

**دانشکده مهندسی کامپیوتر**

**گزارش فاز اول پروژه‌ی درس کامپایلر**

OpenUnderstand

**گروه 7:**

**دیار حامدی**

**سید عماد موسوی**

**افشین زنگنه**

**مرتضی شهرابی فراهانی**

**اردیبهشت 1401**



**فهرست مطالب**

[مقدمه: 3](#_Toc102250417)

[نتایج حاصل شده: 4](#_Toc102250418)

[خلاصه‌ای از کارهای انجام شده: 4](#_Toc102250419)

[ساختار ادامه‌ی گزارش: 4](#_Toc102250420)

[نحوه‌ی تقسیم کار: 4](#_Toc102250421)

[روش پیشنهادی: 5](#_Toc102250422)

[فایل main.py: 5](#_Toc102250423)

[متد getListOfFiles: 8](#_Toc102250424)

[متد getFileEntity: 8](#_Toc102250425)

[متد Parse: 8](#_Toc102250426)

[متد addDefineRefs: 9](#_Toc102250427)

[define\_definein.py: 9](#_Toc102250428)

[کلاس DefineListener: 9](#_Toc102250429)

[انواع listener ها: 9](#_Toc102250430)

[متد addDefineInfo: 13](#_Toc102250431)

[class\_properties.py: 14](#_Toc102250432)

[ارزیابی: 14](#_Toc102250433)

[مشکلات و چالش‌ها: 21](#_Toc102250434)

[نتیجه‌گیری و کارهای آتی: 22](#_Toc102250435)

# مقدمه:

Understand ابزاری قدرتمند برای تجزیه و تحلیل کدها است. در واقع این نرم‌افزار، تمامی موجودیت‌ها که شامل فایل‌ها، پکیج‌ها، توابع، متغیرها و ... است را در پروژه بررسی می‌کند و همچنین نوع رابطه‌ای که بین هر کدام از این موجودیت‌ها وجود دارد، خط و نام فایلی که این رابطه در آن رخ داده‌است و سایر موارد این‌چنینی را بررسی می‌کند. موارد گفته‌شده توسط برنامه‌ی Understand انجام می‌شود اما هدف از این پروژه پیاده‌سازی متن‌باز از Understand Python API برای تجزیه و تحلیل کدهای منبع است. این کار برای کدهای جاوا با استفاده از ابزارهای Python و Antlr-4 انجام می‌شود. با اجرای این پروژه، کارکردهای برنامه‌ی Understand به صورت رایگان و متن باز در اختیار برنامه‌نویسان قرار می‌گیرد.

## نتایج حاصل شده:

در پایان این پروژه API مربوط به define/definIn پیاده‌سازی شده‌است و در نهایت اطلاعات مربوط به موجودیت‌ها و مراجع در پایگاه داده ذخیره می‌شود.

## خلاصه‌ای از کارهای انجام شده:

در فاز اول پروژه، وظیفه‌ی گروه 7 پیاده‌سازی API مربوط به بخش define/defineIn پروژه‌ی Open Understand بوده‌است. در ابتدا مکان‌ پروژه‌ای که قصد بررسی آن را داریم به برنامه داده می‌شود. سپس برای هرکدام از فایل‌ها و بخش‌های مختلف آن پروژه، بخش‌های define/defineIn که کد آن‌ها در این بخش زده شده‌است، پیدا می‌شود و در دیتابیس در نظر گرفته‌شده برای پروژه ریخته می‌شود.

## ساختار ادامه‌ی گزارش:

در ادامه‌ی گزارش، کدها و بخش‌های مختلفی از پروژه که در این فاز توسط گروه 7 تغییر داده شده‌اند، با جزئیات و توضیح بیشتر به ترتیب بررسی می‌شوند و توضیحات هر بخش به صورت جداگانه داده می‌شود.

## نحوه‌ی تقسیم کار:

بیشتر کار پروژه به صورت تیمی و مشورتی انجام شد. در ابتدا اعضای گروه در چند جلسه‌ی حضوری در سایت دانشکده با هم به بررسی پروژه پرداختند و کدهای هر بخش ما مشارکت همه‌ی اعضای گروه زده شده‌است. اگر بخواهیم به صورت ریزتر، برای هر فرد مشخص کنیم که روی چه بخشی تمرکز بیشتری داشته‌است، نحوه‌ی تقسیم کار بدین صورت می‌شود.

دیار حامدی و سید عماد موسوی: بیشتر بررسی کد define / defineIn

افشین زنگنه: نوشتن گزارش پروژه و بخشی از listener

مرتضی شهرابی فراهانی: نوشتن گزارش و بخش تست و مقایسه‌ی نتایج با نرم افزار Understand

\* البته تمامی اعضای گروه در تمام بخش‌های پروژه همکاری داشتند و تقسیم‌بندی بالا تنها بیانگر بخشی است که هریک از افراد در آن بخش تاثیر بیشتری داشته‌اند.

# روش پیشنهادی:

در ابتدا برنامه‌ی Understand توسط هریک از اعضای گروه نصب شد و سعی شد تا با بررسی آن با بخش‌های مختلف آن آشنا شویم. پس از به دست آمدن آشنایی نسبی با برنامه و نحوه‌ی کار آن، چند پروژه‌ی نمونه به برنامه داده‌شد و خروجی آن‌ها برای define و defineIn بررسی شد و حالت‌های مختلفی که define و defineIn رخ می‌داد یادداشت شد تا در بخش کدزنی به آن‌ها پرداخته شود. در ادامه به سراغ پروژه‌ی OpenUnderstand رفتیم.

