به نام خدا

نام: فاطمه زهرا بخشنده **استاد: دکتر سعید پارسا**

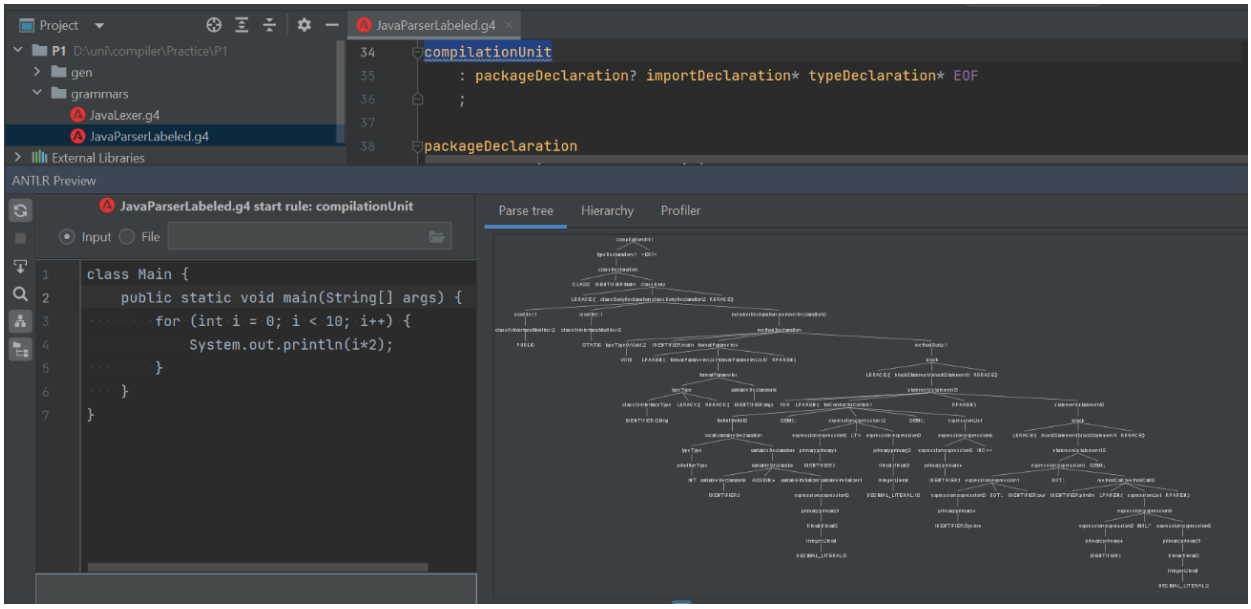
شماره دانشجویی: 98522157 **درس اصول طراحی کامپایلر**

گزارش تمرین 1:

پس از نصب و راه اندازی انتلر، گرامر های جاوا را در پوشه grammars اضافه کردم و فایل های lexer و مابقی فایل های ساخته شده از گرامر نیز در پوشه gen موجود است.

سوال اول:

فایل JavaParserLabeled.g4 را باز می‌کنیم. در خطی که گرامر شروع می‌شود (compilationUnit) کلیک راست می‌کنیم. گزینه Test Rule compilationUnit را می‌زنیم. قسمت ANTLR Preview را باز می‌کنیم. کد جاوا را وارد می‌کنیم. درخت تجزیه به صورت زیر است.



تصویر واضح درخت تجزیه نیز با نام Q1parseTree.png در فایل zip موجود است.

سوال دوم:

