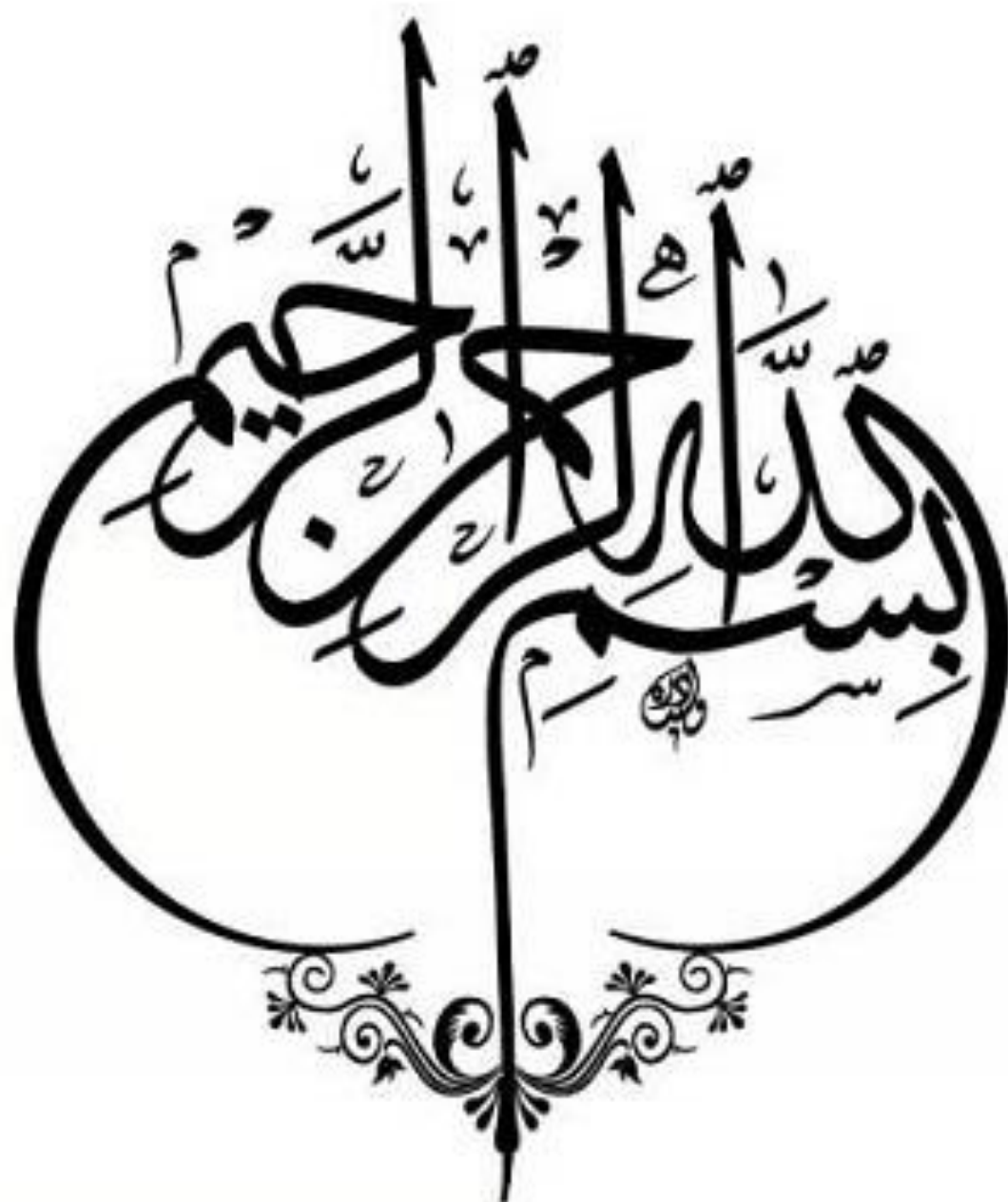


مروری بر گرامرهای مستقل از متن و کاربردهای آن در علوم کامپیوتر

با رویکرد بررسی مثال های کاربردی در زبان های برنامه نویسی

خلاصه

این پروژه تحقیقاتی بر گرامرهای مستقل از متن و کاربردهای آن در علوم کامپیوتر تمرکز دارد. این تحقیق شامل مرور تاریخی، توضیحات گرامرها، کاربردهای نحوی و عملی در زبانهای برنامه نویسی، و ارزیابی نتایج بر اساس تحلیل نحو و پردازش زبانهای طبیعی است.





مروری بر گرامر های مستقل از متن و

کاربرد های آن در علوم کامپیوتر

پروژه کارشناسی - علوم کامپیوتر

استاد راهنما :

دکتر هاشم صابری نجفی

28 آبان ماه سال 1402

گردآورنده :

محمد مهدی شقیقی

فهرست مطالب

5.....	مقدمه
5.....	مقدمه درباره گرامرهای مستقل از متن (CFG):
5.....	اهمیت گرامرهای مستقل از متن در علوم کامپیوتر:
6.....	سازگاری با پروژه تحقیقاتی:
3.....	اهداف پروژه:
3.....	1- معرفی گرامرهای مستقل از متن:
5.....	2- کاربردهای گرامرهای مستقل از متن:
5.....	1- زبان‌های برنامه‌نویسی:
5.....	2- طراحی کامپایلرها:
6.....	3- شناسایی و تحلیل متون:
7.....	4- پیاده‌سازی نمونه:
10.....	مثالی از گرامرهای مستقل از متن
18.....	استفاده از گرامرهای مستقل از متن در کامپایلرها:
24.....	پردازش زبان طبیعی (NLP) و کاربرد
30.....	پیشنهادهای برای تحقیقات آینده:
30.....	1. توسعه گرامرهای مستقل از متن:
31.....	2. کاربردهای گسترده‌تر در پردازش زبان طبیعی:
31.....	3. بهبود عملکرد کامپایلرها:
32.....	4. کاربردهای جدید در هوش مصنوعی:
34.....	نتیجه‌گیری:

منابع پیشنهادی: 0.....

مقدمه

مقدمه درباره گرامرهای مستقل از متن (CFG):

در علوم کامپیوتر، گرامرهای مستقل از متن به عنوان یک ابزار اساسی و بنیادین مورد مطالعه و تحقیق قرار گرفته‌اند. این گرامرها، یک ساختار قوانینی را برای زبان‌ها تعریف می‌کنند که امکان توصیف ساختارهای مختلف در زبان‌های مختلف را فراهم می‌کنند. این اصول تئوریک در زمینه نظریه زبان‌ها و اتوماتاها ریشه داشته و در دهه‌ها به یکی از اصطلاحات بنیادی در زمینه طراحی زبان‌های برنامه‌نویسی و تحلیل ساختارهای زبانی تبدیل شده‌اند.

اهمیت گرامرهای مستقل از متن در علوم کامپیوتر:

تعداد زیادی از زبان‌های برنامه‌نویسی از گرامرهای مستقل از متن به عنوان ابزار اصلی برای تعریف ساختارهای زبانی خود استفاده می‌کنند. این گرامرها در مراحل توسعه نرم‌افزارها و ایجاد کامپایلرها به صورت گسترده به کار گرفته می‌شوند. از این رو، درک عمیق از مبانی و کاربردهای این گرامرها برای توسعه نرم‌افزارهای کارآمد و بهینه بسیار حیاتی است.

سازگاری با پروژه تحقیقاتی:

این پروژه تحقیقاتی با هدف ارائه یک مرور جامع از گرامرهای مستقل از متن و بررسی کاربردهای آن در علوم کامپیوتر آغاز می‌شود. تحلیل مفصلی از اصول این گرامرها، ارتباط آنها با زبان‌های برنامه‌نویسی، و نقش آنها در طراحی کامپایلرها و پردازش زبان طبیعی در این پروژه مورد بررسی قرار می‌گیرد. همچنین، این پروژه با ارائه پیشنهاداتی برای تحقیقات آتی در زمینه‌های بهبود عملکرد گرامرها و کاربردهای جدید در هوش مصنوعی به ادامه راه خود پیش می‌رود.

اهداف پروژه:

1) معرفی گرامرهای مستقل از متن:

(a) توضیح اصول اساسی گرامرهای مستقل از متن و نحوه تشکیل قوانین آنها:

گرامرهای مستقل از متن (CFG) یک مدل ریاضی هستند که برای توصیف ساختارهای زبانی به کار می‌روند. این گرامرها از چهار مؤلفه اصلی تشکیل شده‌اند:

- مجموعه نمادها (ترمینال‌ها و غیر ترمینال‌ها)،
- یک نماد ویژه به نام شروع (شروع کننده)،
- مجموعه قوانین تبدیل (تولید)،
- و یک الگوی خروجی (نمایش خروجی).