برای توضیح روش پیشنهادی، ابتدا فایل main.py که فایل ابتدایی پروژه است بررسی می‌شود و عملکرد آن توضیح داده می‌شود. این فایل شامل توابع مختلفی است که در این بخش توابع مرتبط با پروژه‌ی گروه ما هم بررسی می‌شوند. سپس در ادامه سایر فایل‌هایی که توسط گروه ما به پروژه اضافه شده‌اند نیز بررسی می‌شوند و نقش هریک از آن‌ها توضیح داده می‌شود.

## فایل main.py:

if \_\_name\_\_ == '\_\_main\_\_':  
 p = Project()  
 create\_db("../benchmark\_database.oudb",  
 project\_dir="..\benchmark")  
 main()  
 db = db\_open("../benchmark\_database.oudb")  
 path = r"D:\works\university\term6\compiler\Project\OpenUnderstand\benchmark\xerces2j"  
 files = p.getListOfFiles(path)  
 for file\_address in files:  
 try:  
 file\_ent = p.getFileEntity(file\_address)  
 tree = p.Parse(file\_address)  
 except Exception as e:  
 print("An Error occurred in file:" + file\_address + "\n" + str(e))  
 continue  
 try:  
 *# define* listener = DefineListener()  
 p.Walk(listener, tree)  
 p.addDefineRefs(listener.defines, file\_ent)  
 except Exception as e:  
 print("An Error occurred for reference define in file:" + file\_address + "\n" + str(e))

کد (1) فایل main.py بخش main

این فایل، فایل اصلی و ابتدایی پروژه است. در این فایل ابتدا در بخش main آن، یک شی از کلاس اصلی این فایل که کلاس Project است ساخته می‌شود. سپس پایگاه داده مرتبط با این اجرا ساخته‌می‌شود و در ادامه توسط تابع getListOfFiles که به عنوان ورودی آدرس پروژه را دریافت می‌کند لیست آدرس تمام فایل‌های جاوای پروژه برگردانده می‌شود. سپس به ازای هر کدام از آدرس‌های فایل‌های جاوا، یک بار تابع getFileEntity را صدا می‌زنیم. این تابع به عنوان ورودی آدرس یک فایل جاوا را دریافت می‌کند و در ادامه Entity مربوط به آن فایل را تشخیص می‌دهد و آن را بر می‌گرداند.

در ادامه برای همین فایل تابع Parse را صدا می‌زنیم و با استفاده از این فایل درخت تجزیه‌ی این فایل ساخته می‌شود و برگردانده می‌شود. (توضیحات بیشتر مربوط به هریک از این توابع گفته‌شده در خطوط بالا در صفحات جلوتر آمده‌است. جهت دسترسی سریع‌تر می‌توانید به فهرست مطالب مراجعه کنید.)

پس از ساختن درخت مربوط به این فایل، باید listener مربوط به این فایل صدا زده‌شود. در این بخش از کد که در انتهای فایل main وجود دارد، عملیات پیداکردن روابط را فراخوانی می‌کنیم. توضیح مربوط به listener ی که برای define/defineby ساخته شده‌است، در بخش بعدی به صورت کامل آمده‌است.

class Project():  
 tree = None  
  
 def Parse(self, fileAddress):  
 file\_stream = FileStream(fileAddress)  
 lexer = JavaLexer(file\_stream)  
 tokens = CommonTokenStream(lexer)  
 parser = JavaParserLabeled(tokens)  
 tree = parser.compilationUnit()  
 self.tree = tree  
 return tree  
  
 def Walk(self, listener, tree):  
 walker = ParseTreeWalker()  
 walker.walk(listener=listener, t=tree)  
  
 def getListOfFiles(self, dirName):  
 listOfFile = os.listdir(dirName)  
 allFiles = list()  
 for entry in listOfFile:  
 *# Create full path* fullPath = os.path.join(dirName, entry)  
 if os.path.isdir(fullPath):  
 allFiles = allFiles + self.getListOfFiles(fullPath)  
 elif fnmatch(fullPath, "\*.java"):  
 allFiles.append(fullPath)  
  
 return allFiles  
  
 def getFileEntity(self, path):  
 *# kind id: 1* path = path.replace("/", "\\")  
 name = path.split("\\")[-1]  
 file = open(path, mode='r')  
 file\_ent = EntityModel.get\_or\_create(\_kind=1, \_name=name, \_longname=path, \_contents=file.read())[0]  
 file.close()  
 print("processing file:", file\_ent)  
 return file\_ent  
  
 def addDefineRefs(self, ref\_dicts, file\_ent):  
 for ref\_dict in ref\_dicts:  
 if ref\_dict["scope"] is None: *# the scope is the file* scope = file\_ent  
 else: *# a normal package* scope = self.getPackageEntity(file\_ent, ref\_dict["scope"], ref\_dict["scope\_longname"])  
  
 ent = self.getPackageEntity(file\_ent, ref\_dict["ent"], ref\_dict["ent\_longname"])  
  
 *# Define: kind id 194* define\_ref = ReferenceModel.get\_or\_create(\_kind=194, \_file=file\_ent, \_line=ref\_dict["line"],  
 \_column=ref\_dict["col"], \_ent=ent, \_scope=scope)  
  
 *# Definein: kind id 195* definein\_ref = ReferenceModel.get\_or\_create(\_kind=195, \_file=file\_ent, \_line=ref\_dict["line"],  
 \_column=ref\_dict["col"], \_scope=ent, \_ent=scope)

کد (2) فایل main.py بخش کلاس Project

### متد getListOfFiles:

این تابع به عنوان ورودی، آدرس پروژه را دریافت می‌کند. یک لیست به نام allFiles ساخته می‌شود و در ادامه فایل‌های مختلف موجود در آدرس داده‌شده بررسی می‌شوند و آدرس تمامی فایل‌هایی که به پسوند .java ختم می‌شوند (که همان فایل‌هایی هستند که ما قصد بررسی آن‌ها را داریم)، به لیست allFiles اضافه می‌شوند.

