to_recursion:
out("Recursion")
fn fib(n)
    if n <= 0
        return 0
    else_if n == 1
        return 1
    else
        return fib(n-1) + fib(n-2)
    end
end

i = 0
while i < 10
    out(fib(i)," ", no_break)
    i = i + 1
end

out()

# Class
out("\nClass")

class tes
    fn init(a,b,c)
        this.a = a
        this.b = b
        this.c = c
    end

    fn print()
        out(this.a)
        out(this.b)
        out(this.c)
        out()
    end

    fn get()
        return this
    end
    fn copy_add()
        return tes(this.a + 1, this.b + 1, this.c + 1)
    end
end

fn createTes(a,b,c)
    return tes(a,b,c)
end

createTes(1,2,3).print()
createTes(1,2,3).copy_add().print()
createTes(1,2,3).copy_add().copy_add().print()
createTes(1,2,3).copy_add().copy_add().copy_add().print()
createTes(3,2,7).get().print()

tes(1,2,3).print()
tes(1,2,3).copy_add().print()
tes(1,2,3).copy_add().copy_add().print()
tes(1,2,3).copy_add().copy_add().copy_add().print()

objA = tes(34,21,90)
objB = tes(34,21,90)
objC = objA
objD = objA.get()

fn copy_tes(obj)
    return tes(obj.a, obj.b, obj.c)
end

copy_cls = copy_tes(objA)

conf = ""
while conf != "y" and conf != "n"
    out("Back to recursion ? y / n : ", no_break)
    in(conf)
    if conf == "y"
        goto back_to_recursion
    end
end

#Import
out("Import")
import Lib\vector_class.pyhk

vec1 = vector3(1,2,3)
vec1.print()
vec1.add(9,3,1)
vec1.print()
vec1.substrac(4,1,8)
vec1.print()
```

3.1.2 Lib/vector_class.pyhk

```
class vector3
    fn init(x,y,z)
        this.x = x
        this.y = y
        this.z = z
    end
    fn add(x,y,z)
        this.x = this.x + x
        this.y = this.y + y
        this.z = this.z + z
    end
    fn substrac(x,y,z)
        this.x = this.x - x
        this.y = this.y - y
        this.z = this.z - z
    end
    fn print()
        out("x : ", x)
        out("y : ", y)
        out("z : ", z)
    end
end
```

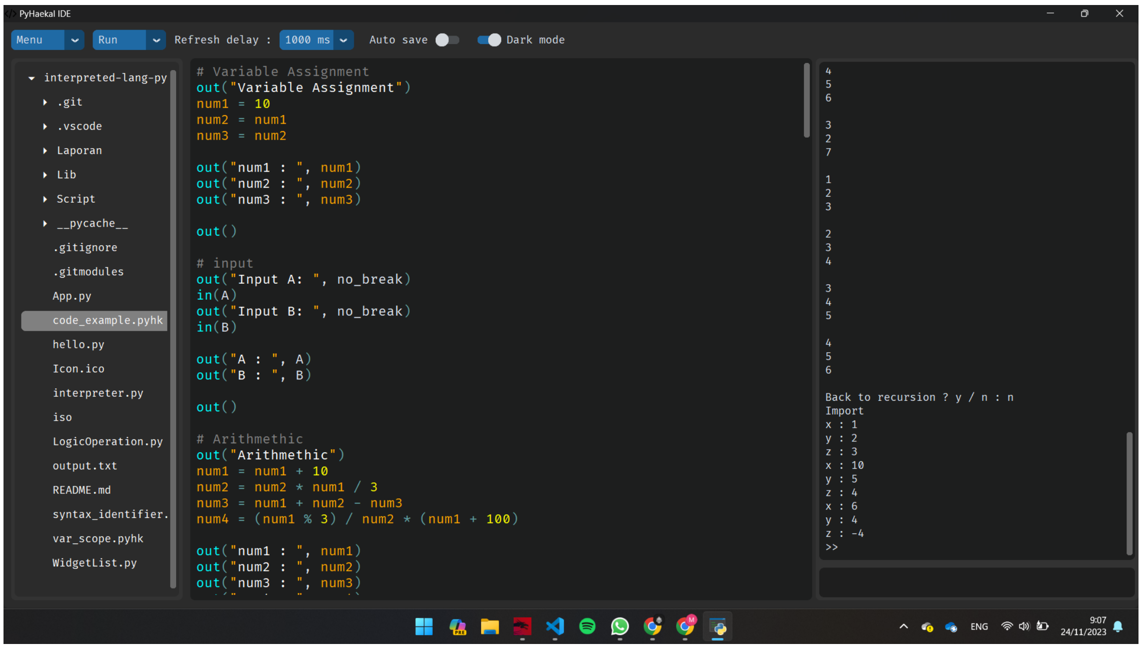
3.2 Output