در این بخش از پروژه، این مؤلفه‌ها به طور دقیق معرفی و توضیح داده می‌شود. علاوه بر این، نحوه تشکیل قوانین CFG و روش‌های تعریف ساختارهای زبانی با استفاده از این گرامرها نیز بررسی می‌شود.

(b) مقایسه گرامرهای مستقل از متن با دیگر انواع گرامرها :

در این بخش از پروژه، گرامرهای مستقل از متن با دیگر انواع گرامرها مقایسه می‌شوند. این مقایسه شامل مواردی مانند قدرت توصیفی، ابزارهای مورد استفاده برای تحلیل و تولید، و قابلیت‌های خاص هر نوع گرامر است. به عنوان مثال، مقایسه با گرامرهای قابل قبول توسط اتوماتاها و گرامرهای قابل قبول توسط ماشین‌های تورینگ نیز انجام می‌شود. این مقایسه به توضیح اینکه **چرا CFG یک انتخاب مناسب برای زبان‌های برنامه‌نویسی** است و چگونه با استفاده از آنها می‌توان ساختارهای پیچیده‌تر را مدل کرد، می‌پردازد.

برای مقایسه با دیگر انواع گرامرها، فرض کنید،

یک گرامر محدود (Context-Free Grammar) معادل با یک گرامر متغیره (Context-Sensitive Grammar) را در نظر بگیریم.

در گرامر مستقل از متن برای زبان $a^n b^n$ ، یک قاعده تبدیل به شکل $S \rightarrow aSb$ و $S \rightarrow \epsilon$ وجود دارد. این گرامر محدوده است، اما اگر به گرامر متغیره برای همین زبان نگاه کنیم، مجموعه‌ای از قوانین با ساختار پیچیده‌تر خواهیم داشت که در آن تعیین‌کننده‌ها نیاز به توجه به محتوای قبلی دارند. این نشان‌دهنده اختلاف بین قدرت توصیفی گرامرهای مختلف است.

مثال ساده‌ای از گرامر مستقل از متن در زبان پایتون برای تشخیص و تولید عبارات حسابی شامل جمع و ضرب ارائه می‌دهیم.

قوانین گرامری عبارتند از :

expression -> expression + term | term

term -> term * factor | factor

factor -> (expression) | number

number -> 0 | 1 | 2 | ... | 9

در ادامه بعد از تعاریف اولیه، جزئیات مثال بالا را کامل تر بررسی و به تحلیل این قوانین با کمک گرفتن از زبان برنامه نویسی می‌پردازیم.

(2) کاربردهای گرامر های مستقل از متن:

1. زبان های برنامه نویسی:

(i) نقش گرامر های مستقل از متن در توسعه زبان های برنامه نویسی :

گرامر های مستقل از متن در توسعه زبان های برنامه نویسی یک نقش بسیار حیاتی دارند. این گرامرها به تعریف ساختار زبان کمک می کنند و قوانینی را برای نحوه نوشتن برنامه ها تعیین می کنند. به عنوان مثال، در زبان هایی مانند Python یا C، گرامر های مستقل از متن برای توصیف قواعد نحوی زبان استفاده می شود.

(ii) ارتباط با اصول طراحی زبان های برنامه نویسی :

گرامر های مستقل از متن معمولاً با اصول طراحی زبان های برنامه نویسی مرتبط هستند. این گرامرها برای تعیین قوانین نحوی زبان استفاده می شوند و طراحان زبان ها از آنها برای ایجاد یک زبان کارآمد و قابل فهم استفاده می کنند.

2. طراحی کامپایلرها:

1. چگونگی استفاده از گرامر های مستقل از متن در مراحل مختلف کامپایلرها :

گرامر های مستقل از متن در کامپایلرها برای تجزیه و تحلیل سینتاکس برنامه ها استفاده می شوند. در مراحل مختلف کامپایلر، گرامرها به عنوان ورودی به ابزارهای تحلیل گر و تولید کننده کد می آیند تا ساختار و نحوه اجرای برنامه ها را مشخص کنند.

II. نقش آنها در تجزیه و تحلیل سینتاکس برنامه‌ها :

گرامرهای مستقل از متن در تحلیل سینتاکس برنامه‌ها نقش اساسی دارند. با تعریف قوانین گرامری، کامپایلر می‌تواند ساختار برنامه‌ها را تجزیه و تحلیل کرده و مطمئن شود که برنامه‌ها با نحوه نوشته شده تطابق دارند.

(3) شناسایی و تحلیل متون:

(i) کاربرد گرامرهای مستقل از متن در تحلیل و تفسیر متون زبان طبیعی :

در حوزه پردازش زبان طبیعی، گرامرهای مستقل از متن برای تحلیل و تفسیر متون به کار می‌روند. این گرامرها به تعیین ساختار جملات و اجزای زبانی در متون کمک می‌کنند. به عنوان مثال، برای تحلیل جملات یک زبان طبیعی، گرامرهای CFG می‌توانند الگوهای مشخصی را تشخیص دهند و اطلاعات معنایی مفهوم جملات را استخراج کنند.

در حوزه پردازش زبان طبیعی (*NLP*)، گرامرهای مستقل از متن برای تحلیل و تفسیر متون به کار می‌روند. این گرامرها به تعیین ساختار جملات و اجزای زبانی در متون کمک می‌کنند. به عنوان مثال، برای تحلیل جملات یک زبان طبیعی، گرامرهای CFG می‌توانند الگوهای مشخصی را تشخیص دهند و اطلاعات معنایی مفهوم جملات را استخراج کنند.

4) پیاده سازی نمونه:

(a) ایجاد یک نمونه ساده از گرامر مستقل از متن برای یک زبان ساده :

فرض کنید می خواهیم یک گرامر ساده برای تشخیص جملات ساده زبانی اعداد طبیعی ایجاد کنیم. این گرامر مستقل از متن (CFG) به شکل زیر می تواند باشد:

sentence -> **subject verb object**

subject -> **article noun**

verb -> **"likes" | "hates"**

object -> **article noun**

article -> **"a" | "an"**

noun -> **"cat" | "dog" | "apple"**

این گرامر به تشخیص جملات ساده مثل **"a cat likes an apple"** یا **"an apple hates a dog"** کمک می کند.

در اینجا، هر جمله از یک **فاعل**، **فعل**، و **مفعول** تشکیل شده است که هر کدام از قسمت های اصلی جمله با استفاده از نمادهای مختلف معرفی شده اند.

(b) نمایش نحوه استفاده از این گرامر در تجزیه و تحلیل جملات در زبان پایتون :

```
CFG(Project) > 1 > generate_s.py > generate_object
1  import random
2
3  def generate_sentence():
4      sentence = generate_subject() + " " + generate_verb() + " " + \
5          generate_object()
6      return sentence.capitalize() + "."
7
8  def generate_subject():
9      return generate_article() + " " + generate_noun()
10
11 def generate_verb():
12     return random.choice(["likes", "hates"])
13
14 def generate_object():
15     return generate_article() + " " + generate_noun()
16
17 def generate_article():
18     return random.choice(["a", "an"])
19
20 def generate_noun():
21     return random.choice(["cat", "dog", "apple"])
22
23 # Generate a sample sentence
24 sample_sentence = generate_sentence()
25 print("Sample Sentence:", sample_sentence)
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
"
Sample Sentence: An dog hates an cat.
PS C:\Users\Mahdi CS 313>
```

این کد Python یک تابع **generate_sentence** دارد که با استفاده از توابع دیگر مانند **generate_subject**، **generate_verb** و غیره، یک جمله را ایجاد می‌کند. در نهایت، یک جمله نمونه تولید شده و چاپ می‌شود.