### متد getFileEntity:

در این تابع ابتدا روی آدرسی که به عنوان ورودی به تابع داده می‌شود، یک سری اصلاحات انجام می‌شود تا به فرمت مورد نظرمان در بیاید. در ادامه این فایل را به صورت یک Entity ذخیره می‌کنیم به طوری که برای فیلد kind مقدار 1، برای فیلد name نام فایل که از روی آدرس آن به دست می‌آید، برای path آدرسی که در تابع به فرمت درست ساخته‌بودیم و در نهایت در contents هم محتوای فایل ذخیره می‌شود و Entity مورد نظر ساخته می‌شود و در نهایت به عنوان خروجی توسط تابع برگردانده می‌شود.

### متد Parse:

هدف نهایی این تابع ساختن درخت تجزیه‌ی فایلی است که آدرس آن به عنوان ورودی به این تابع داده می‌شود. برای این مهم، لازم است مراحل زیر طی شود. این مراحل به ترتیب عبارتند از:

* بازکردن فایل با استفاده از آدرس فایل که به عنوان ورودی به تابع داده شده‌است.
* فراخوانی تابع JavaLexer مربوط به فایل ساخته‌شده و ذخیره خروجی آن در متغیر lexer
* دریافت توکن‌های مربوط به lexer ی که در بخش قبل برگردانده شده‌بود توسط تابع CommonTokenStream
* فراخوانی parser مربوط به زبان جاوا توسط تابع JavaParserLabeled
* ساخت درخت مربوطه توسط فراخوانی parser.compilationUnit()
* برگرداندن درخت ساخته‌شده به عنوان خروجی تابع

### متد addDefineRefs:

هدف از این تابع ساختن و افزودن روابط پیداشده به پایگاه داده است.

## define\_definein.py:

### کلاس DefineListener:

class DefineListener(JavaParserLabeledListener):  
 def \_\_init\_\_(self):  
 self.defines = []  
 self.package = ""  
 self.lambda\_expression\_count = 0

کد (3) فایل define\_definin.py بخش کلاس DefineListener

در این کلاس listener های مرتبط با هر کدام از قسمت‌ها را پیاده‌سازی می‌کنیم. هم‌چنین در ابتدا لیستی از پکیج‌ها و define ها را ایجاد می‌کنیم و در هر listener نتایج مربوطه را در این لیست‌ها ذخیره می‌کنیم.

### انواع listener ها:

def enterPackageDeclaration(self, ctx:JavaParserLabeled.PackageDeclarationContext):  
 self.package = [str(i) for i in ctx.qualifiedName().IDENTIFIER()]  
  
 ent\_start = ctx.qualifiedName().IDENTIFIER()[0]  
 ent\_name = ctx.qualifiedName().IDENTIFIER()[-1].getText()  
 ent\_longname = ".".join(self.package)  
 line = ent\_start.symbol.line  
 column = ent\_start.symbol.column  
 self.defines.append({  
 "scope": None, "ent": ent\_name,  
 "scope\_longname": None, "ent\_longname": ent\_longname,  
 "line": line, "col": column  
 })  
  
def enterClassDeclaration(self, ctx:JavaParserLabeled.ClassDeclarationContext):  
 ent = ctx.IDENTIFIER()  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 self.add\_define\_info(ent, ent\_parents)  
  
def enterInterfaceDeclaration(self, ctx:JavaParserLabeled.InterfaceDeclarationContext):  
 ent = ctx.IDENTIFIER()  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 self.add\_define\_info(ent, ent\_parents)  
  
def enterMethodDeclaration(self, ctx:JavaParserLabeled.MethodDeclarationContext):  
 ent = ctx.IDENTIFIER()  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 self.add\_define\_info(ent, ent\_parents)  
  
def enterAnnotationTypeDeclaration(self, ctx:JavaParserLabeled.AnnotationTypeDeclarationContext):  
 ent = ctx.IDENTIFIER()  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 self.add\_define\_info(ent, ent\_parents)  
  
def enterConstructorDeclaration(self, ctx:JavaParserLabeled.ConstructorDeclarationContext):  
 ent = ctx.IDENTIFIER()  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 self.add\_define\_info(ent, ent\_parents)  
  
def enterVariableDeclarator(self, ctx:JavaParserLabeled.VariableDeclaratorContext):  
 ent = ctx.variableDeclaratorId().IDENTIFIER()  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 self.add\_define\_info(ent, ent\_parents)  
  
def enterEnumConstant(self, ctx:JavaParserLabeled.EnumConstantContext):  
 ent = ctx.IDENTIFIER()  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 self.add\_define\_info(ent, ent\_parents)  
  
def enterEnumDeclaration(self, ctx:JavaParserLabeled.EnumDeclarationContext):  
 ent = ctx.IDENTIFIER()  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 self.add\_define\_info(ent, ent\_parents)  
 self.add\_define\_info(ent, ent\_parents + [ent.getText()], 'values')  
 self.add\_define\_info(ent, ent\_parents + [ent.getText()], 'valueOf')  
  
def enterFormalParameter(self, ctx:JavaParserLabeled.FormalParametersContext):  
 ent = ctx.variableDeclaratorId().IDENTIFIER()  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 self.add\_define\_info(ent, ent\_parents)  
  
def enterLambdaParameters0(self, ctx:JavaParserLabeled.LambdaParameters0Context):  
 self.lambda\_expression\_count += 1  
 ent = ctx.IDENTIFIER()  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 ent\_name = f'(lambda\_expr\_{self.lambda\_expression\_count})'  
 self.add\_define\_info(ent, ent\_parents, ent\_name)  
 self.add\_define\_info(ent, ent\_parents + [ent\_name])  
  
