

به نام خدا

نام استاد: دکتر سعید پارسا

نام درس: کامپایلر

نام و نام خانوادگی: نیوشا یقینی

شماره دانشجویی: 98522346

گزارش تمرین: HW3

تاریخ: 1403/02/07

هدف تمرین:

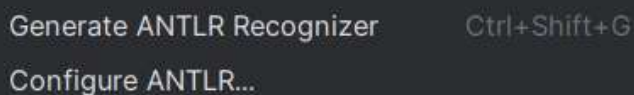
هدف این تمرین این است که داده های ورودی را در قالب جدول خصوصیات آن ها ذخیره کرده و نمایش دهیم. مانند جدول زیر:

ID	Name	Type	Value
1	X	Int	5
2	Y	Float	6.2
...

در این گزارش ابتدا مراحل پیاده سازی آورده شده و در انتها نمونه های مثال چک شده و نتیجه گیری آورده شده است.

مراحل:

1. ابتدا با استفاده از 2 گزینه زیر فولدر gen را ساختیم.



2. سپس با توجه به تمرین سری قبل فایل پایتون Listener را ایجاد کردیم و توابع exit فایل STGrammarListener

از فولدر gen را داخل آن کپی کردیم.

3. مهم ترین نکته برای شروع این است که بتوانیم چند تا ورودی دریافت کنیم، از این رو فایل main.py را به صورت زیر پیاده

سازی می کنیم:

```
from STListener import STGrammarListener
from gen.STGrammarLexer import STGrammarLexer
from gen.STGrammarParser import STGrammarParser

print("Enter an arithmetic expression (or type 'exit' to quit): ")
all_inputs = []

while True:
    new_input = input()

    # Check if the user wants to exit
    if new_input.lower() == 'exit':
        break

    all_inputs.append(new_input)

separator = '\n'
input_expression = separator.join(all_inputs)

# Parse and evaluate the expression
lexer = STGrammarLexer(InputStream(input_expression))
token_stream = CommonTokenStream(lexer)
parser = STGrammarParser(token_stream)
parse_tree = parser.start()
listener = STGrammarListener()
walker = ParseTreeWalker()
walker.walk(listener, parse_tree)
result = listener.result
print("Result :", result)
```

- برای اینکه بتوان چند تا ورودی دریافت کرد، باید لیستی از تمام ورودی ها ایجاد کرد و در نهایت آنها را بصورت concatenate شده به listener داد، که در اینجا ما با استفاده از دستور join آن را انجام داده‌ایم.
- نکته بعدی آن است که باید یکجا این ورودی گرفتن تمام شود، و شرط خاتمه را وارد کردن exit از طرف کاربر گذاشته‌ایم.
- Separator های مختلف من جمله '|' استفاده شد، اما هیچکدام قابل separate شدن در داخل listener نبود، برای همین بهترین گزینه استفاده از '\n' به عنوان separator بود.

4. در مرحله بعدی وارد پیاده سازی با توجه به جدولی که باید پر کنیم property های زیر را در تابع initiate اضافه کردیم:

```
class STGrammarListener(STGrammarListener):
    def __init__(self):
        self.result = []
        self.id_counter = 1
        self.name = ''
        self.current_type = ''
        self.current_value = 0
        self.all_inputs = []
        self.stack_values = []
        self.stack_types = []
        self.db_connection = sqlite3.connect('symbol_table.db')
        self.db_cursor = self.db_connection.cursor()
```

- id_counter: این متغیر برای رصد کردن id ورودی هاست.
- Name: این متغیر برای ذخیره کردن نام variable ورودی استفاده می‌شود.
- Current_type: این متغیر برای ذخیره کردن نوع داده هر variable ورودی استفاده می‌شود.
- Current_value: این متغیر برای ذخیره کردن مقدار هر variable ورودی استفاده می‌شود.
- All_inputs: این آرایه برای ذخیره کردن هر ردیف از table نهایی استفاده می‌شود.
- Stack_values: این پشته برای ذخیره داده های ورودی در پرانتزهای تو در تو تعبیه شده که ترتیب آنها را چک کرده و حفظ کند.
- Stack_types: این پشته برای ذخیره نوع داده های ورودی در پرانتز های تو در تو تعبیه شده که بتواند با توجه به ترتیب، نوع داده ها را چک کند و در صورت لزوم ارور مناسب را برگرداند.
- دو مورد آخر جهت ذخیره جدول داده ها در sqlite هستند.

