import sys

2 import os

3 import re

4

import glob

5

import json

6

7

8

9

from antlr4 import InputStream, CommonTokenStream

from TandemCobolLexer import TandemCobolLexer

from TandemCobolParser import TandemCobolParser

from TandemCobolVisitor import TandemCobo Visitor

10

11

12

13

from collections import OrderedDict

# Mapping types to labels

VAR\_TYPE\_KEYS = {

14

15

"paragraphs": "paragraph",

"divisions": "division",

16

17

"sections": "section",

"filename": "filename",

18

19

20

21

22

23

24

25

"filedefinition": "filecontrolclause",

"dataname": "dataname",

"perform": "performstatement",

"variable": "identifier",

"screenname": "screenname",

"screensection": "screensection",

"sqlstatements": "sqlstatement",

"evaluatestatements": "evaluatestatement"

26

}

#tokens/strings we want to ignore in simple terminal returns

remove\_variables ('END-OF-SECTION', 'EXIT', '.', 'END-PERFORM'}

def find\_after(first\_word, line):

first\_pos line.find(first\_word)

return line[first\_pos + len(first\_word):]

def betwen\_call\_using(first\_word, last\_word, line):

""Return substring between first\_word and last\_word in line".""

" line\_up line.upper()

try:

first\_pos line\_up.index(first\_word.upper())

last\_pos line\_up.index(last\_word.upper(), first\_pos + len(first\_word))

except ValueError:

return ""

return line[first\_pos + len(first\_word): last\_pos].strip()

def copy\_check(source\_folder, input\_file):

copybook=[]

with open(input\_file, 'r') as file:

I

lines file.readlines()

with open(input\_file, 'w') as opfile:

for line in lines:

if not line.strip().startswith("\*"):

if 'COPY' in line.upper():

space\_count len(line)-len(line.lstrip())

copy\_file find\_after('COPY', line).strip().replace('.','')

copybook.append(copy\_file)

copy\_file\_path = source\_folder + "\\" + copy\_file

with open(copy\_file\_path, 'r') as cfile:

1 cfile.readlines()

line\_space = len(1[0])-len(1[0].lstrip()) - space\_count

for 11 in 1:

opfile.write(11[line\_space:])

else:

opfile.write(line)

return opfile, copybook

class AstNode:

def \_init\_\_(self, name, \*\*kwargs):

self.type name

self.attributes = kwargs.copy()

self.ordered\_content = []

self.grouped = {k: [] for k in VAR\_TYPE\_KEYS}

def add\_ordered(self, label, node):

I

self.ordered\_content.append((label, node))

if label in self.grouped:

self.grouped [label].append(node)

def is\_meaningful(self):

78

for v in self.attributes.values():

def is\_meaningful(self):

for v in self.attributes.values():

if v:

return True

if self.ordered\_content:

return Trúe

return False

def \_convert\_attr(self, value):

if isinstance(value, AstNode):

return value.to\_dict()

if isinstance(value, list):

converted = []

for item in value:

if isinstance(item, AstNode):

converted.append(item.to\_dict())

else:

converted.append(item)

return converted

return value

I

def to\_dict(self):

if not self.is\_meaningful():

return None

node\_dict={'type': self.type)

for k, v in self.attributes.items():

node\_dict[k] = self.\_convert\_attr(v)

for label, node in self.ordered\_content:

if node is None:

continue

node\_serial node.to\_dict()

if node serial is None:

continue

if label not in node\_dict:

node\_dict[label] = []

node\_dict[label].append(node\_serial)

for k, Ist in self.grouped.items():

if 1st and k not in node\_dict:

node\_dict[k] [n.to\_dict() for n in Ist if n.to\_dict() is not None]

return node dict

class AstVisitor(TandemCobolVisitor):

definit(self, token\_stream-None):

super().init()

self.token stream token\_stream

I

14

15

16

17

18

19

20

121

122

123

24

125

126

self.dependencies = []

self.declared\_vars OrderedDict()

self.variable\_usage = []

self.control\_flow = []

self.current\_paragraph None

327

128

def get\_text\_with\_spaces(self, ctx):

129

if ctx is None or getattr(ctx, "start", None) is None on getattr(ctx, "stop", None) is None:

return ""

try:

start\_idx= ctx.start.tokenIndex

stop\_idx= ctx.stop.tokenIndex

raw self.token\_stream.getText(start=start\_idx, stop-stop\_idx)

except Exception:

raw ctx.getText()

return re.sub(r'\s+','', raw).strip()

def visitChildren(self, ctx):

name type(ctx).\_name\_.replace('Context', '')

node AstNode (name)

if not getattr(ctx, 'children', None):

return ctx.getText()

for child in ctx.children:

child ast self.visit(child)

if isinstance(child\_ast, str):

norm child\_ast.strip()

