Programming Languages Assignment2

2021312738 소프트웨어학과 김서환

우선 Assignment Goal 1에 대한 내용으로 c코드를 받아와서 parsing하고 abstract syntax tree(AST)를 파이썬 코드로 작성하는 것에 대해 설명드리겠습니다.

저는 우선 node class를 설정하여 ast구조를 담을 노드를 만들었습니다.

```
class node:
    def __init__(self, nodetype, step, name="", childnode=None, parent=None):
        self.nodetype = nodetype
        self.level = step
        self.name = name
        self.childnode = childnode or []
        self.parent = parent

    #step를 부모노드의 step+1이 되도록 설정
    @property
    def step(self):
        if self.parent:
            return self.parent.step + 1
        return self.level
    #그 외의 step를 설정할 일이 있을때
    @step.setter
    def step(self, value):
        self.level = value
    #실제 ast 구조를 보여주는 부분
    def show(self):
        str = "{}{}:".format(("  " * self.step), self.nodetype)
        if self.nodetype == "Decl":
            str += " {}, [], [], [], []\n".format(self.name)
        elif self.nodetype == "TypeDecl":
            str += " {}, [], None\n".format(self.name)
        elif self.nodetype == "IdentifierType":
            str += " ['{}']\n".format(self.name)
        elif self.nodetype == "Constant" or "ID" or "BinaryOp":
            str += " {}\n".format(self.name)
        else:
            str += "\n"
        for node in self.childnode:
            str += node.show()
        return str
```

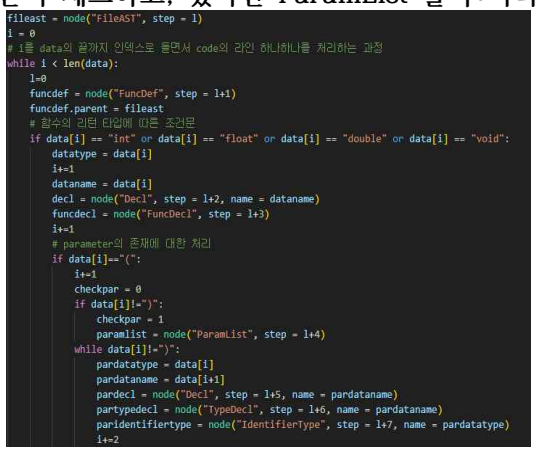
```
def parse(code): # c-code를 읽어와 문자열를 리스트에 담는 과정
    data = []
    textsymbol = ["=","+","-","*","/","&","(",")","{","}",";",",","\"","\'","<",">","%"]
    temp = ""
    skip = False
    # code에 있는 문자열 하나씩 가져와 특수문자 혹은 공백을 분리하여 담는 과정
    # <와 >는 <<, >>로 담아야해서 조건문으로 처리
    for text in code:
        if text in ["<",">"] and skip:
            skip = False
            continue
        if text in textsymbol or text.isspace():
            if text == "<":
                if not temp == "":
                    data.append(temp)
                    temp=""
                data.append("<<")
                skip = True
                continue
            if text == ">":
                if not temp == "":
                    data.append(temp)
                    temp=""
                data.append(">>")
                skip = True
                continue
            if not temp == "":
                data.append(temp)
                temp = ""
            if text in textsymbol:
                data.append(text)
        else:
            temp = temp + text
    if not temp == "":
        data.append(temp)
```

이 노드는 ast구조에 있어서 nodetype(노드 이름), level(step, 계층), name(노드의 들어갈 변수), childnode(자식노드), parent(부모노드)로 구성했습니다.

또한, parse 함수 초반 부분에 code를 읽어와 특수문자를 구분하고, 공백을 저장하지 않으며 일반적인 단어는 연속된 문자면 함께 저장하여 int, float와 같은 단어들로 저장하였습니다.

이 과정에서 > 와 <는 <<, >>로 저장해야 연산자로 인식되어 조건문으로 처리해주었습니다.