def enterLambdaParameters2(self, ctx:JavaParserLabeled.LambdaParameters2Context):  
 self.lambda\_expression\_count += 1  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 ent\_name = f'(lambda\_expr\_{self.lambda\_expression\_count})'  
 identifiers = ctx.IDENTIFIER()  
 self.add\_define\_info(identifiers[0], ent\_parents, ent\_name)  
 for ent in identifiers:  
 self.add\_define\_info(ent, ent\_parents + [ent\_name])  
  
def enterEnhancedForControl(self, ctx:JavaParserLabeled.EnhancedForControlContext):  
 ent = ctx.variableDeclaratorId().IDENTIFIER()  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 self.add\_define\_info(ent, ent\_parents)  
  
def enterCatchClause(self, ctx:JavaParserLabeled.CatchClauseContext):  
 ent = ctx.IDENTIFIER()  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 self.add\_define\_info(ent, ent\_parents)  
  
def enterTypeParameter(self, ctx:JavaParserLabeled.TypeParameterContext):  
 ent = ctx.IDENTIFIER()  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 self.add\_define\_info(ent, ent\_parents)  
  
def enterConstantDeclarator(self, ctx:JavaParserLabeled.ConstantDeclaratorContext):  
 ent = ctx.IDENTIFIER()  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 self.add\_define\_info(ent, ent\_parents)  
  
def enterLastFormalParameter(self, ctx:JavaParserLabeled.LastFormalParameterContext):  
 ent = ctx.variableDeclaratorId().IDENTIFIER()  
 ent\_parents = class\_properties.ClassPropertiesListener.findParents(ctx)  
 self.add\_define\_info(ent, ent\_parents)

کد (4) فایل define\_definin.py بخش انواع listener ها در کلاس DefineListener

در ادامه به توضیح هر کدام از listener ها می‌پردازیم:

* متد enterPackageDeclaration:

این متد برای یافتن تعریف فایل‌ها توسط package ها به کار می‌رود.

* متد enterClassDeclaration:

این متد برای یافتن تعریف class به کار می‌رود.

* متد enterInterfaceDeclaration:

این متد برای یافتن تعریف interface به کار می‌رود.

* متد enterMethodDeclaration:

این متد برای یافتن تعریف متدها به کار می‌رود.

* متد enterAnnotationTypeDeclaration:

این متد برای یافتن تعریف annotation به کار می‌رود.

* متد enterConstructorDeclaration:

این متد برای یافتن تعریف متد constructor به کار می‌رود.

* متد enterVariableDeclarator:

این متد برای یافتن تعریف متغیرها به کار می‌رود.

* متد enterEnumConstant:

این متد برای یافتن تعریف enum به کار می‌رود.

* متد enterEnumDeclaration:

این متد برای یافتن متغیرهای تعریف‌شده در enum به کار می‌رود.

* متد enterFormalParameter:

این متد برای یافتن تعریف پارامترهای توابع به کار می‌رود.

* متد enterLambdaParameters0:

این متد برای یافتن عبارات لاندایی است که یک پارامتر دارند.

* متد enterLambdaParameters2:

این متد برای یافتن عبارات لاندایی است که بیش از یک پارامتر دارند.

* متد enterEnhancedForControl:

این متد برای یافتن پارامترهایی است که داخل foreach جاوا تعریف می‌شوند.

* متد enterCatchClause:

این متد برای یافتن تعریف متغیرها در بخش catch به کار می‌رود.

* متد enterTypeParameter:

این متد برای یافتن تعریف متغیرهای generic به کار می‌رود.

* متد enterConstantDeclarator:

این متد برای یافتن تعریف متغیرهای ثابت به کار می‌رود.

* متد enterLastFormalParameter:

این متد برای یافتن enum آخرین پارامتر یک تابع که به صورت ... است، به کار می‌رود.

### متد addDefineInfo:

def addDefineInfo(self, ent, ent\_parents):  
 ent\_name = ent.getText()  
 line = ent.symbol.line  
 column = ent.symbol.column  
 scope\_longname = ".".join(self.package + ent\_parents)  
 ent\_longname = scope\_longname + "." + ent\_name  
 if len(ent\_parents) == 0:  
 scope\_name = None  
 else:  
 scope\_name = ent\_parents[-1]  
 self.defines.append({  
 "scope": scope\_name, "ent": ent\_name,  
 "scope\_longname": scope\_longname, "ent\_longname": ent\_longname,  
 "line": line, "col": column  
 })

کد (5) فایل define\_definin.py بخش اضافه‌کردن اطلاعات define ها در کلاس DefineListener

در این تابع اطلاعات پیداشده توسط listener ها را در لیست defines اضافه می‌کنیم.

## class\_properties.py:

لازم به ذکر است به علت اینکه فایل class\_properties.py به طور کامل نتایج مورد نظر ما را پیدا نمی‌کرد، تابع findParents آن را کمی تغییر دادیم. بعد از تغییر این تابع به صورت زیر درآمد.