5. در مرحله بعد برای پیاده سازی از داخلی ترین توابع شروع می‌کنیم.

• `exitFactor_is_integer`:

زمانی وارد این فانکشن می‌شویم که داده ای از جنس `integer` به عنوان ورودی دریافت کرده‌ایم، اما نمی‌دانیم آیا این داده یک داده ای از نوع `signed` است یا `unsigned` بنابراین با چک کردن تعداد فرزندان `ctx` این را می‌فهمیم و اگر بیشتر از 1 بود یعنی `signed` است. برای داده های `sign` با توجه به تعریفی که در تابع `exitSign` داریم `current_value` را مثبت یا منفی می‌کنیم که اینجا با ضرب داده در `current_value`، قطب عدد مشخص می‌شود. اگر تعداد فرزندان کمتر از 1 بود یعنی عدد ورودی `unsiged` است، و دیگر تابع `exitSign` صدا زده نشده است، برای همین صرفاً آن را در `current_value` می‌ریزیم. در اینجا بسیار حائز اهمیت است که با استفاده از `getText` خود داده ها را در `stack` ها ذخیره کنیم، چرا که بعداً به تداخل خواهیم خورد. (تداخل: به علت وجود پرانتزها و محاسبه مقادیر داخل آنها، بعد از اضافه شدن شان به `stack` دیگر از جنس `node` نیستند بنابراین هنگامی که می‌خواهیم برابری آنها را که در فانکشن های بعدی مورد نیاز است، چک کنیم، هنگام چک کردن `id` های تکراری، برابری آنها تایید نمی‌شود.) در انتها نیز تایپ داده را در `property` مربوطه که `current_type` است می‌ریزیم. توجه شود که از آنجایی که این تابع یک تابعی است که ورودی ها را دریافت می‌کند، آن ها را در پشته های مربوطه مقدار و نوع ذخیره می‌کنیم.

```
# Exit a parse tree produced by STGrammarParser#factor_is_integer.
def exitFactor_is_integer(self, ctx):
    # print("exitFactor_is_integer")

    # ctx.getChildCount()>1 means we have sign number
    if(ctx.getChildCount()>1):
        # print("child count: ", ctx.getChildCount())
        # print("child[1]: ", ctx.getChild(1))
        self.current_value = int(ctx.getChild(1).getText())*self.current_value
        # print("current value: ", self.current_value)
    else:
        # print("child[0]: ", ctx.getChild(0))
        self.current_value = ctx.getChild(0).getText()
        # print("current value: ", self.current_value)

    self.current_type = 'Integer'
    self.stack_values.append(self.current_value)
    self.stack_types.append(self.current_type)
    # print("stack_values: ", self.stack_values)
    # print("stack_types: ", self.stack_types)
    pass
```

- **:exitFactor_is_float**

زمانی وارد این فانکشن می‌شویم که داده ای از جنس float به عنوان ورودی دریافت کرده‌ایم، در این تابع نیز مشابه حالت integer رفتار می‌کنیم.

```
# Exit a parse tree produced by STGrammarParser#factor_is_float.
def exitFactor_is_float(self, ctx):
    # print("exitFactor_is_float")

    # ctx.getChildCount()>1 means we have sign number
    if(ctx.getChildCount()>1):
        self.current_value = float(ctx.getChild(1).getText())*self.current_value
    else:
        self.current_value = ctx.getChild(0).getText()

    self.current_type = 'Float'
    self.stack_values.append(self.current_value)
    self.stack_types.append(self.current_type)
    pass
```

- **:exitFactor_is_string**

زمانی وارد این فانکشن می‌شویم که داده ای از جنس string به عنوان ورودی دریافت کرده‌ایم، در این تابع صرفاً کافیسست داده های مربوط به مشخصه های ورودی را در property های مناسب ذخیره کنیم.

```
# Exit a parse tree produced by STGrammarParser#factor_is_string.
def exitFactor_is_string(self, ctx):
    self.current_value = ctx.getChild(0).getText()
    self.current_type = 'String'
    self.stack_values.append(self.current_value)
    self.stack_types.append(self.current_type)
    # print("stack_values: ", self.stack_values)
    # print("stack_types: ", self.stack_types)
    pass
```