그 후 while문을 통해 ast구조를 만들어나가기 시작했고 위의 과정을 통해 얻은 문자열 리스트를 바탕으로 하나씩 data를 처리해줬습니다. 함수의 파라미터에 대한 처리는 파라미터가 있는지 체크하고, 있다면 ParamList 출력+파라미터 TypeDecl, IdentifierType 출력하였습니다.



```
fileast = node("FileAST", step = 1)
i = 0
# i를 data의 끝까지 인덱스로 돌면서 code의 라인 하나하나를 처리하는 과정
while i < len(data):
    l=0
    funcdef = node("FuncDef", step = l+1)
    funcdef.parent = fileast
    # 함수의 리턴 타입에 따른 조건문
    if data[i] == "int" or data[i] == "float" or data[i] == "double" or data[i] == "void":
        datatype = data[i]
        i+=1
        dataname = data[i]
        decl = node("Decl", step = l+2, name = dataname)
        funcdecl = node("FuncDecl", step = l+3)
        i+=1
        # parameter의 존재에 대한 처리
        if data[i]=="(":
            i+=1
            checkpar = 0
            if data[i]!=")":
                checkpar = 1
                paramlist = node("ParamList", step = l+4)
            while data[i]!=")":
                pardatatype = data[i]
                pardataname = data[i+1]
                pardecl = node("Decl", step = l+5, name = pardataname)
                partypedecl = node("TypeDecl", step = l+6, name = pardataname)
                paridentifiertype = node("IdentifierType", step = l+7, name = pardatatype)
                i+=2
```

```
                i+=2
                if data[i]==",":
                    i+=1
                paridentifiertype.parent = partypedecl
                partypedecl.childnode.append(paridentifiertype)
                partypedecl.parent = pardecl
                pardecl.childnode.append(partypedecl)
                pardecl.parent = paramlist
                paramlist.childnode.append(pardecl)
            if checkpar == 1:
                paramlist.parent = funcdecl
                funcdecl.childnode.append(paramlist)
            typedecl = node("TypeDecl", step = l+4, name = dataname)
            identifiertype = node("IdentifierType", step = l+5, name = datatype)
            identifiertype.parent = typedecl
            typedecl.childnode.append(identifiertype)
            typedecl.parent = funcdecl
            funcdecl.childnode.append(typedecl)
        funcdecl.parent = decl
        decl.childnode.append(funcdecl)
        decl.parent = funcdef
        funcdef.childnode.append(decl)
        i+=1
```

이 과정에서 step을 루트노드인 FileAst의 step을 이용해 정적으로 처리해주려고 처음에 시도했지만, 후에 연산하는 과정에 있어서 너무 꼬임이 심해 node 클래스의 property-setter로 동적으로 부모노드의 값 +1이 될 수 있도록 설정해줬습니다. 그 이후 step을 동적 처리 이후에 제가 노드간의 관계를 파악할 때 활용했어서 지우지 않았습니다.

그 이후 함수의 내용, Body에 해당하는 Compound 부분을 변수선언, 재할당, 함수호출 3가지로 구분하여 코드를 작성했습니다. data가 {부터 }까지 돌면서 구분해주었고, int a = ? -> int a = 까지는 아래의 사진과 같이 처리해줬고, ?에 해당하는 것은 operation에서 처리해줬습니다.