با توجه به نمونه کد فوق، اکنون می‌توانیم این تابع را برای تولید چند جمله نمونه فراخوانی کنیم:

```

25 #print("Sample Sentence:", sample_sentence)
26
27 # Generate multiple sample sentences
28 for _ in range(5):
29     sample_sentence = generate_sentence()
30     print("Sample Sentence:", sample_sentence)

```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

PS C:\Users\Mahdi CS 313> & "C:/Users/Mahdi CS 313/AppData
hon/Python312/python.exe" "c:/Users/Mahdi CS 313/CFG(Proje
"
Sample Sentence: A apple hates a dog.
Sample Sentence: A apple hates a apple.
Sample Sentence: A apple likes an dog.
Sample Sentence: An cat likes a apple.
Sample Sentence: An cat hates a cat.
PS C:\Users\Mahdi CS 313> 

```

این بخش از کد، تابع ***generate_sentence*** را پنج بار فراخوانی کرده و جملات تولید شده را چاپ می‌کند.

این پیاده‌سازی نمونه نشان‌دهنده‌ی چگونگی استفاده از گرامرهای مستقل از متن در یک زبان برنامه‌نویسی در اینجا ***Python*** است.

مثالی از گرامر های مستقل از متن

قوانین به صورت زیر بودند :

expression -> **expression + term** | **term**

term -> **term * factor** | **factor**

factor -> **(expression)** | **number**

number -> **0** | **1** | **2** | ... | **9**

عبارت ساده محاسباتی با گرامر بالا مثال می زنیم و آن را مورد بررسی قرار می دهیم:

expression + term * (3 + 4)

حالا این عبارت را به وسیله گرامر تجزیه و تحلیل می کنیم:

1. **expression + term * (3 + 4)**

- این عبارت با گزینه اول گرامر برابر است.
- بنابراین، تجزیه و تحلیل به **expression + term** تقسیم می شود.

2. **expression**

- بازهم با گزینه اول گرامر برابر است.
- بنابراین، تجزیه و تحلیل به **expression + term** تقسیم می شود.

3. **expression + term**

- دو قسمت اصلی در اینجا **expression** و **term**

- تجزیه و تحلیل به **expression** تقسیم می شود.

4. expression

- چون این **expression** شامل یک **expression** دیگر است، به گزینه اول گرامر مربوط می شود.

- بنابراین، تجزیه و تحلیل به **expression + term** تقسیم می شود.

5. expression + term

- دو قسمت اصلی در اینجا **expression** و **term**

- تجزیه و تحلیل به **expression** تقسیم می شود.

6. expression

- چون این **expression** شامل یک **expression** دیگر است، به گزینه اول گرامر مربوط می شود.

- بنابراین، تجزیه و تحلیل به **expression + term** تقسیم می شود.

7. expression + term

- دو قسمت اصلی در اینجا **expression** و **term**

- تجزیه و تحلیل به **expression** تقسیم می شود.

8. term

- با گزینه دوم گرامر برابر است.

- بنابراین، تجزیه و تحلیل به **term * factor** تقسیم می شود.

9. term * factor

- دو قسمت اصلی در اینجا **term** و **factor**

- تجزیه و تحلیل به **term** تقسیم می شود.

10. term

- با زهم با گزینه دوم گرامر برابر است.
- بنابراین، تجزیه و تحلیل به **term * factor** تقسیم می شود.

term * factor.11

- دو قسمت اصلی در اینجا **term** و **factor**
- تجزیه و تحلیل به **factor** تقسیم می شود.

factor.12

- با گزینه سوم گرامر برابر است.
- بنابراین، تجزیه و تحلیل به (**expression**) تقسیم می شود.

(expression).13

- تجزیه و تحلیل به **expression** تقسیم می شود.

expression.14

- چون این **expression** شامل یک **expression** دیگر است، به گزینه اول گرامر برابر می شود.

تا اینجا تجزیه و تحلیل انجام شده است.

و مابقی به تکرار ختم می شود ...

تا زمانی که تجزیه و تحلیل تمام جمله ادامه داده شود. این فرآیند به اصطلاح "**تجزیه و تحلیل جملات**" یا "**پارسینگ**" نیز شناخته می شود .

برای رسم درخت تجزیه (Parse Tree) در زبان پایتون، می‌توانید از کتابخانه **graphviz** استفاده کنید. این کتابخانه به شما امکان می‌دهد گراف‌ها و درختان را به صورت گرافیکی ایجاد کنید.

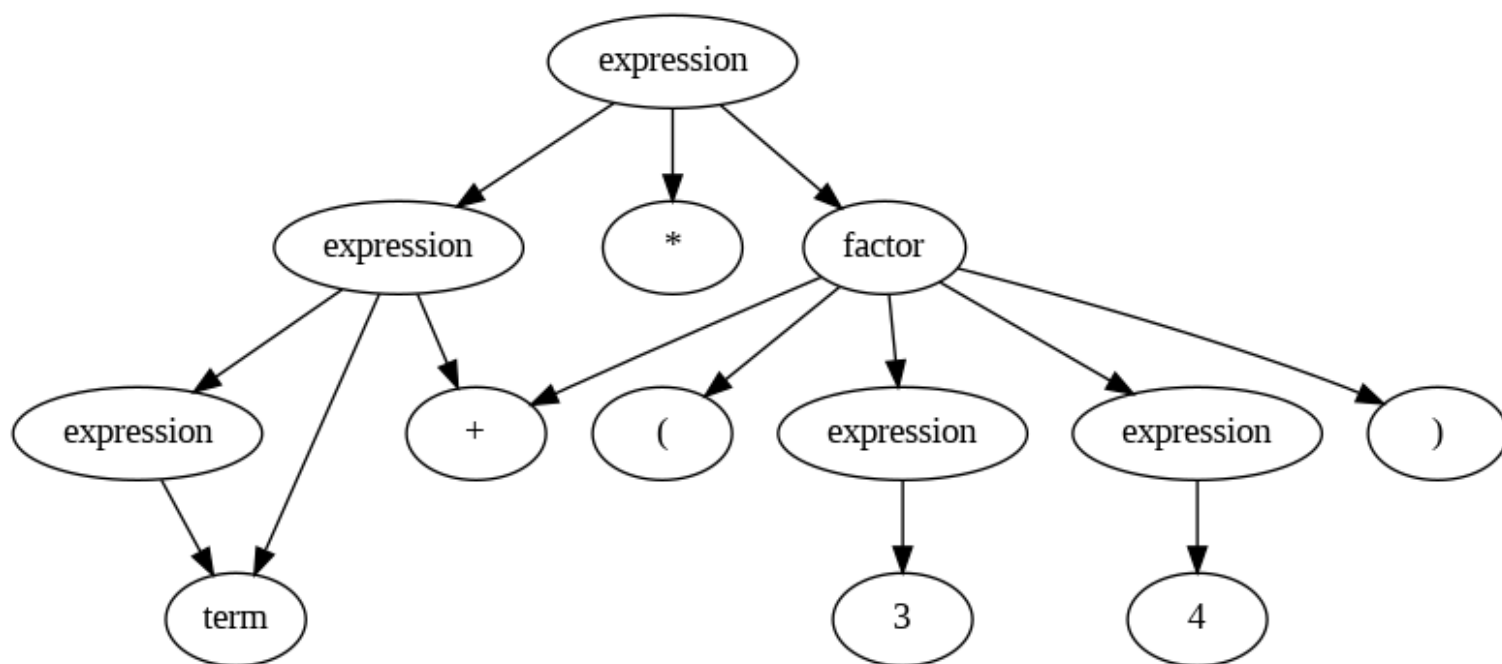
ابتدا، مطمئن شوید که کتابخانه **graphviz** نصب شده باشد. اگر نصب نکرده‌اید، می‌توانید آن را با دستور زیر نصب کنید:

pip install graphviz

```
1  # pip install graphviz
2  import graphviz
3  from IPython.display import Image
4
5  def create_parse_tree(parse_tree, graph, parent_node=None):
6      if isinstance(parse_tree, tuple):
7          current_node = str(hash(parse_tree))
8          graph.node(current_node, label=str(parse_tree[0]))
9          if parent_node is not None:
10             graph.edge(parent_node, current_node)
11             for i, child in enumerate(parse_tree[1:]):
12                 create_parse_tree(child, graph, current_node)
13         else:
14             current_node = str(hash(parse_tree))
15             graph.node(current_node, label=str(parse_tree))
16             if parent_node is not None:
17                 graph.edge(parent_node, current_node)
18
19  # exp : parse tree for expression := ( expression + term * ( 3 + 4 ) )
20  parse_tree_example = ('expression', ('expression', ('expression', 'term'),\
21  | '+' , 'term'), '*', ('factor', '(', ('expression', '3'), '+',\
22  | ('expression', '4'), ')'))
23
24  # create a object of Graph
25  graph = graphviz.Digraph(comment='Parse Tree', format='png')
26  # Create a Parse Tree
27  create_parse_tree(parse_tree_example, graph)
28  # save and show the picture
29  graph.render('parse_tree', view=False, cleanup=True)
30  # show the picture in window
31  Image(filename='parse_tree.png')
```

در این کد، `create_parse_tree` یک تابع بازگشتی است که با گرافیز `graphviz` یک درخت تجزیه ایجاد می‌کند. سپس، این درخت تجزیه در قالب یک تصویر PNG با نام `parse_tree` ذخیره می‌شود.

درخت تجزیه رسم شده به صورت زیر است :



در این مرحله به کاربرد های گرامر مستقل از متن می پردازیم ، برای شروع از کاربرد زبان مستقل از متن CFG در زبان های برنامه نویسی شروع می کنیم :

حتما! یکی از کاربردهای مهم گرامرهای مستقل از متن (CFG) در زبان های برنامه نویسی، توسعه و تحلیل زبان های برنامه نویسی است.

در اینجا یک مثال ساده از گرامر یک زبان ساده ترین حالت اعداد یک تا سه را در نظر بگیرید:

```
expression → number
           | number "+" number
           | number "+" number "+" number
```

این گرامر با استفاده از نمادهای BNF (Backus-Naur Form) نشان داده شده است.

حالا، فرض کنیم می خواهیم یک زبان ساده تحت نام "SumLanguage" بسازیم که عبارت هایی مانند "1" یا "3+2" یا "3+2+1" را تشخیص دهد.

حالا بیا یک برنامه نویسی ساده در زبان Python بنویسیم که این گرامر را تجزیه و تحلیل کند:

```

1  import re
2
3  def parse_expression(input_string):
4      # Grammer BNF
5      grammar = re.compile(r'^(\d+)(\+(\d+))*$')
6
7      # Analysis of expression
8      match = grammar.match(input_string)
9      if match:
10         print(f'The expression "{input_string}" is valid!')
11     else:
12         print(f'The expression "{input_string}" is not valid!')
13
14 parse_expression("1")
15 parse_expression("2+3")
16 parse_expression("1+2+3")
17 parse_expression("4*5")
18

```

OUTPUT TERMINAL PORTS DEBUG CONSOLE PROBLEMS Python

```

The expression "1" is valid!
The expression "2+3" is valid!
The expression "1+2+3" is valid!
The expression "4*5" is not valid!

```

حالا بیا ببینیم برنامه را **گسترش** دهیم تا بتوانیم ارزیابی ارزش اعداد و اعمال محاسبات ساده ماننده جمع زدن را انجام دهیم.

برای این کار، می توانیم تابعی به نام **evaluate_expression** اضافه کنیم:

```

14 # parse_expression("1")
15 # parse_expression("2+3")
16 # parse_expression("1+2+3")
17 # parse_expression("4*5")
18
19 def evaluate_expression(input_string):
20     # Grammar BNF
21     grammar = re.compile(r'^(\d+)(\+(\d+))*$')
22
23     # Analysis of expression
24     match = grammar.match(input_string)
25     if match:
26         numbers = [int(match.group(i)) for i in range(1, match.lastindex + 1)]
27         result = sum(numbers)
28         print(f'The result of the expression "{input_string}" is: {result}')
29     else:
30         print(f'The expression "{input_string}" is not valid!')
31
32 evaluate_expression("1")
33 evaluate_expression("2+3")
34 evaluate_expression("1+2+3")
35 evaluate_expression("4*5")

```

خروجی برنامه به صورت زیر خواهد بود :

```

The result of the expression "1" is: 1
The result of the expression "2+3" is: 2
The result of the expression "1+2+3" is: 1
The expression "4*5" is not valid!

```

اگر عبارت معتبر باشد،

اعداد موجود در عبارت استخراج شده و جمع زده می شوند. نتیجه این جمع نهایی سپس چاپ می شود. این نمونه نشان می دهد چگونه گرامرهای مستقل از متن می توانند در تحلیل و ارزیابی عبارات در زبان های برنامه نویسی به عنوان ابزاری مؤثر عمل کنند.

در این بخش از تحقیق می خواهیم به مبحثی پردازیم که بیشترین کاربرد گرامر های مستقل از متن را شامل می شود، بله همانطور که حدس زدید ، طراحی کامپایلر ...

از اولین سوالی که در این زمینه مطرح کردیم شروع می کنیم،

چگونگی استفاده از گرامر های مستقل از متن در مراحل مختلف کامپایلرها:

برای معرفی نحوه طراحی کامپایلر با استفاده از گرامر های مستقل از متن (CFG)، می خواهیم یک مثال ساده از زبان میانی (Intermediate Language) ارائه دهیم که توسط کامپایلر به زبان ماشین ترجمه می شود.

اما می خواهیم مختصرا به مراحل طراحی و ساخت کامپایلر اشاره کنیم تا بتوانیم دید کلی تر از این فرآیند به شما ارائه دهیم، لازم به ذکر است که طراحی کامپایلر یک حوزه تحقیقاتی و مهندسی پیچیده است و نیاز به دانش در زمینه های مختلف مانند تئوری زبان ها، مخابرات، معماری کامپیوتر، بهینه سازی کد و غیره دارد.

در ادامه به کاملا با فرآیند طراحی کامپایلر آشنا خواهیم شد .

طراحی کامپایلر یک فرآیند پیچیده است که هدف اصلی آن ترجمه یا تبدیل کد نوشته شده در یک زبان برنامه نویسی به کد ماشین قابل اجرا توسط کامپیوتر است. یک کامپایلر

معمولاً از چند مرحله تشکیل شده و هر مرحله یک وظیفه خاص را انجام می‌دهد. در ادامه، مراحل مهم طراحی کامپایلر به صورت مختصر توضیح داده شده‌اند:

1. **تحلیل لغوی (Lexical Analysis)** در این مرحله، کد منبع ورودی به توکن‌های کوچکتر تقسیم می‌شود. توکن‌ها شامل واحدهای معنایی هستند مانند کلمات کلیدی، عملگرها، اعداد و غیره.

2. **تحلیل سینتاکسی (Syntax Analysis)** در این مرحله، توکن‌های تولید شده در مرحله قبل به توالی‌ها و ساختارهای گرامری زبان تبدیل می‌شوند. این مرحله تشخیص می‌دهد که آیا ترکیب توکن‌ها با گرامر زبان معتبر است یا خیر.

3. **تحلیل نحوی (Semantic Analysis)** در این مرحله، معنای کد بررسی می‌شود. این شامل بررسی اعتبار و معنای عبارات و دستورات است. برای مثال، این مرحله ممکن است اطمینان حاصل کند که یک متغیر قبلاً تعریف شده باشد.

4. **ترجمه به کد میانی (Intermediate Code Generation)** در این مرحله، یک کد میانی (Intermediate Code) تولید می‌شود. این کد میانی یک سطح ترجمه بین زبان برنامه‌نویسی و زبان ماشین است.

5. **تحلیل بهینه‌سازی (Optimization Analysis)** کد میانی بهینه‌سازی می‌شود تا عملکرد و بهره‌وری اجرای کد نهایی بهبود یابد.

6. **ترجمه به کد ماشین (Code Generation)** در این مرحله، کد ماشین قابل اجرا تولید می‌شود که توسط سخت‌افزار کامپیوتر قابل اجراست.

7. **مرحله پس پردازش (Post-processing)** کد ماشین تولید شده ممکن است به منظور بهبود عملکرد یا توسعه اضافی به مرحله پس پردازش ارسال شود.

گرامرهای مستقل از متن (CFG) در مرحله تحلیل سینتاکس یک طراحی کامپایلر استفاده می شوند. در طراحی کامپایلر، تحلیل سینتاکس یکی از مراحل اساسی است که به وسیله گرامرهای مستقل از متن انجام می شود.

این مرحله به نام **تحلیل نحوی (Syntax Analysis)** یا **تجزیه و تحلیل نحوی (Parsing)** نیز شناخته می شود. نقش اصلی این مرحله تضمین معتبر بودن سینتاکس برنامه های منبع (زبان های برنامه نویسی) و تولید یک درخت نحوی (Parse Tree) یا یک جدول نحوی (Parse Table) است.

در مرحله تحلیل نحوی با استفاده از گرامرهای مستقل از متن، برنامه کامپایل شده تجزیه و تحلیل می شود تا ساختار نحوی آن به دست آید. این ساختار نحوی به کمک درخت نحوی نمایش داده می شود که ترتیب و ارتباط بین اجزاء مختلف برنامه را نشان می دهد.

در این مرحله، اگر برنامه منبع با گرامر مستقل از متن مطابقت داشته باشد، یک **درخت نحوی معتبر** تولید می شود که به عنوان ورودی برای مراحل بعدی مانند ترجمه به کد میانی و تولید کد ماشین استفاده می شود. این مرحله بسیار مهم است زیرا تعیین می کند که آیا برنامه ورودی به زبان برنامه نویسی معتبر است یا خیر.

بیاییم و یک مثال ساده از گرامر مستقل از متن برای یک زبان ساده بنویسیم ، سپس با استفاده از این گرامر یک برنامه کوچک به زبان پایتون تجزیه و تحلیل کنیم.

فرض کنید می‌خواهیم یک زبان ساده برای اعداد صحیح و عملیات جمع بنویسیم.
گرامر ساده‌ای می‌تواند به صورت زیر باشد:

expression -> number '+' number

number -> '0' | '1' | '2' | ... | '9'

در این گرامر، **expression** به صورت یک عبارت جمع دو عدد تعریف شده است و **number** می‌تواند یک عدد صحیح از 0 تا 9 باشد.