```
py interpreter.py  .\code_example.pyhk
Variable Assignment
num1 : 10
num2 : 10
num3 : 10

Input A: 3
Input B: 4
A : 3
B : 4

Arithmethic
num1 : 20
num2 : 66.66666666666667
num3 : 76.66666666666667
num4 : 3.5999999999999996

hello : Hello World
gbye : Good Bye World
c : Hello World and Good Bye World
d : Hello World and Good Bye World, Hello World and Good Bye World, Hello World and Good Bye Worl

num1 : 20
num2 : 66.66666666666667
num3 : 76.66666666666667
num4 : 3.5999999999999996

num1 > num2 : 0
num2 > num3 : 0
num3 > num4 : 1

Conditional Statement
This statement is True: num1 < num2 and num2 < num3
```

```
Loop
0 1 2 3 4 5 6 7 8 9

List
ls_A : [-3, -2, -99, 3, 1, 7, 6, 4]
ls_B : []
sizeof ls_A : 8
sizeof ls_B : 0

Sorting
ls_A : [-99, -3, -2, 1, 3, 4, 6, 7]

Function
sorted in 3 iterations
list_example : [23, 56, -99, -3, 0.1, 45.6, 100, 23, 45, 45, 45]
sorted_list : [-99, -3, 0.1, 23, 23, 45, 45, 45, 45.6, 56, 100]
Recursion
0 1 1 2 3 5 8 13 21 34

Class
1
2
3

2
3
4

3
4
5

4
5
6

3
2
7

1
2
3

2
3
4

3
4
5

4
5
6
```

```
Back to recursion ? y / n : y
Recursion
0 1 1 2 3 5 8 13 21 34

Class
1
2
3

2
3
4

3
4
5

4
5
6

3
2
7

1
2
3

2
3
4

3
4
5

4
5
6

Back to recursion ? y / n : n
Import
x : 1
y : 2
z : 3
x : 10
y : 5
z : 4
x : 6
y : 4
z : -4

Press Enter to continue...
```

3.3 Screenshot



```
# Variable Assignment
out("Variable Assignment")
num1 = 10
num2 = num1
num3 = num2

out("num1 : ", num1)
out("num2 : ", num2)
out("num3 : ", num3)

out()

# input
out("Input A: ", no_break)
in(A)
out("Input B: ", no_break)
in(B)

out("A : ", A)
out("B : ", B)

out()

# Arithmethic
out("Arithmethic")
num1 = num1 + 10
num2 = num2 * num1 / 3
num3 = num1 + num2 - num3
num4 = (num1 % 3) / num2 * (num1 + 100)

out("num1 : ", num1)
out("num2 : ", num2)
out("num3 : ", num3)
```

```
4
5
6

3
2
7

1
2
3

2
3
4

3
4
5

4
5
6

Back to recursion ? y / n : n
Import
x : 1
y : 2
z : 3
x : 10
y : 5
z : 4
x : 6
y : 4
z : -4
>>
```

Figure 1: Screenshot Text Editor