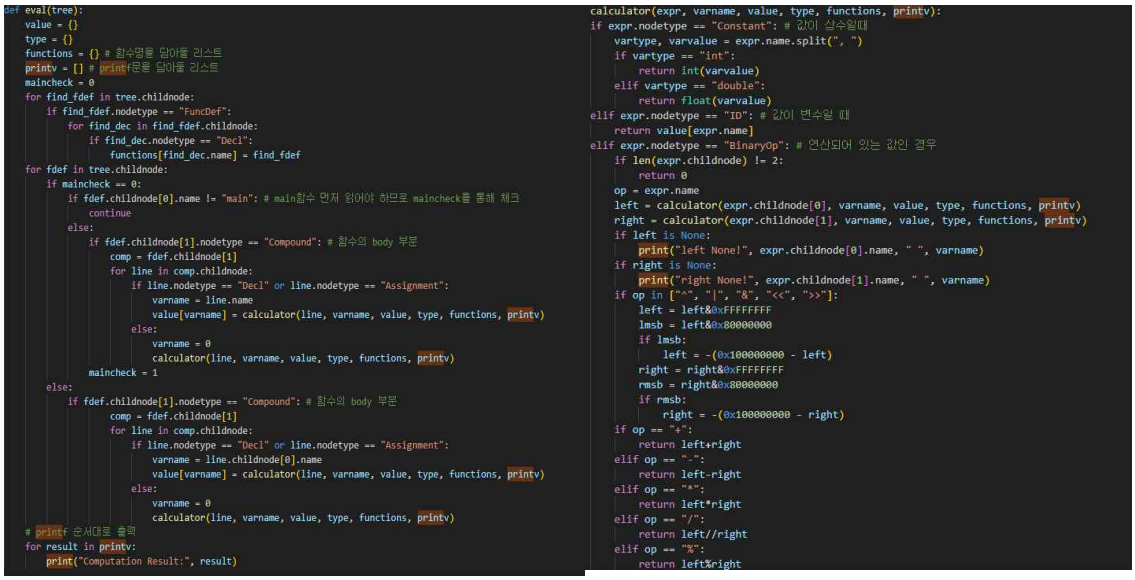
```
while data[i]!="}":
    l=fixl
    if data[i] == "int" or data[i] == "float" or data[i] == "double": # 처음 선언되는 변수 저장
        bodydatatype = data[i]
        bodydataname = data[i+1]
        bodydecl = node("Decl", step = l+1, name = bodydataname)
        bodytypedecl = node("TypeDecl", step = l+2, name = bodydataname)
        bodyidentifiertype = node("IdentifierType", step = l+3, name = bodydatatype)
        bodyidentifiertype.parent = bodytypedecl
        bodytypedecl.childnode.append(bodyidentifiertype)
        bodytypedecl.parent = bodydecl
        bodydecl.childnode.append(bodytypedecl)
        i+=2
        if data[i]==";":
            i+=1
            bodydecl.parent = compound
            compound.childnode.append(bodydecl)
            continue
        i+=1
        opexp = operation(data, l+2, i, [";"],0)
        i=opexp["index"]+1
        opexpnode = opexp["node"]
        opexpnode.parent = bodydecl
        bodydecl.childnode.append(opexpnode)
        bodydecl.parent = compound
        compound.childnode.append(bodydecl)
```

```
def operation(data, arglevel, start, endsymbol,recursion):
    priority = {"*": 1, "/": 1, "%": 1, "+": 2, "-": 2, "<<": 3, ">>": 3, "&": 4, "^": 5, "|": 6} # 값이 낮을
    opsymbol = ["+","-","*","/","%","^","|","&","<<",">>"]
    i = start
    nodelist = []
    oplist = [] # 없애서부터 하나씩 읽으며 연산자, 인덱스, 그에 따른 우선순위를 전부 저장하는 딕셔너리 리스트
    ops = [] # 최종 우변의 식에 존재하는 연산자, 인덱스, 그에 따른 우선순위를 전부 저장한 딕셔너리 리스트 단, 괄호로
    rec = recursion
    opcount = 0 # 연산자의 갯수
    bracket = 0 # 괄호의 갯수
    # 여는 괄호와 닫는 괄호의 갯수가 맞을때만 ops저장
    for k in range(start,len(data)):
        if rec>=1:
            if data[k]==")":
                break
        if data[k]=="(":
            bracket+=1
            continue
        if data[k]==")":
            bracket-=1
            continue
        if bracket==0:
            if data[k] in opsymbol:
                ops.append({"op": data[k], "index": k, "priority": priority.get(data[k])})
                opcount+=1
        if data[k]==";":
            break

    level=arglevel
```

operation 함수는 ? 즉, 등호 우변의 식을 처리해주는 함수입니다. 등호 우변의 식은 함수호출, 연산, 괄호 등의 여러 가지 복잡한 식이 얽힌 식입니다. 따라서 앞으로 우변의 식을 그냥 식이라고 하겠습니다. 식을 연산 우선순위에 따라서 ast구조를 다르게 만들어야 하기에 괄호 안의 있는 식을 제외한 연산자의 개수를 파악하여 opcount로 계층을 조작했습니다. 또한, ops라는 식의 모든 연산자를 딕셔너리 리스트로 담아 oplist(내가 사용할 연산자 딕셔너리 리스트)와 비교하여 연산해줬습니다. 괄호 -> 재귀연산, 연산자 -> 연산자가 하나라도 존재하고 현재 연산자가 직전에 추가한 연산자보다 우선순위가 낮거나 같을때만 반복하여 직전에 추가한 연산자 pop을 하여 계산하는과정, 아니라면 그냥 추가하고 그 이후에 while문을 다 돌고 남은 연산자들을 처리해주는 연산과정까지 설정해줬습니다. 함수/변수/상수 -> ast구조가 잘 나오도록만 코드를 작성해줬습니다. 이렇게 해서 변수선언과 재할당, 리턴(리턴값이 연산인 경우)operation함수를 통해 식을 계산해줬습니다. 또한, 함수호출에 대한 처리도 출력이 잘되도록 코드를 작성했습니다.