کد این سوال در Q2.py قرار دارد. ابتدا یک کلاس به نام FindMethods ایجاد می‌کنیم که از کلاس JavaParserLabeledListener ارث بری می‌کند. در کلاس ایجاد شده یک لیست به نام methods داریم که وظیفه نگه داری نام توابع را دارد. همچنین از یک متغیر به نام class_name استفاده شده که وظیفه نگه داری نام کلاس را دارد. لیست classes_tuple نیز وظیفه نگه داری زوج مرتب ها را دارد که عضو اول آن ها نام کلاس و عضو دوم آن ها لیستی از نام توابع موجود در آن کلاس است. تابع enterMethodDeclaration را در این کلاس Override می‌کنیم. کانتکست آن را MethodDeclarationContext در نظر می‌گیریم و هرگاه که وارد حوزه تعریف یک تابع شدیم نام آن تابع (IDENTIFIER) را درون لیست methods اضافه می‌کنیم. تابع enterClassDeclaration را در این کلاس Override می‌کنیم. کانتکست آن را ClassDeclarationContext در نظر می‌گیریم و هرگاه که وارد حوزه تعریف یک کلاس شدیم نام آن تابع (IDENTIFIER) را در class_name نگه می‌داریم. تابع exitClassDeclaration را در این کلاس Override می‌کنیم. کانتکست آن را ClassDeclarationContext در نظر می‌گیریم و هرگاه که از حوزه تعریف یک کلاس خارج شدیم زوج مرتب (class_name, methods) را به classes_tuple اضافه کرده و methods را خالی می‌کنیم.

```
class FindMethods(JavaParserLabeledListener):
    def __init__(self):
        self.methods = []
        self.class_name = ""
        self.classes_tuple = []

    def enterMethodDeclaration(self, ctx: JavaParserLabeled.MethodDeclarationContext):
        self.methods.append(str(ctx.IDENTIFIER()))

    def enterClassDeclaration(self, ctx: JavaParserLabeled.ClassDeclarationContext):
        self.class_name = str(ctx.IDENTIFIER())

    def exitClassDeclaration(self, ctx: JavaParserLabeled.ClassDeclarationContext):
        self.classes_tuple.append((self.class_name, self.methods))
        self.methods = []
```

یک تابع به نام extract_methods ایجاد می‌کنیم که با ورودی گرفتن مسیر فایل جاوا و ایجاد Listener، Lexer، Parser و انجام عملیات Walk نام کلاس ها و توابع آن را استخراج می‌کند. خروجی این تابع مجموعه ای از زوج مرتب ها هستند. عضو اول این زوج مرتب ها نام کلاس است و عضو دوم آن لیستی از نام توابع موجود در آن کلاس می‌باشد.

```python
def extract_methods(file_path):

    if file_path.split('.')[-1] == 'java':
        stream = FileStream(r""+file_path, encoding="utf8")

        lexer = JavaLexer(stream)
        token_stream = CommonTokenStream(lexer)

        parser = JavaParserLabeled(token_stream)
        tree = parser.compilationUnit()

        listener = FindMethods()
        walker = ParseTreeWalker()

        walker.walk(listener, tree)

    return listener.classes_tuple
```

با استفاده از argparse برای اسکریپت پایتون help هم ایجاد شده است.

```
PS D:\uni\compiler\assignments\HW1\HW1_98522157> python Q2.py -h
usage: Q2.py [-h] [-n FILE]

Java class methods extractor

optional arguments:
  -h, --help            show this help message and exit
  -n FILE, --file FILE  Java file path
```

در نهایت در خروجی نام کلاس ها به همراه نام توابع موجود در آن کلاس ها را خواهیم داشت.

```
PS D:\uni\compiler\assignments\HW1\HW1_98522157> python Q2.py -n Game.java
[('Piece', ['move']), ('Castle', ['move', 'move0', 'move1', 'move2']), ('Minister', ['move', 'moveCastle0', 'moveCastle1',
'moveCastle2', 'moveElephant0', 'moveElephant1']), ('Game', ['setPieces', 'choosePiece', 'collision', 'win'])]
```

سوال سوم:

کد این سوال در Q3.py قرار دارد. با وارد کردن یک کد در ANTLR Preview و چک کردن Parse tree و Hireachy آن متوجه شدم عملیات ریاضی مربوط به expression9 و expression10 هستند. سپس با چک کردن فایل JavaParserLabeled متوجه شدم عملیات ضرب و تقسیم مربوط به expression9 و عملیات جمع و تفریق مربوط به expression10 هستند.

```
| expression bop=('*'|'/'|'%') expression #expression9
| expression bop=('+'|'-') expression #expression10
```