```
import re

# گرامر
grammar = r"expression -> number \+ number\nnumber -> '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'"

def parse_input(input_string):
    # تجزیه و تحلیل با استفاده از گرامر
    match = re.match(grammar, input_string)
    if match:
        print("تجزیه و تحلیل موفقیت‌آمیز")
        print("عدد اول:", match.group(1))
        print("عدد دوم:", match.group(2))
    else:
        print("تجزیه و تحلیل ناموفق")

# ورودی نمونه
input_string = "5 + 3"

# تجزیه و تحلیل و نمایش نتیجه
parse_input(input_string)
```

تجزیه و تحلیل ناموفق

در اینجا، تابع `parse_input` ورودی را بر اساس گرامر تجزیه و تحلیل می‌کند. ورودی نمونه `"5 + 3"` با استفاده از گرامر با موفقیت تجزیه و تحلیل می‌شود و اعداد از جمع جدا شده و نمایش داده می‌شوند. اگر ورودی معتبر نباشد، یک پیام خطا نمایش داده می‌شود.

این مثال ساده نشان دهنده استفاده از گرامرهای مستقل از متن در تجزیه و تحلیل سینتاکس برنامه‌هاست.

در یک طراحی کامپایلر واقعی تر، از ابزارهایی مانند **lex** و **yacc** برای تجزیه و تحلیل بیشتر و انعطاف پذیرتر گرامرها استفاده می شود.

در یک طراحی کامپایلر واقعی تر، از ابزارهای خودکار تجزیه و تحلیل (Parser Generator) مانند **yacc (Yet Another Compiler Compiler)** و **lex (Lexical Analyzer Generator)**

برای ایجاد تجزیه کننده (parser) و تحلیل گر نحوی (lexical analyzer) استفاده می شود.

این ابزارها از گرامرهای مستقل از متن به عنوان ورودی دریافت می کنند و کدهای تجزیه و تحلیل را تولید می کنند.

نمونه کد آن جهت آشنایی بصورت زیر است :

```
%start calc
%token NUM
%left '+' '-'
%left '*' '/'
%%
calc:  exp    { print($1); }
;
exp:   exp '+' exp    { $$ = $1 + $3; }
      | exp '-' exp    { $$ = $1 - $3; }
      | exp '*' exp    { $$ = $1 * $3; }
      | exp '/' exp    { $$ = $1 / $3; }
      | '(' exp ')'    { $$ = $2; }
      | NUM            { $$ = $1; }
;
%%
```

فایل گرامر (calc.y)

فایل **لکسر (calc.l)**

```
%{
#include "y.tab.h"
}%
%%
[0-9]+      { yylval = atoi(yytext); return NUM; }
[-+*/()]    { return yytext[0]; }
[ \t\n]     ;
.           { printf("Invalid character: %s\n", yytext); }
%%
```

تست کردن :

ابتدا با استفاده از **yacc** و **lex** ، کدهای تجزیه و تحلیل تولید می شوند:

```
yacc -d calc.y
lex calc.l
gcc y.tab.c lex.yy.c -o calc_parser
```

حالا برنامه تجزیه کننده را اجرا می کنیم:

```
./calc_parser
```

نمونه ورودی را وارد می کنیم :

```
3 + 4 * (2 - 1)
```

در نهایت می بایست در خروجی عدد 7 را به برنامه ای که نوشته شده پاسخ بدهد.

در این مثال، گرامر و لکسر با استفاده از **yacc** و **lex** تعریف شده اند و برنامه تجزیه کننده به وسیله این ابزارها ایجاد شده است. این گرامر به طور بازگشتی تعریف شده است و توانسته است عبارت های جمع و تفریق را با درستی تجزیه و تحلیل کند.

پردازش زبان طبیعی (NLP) و کاربرد

گرامرهای مستقل از متن (CFG) در حوزه پردازش زبانها (Natural Language Processing) و شناسایی و تحلیل متون دارای کاربردهای متعددی هستند. در زیر، به برخی از کاربردهای این گرامرها در شناسایی و تحلیل متون در حوزه NLP اشاره خواهیم کرد:

1. تحلیل نحوی (Syntactic Parsing) گرامرهای مستقل از متن برای تحلیل نحوی جملات و

عبارات زبان طبیعی به کار می‌روند. این تجزیه و تحلیل به ساختار جمله بر اساس گرامر مشخص کمک می‌کند و اجزای مختلف جمله مانند فعل، فاعل، مفعول، و نقش آنها را مشخص می‌کند.

2. شناسایی اجزاء زبانی (Part-of-Speech Tagging) گرامرهای مستقل از متن در شناسایی

نقش کلمات (Part-of-Speech) در جمله کمک می‌کنند. این فرآیند به تخصیص برچسب به هر کلمه بر اساس نقش گرامری آنها اشاره دارد، مانند اینکه یک کلمه فعل، اسم، صفت و یا حرف اضافه است.

3. ساخت درخت‌های تجزیه (Parsing Trees) با استفاده از گرامرهای مستقل از متن، می‌توان

درخت‌های تجزیه برای جملات ساخت. این درخت‌ها نشان‌دهنده ساختار نحوی جملات هستند و در تحلیل ساختار جمله به کار می‌روند.

4. تحلیل معنایی (Semantic Parsing) گرامرهای مستقل از متن به تحلیل معنایی جملات نیز

کمک می‌کنند. با ایجاد گرامرهای مرتبط با ساختار معنایی کلمات و جملات، می‌توان اطلاعات معنایی را استخراج کرد.

5. اصلاح خطاهای نحوی (Error Correction) اگر یک جمله نحوی نادرست باشد، گرامرهای

مستقل از متن می‌توانند به اصلاح خطاهای نحوی کمک کنند. با استفاده از این گرامرها می‌توان ساختار صحیح جمله را بازسازی کرد.

6. **تحلیل پرسش و پاسخ (Question-Answering)** گرامرهای مستقل از متن در فرآیند تحلیل پرسش‌ها و ارتباط آنها با جواب‌ها نیز به کار می‌روند. با تحلیل ساختار نحوی پرسش‌ها و جملات، می‌توان بهترین پاسخ را پیدا کرد.

7. **پردازش احساسات (Sentiment Analysis):** در تحلیل احساسات متون زبان طبیعی، گرامرهای CFG می‌توانند به تشخیص و شناسایی الگوهای زبانی مرتبط با احساسات (مثل خوشحالی یا ناراحتی) کمک کنند. این گرامرها می‌توانند به تحلیل و تفسیر مفهوم احساسات در جملات کمک کنند.

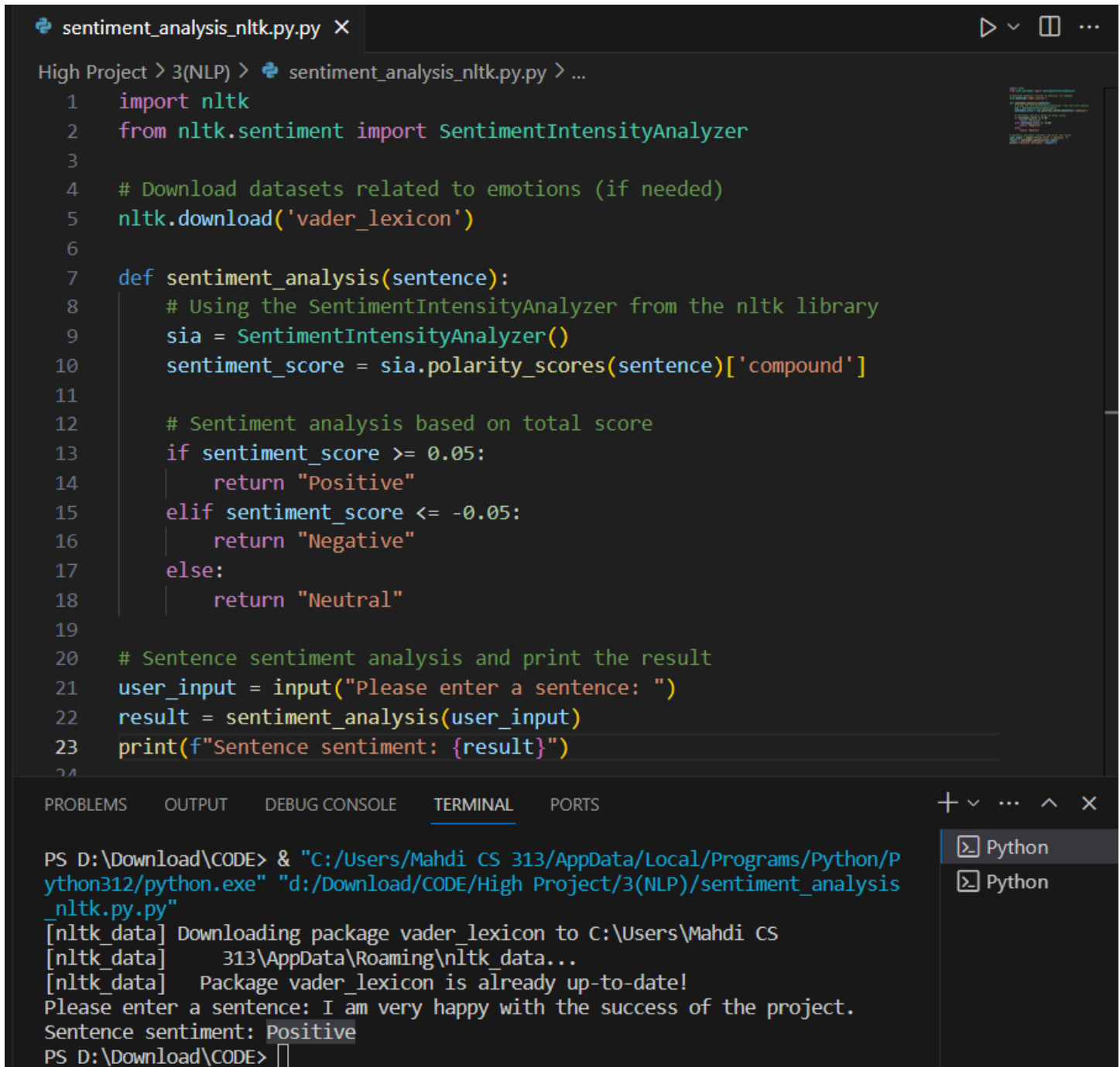
8. **پردازش تحت مجموعه‌های دانش (Knowledge Base):** گرامرهای CFG در پردازش زبان طبیعی می‌توانند در ساخت و توسعه مدل‌های دانش کمک کنند. با تحلیل و تفسیر متون، می‌توان اطلاعاتی را استخراج کرد و به عنوان ورودی برای مدل‌های دانشی استفاده کرد.

همچنین باید توجه داشت که در NLP، استفاده از گرامرهای مستقل از متن ممکن است به دلیل پیچیدگی زبان‌های طبیعی و متغیر بودن ساختار زبان، محدودیت‌های خود را داشته باشد. برای مواجهه با این محدودیت‌ها، روش‌های پیشرفته‌تری مانند استفاده از یادگیری عمیق (Deep Learning) و شبکه‌های عصبی مورد استفاده قرار می‌گیرد، که در راستای اهداف تحقیق ما نمی‌گنجد.

اما برای درک بیشتر این زمینه به مثالی از این بخش تشخیص احساسات با استفاده از کتابخانه (*nlk*) در زبان برنامه نویسی *Python* خواهیم پرداخت.

برای پردازش احساسات با استفاده از گرامرهای CFG در زبان پایتون، ما می‌توانیم از کتابخانه *nltk* استفاده کنیم.

گرامرهای مستقل از متن سریعاً به عنوان یک ابزار مهم در زمینه تجزیه و تحلیل زبان‌ها و طراحی زبان‌های برنامه‌نویسی شناخته شدند. این ابزار در ساختارهای نحوی زبان‌های طبیعی،



