داکیومنت پروژه ی کامپایلر

گروه 8

اعضای گروه:

محسن اکبری **96521047**

علی گرانمایه **96521452**

سید محسن اسلام پناه **97521063**

سید مهدی هاشمی

1. مقدمه

در این پروژه، افراد این گروه سعی در پیاده سازی متن باز از Understand Python API برای تجزیه و تحلیل کدهای منبع بوده اند. در واقع این کار توسعه یک پیاده‌سازی منبع باز Understand Python API، بوده است.
ابتدا بر پیاده سازی API برای برنامه های جاوا با استفاده از زبان های برنامه نویسی پایتون و ابزارهای کامپایلر مانند Antlr4 کار شده است. و ساختارهایی را که توسط Understand برای تجزیه و تحلیل کدهای منبع استفاده می‌شود، بررسی شده است.
برای گروه 4G باید تجزیه و تحلیلی را برای یافتن زیر مجموعه ای از انواع مراجع فهرست شده برای implement Couple_implementbyCouple by،Create Create by و declare declare in به همراه موجودیت های مربوطه خود ایجاد می شد.
مواردی که در پروژه استفاده میشود و بهتر است این جا معرفی شوند:

- Entity: هر چیزی در کد است که Understand اطلاعات مربوط به آن را می گیرد: به عنوان مثال، یک فایل، یک کلاس، یک متغیر، یک تابع و غیره. برای ذخیره موجودیت های جاوا در پروژه استفاده می شود . این جدول در طول تجزیه و تحلیل استاتیک برنامه توسط ANTLR Listener پر می شود. دارای مجموعه ای منحصر به فرد از ویژگی ها است که می تواند توسط API جست و جو شود
- Reference: مکان خاصی که یک موجودیت در کد ظاهر می شود. یک مرجع همیشه به عنوان رابطه بین دو موجودیت تعریف شود.دارای هر دو نهاد مرتبط با آن و همچنین فایل، خط و ستونی است که مرجع در آن قرار می گیرد و نوع مرجعیت آن است. دارای مجموعه ای منحصر به فرد از ویژگی ها است که می تواند توسط API جست و جو شود
- Project: برای ذخیره برخی از اطلاعات اولیه پروژه در حال تحلیل مانند نام پروژه، زبان های برنامه نویسی و غیره. این جدول به صورت خودکار پر می شود.
- Kind: برای ذخیره هر دو نوع موجودیت و مرجع که با پر شدن خودکار دیتابیس و اعداد لازم و مرتبط با بخش های مختلف قابل تفکیک است.

برای بخش پایگاه داده این پروژه از کتابخانه peewee و 3SQLite استفاده شده است و بیشتر داده‌های جمع‌آوری‌شده شامل Entity, Reference است.
در ادامه به ترتیب

- به روش پیشنهادی که شامل
  - کد اصلی در main، نحوه ی پیدا کرد
  - ست کردن های entity , parent
  - پیاده سازی Throw و ThrowBy
  - نحوه ی استفاده از Throw و ThrowBy
  - پیاده سازی DotRef و DotRefBy
  - نحوه ی استفاده از DotRef و DotRefBy
- ارزیابی
  - که جواب های بدست آمده را با جواب هایی که understand به ما میدهد، مقایسه می شود.
- نتیجه گیری و کارهای آتی
  - نشان دادن نتیجه کلی برنامه ی زده شده نوشته می شود.

می پردازیم

2. روش پیشنهادی

کد کلی برای تمام قسمتها (**main.py**)

در این قسمت ابتدا دیتابیس پایه ساخته می‌شود و سپس روی تمام فایل‌های جاوای درون آدرس فولدر داده شده (path) برای هر کدام از رفرنس‌های پروژه یک حلقه می‌زند.

به ازای هر کدام از فایل‌ها در این حلقه ابتدا entity فایل برگردانده می‌شود (getFileEntity in Project class) و سپس listener مربوط به رفرنس ساخته می‌شود و در متد ParseAndWalk آن listener و آدرس فایل برای ساخت FileStream داده می‌شود. این متد درخت را walk کرده و listener با لیستی از خصوصیات رفرنس‌های پیدا شده در فایل پر می‌شود. سپس متد مخصوص به آن رفرنس برای پر کردن دیتابیس صدا زده می‌شود. همچنین در اینجا error handling صورت گرفته تا اگر یک فایل به مشکل خورد بتواند به اجرا ادامه دهد و دیگر فایلها را پردازش کند.