@staticmethod  
def findParents(c: ParserRuleContext): # includes the ctx identifier  
 parents = []  
 current = c.parentCtx  
 while current is not None:  
 if current.getRuleIndex() in [  
 JavaParserLabeled.RULE\_classDeclaration,  
 JavaParserLabeled.RULE\_methodDeclaration,  
 JavaParserLabeled.RULE\_enumDeclaration,  
 JavaParserLabeled.RULE\_interfaceDeclaration,  
 JavaParserLabeled.RULE\_constructorDeclaration,  
 JavaParserLabeled.RULE\_annotationTypeDeclaration,  
 ]:  
 parents.append(current.IDENTIFIER().getText())  
 current = current.parentCtx  
 return list(reversed(parents))

کد (6) فایل class\_properties.py متد findParents

# ارزیابی:

در ابتدای امر، برای ارزیابی برنامه چند فایل کوچک را به Understand و OpenUnderstand دادیم و نتایج آن‌ها را با بررسی یک به یک نتایج این دو برنامه، مقایسه کردیم. سپس بعد از اینکه ایرادات مربوط به این فایل‌های کوچک برطرف شدند، برای بررسی فایل‌ها و پروژه‌های بزرگ‌تر نیاز بود تا یک فایل برای بررسی دقیق‌تر نتایج Understand ایجاد شود. برای این کار از API هایی که خود نرم‌افزار Understand در اختیار ما قرار می‌داد استفاده کردیم.

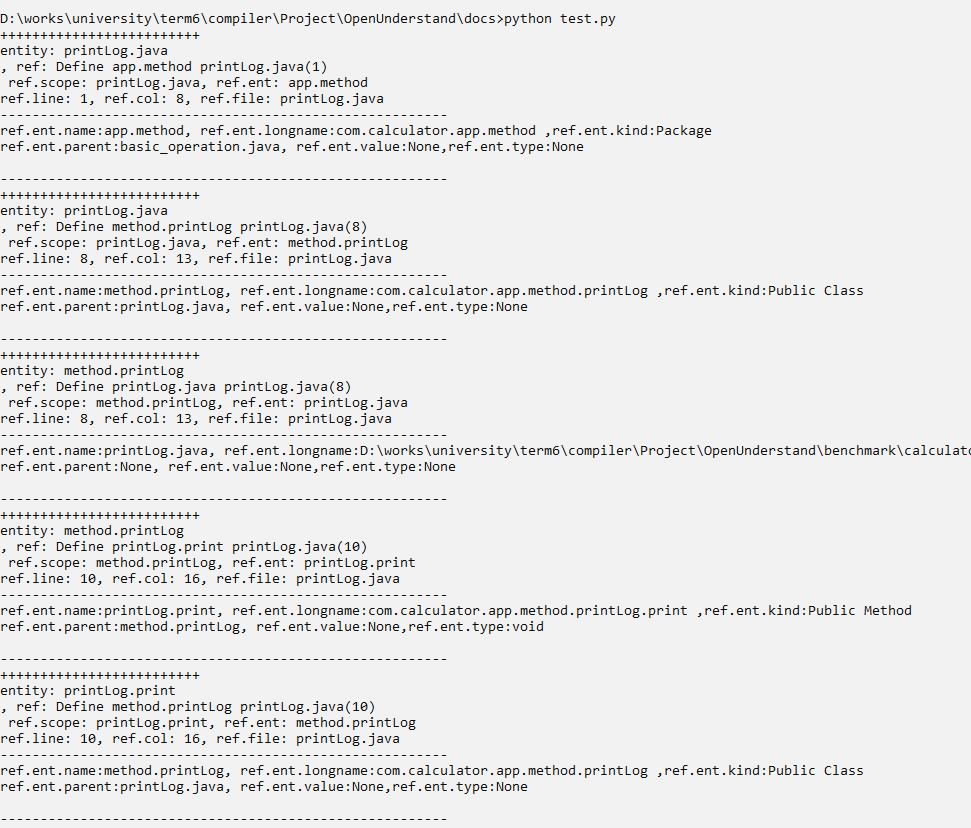
ابتدا لازم است تا پروژه را به نرم افزار Understand بدهیم تا فایلی با پسوند .udb توسط نرم‌افزار ساخته شود. سپس با استفاده از کد زیر می‌توانیم به محتوای تولید شده توسط نرم‌افزار understand دسترسی داشته باشیم.

در ادامه می‌توانیم به نتایج به شیوه‌ی دلخواه دسترسی داشته باشیم. برای این منظور گروه ما نتایج را در کنسول پرینت می‌کرد. همچنین یک متغیر counter هم برای محاسبه‌ی تعداد فیلدهای ساخته شده تعریف شده‌است. محتوای کامل فایل بالا با نام test.py در زیر آورده شده‌است.