이제 Assingment Goal 2인 실제 printf()가 있는 문장을 출력하는 과제에 대해 설명드리겠습니다.
eval()함수를 통해 ast tree를 받아주었고, main 함수를 먼저 읽어야하므로 그부분도 처리해줬습니다. Compound 부분을 calculator 함수로 처리해줬습니다.



```
def eval(tree):
    value = {}
    type = {}
    functions = {} # 함수명을 담아둘 리스트
    printv = [] # printf문을 담아둘 리스트
    maincheck = 0
    for find_fdef in tree.childnode:
        if find_fdef.nodetype == "FuncDef":
            for find_dec in find_fdef.childnode:
                if find_dec.nodetype == "Decl":
                    functions[find_dec.name] = find_fdef
    for fdef in tree.childnode:
        if maincheck == 0:
            if fdef.childnode[0].name != "main": # main함수 먼저 읽어야 하므로 maincheck를 통해 체크
                continue
            else:
                if fdef.childnode[1].nodetype == "Compound": # 함수의 body 부분
                    comp = fdef.childnode[1]
                    for line in comp.childnode:
                        if line.nodetype == "Decl" or line.nodetype == "Assignment":
                            varname = line.name
                            value[varname] = calculator(line, varname, value, type, functions, printv)
                        else:
                            varname = 0
                            calculator(line, varname, value, type, functions, printv)
                    maincheck = 1
        else:
            if fdef.childnode[1].nodetype == "Compound": # 함수의 body 부분
                    comp = fdef.childnode[1]
                    for line in comp.childnode:
                        if line.nodetype == "Decl" or line.nodetype == "Assignment":
                            varname = line.childnode[0].name
                            value[varname] = calculator(line, varname, value, type, functions, printv)
                        else:
                            varname = 0
                            calculator(line, varname, value, type, functions, printv)
    # printf 순서대로 출력
    for result in printv:
        print("Computation Result:", result)
```

```
calculator(expr, varname, value, type, functions, printv):
if expr.nodetype == "Constant": # 값이 상수일때
    vartype, varvalue = expr.name.split(", ")
    if vartype == "int":
        return int(varvalue)
    elif vartype == "double":
        return float(varvalue)
elif expr.nodetype == "ID": # 값이 변수일 때
    return value[expr.name]
elif expr.nodetype == "BinaryOp": # 연산되어 있는 값인 경우
    if len(expr.childnode) != 2:
        return 0
    op = expr.name
    left = calculator(expr.childnode[0], varname, value, type, functions, printv)
    right = calculator(expr.childnode[1], varname, value, type, functions, printv)
    if left is None:
        print("left None!", expr.childnode[0].name, " ", varname)
    if right is None:
        print("right None!", expr.childnode[1].name, " ", varname)
    if op in ["^", "|", "&", "<<", ">>"]:
        left = left&0xFFFFFFFF
        lmsb = left&0x80000000
        if lmsb:
            left = -(0x100000000 - left)
        right = right&0xFFFFFFFF
        rmsb = right&0x80000000
        if rmsb:
            right = -(0x100000000 - right)
    if op == "+":
        return left+right
    elif op == "-":
        return left-right
    elif op == "*":
        return left*right
    elif op == "/":
        return left//right
    elif op == "%":
        return left%right
```

calculator함수는 상수일 때, 변수일땐 그냥 value에 있는 값을 저장해서 리턴하였고, 연산의 경우 산술연산과 비트연산을 진행하여 리턴하였습니다. bitwise연산은 32비트의 정수일경우만 고려해줬고 재할당, 변수 선언의 경우에도 type과 value를 저장하여 리턴했습니다. 함수호출의 경우 printf, return, user-defined 함수를 처리해줬습니다. 함수 호출이 되면 함수명을 받아 그 함수의 body에 있는 내용을 읽어 적용시킬 수 있도록 했습니다. 그리고 printf를 발견할때마다 Computation Result:를 printv라는 리스트에 저장하여 마지막에 모두 순서대로 출력했습니다.