یافتن فیلدهای مربوط به یک کلاس یا اینترفیس (**class_properties.py**)

```
def enterClassDeclaration(self, ctx:JavaParserLabeled.ClassDeclarationContext):
  if self.class_properties:  # already found the class
    return
  if self.class_longname[-1] == ctx.IDENTIFIER().getText():
    if self.checkParents(ctx):
      # this is the exact class we wanted.
      self.class_properties = {}
      self.class_properties["name"] = self.class_longname[-1]
      self.class_properties["longname"] = ".".join(self.class_longname)

      if len(self.class_longname) == 1:
        self.class_properties["parent"] = None
      else:
        self.class_properties["parent"] = self.class_longname[-2]
      self.class_properties["modifiers"] =
ClassPropertiesListener.findClassOrInterfaceModifiers(ctx)
      self.class_properties["contents"] = ctx.getText()

class InterfacePropertiesListener(JavaParserLabeledListener):
  interface_longname = []
  interface_properties = None

  def checkParents(self, c):
```

```
        return set(ClassPropertiesListener.findParents(c)) &
set(list(reversed(self.interface_longname)))

    def enterInterfaceDeclaration(self, ctx:JavaParserLabeled.InterfaceDeclarationContext):
        if self.interface_properties:  # already found the class
            return
        if self.interface_longname[-1] == ctx.IDENTIFIER().getText():
            if self.checkParents(ctx):
                # this is the exact class we wanted.
                self.interface_properties = {}
                self.interface_properties["name"] = self.interface_longname[-1]
                self.interface_properties["longname"] = ".".join(self.interface_longname)

                if len(self.interface_longname) == 1:
                    self.interface_properties["parent"] = None
                else:
                    self.interface_properties["parent"] = self.interface_longname[-2]
                self.interface_properties["modifiers"] =
ClassPropertiesListener.findClassOrInterfaceModifiers(ctx)
                self.interface_properties["contents"] = ctx.getText()
```

برای یافتن فیلدهای یک کلاس یا اینترفیس که در listener مربوط به آن تنها long name آن در دسترس بود از این دو listener استفاده می‌شود. (برای رفرنس‌های Implement/ImplementBy و Create/CreateBy)
این دو listener تفاوت زیادی باهم ندارند، تنها به دلیل متفاوت بودن ruleهای مورد نیازشان (که ClassDeclaration باشد یا InterfaceDeclaration) این دو کلاس ClassPropertiesListener و InterfacePropertiesListener جدا ساخته شده‌اند.به همین دلیل تنها به شرح ClassPropertiesListener پرداخته شده است.
برای استفاده از listener ابتدا longname کلاس به صورت یک لیست که از استرینگ نام بالاترین parent تا نام خود کلاس را دارد داده می‌شود. همچنین یک class_properties نیز تعریف شده است که در صورتی که کلاس مورد نظر پیدا شود آن متغیر با یک دیکشنری از فیلدهای موردنیاز انتتی کلاس پر خواهد شد.
در این کلاس تنها یک rule بررسی شده است: classDeclaration.
توضیح enterClassDeclaration: با رسیدن به هر class declaration در درخت پارسر ابتدا چک می‌شود که در صورت پیدا شدن کلاس درست تا آن زمان ادامه متد اجرا نشود. اما اگر class_properties هنوز None باشد ابتدا نام کلاسی که به آن رسیده است (IDENTIFIER) با نام کلاس موردنظر که آخرین المان class_longname است مقایسه می‌شود. اگر نامشان برابر باشد باید چک شود که آیا parentهای این کلاس با کلاس موردنظر یکی است یا نه، به این دلیل که چند کلاس ممکن است با نام‌های یکسان در فایل موجود باشند، پس لازم است کلاسی که پرنت‌هایش با پرنت‌های کلاس موردنظر یکسان باشد را بیابیم. پرنت‌های کلاس را به کمک متد استاتیک findParents پیدا می‌کنیم که در قسمتی جدا توضیح داده شده است.
در صورتی که هم نام هم parentهای کلاس یکی باشد باید فیلدهای مورد نیاز برای ساخت انتیتی را پیدا کند. name, longname و parent از longname داده شده به دست می‌آیند. برای گرفتن modifierهای کلاس (مثلا public یا static بودن) از متد استاتیک findClassOrInterfaceModifiers استفاده کردم. این modifier برای پیدا کردن

kind مرتبط با کلاس استفاده می‌شود. همچنین با گرفتن کل متن class declaration قسمت contents انتتی را به دست آوردیم.

