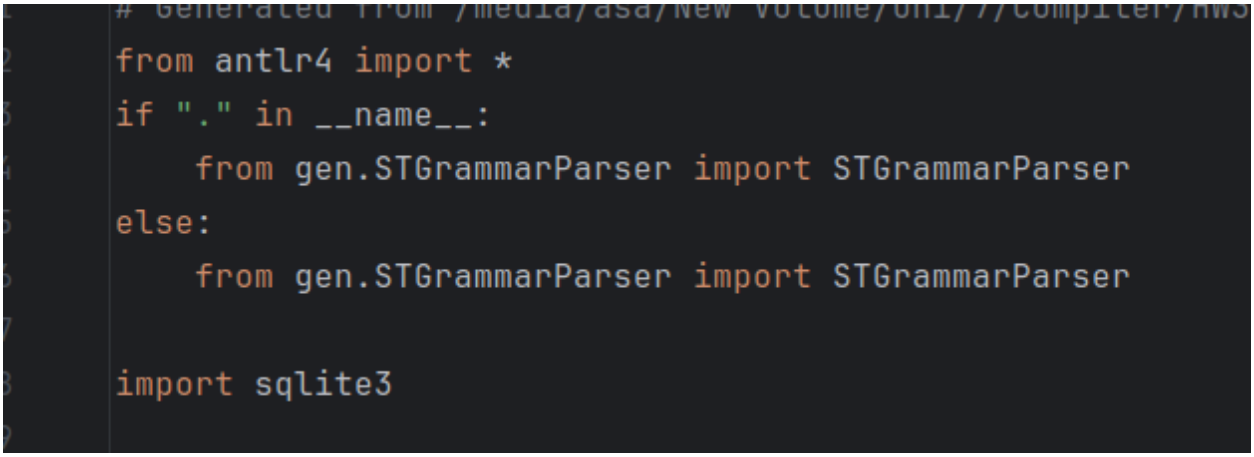
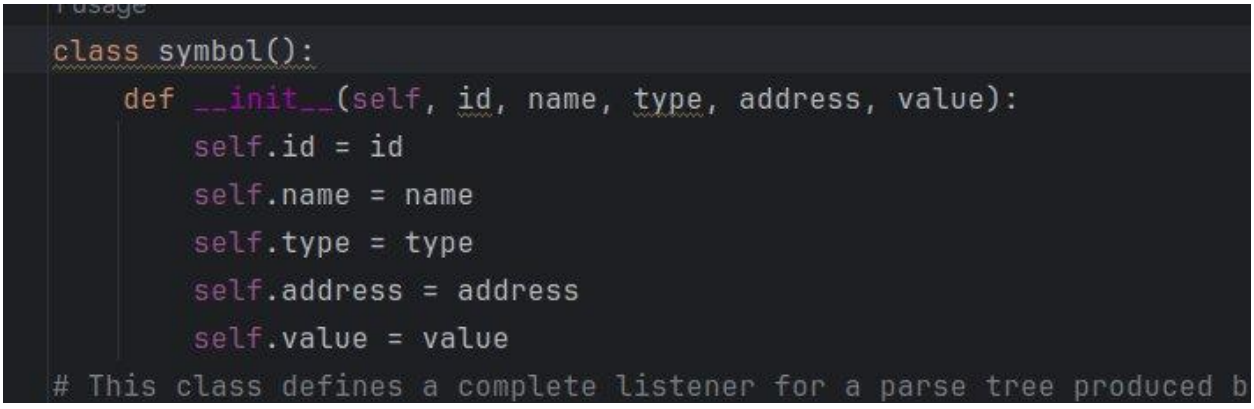
به نام خدا

import



در این بخش، ما از بسته‌های مختلفی از جمله antlr4 و sqlite3 استفاده می‌کنیم. همچنین، بسته‌های تولید شده توسط ANTLR نیز import می‌شوند.

Symbol class



این کلاس symbol یک مدل برای نمایش عناصر جدول نماد است. هر عنصر جدول نماد شامل شناسه، نام، نوع، آدرس و مقدار است.

MySTGrammarListener class

```
class MySTGrammarListener(ParseTreeListener):

    def __init__(self):
        self.symbol_table = {}
        self.connection = sqlite3.connect("Symbols.db")
        self.cursor = self.connection.cursor()
        self.cursor.execute("""CREATE TABLE IF NOT EXISTS symbol (
                            id integer,
                            name TEXT,
                            type TEXT,
                            address integer,
                            value TEXT
                            )""")
        self.connection.commit()
```

این کلاس یک لیسنر است که برای پردازش درخت تجزیه و تحلیل متن تولید شده توسط ANTLR برای زبان STGrammar استفاده می‌شود.