I

if not norm or norm.upper() in remove\_variables:

continue

if child ast is None or (isinstance(child ast, str) and not child\_ast.strip()):

continue

if isinstance(child\_ast, str):

if hasattr(child, 'getRuleIndex'):

key TandemCobolParser.ruleNames [child.getRuleIndex()]

else:

key 'value'

prev node.attributes.get(key)

if prev is None:

node.attributes[key] = child\_ast

elif isinstance(prev, list):

prev.append(child\_ast)

else:

node.attributes[key] = f"(prev) (child\_ast}"

continue

Iname (child\_ast.type or "").lower()

try:

if hasattr(child, "start") and hasattr(child, "stop"):

if 'datadescriptionentry In Iname:

data\_node self.extract\_datadescription\_entry(child)

if data\_node:

node.add\_ordered("datadescription", data\_node) continue

if Iname='paragraph":

para name child.getText()

self.current\_paragraph para name

if 'movestatement' in Iname:

move\_node = self.extract\_move\_statement(child)

if move\_node:

node.add\_ordered("movestatements", move\_node)

continue

if 'gotostatement' in Iname:

goto\_node = self.extract\_goto\_statement(child)

if goto\_node:

node.add\_ordered("gotostatements", goto\_node)

continue

if 'performstatement' in Iname:

perform\_node = self.extract\_perform\_statement(child)

if perform\_node:

node.add\_ordered("performstatements", perform\_node)

continue

if 'ifstatement in Iname:T

if\_node self.extract\_if\_statement(child)

if if\_node:

node.add\_ordered("ifstatements", if\_node)

continue

if 'evaluatestatement in Iname:

eval\_node = self.extract\_evaluate\_statement(child)

it eval\_node:

node.add ordered("evaluatestatements", eval\_node)

continue

if 'callstatement in Iname:

call\_node self.extract\_call\_statement(child)

if call\_node:

node.add\_ordered("calltatements", call\_node)

continue

if 'sqlstatement' in Iname or 'exec' in 1name:

exec\_node self.extract\_exec\_statement(child)

if exec\_node:

node.add\_ordered("sqlstatements", exec\_node)

continue

matched False

for key, value in VAR\_TYPE\_KEYS.items():

if value in Iname and 1name = 'sectionheader":

node.add\_ordered(key, Child\_ast)

matched True

break

if not matched:

node.add\_ordered("statements", child\_ast)

except Exception:

node.add\_ordered("statements", child\_ast)

return node

def visitTerminal(self, node):

return node.getText()

def extract datadescription\_entry(self, ctx):

txt self.get text\_with\_spaces(ctx)

238

239

240

241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

if not txt:

return None

children text txt.upper().replace(".", "")

tokens children\_text.split()

if len(tokens) >= 2:

level tokens [6]

name tokens [1]

pic clause None

if "PIC" in tokens:

pic index tokens.index("PIC")

if pic index + 1 < len(tokens)

pic clause tokens [pic\_index+1]

self.declared\_vars[name] = {

"level":level,

"pic":pic\_clause

return AstNode("variable", var\_name-nane, level-level, picspic\_clause)

return None

def extract perform statement(self, ctx):

raw\_text= self.get\_text\_with\_spaces(ctx).upper().replace(".

if raw text.startswith('PERFORM'): target raw text.replace('PERFORM','").strip() if self.current \_paragraph:

self.control\_flow.append(("from":self.current\_paragraph, "to": target))

return AstNode("perform", paragraph-raw\_text)

def extract\_move\_statement(self, ctx):

raw\_text= self.get\_text\_with\_spaces(ctx).upper().replace(".", "")

if not raw text or 'MOVE' not in raw text:

return None

170

271

272

273

274

after nove raw\_text.split('NOVE', 1)[1].strip() parts re.split(r'\bTO\b', after move, maxsplit=1)

if len(parts) = 2:

275

276

277

278

279

280

281

282

283

284

return AstNode('move", source=left, target right)

left parts[0].strip() right parts[1].strip() self.variable\_usage.append(( "action": "HOVE", I "from": left, "to": right })

return AstNode('nove', source=after nove, target-lione)

def extract\_if\_statement(self, ctx):

raw text self.get\_text\_with spaces(ctx)

if not raw\_text:

return None

raw\_up = raw\_text.upper().replace(".", "")

if not raw\_up.strip().startswith("IF"):

return None

remain = raw\_text.strip()

up\_remain remain.upper()

else\_part = None

if" ELSE in up\_remain:

parts = re.split(r'\bELSE\b', remain, maxsplit=1)

if\_then\_part = parts[0]

else\_part parts[1].strip()

else:

if\_then\_part = remain

action\_keywords = ['DISPLAY', 'MOVE', 'GOTO', 'COMPUTE', 'IF', 'INITIALIZE', 'PERFORM', 'CALL', 'EXEC', 'SET']

up\_if\_then if\_then\_part.upper()

I

condition =

then\_block =

if up\_if\_then.strip().startswith("IF"):

up\_after\_if = up\_if\_then.strip()[2:].strip()

first\_match\_idx = None

first\_kw None

for kw in action\_keywords:

mre.search(r'\b'+ re.escape(kw) + r'\b', up\_after\_if)

if m:

idx= m.start()

if first\_match\_idx is None or idx < first\_match\_idx:

first\_match\_idx = idx

first\_kw = kw

if first\_match\_idx is not None:

condition = up\_after\_if[:first\_match\_idx].strip()

then\_block if\_then\_part.strip() [len(if\_then\_part) len(up\_after\_if) + first\_match\_idx:].strip()

else:

condition = up\_after\_if.strip()

then\_block = ""