یافتن **parent**های یک **entity** به کمک متد **findParents**

```
@staticmethod
def findParents(c):  # includes the ctx identifier
  parents = []
  current = c
  while current is not None:
    if type(current).__name__ == "ClassDeclarationContext" or type(current).__name__ ==
"MethodDeclarationContext"\
        or type(current).__name__ == "EnumDeclarationContext"\
        or type(current).__name__ == "InterfaceDeclarationContext"\
        or type(current).__name__ == "AnnotationTypeDeclarationContext":
      parents.append(current.IDENTIFIER().getText())
    current = current.parentCtx
  return list(reversed(parents))
```

این متد context رول مربوط به آن انتتی را می‌گیرد و روی contextهای parent آن حلقه می‌زند. در این حلقه اگر contextهای پیدا شده مربوط به رول‌های ClassDeclaration، EnumDeclaration ، InterfaceDeclaration و یا AnnotationTypeDeclaration باشد که همه درون خود یک IDENTIFIER (نام آن کلاس / enum / interface / annotation type است) را دارند، متن IDENTIFIER به عنوان parent به لیست اضافه می‌شود. در پایان چون parentها از خود کلاس شروع شده و از پایین به بالاست، این لیست reverse شده است.

یافتن **modifier**های یک **entity** کلاس یا **interface** یا **enum** به کمک متد **findClassOrInterfaceModifiers**

```
@staticmethod
def findClassOrInterfaceModifiers(c):
  m = ""
  modifiers=[]
  current = c
  while current is not None:
    if "typeDeclaration" in type(current.parentCtx).__name__:
      m=(current.parentCtx.classOrInterfaceModifier())
      break
    current = current.parentCtx
  for x in m:
```

```
        modifiers.append(x.getText())
    return modifiers
```

با دقت در رول type declaration می‌یابیم که تمام modifierهای انواع انتتی‌های مورد نیاز ما برای پروژه می‌تواند با رسیدن به این رول و گرفتن classOrInterfaceModifier به دست آید. به همین دلیل مانند قبل parentهای این context را بالا رفته تا به یک type declaration برسد. سپس modifierهای درون آن به لیست modifiers اضافه شده است.

یافتن یا ایجاد مدل انتتی‌ها در کلاس **Project** (فایل **main.py**)

انتتی فایل (**getFileEntity**)

```
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
```

این متد فیلدهای نام و longname انتتی فایل را با استفاده از آدرس آن به دست می‌آورد و همچنین با open کردن آن contents درونش را خوانده و تمام این فیلدها را بعلاوه kind id آن (که در دیتابیس ۱ است) برای گرفتن یا ساخت انتتی فایل به کار می‌برد.

انتتی پکیج (**getPackageEntity**)

```
def getPackageEntity(self, file_ent, name, longname):
    # package kind id: 72
    ent = EntityModel.get_or_create(_kind= 72, _name=name, _parent=file_ent,
                                    _longname=longname, _contents="")
    return ent[0]
```

فیلدهای پکیج عبارت است از kind آن، نام و parent و longname. همچنین contents آن استرینگ خالی در نظر گرفته می‌شود.

انتتی پکیج بدون نام (getUnnamedPackageEntity)

```
def getUnnamedPackageEntity(self, file_ent):
  # unnamed package kind id: 73
  ent = EntityModel.get_or_create(_kind= 73, _name="(Unnamed_Package)",
_parent=file_ent,
                                  _longname="(Unnamed_Package)", _contents="")
  return ent[0]
```

فرق این متد با متد بالا در kind id و نام و longname است که دو مورد آخر (Unnamed Package) در نظر گرفته می‌شوند.

انتتی کلاس (getClassEntity)

```
def getClassEntity(self, class_longname, file_address):
  props = p.getClassProperties(class_longname, file_address)
  if not props:  # This class is unknown, unknown class id: 84
     ent = EntityModel.get_or_create(_kind=84, _name=class_longname.split(".")[-1],
                                  _longname=class_longname, _contents="")
  else:
     if len(props["modifiers"]) == 0:
        props["modifiers"].append("default")
     kind = self.findKindWithKeywords("Class", props["modifiers"])
     ent = EntityModel.get_or_create(_kind=kind, _name=props["name"],
                                  _longname=props["longname"],
                                  _parent= props["parent"] if props["parent"] is not None else file_ent,
                                  _contents=props["contents"])
  return ent[0]
```

برای گرفتن انتتی یک کلاس به کمک آدرس فایل و نام کامل (longname) کلاس از این متد استفاده می‌شود. ابتدا به کمک این دو ClassPropertiesListener تلاش می‌کند این کلاس را در فایل پیدا کند. اگر نتواند این کار را بکند یعنی این کلاس در فایل تعریف نشده و لازم است انتتی از نوع unknown class باشد. در غیر این صورت باید نوع دقیق کلاس پیدا شود. برای این کار از متد findKindWithKeywords استفاده شده است که نوع انتتی ("Class") و modifierهای آن را میگیرد و مناسبترین kind را بازمیگرداند. سپس انتتی کلاس به کمک get_or_create به دست می‌آید. اگر در دیکشنری parent نداشته باشد یعنی parent آن خود فایل است.