ابتدا یک کلاس به نام CountOperations ایجاد می‌کنیم که از کلاس JavaParserLabeledListener ارث بری می‌کند. در کلاس ایجاد شده یک دیکشنری به نام operations داریم که وظیفه نگه داری operation ها به همراه تعدادشان را دارد. همچنین از یک متغیر به نام op_count استفاده شده که وظیفه نگه داری تعداد کل operation های برنامه را دارد. تابع enterExpression10 را در این کلاس Override می‌کنیم.
کانتکست آن را Expression10Context در نظر می‌گیریم و هرگاه که وارد یک expression10 شدیم اگر عملیات SUB داشتیم به تعداد (-) ها در دیکشنری یکی اضافه می کنیم در غیر این صورت به تعداد (+) ها در دیکشنری یکی اضافه می کنیم. به تعداد کل op_count هم یکی اضافه می‌شود.
تابع enterExpression9 را در این کلاس Override می‌کنیم. کانتکست آن را Expression9Context در نظر می‌گیریم و هرگاه که وارد یک expression9 شدیم اگر عملیات MUL داشتیم به تعداد (*) ها در دیکشنری یکی اضافه می کنیم و اگر عملیات DIV داشتیم به تعداد (/) ها در دیکشنری یکی اضافه می کنیم. در هر دو صورت به تعداد کل op_count هم یکی اضافه می‌شود.

```python
class CountOperations(JavaParserLabeledListener):
    def __init__(self):
        self.op_count = 0
        self.operations = {}

    def enterExpression10(self, ctx:JavaParserLabeled.Expression10Context):
        if ctx.SUB():
            self.operations['-'] = self.operations.get('-', 0) + 1
        else:
            self.operations['+'] = self.operations.get('+', 0) + 1
        self.op_count += 1

    def enterExpression9(self, ctx:JavaParserLabeled.Expression9Context):
        if ctx.MUL():
            self.operations['*'] = self.operations.get('*', 0) + 1
            self.op_count += 1
        elif ctx.DIV():
            self.operations['/'] = self.operations.get('/', 0) + 1
            self.op_count += 1
```

برای این سوال هم یک تابع ایجاد می‌کنیم که با ورودی گرفتن مسیر فایل جاوا و ایجاد Listener، Lexer، Parser و انجام عملیات Walk تعداد هر عملیات اصلی ریاضی و تعداد کل عملیات ریاضی در برنامه را خروجی می‌دهد.

```
def extract_operations(file_path):

    if file_path.split('.')[-1] == 'java':
        stream = FileStream(r""+file_path, encoding="utf8")

        lexer = JavaLexer(stream)
        token_stream = CommonTokenStream(lexer)

        parser = JavaParserLabeled(token_stream)
        tree = parser.compilationUnit()

        listener = CountOperations()
        walker = ParseTreeWalker()

        walker.walk(listener, tree)

    return listener.operations, listener.op_count
```

با استفاده از argparse برای اسکریپت پایتون help هم ایجاد شده است.

```
PS D:\uni\compiler\assignments\HW1\HW1_98522157> python Q3.py -h
usage: Q3.py [-h] [-n FILE]

Java operation counter

optional arguments:
  -h, --help            show this help message and exit
  -n FILE, --file FILE  Java file path
```

خروجی برنامه:

```
PS D:\uni\compiler\assignments\HW1\HW1_98522157> python Q3.py -n Game.java
operations: {'*': 4, '+': 4, '/': 3, '-': 1}
all operations count: 12
```

سوال چهارم:

برای این سوال ابتدا این لینک را چک کردم. طبق این منبع، زبان COW، 12 تا instruction دارد که به صورت زیر هستند:

Instruction	Description
moo	This command is connected to the MOO command. When encountered during normal execution, it searches the program code in reverse looking for a matching MOO command and begins executing again starting from the found MOO command. When searching, it skips the instruction that is immediately before it (see MOO).
mOo	Moves current memory position back one block.
moO	Moves current memory position forward one block.
mOO	Execute value in current memory block as if it were an instruction. The command executed is based on the instruction code value (for example, if the current memory block contains a 2, then the moO command is executed). An invalid command exits the running program. Value 3 is invalid as it would cause an infinite loop.
Moo	If current memory block has a 0 in it, read a single ASCII character from STDIN and store it in the current memory block. If the current memory block is not 0, then print the ASCII character that corresponds to the value in the current memory block to STDOUT.
MOo	Decrement current memory block value by 1.
MoO	Increment current memory block value by 1.
MOO	If current memory block value is 0, skip next command and resume execution after the next matching moo command. If current memory block value is not 0, then continue with next command. Note that the fact that it skips the command immediately following it has interesting ramifications for where the matching moo command really is. For example, the following will match the second and not the first moo: OOO MOO moo moo
OOO	Set current memory block value to 0.
MMM	If no current value in register, copy current memory block value. If there is a value in the register, then paste that value into the current memory block and clear the register.
OOM	Print value of current memory block to STDOUT as an integer.
oom	Read an integer from STDIN and put it into the current memory block.

8 حالت مختلف capitalization کلمه moo و 4 کلمه استثنای MMM و OOO و oom و OOM.