1

else:

first\_match\_idx = None

for kw in action\_keywords:

mre.search(r'\b'+ re.escape(kw) + r'\b', up\_if\_then)

if m:

idx= m.start()

I

if first\_match\_idx is None or idx < first\_match\_idx:

first\_match\_idx = idx

if first\_match\_idx is not None:

condition = up\_if\_then[:first\_match\_idx].strip()

then\_block = if\_then\_part.strip() [first\_match\_idx:].strip()

else:

condition = if\_then\_part.strip()

then\_block=""

self.variable\_usage.append({"action": "CHECK", "variable": condition))

return AstNode('if', condition=condition, then then\_block, else\_block=(else\_part.strip() if else\_part else None)

def extract\_goto\_statement(self, ctx):

raw\_text= self.get\_text\_with\_spaces(ctx).strip()

if not raw\_text:

return None

upraw\_text.upper()

if 'GOTO' in up:

target = re.sub(r'(Pi)\bGOTO\b','", up).strip()

if self.current\_paragraph:

self.control\_flow.append({"from": self.current\_paragraph, "to": target))

return AstNode("goto", target=target)

return AstNode("goto", target=raw\_text)

def extract\_evaluate\_statement(self, ctx): I

raw\_text= self.get\_text\_with\_spaces(ctx).strip()

if not raw\_text:

return None

return AstNode("evaluate", text=raw\_text)

def extract\_call\_statement(self, ctx):

raw\_text= self.get\_text\_with\_spaces(ctx)

if not raw\_text:

return None

up raw\_text.upper()

if 'CALL' in up:

if 'USING' in up:

value betwen\_call\_using("CALL", "USING", raw\_text)

else:

parts re.split(r'(?i)\bCALL\b', raw\_text, maxsplit-1)

value parts[1].strip() if len(parts) > 1 else

value value.strip().strip("\"")

if value and value not in self.dependencies:

self.dependencies.append(value)

self.variable\_usage.append({"action": "CALL ARG", "variable": value))

return AstNode('call', value value)

return None

def extract\_exec\_statement(self, ctx):

raw\_text= self.get\_text\_with\_spaces(ctx)

if not raw\_text:

I

return None

return AstNode('exec', text=raw\_text)

e

1

2

83

34

main(source\_code, source\_path, copybook):

file\_base os.path.splitext(os.path.basename(source\_path))[8]

prog match re.search('PROGRAM-10\.\s\*([\-]+)\.', source\_code, re. IGNORECASE)

prog\_name prog\_match.group(1) if prog match else file\_base

35

6

input stream InputStream(source\_code)

lexer TandemCobolLexer(input\_stream)

tokens CommonTokenStream (lexer)

9

e

1

2

3

4

6

parser TandemCobolParser (tokens)

tree = parser.compilationUnit()

visitor AstVisitor(token\_stream-tokens)

ast visitor.visit(tree)

ast\_dict OrderedDict()

ast\_dict['dialect'] = "ibm"

rawast.to\_dict() if isinstance(ast, AstNode) else None

if raw:

for k, v in raw.items():

1

2

ast\_dict[k] = v

return {

"ast": ast\_dict,

"dependencies": {

},

"CALL": visitor.dependencies,

"COPY": list(copybook)

I

"variables": {

},

"declared\_variables": visitor.declared\_vars,

"variable\_usage": list(visitor.variable\_usage)

"control\_flow": visitor.control\_flow,

"dialect": "ibm"

}

name\_\_main\_\_':

417

source\_folder r"C:\Users\2782313\Desktop\pythonprit\Cobol Codebase Trimmed\COBOL program"

418

output\_path = r"C:\Users\2782313\Downloads\loan\_data\_analyzer\Output\Cobol\_ast\_to\_visitor\_final.json"

419

if not os.path.isdir(source\_folder):

420

print(f"Error: folder not found: (source\_folder)")

421

sys.exit(1)

422

423

cobol\_files = sorted(glob.glob(os.path.join(source\_folder, \*\*.cbl")) +

424

glob.glob(os.path.join(source\_folder, "\*.cob")))

427

425

if not cobol\_files:

426

print("No COBOL files found in folder")

428

sys.exit(0)

429

430

combined\_output = OrderedDict()

431

copybook=[]

432

for file path in cobol\_files:

433

file, copybook copy\_check(source\_folder, file\_path)

434

with open(file\_path, 'r', encoding='utf-8') as f:

435

source\_code = f.read()

I

436

print(f"Parsing {file\_path) ...")

437

file\_result = main(source\_code, file\_path, copybook)

438

file\_key= os.path.splitext(os.path.basename(file\_path))[0]

439

combined\_output[file\_key] = file\_result

440

os.makedirs(os.path.dirname(output\_path), exist\_ok=True)

441

with open(output\_path, 'w', encoding='utf-8') as f:

442

json.dump(combined\_output, f, indent=2)

print(f"Combined AST saved to (output\_path}")