__init__: در این متد، یک اتصال به پایگاه داده SQLite برقرار می‌شود و یک جدول به نام symbol ایجاد می‌شود.

در متد __init__، یک اتصال به پایگاه داده SQLite برقرار می‌شود و یک جدول به نام symbol ایجاد می‌شود.

exitAssigns

```
def exitAssigns(self, ctx:STGrammarParser.AssignsContext):
    self.cursor.execute("SELECT * FROM symbol")
    rows = self.cursor.fetchall()
    for row in rows:
        print(row)
    self.connection.close()
```

این متد از جدول symbol اطلاعات را بازیابی کرده و نمایش می‌دهد

exitAssign

```
def exitAssign(self, ctx:STGrammarParser.AssignContext):
    identifier = ctx.getChild(0).getText()
    result = ctx.getChild(2)
    if identifier not in self.symbol_table:
        self.symbol_table[identifier] = symbol( id: None, identifier,  type: None, id(identifier), value: 0)
    self.symbol_table[identifier].type = result.rule_type
    self.symbol_table[identifier].value = result.value
    self.cursor.execute("SELECT COUNT(*) FROM symbol")
    num = self.cursor.fetchall()[0][0]
    self.symbol_table[identifier].id = int(num)
    self.cursor.execute("SELECT * FROM symbol WHERE name = '{}' LIMIT 1".format(identifier))
    items = self.cursor.fetchall()
    if (len(items) == 0):
        try:
            # Executing the SQL command and commit
            _id = self.symbol_table[identifier].id
            name = self.symbol_table[identifier].name
            type = self.symbol_table[identifier].type
            address = self.symbol_table[identifier].address
            value = str(self.symbol_table[identifier].value)
            params = (_id, name, type, address, value)
            self.cursor.execute( _sql: f"INSERT INTO symbol VALUES (?, ?, ?, ?, ?)", params)
            self.connection.commit()
            print("Data inserted")

        except:
            # Rolling back in case of error
            pass
    else:
        self.cursor.execute( _sql: f"UPDATE symbol SET value = ?, type =  ? WHERE name = ?",  _parameters: (str(self.sy
        self.connection.commit()
```

این متد وظیفه اضافه کردن یا به‌روزرسانی عناصر جدول نماد را دارد. اگر عنصر موجود نباشد، آن را در جدول اضافه می‌کند و در غیر این صورت مقادیر آن را به‌روز می‌کند.

exitExpr

```
def exitExpr(self, ctx:STGrammarParser.ExprContext):
    if ctx.getChildCount() == 1:
        ctx.value = ctx.getChild(0).value
        ctx.rule_type = ctx.getChild(0).rule_type
    else:
        op = ctx.getChild(1).getText()
        if op == '+':
            left_child_type = ctx.getChild(0).rule_type
            right_child_type = ctx.getChild(2).rule_type
            if left_child_type == "String":
                if right_child_type in ["Integer", "Float"]:
                    ctx.rule_type = "String"
                    ctx.value = ctx.getChild(0).value + str(ctx.getChild(2).value)
            elif left_child_type in ["Integer", "Float"]:
                if left_child_type == "Integer" and right_child_type == "Integer":
                    ctx.rule_type = "Integer"
                    ctx.value = ctx.getChild(0).value + ctx.getChild(2).value
                else:
                    match right_child_type:
                        case "Integer" | "Float":
                            ctx.rule_type = "Float"
                            ctx.value = ctx.getChild(0).value + ctx.getChild(2).value
                        case "String":
                            ctx.rule_type = "Error"
                            ctx.value = ""
            else:
                ctx.rule_type = "Float"
                ctx.value = ctx.getChild(0).value + ctx.getChild(2).value
        else:
```

```
def exitExpr(self, ctx:STGrammarParser.ExprContext):
                ctx.value = ctx.getChild(0).value + ctx.getChild(2).value
        else:
            left_child_type = ctx.getChild(0).rule_type
            right_child_type = ctx.getChild(2).rule_type
            if left_child_type == "String" or right_child_type == "String":
                ctx.rule_type = "Error"
                ctx.value = ""
            elif left_child_type in ["Integer", "Float"]:
                if left_child_type == "Integer" and right_child_type == "Integer":
                    ctx.rule_type = "Integer"
                    ctx.value = ctx.getChild(0).value - ctx.getChild(2).value
                else:
                    ctx.rule_type = "Float"
                    ctx.value = ctx.getChild(0).value - ctx.getChild(2).value
            else:
                ctx.rule_type = "Float"
                ctx.value = ctx.getChild(0).value - ctx.getChild(2).value

# Exit a parse tree produced by STGrammarParser#term
```