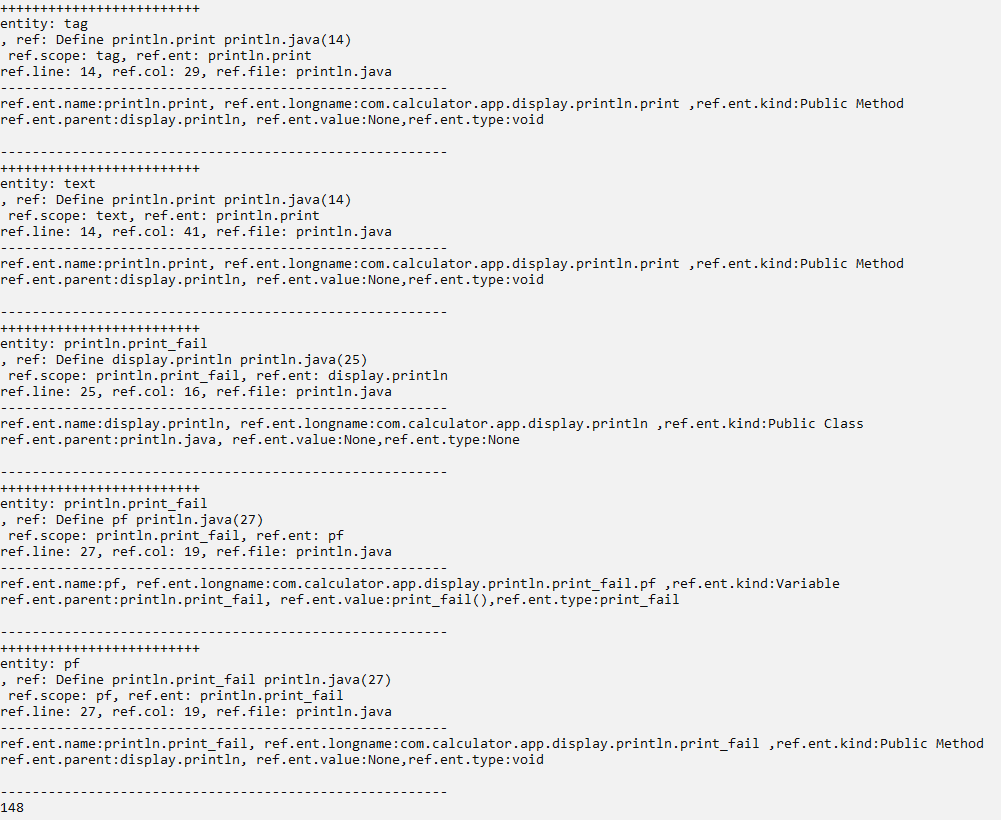
from pprint import pprint  
  
try:  
 import understand  
except ImportError:  
 print("Can not import understand")  
  
db = understand.open(r"D:\works\university\term6\compiler\Project\OpenUnderstand\benchmark\calculator\_app\calculator\_app\calculator\_app1.udb")  
  
counter = 0  
  
for ent in db.ents():  
 for ref in ent.refs():  
  
 if ref.kindname() == "Define":  
 counter += 1  
   
 print("+++++++++++++++++++++++++")  
  
 print(f"entity: {ent}\n, ref: {ref}\n ref.scope: {ref.scope()}, ref.ent: {ref.ent()}\n"  
 f"ref.line: {ref.line()}, ref.col: {ref.column()}, ref.file: {ref.file().name()}")  
 print("--------------------------------------------------------")  
 print(f"ref.ent.name:{ref.ent().name()}, ref.ent.longname:{ref.ent().longname()} ,ref.ent.kind:{ref.ent().kind()}\n"  
 f"ref.ent.parent:{ref.ent().parent()}, ref.ent.value:{ref.ent().value()},ref.ent.type:{ref.ent().type()}\n")  
 print("--------------------------------------------------------")  
  
print(counter)

کد (7) فایل test.py

در ادامه نتایج بدست آمده از اجرای برنامه‌ی calculator\_app توسط برنامه‌ی Understand و همچنین توسط کد نوشته‌شده‌ی ما آورده شده‌است.

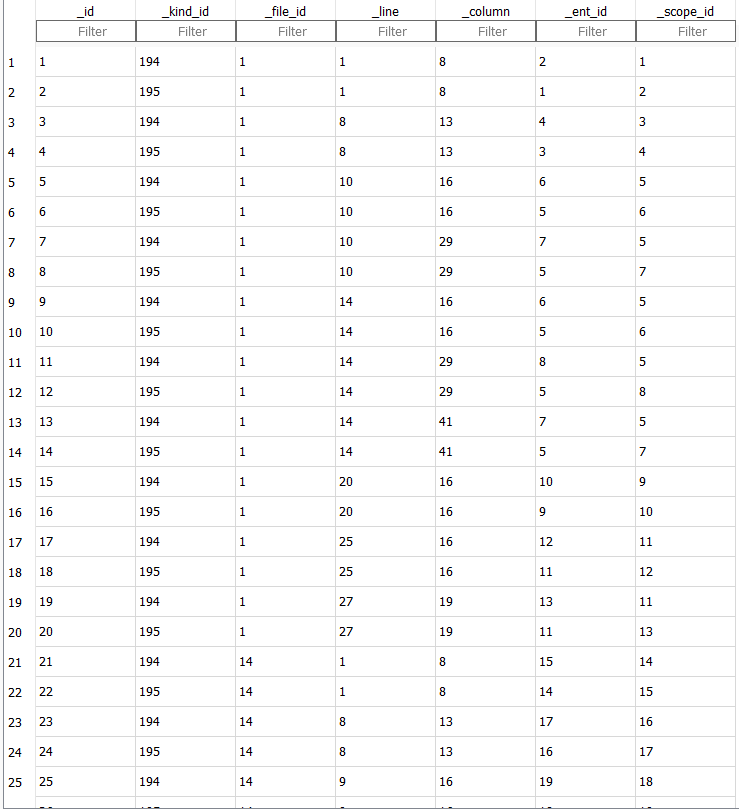
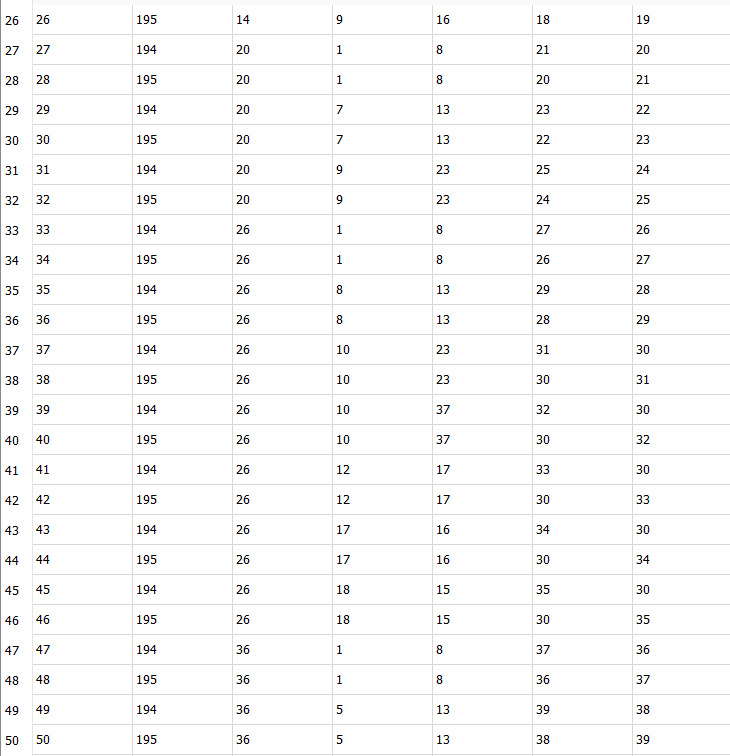
تصویر (1) نتایج پایگاه داده Understand