- **:exitSign**

زمانی وارد این فانکشن می‌شویم که داده ی ورودی ما از نوع عددی ست و sign دارد. برای مشخص شدن آن چک می‌کنیم که sign مثبت است یا منفی، سپس برای مثبت عدد 1 و برای منفی عدد -1 را در current_value قرار می‌دهیم، این کار انجام می‌شود تا در مرحله بعد که عدد گرفته شد در آن ضرب شود و مثبت یا منفی بودن آن اعمال شود.

```
# Exit a parse tree produced by STGrammarParser#sign.
def exitSign(self, ctx):
    # print("exitSign")
    # print(ctx.getChild(0))

    if(ctx.getChild(0).getText()=="+" ):
        self.current_value = 1
    elif(ctx.getChild(0).getText()=="-"):
        self.current_value = -1
    # print("current value: ", self.current_value)

    pass
```

• :exitFactor_is_expression

زمانی که داخل پرانتزهای تو در تو باشیم این تابع صدا زده میشود و ctx آن دارای 3 فرزند است، که اولی و آخری پرانتزها هستند، در این تابع تغییری اعمال نمی‌کنیم چرا که هندل کردن اعداد با پشته ها اعمال شده است.

```
# Exit a parse tree produced by STGrammarParser#factor_is_expression.
def exitFactor_is_expression(self, ctx):
    pass
```

• :exitFactor_is_expression

این تابع زمانی صدا زده می‌شود که ما داخل ورودی هایمان، ورودی ای از جنس id داشته باشیم، در این حالت باید چک شود که آیا این id داده شده واقعا قبلا در ورودی ها گرفته شده یا نه، و اگر گرفته شده، داده آن بازیابی شود و اگر محاسبه ای نیاز است با آن انجام شود. در صورت عدم وجود id مقدار دهی شده از قبل، ارور مناسب برگردانده می‌شود.

```
# Exit a parse tree produced by STGrammarParser#factor_is_id.
def exitFactor_is_id(self, ctx):
    # print("exitFactor_id")
    # print(ctx.getChild(0))
    my_id = ctx.getChild(0)
    valid_id_flag = False
    for i in self.all_inputs:
        if(i[1].getText() == my_id.getText()):
            self.current_type = i[2]
            self.current_value = i[3]
            valid_id_flag = True
            self.stack_values.append(self.current_value)
            self.stack_types.append(self.current_type)
            # print("stack_values: ", self.stack_values)
            # print("stack_types: ", self.stack_types)

    if(valid_id_flag==False):
        print("This id =>", my_id, "does not exist.")
        self.current_value = None
    pass
```

تا اینجا تمام ورودی های مرحله اول گرفته شده و پیاده سازی شده اند، حال به مرحله ی پیاده سازی توابع با سلسله مراتب بالاتر می رویم.

6. در این مرحله ابتدا تابع `exitTerm` و سپس تابع `exitExpr` پیاده سازی می شود، تابع اول مربوط به محاسبات ضرب و تقسیم است و از لحاظ سلسله مراتبی بالاتر است، تابع دوم نیز مسئول محاسبات جمع و تفریق می باشد.

• `exitTerm`:

در این تابع ابتدا `operator` اعمال شده را شناسایی می کنیم، سپس دو مقداری که آخر از همه در `stack` ها ذخیره شده اند را `pop` می کنیم، تایپ این مقادیر و `operator` بدست آمده، تعیین کننده نوع داده بعد از اعمال عملیات است، در ثانی اگر به ارور `type error` برخوردیم در اینجا اعمال شده و مشخص می شود.

```
# Exit a parse tree produced by STGrammarParser#term.
def exitTerm(self, ctx):
    if ctx.getChildCount() == 3: # Check if it's a binary operation
        operator = ctx.getChild(1).getText() # Get the operator

        operand1_value = self.stack_values.pop()
        operand1_type = self.stack_types.pop()
        operand2_value = self.stack_values.pop()
        operand2_type = self.stack_types.pop()
        type_check = self.Term_errorChecker(operand1_type, operand2_type, operator)
```