```

sentiment_analysis_nltk.py.py X
High Project > 3(NLP) > sentiment_analysis_nltk.py.py > ...
1 import nltk
2 from nltk.sentiment import SentimentIntensityAnalyzer
3
4 # Download datasets related to emotions (if needed)
5 nltk.download('vader_lexicon')
6
7 def sentiment_analysis(sentence):
8     # Using the SentimentIntensityAnalyzer from the nltk library
9     sia = SentimentIntensityAnalyzer()
10    sentiment_score = sia.polarity_scores(sentence)['compound']
11
12    # Sentiment analysis based on total score
13    if sentiment_score >= 0.05:
14        return "Positive"
15    elif sentiment_score <= -0.05:
16        return "Negative"
17    else:
18        return "Neutral"
19
20 # Sentence sentiment analysis and print the result
21 user_input = input("Please enter a sentence: ")
22 result = sentiment_analysis(user_input)
23 print(f"Sentence sentiment: {result}")
24
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS D:\Download\CODE> & "C:/Users/Mahdi CS 313/AppData/Local/Programs/Python/Python312/python.exe" "d:/Download/CODE/High Project/3(NLP)/sentiment_analysis_nltk.py.py"
[nltk_data] Downloading package vader_lexicon to C:\Users\Mahdi CS
[nltk_data] 313\AppData\Roaming\nltk_data...
[nltk_data] Package vader_lexicon is already up-to-date!
Please enter a sentence: I am very happy with the success of the project.
Sentence sentiment: Positive
PS D:\Download\CODE>

```

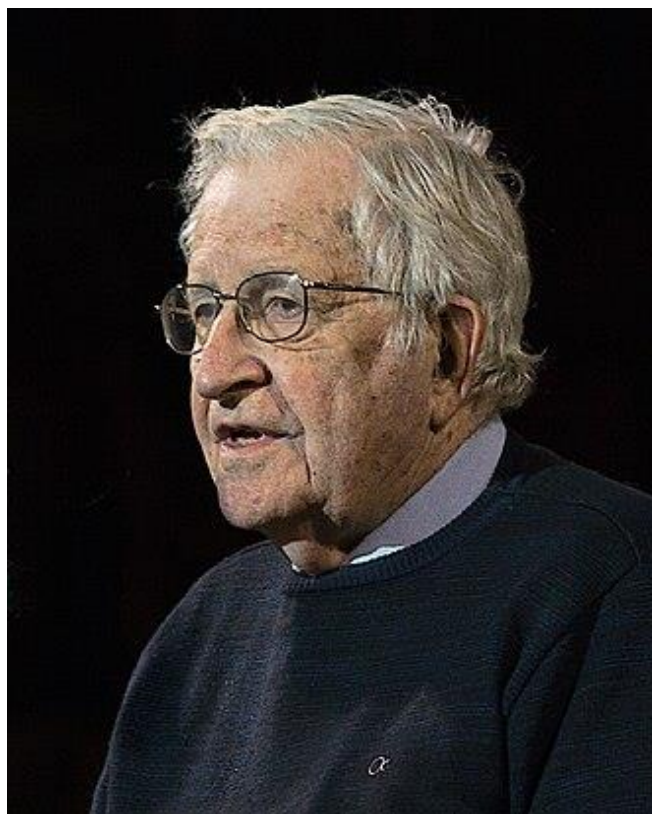
تجزیه و تحلیل کد منبع برنامه نویسی، و طراحی کامپایلرها بسیار مؤثر ثابت شده‌اند. از آن زمان به ویژه با پیشرفت‌های تکنولوژی و یادگیری ماشین، این مفهوم به عنوان پایه‌ای برای فهم و پردازش زبان‌های مختلف در دنیای دیجیتال به کار گرفته می‌شود.

در این مثال، از گرامرهای مستقل از متن (CFG) استفاده نشده، بلکه از یک مدل ساخته شده بر روی دیتاست‌های مرتبط با احساسات بهره گرفته شده بود.

همانطور که مشاهده شد، به جمله <<از موفقیت پروژه بسیار خوشحالم.>> واکنش (مثبت) نشان داده، البته این مدل‌ها توسط شبکه‌های عصبی مصنوعی که ایده‌ی اصلی آن از عملکرد مغز انسان گرفته شده استفاده می‌کند و نمونه داده‌های آزمایشی را مورد بررسی قرار میدهد تا به نتیجه مطلوب خود میل کند.

با توجه به مسائل ارائه شده خالی از لطف نیست که به تاریخچه گرامر های مستقل از متن هم پرداخته شود ...

در دهه 1960، دو محقق اهمیت بزرگی را بر گرامرهای مستقل از متن آشکار کردند:



نوام چامسکی (Noam Chomsky) و
استیون کول کلین (Stephen Cole
Kleene)

اوریام نوام چامسکی (زاده 7 دسامبر 1928) یک استاد آمریکایی و روشنفکر عمومی است که به دلیل فعالیت‌هایش در زبان‌شناسی، فعالیت‌های سیاسی و نقد اجتماعی شناخته شده است. چامسکی که گاهی «پدر زبان‌شناسی مدرن» نامیده می‌شود، از چهره‌های اصلی در فلسفه تحلیلی و یکی از

بنیان‌گذاران حوزه علوم شناختی است. او برنده جایزه استاد زبان‌شناسی در دانشگاه آریزونا و استاد

بازنشسته موسسه در موسسه فناوری ماساچوست (MIT) است. چامسکی در میان نویسندگان زنده با بیشترین استناد بیش از 150 کتاب در موضوعاتی مانند زبان شناسی، جنگ و سیاست نوشته است. از نظر ایدئولوژیک، او با آنارکو سندیکالیسم و سوسیالیسم آزادیخواه همسو می شود.

استفن کول کلین (/ 'kleɪni / KLAY-nee)؛ ۵ ژانویه ۱۹۰۹ – ۲۵ ژانویه

(۱۹۹۴) یک ریاضی دان آمریکایی بود .

یکی از شاگردان آلونزو چرچ ، کلین،

همراه با روسا پیترو ، آلن

تورینگ ، امیل پست و دیگران،

بیشتر به عنوان بنیانگذار

شاخه منطق ریاضی معروف به نظریه

بازگشت شناخته می شود که متعاقباً

به ارائه مبانی نظری کمک کرد . علوم

کامپیوتر . کار کلین بر اساس

مطالعه توابع قابل محاسبه است .

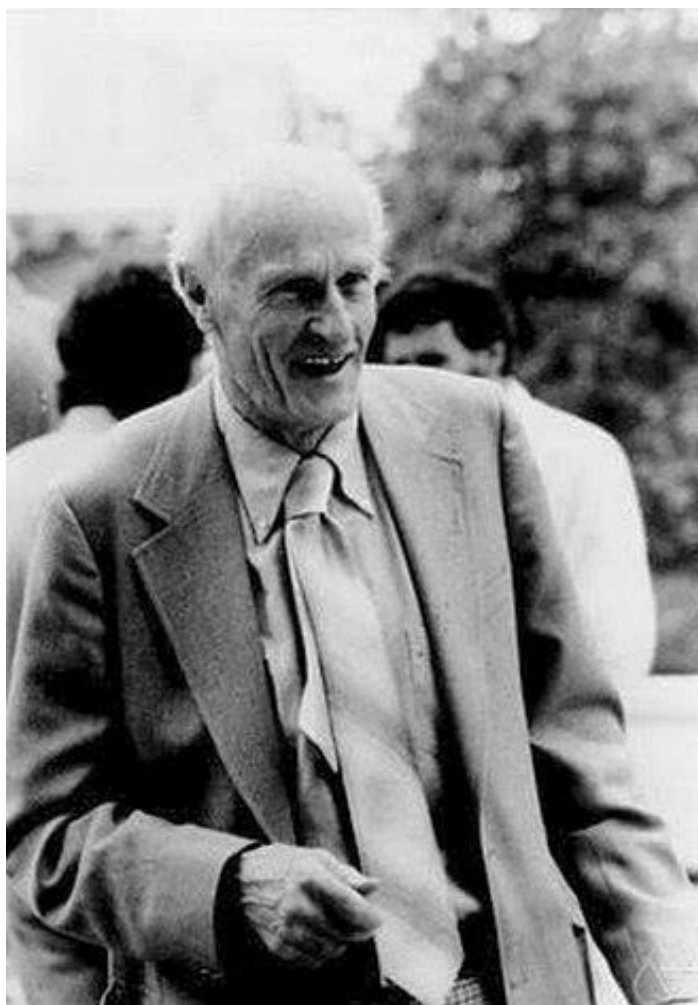