نمونه کد برای تولید دنباله فیبوناتچی با زمان COW:

```
MoO moO MoO mOo MOO OOM MMM moO moO
MMM mOo mOo moO MMM mOo MMM moO moO
MOO MOo mOo MoO moO moo mOo mOo moo
```

نمونه کد بعدی:

```
1  moOMoOMoOMoOMoOmoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMMMmoOMMMMoOMoOMoOMoOMoOMoO
2  MoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMMMmoOMMMommMoOMoOMoOMoOMoO
3  MoOMoOMoOMoOMoOMoOMoOMMMmoOMMMMoOMoOMMMmoOMMMMoOMoOMoOMoOMoOMoOMoOMoOMoOMoO
4  MoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoO
5  MoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMoOMMMmoOMMMMoOMMMmOomOomOomOo
6  mOomOoMMMMoOmoOmoOmoOmoOmoOmoOMMMmoOMMMMoOMoOMMMmoOMMMMoOMMMmoOMMMMoOMMMmoO
7  MMMMoOMoOMMMmoOMMMMoOMMMmoOMMMMoOMoOMMMmoOMMMMoOMMMmoOMMMMoOMoOMMMmoOMMMMoO
8  MMMmoOMMMMoOMMMmoOMMMMoOMoOMMMmoOMMMMoOMMMmoOMMMMoOMMMmoOMMMMoOMMMmoOMMMMoO
9  MoOmOomOomOomOomOomOomOomOomOomOomOomOomOomOomOomOomOomOomOomOomOoMOOOOMmoO
10 moOmoOmoOmoOmoOMoomoOmoOmoOmoOmoOmoOmoOmoOmoOMoomoOmoOmoOmoOmoOMooMoomOomOomOo
11 mOomOomOoMoomOomOomOomOomOoMoomOomOomOomOomOomOomOomOoMMMMOoMOOmoOmoOmoOmoO
12 moOmoOmoOmoOmoOmoOmoOmoOmoOmoOmoOmoOmoOmoOmoOMoomOomOomOomOomOomOomOomOomOo
```

از روی این کد متوجه می‌شویم علاوه بر instruction ها باید Whitespace و Newline هم در گرامر در نظر بگیریم. (هم با whitespace هم بدون آن می توان به این زبان کد زد.)

کد گرامر COW در فایل Cow.g4 در پوشه grammars موجود است. همچنین فایل های lexer و مابقی فایل های ساخته شده از این گرامر نیز در پوشه gen موجود است.

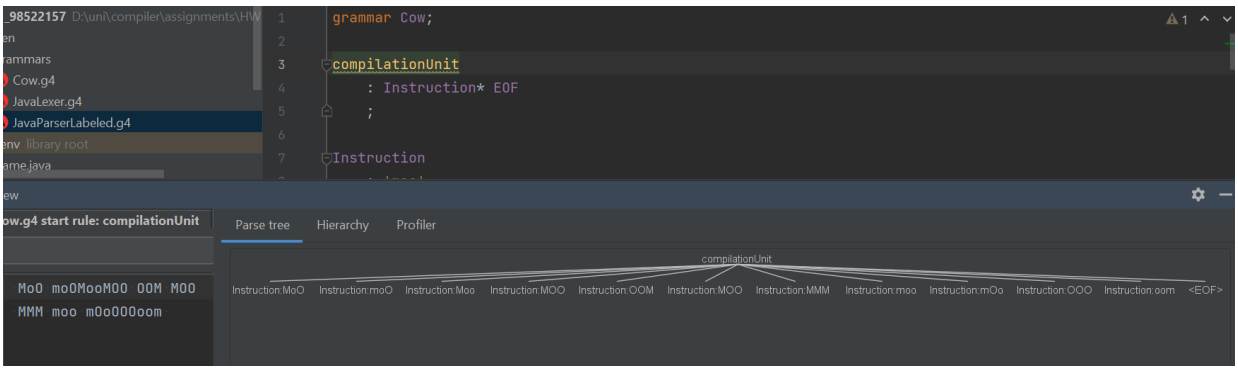
```
grammar Cow;

compilationUnit
    : Instruction* EOF
    ;

Instruction
    : 'moo'
    | 'moO'
    | 'mOo'
    | 'mOO'
    | 'Moo'
    | 'MoO'
    | 'MOo'
    | 'MOO'
    | 'OOO'
    | 'MMM'
    | 'OOM'
    | 'oom'
    ;

Newline: '\n' -> skip;
Whitespace: [ \t\r]+ -> channel(HIDDEN);
```

با استفاده از این گرامر درخت تجزیه را برای یک کد به زبان COW می کشیم. این کد شامل Newline و دستور هایی با whitespace و بدون whitespace است. و این گرامر هر دو نوع را تشخیص می‌دهد.



تصویر واضح درخت تجزیه نیز با نام Q4parseTree.png در فایل zip موجود است.