تابع صدا زده شده در اینجا Term_errorChecker است که برای مشخص کردن نوع داده ی خروجی یا ارور مربوطه در فانکشن exitTerm است. که به صورت زیر شرط گذاری های آن انجام شده است.

```
1 usage
def Term_errorChecker(self, type1, type2, operator):
    if(type1=='Integer' and type2=='Integer' and operator=='*'):
        return 'Integer'
    elif(type1=='Integer' and type2=='Integer' and operator=='/'):
        return 'Float'
    elif(type1=='Integer' and type2=='Float'):
        return 'Float'
    elif(type1=='Float' and type2=='Integer'):
        return 'Float'
    elif(type1=='Float' and type2=='Float'):
        return 'Float'
    else:
        return 'Type Error!'
```

در ادامه تابع exitTerm، در صورت عدم وجود ارور، داده های pop شده را به جنس داده خودشان cast می کنیم و سپس عملیات مورد نیاز را انجام می دهیم. دقت شود که پس از انجام عملیات ها نوع داده خروجی از این عملیات و جنس آن نیز در property های مربوطه جهت عملیات های آتی ذخیره می شود. در انتها نیز در صورت وجود ارور مقدار current_valut، none می شود چرا که برای عملیات بعدی مشخص شود در مرحله قبل ارور داشته ایم، و نوع ارور نیز چاپ می شود.

```
if (type_check != 'Type Error!'):
    if (operand1_type == 'Integer'):
        operand1_value = int(operand1_value)
    elif (operand1_type == 'Float'):
        operand1_value = float(operand1_value)

    if (operand2_type == 'Integer'):
        operand2_value = int(operand2_value)
    elif (operand2_type == 'Float'):
        operand2_value = float(operand2_value)

    if operator == "/":
        self.current_value = operand2_value / operand1_value
        self.stack_values.append(self.current_value)
        self.current_type = type_check
        self.stack_types.append(self.current_type)
    elif operator == "*":
        self.current_value = operand2_value * operand1_value
        self.stack_values.append(self.current_value)
        self.current_type = type_check
        self.stack_types.append(self.current_type)
else:
    print(type_check)
    self.current_value = None
```

pass

- **exitExpr**:

تابع بعدی تابع **exitExpr** که همانطور که اشاره شد عملیات های مربوط به جمع و تفریق را انجام می دهد. این تابع مشابه تابع **exitTerm** پیاده سازی شده است، با این تفاوت که از فانکشن **Expr_errorChecker** برای چک کردن نوع داده خروجی در تابع **exitExpr** و وجود یا عدم وجود **error type** بهره گرفته ایم.

```
1 usage
def Expr_errorChecker(self, type1, type2, operator):
    if(type1=='Integer' and type2=='Integer'):
        return 'Integer'
    elif(type1=='Integer' and type2=='Float'):
        return 'Float'
    elif(type1=='Float' and type2=='Integer'):
        return 'Float'
    elif(type1=='Float' and type2=='Float'):
        return 'Float'
    elif(type1=='String' and type2=='String' and operator=='+'):
        return 'String'
    else:
        return 'Type Error!'
```

7. مرحله بعدی مرحله پیاده سازی **exitAssign** و **exitAssigns** است.

- فانکشن اول، مرحله ای است که ما از یک **expression** خارج شده ایم و حال می خواهیم داده های دریافتی و محاسبه شده را در جدول مربوطه بارگزاری کنیم، این جدول یک لیستی است که در **property** های ما به نام **all_inputs** مقدار دهی اولیه شده است، در اینجا همچنین چک می کنیم که آیا داده ورودی تکراری هست یا نه، این چک هم برای وارد کردن داده ها به **sqlite** نیاز است، هم باید چک کنیم که در صورت تکرار **id** ها و **assign** دو مقدار متفاوت به آنها مقدار نهایی را به آن اطلاق کنیم.
- فانکشن دوم نیز زمانی صدا زده می شود که تمام **expression** ها تمام شده و در انتهای ورودی ها قرار داریم، پس به عبارتی این فانکشن از لحاظ سلسله مراتبی در سلسله مراتب بالاتری قرار دارد.

```
# Exit a parse tree produced by STGrammarParser#assigns.
def exitAssigns(self, ctx):
    pass

# Exit a parse tree produced by STGrammarParser#assign.
def exitAssign(self, ctx):
    self.name = ctx.getChild(0)

    if(self.current_value != None):
        repetition_flag = False
        for i in self.all_inputs:
            if (i[1].getText() == self.name.getText()):
                i[2] = self.current_type
                i[3] = self.current_value
                repetition_flag = True

        if(repetition_flag == False):
            self.all_inputs.append([self.id_counter, self.name, self.current_type, self.current_value])
            self.id_counter += 1

    pass
```

8. مرحله بعدی تابع exitStart است.