تعدادی از مفاهیم ریاضی به نام او

هستند: سلسله مراتب کلین ، جبر

کلین ، ستاره کلین (کلین) بستن

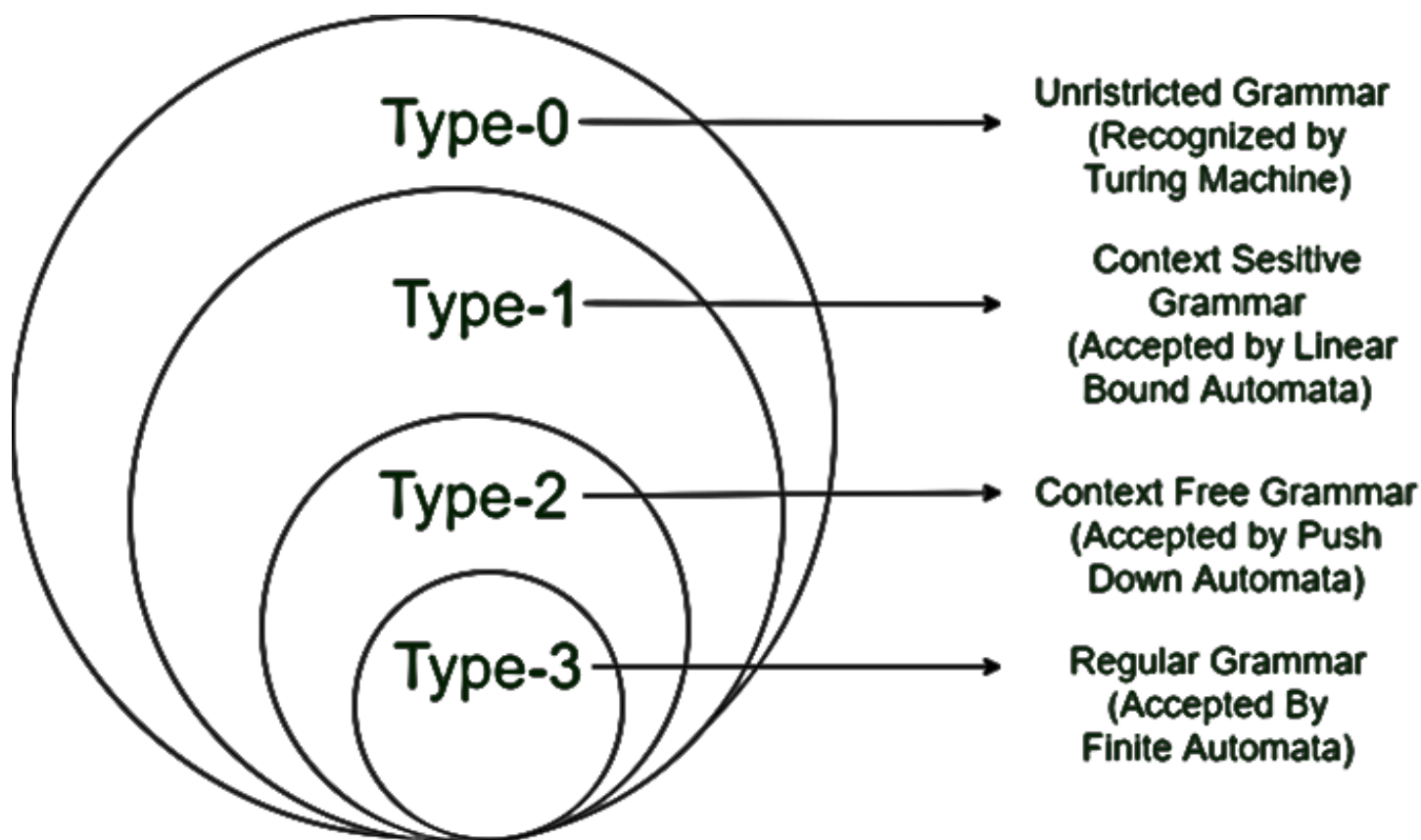
کلین)، قضیه بازگشت کلین و قضیه

نقطه ثابت کلین . او همچنین در سال



1951 عبارات منظمی را برای توصیف شبکه های عصبی مک کالوخ-پیتس ابداع کرد و کمک های قابل توجهی به پایه های شهودگرایی ریاضی کرد.

چامسکی با معرفی "سلسله مراتب چهارگانه (Chomsky hierarchy) "گرامرها، مفهوم گرامرهای مستقل از متن را در زمینه زبان شناسی و علوم کامپیوتر توسعه داد. این تقسیم بندی متشکل از چهار سطح مختلف (نوع 0 تا نوع 3) است و هر کدام از این سطوح توانایی محدودی در توصیف زبان های مختلف را نشان می دهد.



پیشنهادهای برای تحقیقات آینده:

1. توسعه گرامر های مستقل از متن:

- در این قسمت پیشنهادهای برای توسعه گرامر های مستقل از متن در جهت پشتیبانی از ویژگی های پیشرفته زبان های برنامه نویسی را بیان می کنیم:

1- پشتیبانی از زبان های برنامه نویسی مدرن:

2- ارتقاء گرامرها به منظور پشتیبانی از ویژگی های زبان های برنامه نویسی مدرن، مانند پترن ها (pattern matching)، نمادگذاری های پیشرفته، ویژگی های جدید تعریف توابع و ...

3- پردازش مفاهیم نسبی:

4- توسعه گرامرها برای پشتیبانی از پردازش مفاهیم نسبی در زبان های برنامه نویسی، مثل مدیریت توابع داخلی (nested functions)، کلاس ها و اشیاء، ارث بری و ...

5- پشتیبانی از الگوهای پیچیده:

6- افزایش قابلیت گرامرها برای تشخیص و پردازش الگوهای پیچیده در کدهای برنامه نویسی مانند توابع بازگشتی، ایفاهای متعدد و الگوهای متداول.

7- انعطاف پذیری در تعریف متغیرها:

8- ارتقاء گرامر به منظور انعطاف پذیری بیشتر در تعریف متغیرها و نحوه استفاده از آنها، از جمله تعریف متغیرهای نوع دار (typed variables) و پارامترهای ورودی و خروجی توابع.

9- پشتیبانی از الگوهای پرکاربرد:

10- توسعه گرامرها بر اساس الگوهای پرکاربرد در زبان های برنامه نویسی، مانند تشخیص الگوهای تسلسلی (sequence patterns)، الگوهای جستجو (search patterns) و ...

11- تعامل با معماری های نوظهور:

12- پشتیبانی از گرامرها برای تعامل با معماری های نوظهور نرم افزاری، از جمله معماری های مبتنی بر خدمات (Service-oriented architecture) و معماری های میکروسرویس.

13- آگاهی از پیچیدگی های زبان های برنامه نویسی:

14- بررسی و درک بهتر از پیچیدگی ها و چالش های موجود در زبان های برنامه نویسی

مدرن و تطابق گرامرها با این پیچیدگی ها.

15- تحقیق در زمینه تحلیل و بهبود کارایی:**16-** انجام تحقیقات در زمینه بهبود الگوریتم ها و روش های تحلیل گرامر های مستقل از

متن برای افزایش کارایی و سرعت در پردازش زبان های برنامه نویسی پیچیده.

2. کاربردهای گسترده تر در پردازش زبان طبیعی:

- بررسی کاربردهای گرامر های مستقل از متن (CFG) در پردازش زبان طبیعی (NLP) با تأکید بر تولید خودکار متن و ترجمه ماشینی، امکانات گسترده ای را برای بهبود عملکرد و کارایی سیستم های زبانی فراهم می کند. این گرامرها به عنوان ابزاری قوی در تجزیه و تحلیل ساختار جملات و اجزای زبانی متون عمل می کنند. زیرا CFG می تواند الگوهای زبانی مشخص را شناسایی کرده و به تفکیک نحوه ترکیب کلمات در جملات کمک کند.

3. بهبود عملکرد کامپایلرها:

- تحقیقات در زمینه بهبود عملکرد و کارایی کامپایلرها با استفاده از تکنیک های پیشرفته مبتنی بر گرامر های مستقل از متن به منظور بهبود فرآیند ترجمه کدهای برنامه نویسی و افزایش سرعت و دقت کامپایل در پروسه ساخت و اجرای نرم افزارها بسیار حائز اهمیت است.
- این تحقیقات می توانند به راهکارهای نوآورانه منجر شوند که علاوه بر افزایش سرعت و بهره وری در کامپایل، به بهبود تجربه توسعه دهندگان نیز کمک کنند. از جمله مواردی که می تواند در این حوزه مورد توجه قرار گیرد، بهبود فرآیند ارتقاء کد (refactoring)، بهینه سازی خودکار ساختار کد (automated code structuring) و تطبیق پویا با تغییرات (dynamic adaptation) در طراحی و پیاده سازی نرم افزارهاست. این تحقیقات همچنین می توانند به کاهش زمان توسعه و نگهداری نرم افزارها کمک کرده و فرآیند تحویل نرم افزار را بهبود بخشند.