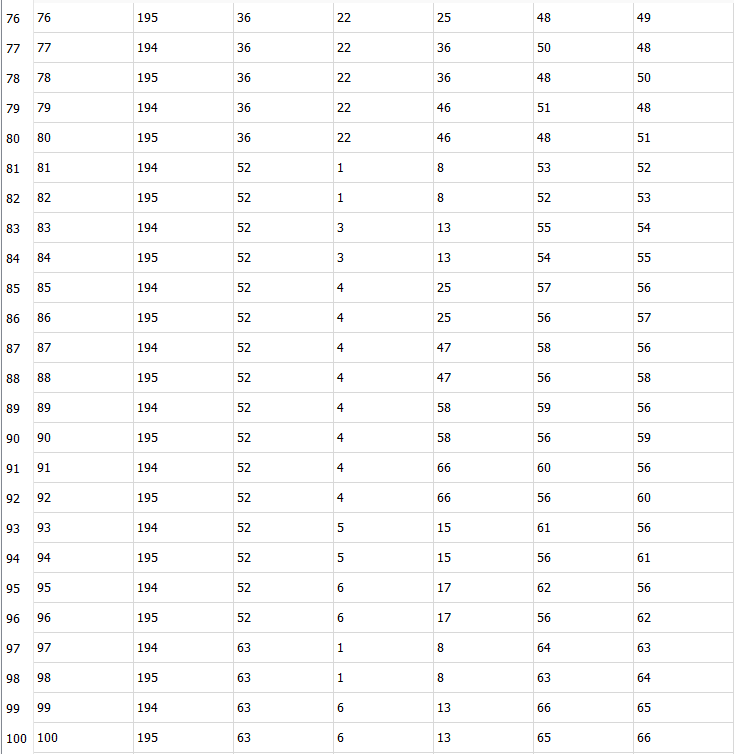
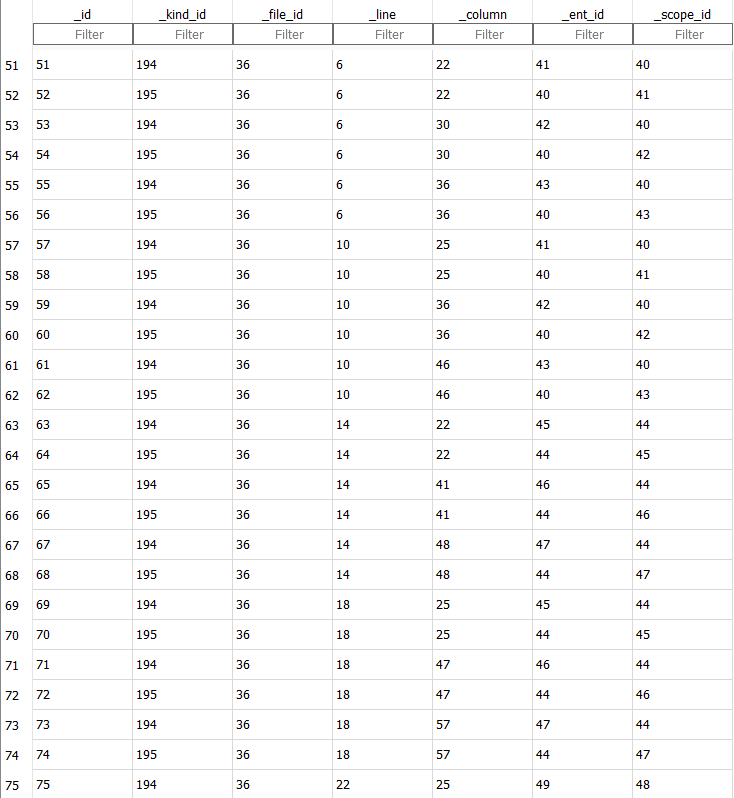
در این پروژه در مجموع 148 مورد define/defineIn پیدا شد. به علت اینکه پوشش دادن تمام این موارد در فایل باعث زیاد شدن حجم گزارش می‌شود، بنابراین عکس سایر موارد به طور کامل آورده نشده‌است و در تصویر بعدی تعداد define/defineIn ها آورده شده‌است تا بتوانیم با بخش بعدی که نتایج Open Understand است بهتر مقایسه انجام بدهیم.