این متد عملگرهای عبارت ریاضی را پردازش می‌کند و نتیجه و نوع عبارت را محاسبه می‌کند.

exitTerm

```python
# Exit a parse tree produced by STGrammarParser#term.
def exitTerm(self, ctx:STGrammarParser.TermContext):
    if ctx.getChildCount() == 1:
        ctx.value = ctx.getChild(0).value
        ctx.rule_type = ctx.getChild(0).rule_type
    else:
        op = ctx.getChild(1).getText()
        if op == '*':
            left_child_type = ctx.getChild(0).rule_type
            right_child_type = ctx.getChild(2).rule_type
            if left_child_type == "String" or right_child_type == "String":
                ctx.value = ""
                ctx.rule_type = "Error"
            else:
                if left_child_type == "Integer" and right_child_type == "Integer":
                    ctx.rule_type = "Integer"
                else:
                    ctx.rule_type = "Float"
                ctx.value = ctx.getChild(0).value * ctx.getChild(2).value
        else:
            left_child_type = ctx.getChild(0).rule_type
            right_child_type = ctx.getChild(2).rule_type
            if left_child_type == "String" or right_child_type == "String":
                ctx.value = ""
                ctx.rule_type = "Error"
            else:
                ctx.rule_type = "Float"
                ctx.value = ctx.getChild(0).value / ctx.getChild(2).value
```

این متد همانند exitExpr عملگرهای عبارت ریاضی را پردازش می‌کند، اما برای عملگرهای ضرب و تقسیم مورد استفاده قرار می‌گیرد.

exitFactor_is_string، exitFactor_is_integer، exitFactor_is_float، exitFactor_is_expression، exitFactor_is_id

---

```python
# Exit a parse tree produced by STGrammarParser#factor_is_string.S
def exitFactor_is_string(self, ctx:STGrammarParser.Factor_is_stringContext):
    ctx.rule_type = "String"
    ctx.value = ctx.getText()
# Exit a parse tree produced by STGrammarParser#factor_is_integer.

def exitFactor_is_integer(self, ctx:STGrammarParser.Factor_is_integerContext):
    ctx.rule_type = "Integer"
    ctx.value = int(ctx.getText())

# Exit a parse tree produced by STGrammarParser#factor_is_float.

def exitFactor_is_float(self, ctx:STGrammarParser.Factor_is_floatContext):
    ctx.rule_type = "Float"
    ctx.value = float(ctx.getText())

# Exit a parse tree produced by STGrammarParser#factor_is_expression.

def exitFactor_is_expression(self, ctx:STGrammarParser.Factor_is_expressionContext):
    ctx.rule_type = ctx.getChild(1).rule_type
    ctx.value = ctx.getChild(1).value
```

این متدها مقادیر مختلف را مانند رشته، عدد صحیح، عدد اعشاری و ... تجزیه و تحلیل می‌کنند.

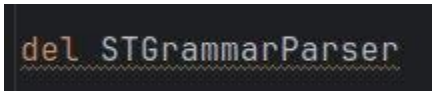
```python
# Exit a parse tree produced by STGrammarParser#factor_is_id.
def exitFactor_is_id(self, ctx:STGrammarParser.Factor_is_idContext):
    identifier = ctx.getText()
    self.cursor.execute("SELECT * FROM symbol WHERE name = '{}' LIMIT 1".format(identifier))
    items = self.cursor.fetchall()
    if len(items) != 0:
        ctx.rule_type = items[0][2]
        match ctx.rule_type:
            case "Integer":
                ctx.value = int(items[0][4])
            case "Float":
                ctx.value = float(items[0][4])
            case _:
                ctx.value = items[0][4]

    else:
        ctx.rule_type = "ERROR"
        ctx.value = ""

# Exit a parse tree produced by STGrammarParser#sign.
def exitSign(self, ctx:STGrammarParser.SignContext):
    ctx.value = ctx.getText()
```

exitSign

```python
# Exit a parse tree produced by STGrammarParser#sign.
def exitSign(self, ctx:STGrammarParser.SignContext):
    ctx.value = ctx.getText()
```

این متد علامت‌های منفی یا مثبت را پردازش می‌کند.

del STGrammarParser



این دستور آخر کد به صورت متمرکز، کلاس STGrammarParser را حذف می‌کند تا حافظه را آزاد کند.

علاوه بر تجزیه و تحلیل اعلان‌ها و اختصاص‌ها، عملیات ریاضی از جمله جمع، تفریق، ضرب و تقسیم را نیز پردازش می‌کند و اطلاعات نمادها را در یک پایگاه داده SQLite ذخیره می‌کند.

علی شیخ عطار