انتتی اینترفیس (getInterfaceEntity)

```
def getInterfaceEntity(self, interface_longname, file_address): # can't be of unknown kind!
  props = p.getInterfaceProperties(interface_longname, file_address)
```

```
  if not props:
    return None
  else:
    kind = self.findKindWithKeywords("Interface", props["modifiers"])
    ent = EntityModel.get_or_create(_kind=kind, _name=props["name"],
                                    _longname=props["longname"],
                                    _parent= props["parent"] if props["parent"] is not None else file_ent,
                                    _contents=props["contents"])
  return ent[0]
```

برای گرفتن انتتی اینترفیس به کمک آدرس فایل و نام کامل (longname) اینترفیس از این متد استفاده می‌شود. ابتدا به کمک این دو InterfacePropertiesListener تلاش می‌کند آن را در فایل پیدا کند. اگر نتواند این کار را بکند از آنجا که اینترفیس از نوع unknown نداریم None برمیگرداند تا متدی که این متد را صدا زده به جای اینترفیس آن را کلاس در نظر بگیرد و اگر جایی تعریف نشده بود unknown class برگرداند. در غیر این صورت باید نوع دقیق اینترفیس پیدا شود. برای این کار از متد findKindWithKeywords استفاده شده است که نوع انتتی ("Interface") و modifierهای آن را میگیرد و مناسبترین kind را بازمیگرداند. سپس انتتی اینترفیس به کمک get_or_create به دست می‌آید. اگر در دیکشنری parent نداشته باشد یعنی parent آن خود فایل است.

متد کمکی **findKindWithKeywords** (فایل **main.py**)

```
def findKindWithKeywords(self, type, modifiers):
  if len(modifiers) == 0:
    modifiers.append("default")
  leastspecific_kind_selected = None
  for kind in KindModel.select().where(KindModel._name.contains(type)):
    if self.checkModifiersInKind(modifiers, kind):
      if not leastspecific_kind_selected \
          or len(leastspecific_kind_selected._name) > len(kind._name):
        leastspecific_kind_selected = kind
  return leastspecific_kind_selected


def checkModifiersInKind(self, modifiers, kind):
  for modifier in modifiers:
    if modifier.lower() not in kind._name.lower():
      return False
  return True
```

در این متد type منظور نوع اصلی (مثلا "Class" یا "Interface" یا "Enum" یا بودن نوع) و modifiers خصوصیات فرعی نوع (مثلا پابلیک بودن یا استاتیک بودن) را دارند. اگر modifiers خالی باشد یک default به آن اضافه میکنیم که نوع دیفالت آن پیدا شود.

سپس بین تمام kindهایی که درونشان type را دارند چک میکنیم و بین آن‌هایی که modifierها را هم در نامشان دارند kind با کوتاهترین نام را برمی‌گردانیم که احتمالا خصوصیات اضافی که در modifiers نیامده را ندارد. به طور مثال اگر modifiers تنها public داشته باشد و به دنبال یک کلاس باشیم، بین دو نوع Java Abstract Class Type و Public Member و Java Class Type Public Member آن که کوتاه تر است را انتخاب می‌کند که نوع مورد نظر ماست.

رفرنس‌های DotRefBy و DotRef (فایل DotRef_DotRefBy.py)

```
class DotRef_DotRefBy(JavaParserLabeledListener):
    state = False
    class_name =[]

    def enterPackageDeclaration(self,
ctx:JavaParserLabeled.PackageDeclarationContext):
        all_pac = ctx.qualifiedName().IDENTIFIER()
        self.class_name.append(ctx.qualifiedName().getText())
        if len(all_pac)>0:
            self.state = True
```

.