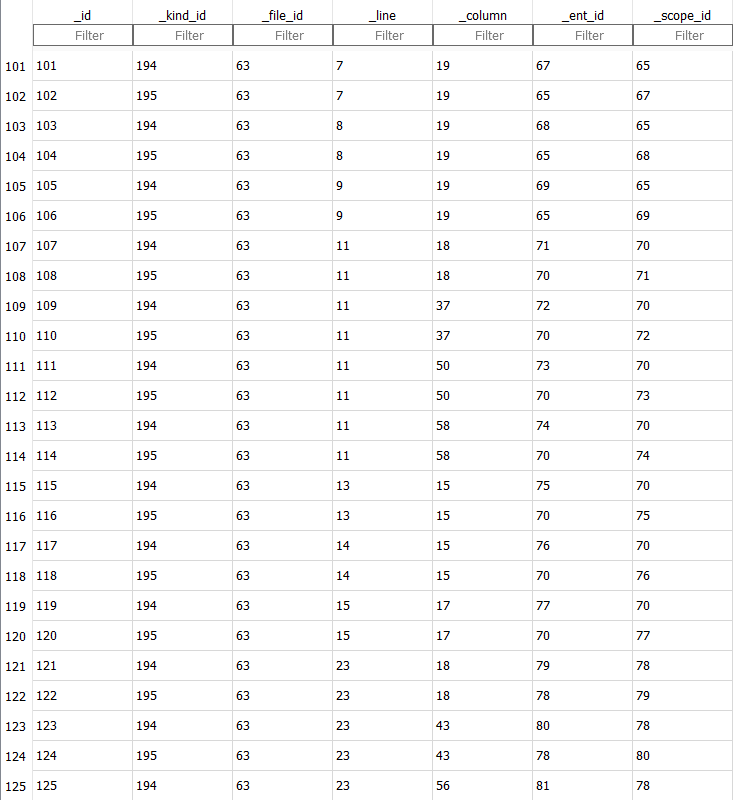
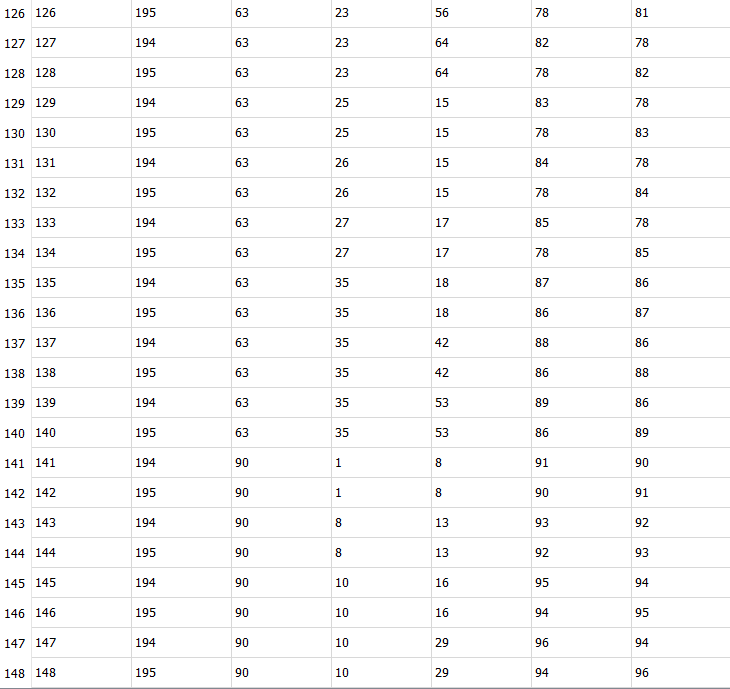


تصویر (2) ادامه نتایج پایگاه داده Understand

در تصویرهای بعدی، نتایج مربوط به دیتابیس Open Understand آورده شده‌است.





تصویر (3) مجموعه نتایج پایگاه داده Open Understand

با بررسی نتایج متوجه می‌شویم که هم دیتابیس OpenUnderstand به ما 148 خروجی می‌دهد و هم خروجی فایل test.py. همچنین در صورتی که دقیق‌تر بررسی کنیم، هر یک از سطرهای این دو خروجی باهم مطابقت دارند. بنابراین برنامه به درستی کار می‌کند.

# مشکلات و چالش‌ها:

یکی از مشکلات به‌وجود آمده در پروژه، تلاش برای ساخت فایل تست بود. در ابتدا ما فکر می‌‎کردیم که امکان دسترسی به پایگاه داده Understand وجود ندارد و به همین علت زمان زیادی را صرف بررسی دستی نتایج برنامه‌ی خودمان با Understand کردیم. پس از آن‌که متوجه شدیم امکان بررسی بهتر نتایج وجود دارد، با بررسی API های سایت Understand متوجه شدیم که اگر آدرس فایل udb ساخته‌شده را به متد open از کتابخانه‌ی understand بدهیم، می‌توانیم به داده‌ها دسترسی داشته باشیم. اما در ابتدا با دادن پروژه‌های مختلف به برنامه‌ی Understand فایل udb توسط برنامه ساخته نمی‌شد. در اینجا متوجه شدیم که باید نسخه‌های قدیمی‌تر از Understand را نصب می‌کردیم و ورژن کنونی با کرک، همچین فایلی نمی‌سازد.

پس از نصب دوباره‌ی نسخه‌های قدیمی‌تر برنامه، وقتی تلاش کردیم تا فایل test.py را اجرا کنیم، این بار برنامه متد open را شناسایی نمی‌کرد. پس از پرس‌و‌جو از سایر دانشجویان و همچنین دستیاران آموزشی، متوجه شدیم که بعد از نصب برنامه‌ی Understand لازم بود تا یک فایل از پوشه‌ی Scripts به نام api\_install\_test.py را اجرا کنیم و پس از رفع ارورهای آن امکان شناسایی متد open وجود دارد.

پس از اجرای مراحل بالا، درنهایت توانستیم فایل test.py را اجرا کنیم.

# نتیجه‌گیری و کارهای آتی:

با استفاده از بخش پیاده‌سازی‌شده می‌توان رابطه define/definein را در کدهای جاوا پیدا و تجزیه و تحلیل کرد. در آینده می‌توان راهی برای مقایسه نتایج ابزار Open Understand و ابزار Understand ارائه داد تا بتوان به صورت خودکار نتایج را مقایسه کرد و از صحت نتایج اطمینان یافت.