- این تابع زمانی صدا زده می شود که کار تمام توابع داخلی به اتمام رسیده باشد، در این تابع ما جدول نهایی را چاپ می کنیم، آن را در فایل txt ذخیره می کنیم، همچنین در sqlite می ریزیم.

```
# Exit a parse tree produced by STGrammarParser#start.
def exitStart(self, ctx):
    # print("exitStart")
    # print(ctx.getChild(0))

    print("ID Name Type Value")
    for i in range(len(self.all_inputs)):
        for j in range(len(self.all_inputs[i])):
            print(self.all_inputs[i][j], end=' ')
        print()

    #text file part (writing to output.txt)
    self.write_in_text_file()

    #sqlite part (Create a table to store the symbol table data)
    self.write_in_sqlite_file()

    self.print_symbol_table_data()

    pass
```

- پیاده سازی توابع استفاده شده در این فانکشن نیز به شرح زیر است:

```
1 usage
def write_in_text_file(self):
    with open("output.txt", "w") as file:
        file.write("ID Name Type Value\n")
        for entry in self.all_inputs:
            file.write(" ".join(map(str, entry)) + "\n")
```

```
1 usage
def write_in_sqlite_file(self):
    # make table
    self.db_cursor.execute('''CREATE TABLE IF NOT EXISTS symbol_table
                               (ID INTEGER PRIMARY KEY,
                                Name TEXT,
                                Type TEXT,
                                Value TEXT)''')

    # Insert symbol table data into the database
    try:
        for entry in self.all_inputs:
            self.db_cursor.execute('''INSERT INTO symbol_table (ID, Name, Type, Value)
                                   VALUES (?, ?, ?, ?)''', entry)

            self.db_connection.commit()
            print("Entry added successfully.")
    except sqlite3.IntegrityError:
        print("Entry is not unique.")

    # Commit changes and close the database connection
    # self.db_connection.commit()
    self.db_connection.close()
```

```
1 usage
def print_symbol_table_data(self):
    try:
        self.db_cursor.execute("SELECT * FROM symbol_table")
        rows = self.db_cursor.fetchall()
        print("ID Name Type Value")
        for row in rows:
            print(row)
    except sqlite3.Error as e:
        print("An error occurred while retrieving data:", e)
```

نمونه اجرا:

در مثال های زیر تمام حالت های موجود چک شده، نمونه ای هم که ارور داشت در جدول آورده نشده و صرفا ارور آن پرینت شده است.

```
Enter an arithmetic expression (or type 'exit' to quit):
a = 5 - 4
b = 6+-2
c = "niu"
d = c + "sha"
e = 18
e = 5
f = e * (b+a)
h = 4.2/2
k = "ttt" + 6
exit
ANTLR runtime and generated code versions disagree: 4.11.1!=4.13.1
ANTLR runtime and generated code versions disagree: 4.11.1!=4.13.1
Type Error!
ID Name Type Value
1 a Integer 1
2 b Integer 4
3 c String "niu"
4 d String "niussha"
5 e Integer 5
6 f Integer 25
7 h Float 2.1

Entry added successfully in sqlite.
Result : []
```

خروجی داخل txt:

main.py		output.txt		STGrammar.g4		STListener.py	
1		1	ID Name Type Value				
2		2	1 a Integer 1				
3		3	2 b Integer 4				
4		4	3 c String "niu"				
5		5	4 d String "niussha"				
6		6	5 e Integer 5				
7		7	6 f Integer 25				
8		8	7 h Float 2.1				
9		9					

نتیجه گیری:

- حالت امتیازی برای unary اعمال شده است.
- جلوی حالت های تکراری گرفته شده است.
- در فایل output.txt جدول نهایی ذخیره می شود.
- در دیتابیس sqlite جدول نهایی ذخیره می شود.