ابتدا لیست Class_name را می سازیم و نام تمام کلاس ها را درون آن قرار می دهیم.
در متغییر state اگر در فایل ما نامی برای پکیج داشته باشد True و در غیر این صورت ، False می شود و این کار را enterPackageDeclaration انجام می دهد.

```
def enterClassDeclaration(self,
ctx:JavaParserLabeled.ClassDeclarationContext):
    self.class_name.append(ctx.IDENTIFIER().getText())
```

این تابع اسامی کلاس ها را اضافه می کند.

```
def enterExpression1(self, ctx:JavaParserLabeled.Expression0Context):

    if ctx.DOT():
        if ctx.expression() and ("DOT" not in dir(ctx.expression())):
            modifiers = self.findmethodacess(ctx)
            mothodedreturn, methodcontext = self.findmethodreturntype(ctx)

            if self.state:
                refEntName = ctx.expression().getText()
            else:
                refEntName = None

            allrefs =
class_properties.ClassPropertiesListener.findParents(ctx)  #
self.findParents(ctx)
            refent = allrefs[-1]
            if refEntName in self.class_name:
                entlongname = ".".join(allrefs)
                [line, col] = str(ctx.start).split(",")[3].split(":")

                self.implement.append({"scopename": refent, "scopelongname":
```

```
entlongname, "scopemodifiers": modifiers,
                                    "scopereturntype": mothodedreturn,
"scopecontent": methodcontext,
                                    "line": line, "col": col[:-1], "refent":
refEntName,
                                    "scope_parent": allrefs[-2] if
len(allrefs) > 2 else None,
                                    "potential_refent": ".".join(
                                        allrefs[:-1]) + "." + refEntName if
refEntName else ""})
```

هنگام وارد شدن به Expression1 ابتدا چک می کند که آیا Dot درون آن وجود دارد و بعد از آن چک می کند که درون آن Expression1 دیگری هم هست یا خیر ، سپس نام انتیتی قبل از Dot را می گیرد و بررسی می کند که آیا درون لیست class_name هست، اگر بله ، با استفاده از توابعی که قبل تر از این توضیح داده شده است ، اطلاعات لازم برای دیتابیس را ذخیره می کند.

استفاده از **DotRef and DotRefBy Listener** در فایل **main.py**

از همان تابعی که برای Throws ThrowsBy استفاده شد، دوباره استفاده می کند با این تفاوت که مقدار متغییر Throw را False می گذارد.

```
if not Throw:
    if ref_dict["refent"] is None:
        ent = self.getUnnamedPackageEntity(file_ent)
    else:
        ent = self.getPackageEntity(file_ent, ref_dict["refent"],
ref_dict["refent"])
else:
    ent = self.getThrowEntity(ref_dict["refent"], file_address)
```

در این جا با استفاده از getUnnamedPackageEntity و getPackageEntity که قبلا توضیح داده شده است ، وضعیت نام package مشخص میشود.

رفرنس های، **Throw , Throwby** (فایل **Throws_ThrowsBy.py**)

```
def enterMethodDeclaration(self,
ctx:JavaParserLabeled.EnumDeclarationContext):

    if ctx.THROWS():
        modifiers = self.findmethodacess(ctx)
        mothodedreturn, methodcontext = self.findmethodreturntype(ctx)
        refEntName = ctx.qualifiedNameList().getText()
        if refEntName:
            allrefs =
class_properties.ClassPropertiesListener.findParents(ctx)  #
self.findParents(ctx)
            refent = allrefs[-1]
```

```
            entlongname = ".".join(allrefs)
            [line, col] = str(ctx.start).split(",")[3].split(":")

            self.implement.append({"scopename": refent, "scopelongname":
entlongname, "scopemodifiers": modifiers,
                                   "scopereturntype": mothodedreturn,
"scopecontent": methodcontext,
                                   "line": line, "col": col[:-1], "refent":
refEntName,
                                   "scope_parent": allrefs[-2] if len(allrefs) >
2 else None,
                                   "potential_refent": ".".join(
                                       allrefs[:-1]) + "." + refEntName})
```

در این تابع ابتدا چک می کند که آیا Throws وجود دارد ، اگر داشت در ادامه مقادیری که باید وارد دیتابیس شود را به عنوان یک دیکشنری درون لیست در نظر گرفته شده درون کلاس append می کند و برای پیدا کردن مقادیر مختلف از توابعی استفاده می کند که قبل تر این توضیحشان را داده ایم ، مانند findParents و ...

استفاده از **Listener Throw , Throwby** در فایل **main.py** :

```
def addThrows_TrowsByRefs(self, ref_dicts, file_ent,
file_address,id1,id2,Throw):
    for ref_dict in ref_dicts:

        scope =
EntityModel.get_or_create(_kind=self.findKindWithKeywords("Method",
ref_dict["scopemodifiers"]),
                                          _name=ref_dict["scopename"],
                                          _parent= ref_dict["scope_parent"]
if ref_dict["scope_parent"] is not None else file_ent,

_longname=ref_dict["scopelongname"],

_contents=ref_dict["scopecontent"])[0]
```

در این قسمت فقط اطلاعات را درون دیتابیس ادد می کنیم.