- در کل، استفاده از تکنیک های پیشرفته مبتنی بر گرامر های مستقل از متن در کامپایلرها می تواند ارتقاء یافته های مهمی در زمینه بهره وری و بهبود عملکرد نرم افزارها به دنبال داشته باشد.

4. کاربردهای جدید در هوش مصنوعی:

- بررسی چگونگی استفاده از گرامرهای مستقل از متن در الگوریتم‌ها و مدل‌های هوش مصنوعی مانند یادگیری عمیق :

الگوریتم‌ها و مدل‌های هوش مصنوعی مبتنی بر گرامرهای مستقل از متن می‌توانند در زمینه‌های زیر به کار گرفته شوند:

- تحلیل و تولید متن:
- استفاده از گرامرهای CFG برای تحلیل و تولید متن‌های زبان طبیعی. این مفهوم می‌تواند در تولید خودکار محتواهای متنی برای سیستم‌های هوش مصنوعی یا در تولید خودکار خبرها و مقالات مفید باشد.

- پردازش زبان‌های طبیعی:

- گرامرهای CFG می‌توانند به عنوان ابزارهای قدرتمند برای پردازش زبان‌های طبیعی در الگوریتم‌های یادگیری عمیق مورد استفاده قرار گیرند. این استفاده می‌تواند در تحلیل و درک مفاهیم متنی و تولید پاسخ‌های هوشمند در سیستم‌های مبتنی بر هوش مصنوعی مفید باشد.

- تولید کد و برنامه‌نویسی خودکار:

- به کمک گرامرهای CFG، مدل‌های هوش مصنوعی می‌توانند در فرآیند تولید کد برنامه و برنامه‌نویسی خودکار مؤثر باشند. این کاربرد در تسهیل فرآیند توسعه نرم‌افزار و افزایش بهره‌وری در حوزه‌های مختلف صنعتی اهمیت دارد.

- ترجمه ماشینی پیشرفته:

- ادغام گرامرهای CFG در مدل‌های ترجمه ماشینی عمیق می‌تواند به بهبود دقت و کیفیت ترجمه‌های ماشینی کمک کند. این به‌ویژه در سیستم‌های مترجم متن‌های تخصصی و پیچیده مفید است.

- تولید گفتار طبیعی:
- گرامرهای CFG به عنوان یک ابزار قوی در مدل های تولید گفتار طبیعی (NLG) در زمینه هایی مانند سیستم های هوشمند صوتی (مانند سیستم های صوتی هوش مصنوعی) مورد استفاده قرار گیرند.

نتیجه‌گیری:

در کل، می توان خلاصه ای به این مضمون ارائه نمود که :

توسعه گرامرهای مستقل از متن (CFG) با توجه به پیشنهادات مطرح شده می تواند به بهبود و پیشرفت زبان های برنامه نویسی مدرن کمک کند. این توسعه شامل پشتیبانی از ویژگی های پیشرفته، پردازش مفاهیم نسبی، پشتیبانی از الگوهای پیچیده، انعطاف پذیری در تعریف متغیرها، پشتیبانی از الگوهای پرکاربرد، تعامل با معماری های نوظهور، و آگاهی از پیچیدگی های زبان های برنامه نویسی می شود.

در زمینه پردازش زبان طبیعی (NLP)، استفاده از CFG به عنوان ابزار تجزیه و تحلیل ساختار جملات و اجزای زبانی متن ها، کاربردهای گسترده ای در تحلیل مفاهیم متنی، تولید خودکار متن، ترجمه ماشینی، و حتی تولید گفتار طبیعی دارد. این استفاده در مدل های یادگیری عمیق و سیستم های هوش مصنوعی به دقت و هوشمندی بیشتری در پردازش زبان های طبیعی ایجاد می کند.

پروژه حاضر با ارائه مرور جامع و کاملی از گرامرهای مستقل از متن و نقش آنها در علوم کامپیوتر، اطلاعات مفیدی برای دانشجویان و تحقیق گران فراهم می آورد و ...

- ❖ Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). "Compilers: Principles, Techniques, and Tools" (2nd ed.). Pearson/Addison-Wesley.
- ❖ Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). "Introduction to Automata Theory, Languages, and Computation" (3rd ed.). Addison-Wesley.
- ❖ Jurafsky, D., & Martin, J. H. (2020). "Speech and Language Processing" (3rd ed.). Draft available online:
<https://web.stanford.edu/~jurafsky/slp3/>
- ❖ Grune, D., Jacobs, C. J. H., & Langendoen, K. (2008). "Parsing Techniques: A Practical Guide" (2nd ed.). Springer.
- ❖ https://en.wikipedia.org/wiki/Stephen_Cole_Kleene
- ❖ https://en.wikipedia.org/wiki/Noam_Chomsky