

اعضای گروه: سپهر قارداش 40220143

هدیه طهمورثی 40219263

درس: نظریه زبان‌ها و ماشین‌ها

Project Brief

پیش‌نمایش کلی پروژه:

در این پروژه، هدف ما طراحی و پیاده‌سازی سیستمی است که یک گرامر را به عنوان ورودی دریافت کرده و پس از تحلیل آن، ماشین حالت متناظر را تولید کند. سپس خروجی به دست آمده را از جنبه‌های مختلف مورد بررسی قرار داده و به صورت گرافیکی نمایش خواهیم داد. همچنین خطاهای احتمالی بررسی و مدیریت می‌شوند. این گزارش به صورت گام‌به‌گام مراحل انجام پروژه را تحلیل کرده و پیاده‌سازی آن را توضیح می‌دهد.

تولید ماشین حالت

- با استفاده از گرامر تولید شده در مراحل قبلی باید یک ماشین حالت بسازیم
- برای ساخت ماشین حالت باید از متغیر شروع طی کنیم و همه مسیرها را تا رسیدن به حالتنهای بیابیم

دریافت و تجزیه ی گرامر ورودی

- گرامر ورودی ما یک گرامر به فرم EBNF است
- باید ورودی که می‌گیریم را بررسی کنیم و آن را به ترمینال‌ها و غیر ترمینال‌ها تجزیه کنیم تا در صورت لزوم خطاهای مورد نیاز را نشان دهیم یا آن قانون را به گرامر اضافه کنیم

تولید خروجی و خطایابی

- باید عکس ماشین حالت را در فرم PNG ذخیره کنیم
- خطایابی‌ها در مرحله اول انجام شدند.

نمایش گراف یک ماشین حالت

- ماشین حالت ساخته شده در مرحله قبل را با یک عکس نمایش میدهیم

تحلیل و گزارش ماشین حالت

- در یک فایل جداگانه اطلاعات مربوط به این ماشین حالت را ذخیره می‌کنیم.

کلاس ها



Variable

این کلاس مسئول مدیریت هر متغیر در گرامر است. هر متغیر شامل نام، قواعد و اطلاعات مربوط به ترمینال‌ها و غیرترمینال‌ها است

این کلاس شامل ویژگی‌های زیر است که توضیحات آن‌ها در بخش‌های بعدی داده خواهد شد

```
def __init__(self, grammar, all_rules):
    self.grammar = grammar
    left_right = all_rules.split("=::")
    self.name = left_right[0].strip()
    self.RHS =
    left_right[1].strip().replace("\\" + "eps" + "\",")
    self.all = all_rules.strip()
    self.rules = self.RHS.split("|")
    self.isLoop = self.RHS.startswith("{") and
    self.RHS.endswith("}")
    self.isOptional = self.RHS.startswith("[") and
    self.RHS.endswith("]")
    self.isValid = True
    self.isDeterministic = True
    self.adjust_loops()
    self.adjust_optionals()
    self.isValid = self.is_grammar()
    self.all = f"{{self.name}} =::"
    {"|".join(self.rules)}"
    self.find_terminals()
```

Grammar

این کلاس مسئول مدیریت کل گرامر و تولید ماشین حالت است. در این کلاس، متغیرها، ترمینال‌ها و قواعد گرامر ذخیره می‌شوند. همچنین، این کلاس مسئول ایجاد گراف ماشین حالت و تحلیل آن است

این کلاس شامل ویژگی‌های زیر است که توضیحات آن‌ها در بخش‌های بعدی داده خواهد شد

```
def __init__(self):
    self.allVars = []
    self.start = None
    self.G = nx.MultiDiGraph()
    self.visitedVars = []
    self.terminals = []
    self.ogGrammar = []
    self.ogVariables = []
```

Grammar



`self.allVars:`

این فیلد یک لیست است که تمام متغیرهای موجود در گرامر را نگهداری می‌کند. هر متغیر می‌تواند یک شئ باشد که اطلاعات مربوط به یک متغیر خاص در گرامر را ذخیره می‌کند. در این لیست متغیرهای مضافقی که در مرحله تحلیل متغیرها ساخته می‌شود نیز قرار دارند.



`self.start:`

این فیلد نشان‌دهنده متغیر شروع در گرامر است.



`self.G:`

این فیلد یک گراف جهت‌دار چندگانه (MultiDiGraph) از کتابخانه `networkx` است که برای نمایش ماشین حالت استفاده می‌شود. این گراف شامل گره‌ها و یال‌ها است که به ترتیب نشان‌دهنده حالت‌ها و انتقال‌ها بین آن‌ها هستند.



`self.visitedVars:`

این فیلد یک لیست از مقادیر بولین است که نشان می‌دهد آیا هر متغیر در گرامر طی فرآیند تولید ماشین حالت بازدید شده است یا خیر. این فیلد برای تشخیص حالت‌های غیرقابل دسترس استفاده می‌شود.



`self.terminal:`

این فیلد یک لیست است که شامل تمام ترمینال‌های در گرامر است.



`self.ogGrammar:`

این فیلد یک لیست است که گرامر اصلی را به صورت رشته‌های متنی ذخیره می‌کند. این فیلد برای نگهداری نسخه اصلی گرامر قبل از هر گونه تغییر یا پردازش استفاده می‌شود.



`self.ogVariables:`

این فیلد یک لیست است که شامل تمام متغیرهای اصلی در گرامر است. این فیلد نام متغیرهای ورودی را ذخیره می‌کند.

Variable



`self.grammar:`

این فیلد یک شیء از کلاس Grammar است که گرامر اصلی را نشان می‌دهد. این فیلد برای دسترسی به اطلاعات کلی گرامر و متغیرهای دیگر استفاده می‌شود



`self.name:`

این فیلد نام متغیر را ذخیره می‌کند. این نام از قسمت سمت چپ قاعده استخراج می‌شود



`self.RHS:`

این فیلد نشان‌دهنده سمت راست قاعده (Right-Hand Side) است. این بخش شامل تمام قواعدی است که این متغیر می‌تواند به آن‌ها تبدیل شود



`self.all:`

این فیلد یک رشته است که کل قاعده را به صورت کامل ذخیره می‌کند. این شامل نام متغیر و سمت راست قاعده است. از آنجایی که قواعد در مراحل بررسی ممکن است تغییر کنند در نهایت این را معادل اتصال همه قواعد یک متغیر به وسیله | می‌گذاریم.



`self.rules:`

این فیلد یک لیست است که قواعد مربوط به این متغیر را نگهداری می‌کند. هر قاعده با کاراکتر | از دیگر قواعد جدا می‌شود



`self.isLoop:`

این فیلد یک مقدار بولین است که نشان می‌دهد آیا این متغیر شامل یک حلقه (Loop) است یا خیر. اگر سمت راست قاعده با { شروع و با } پایان یابد، این فیلد True خواهد بود



`self.isOptional:`

این فیلد یک مقدار بولین است که نشان می‌دهد آیا این متغیر اختیاری است یا خیر. اگر سمت راست قاعده با [شروع و با] پایان یابد، این فیلد True خواهد بود.



`self.isValid:`

این فیلد یک مقدار بولین است که نشان می‌دهد آیا این متغیر معتبر است یا خیر. این فیلد با استفاده از متند `is_grammar()` مقداردهی می‌شود.



`var_counter:`

این فیلد یک متغیر کلاس (Class Variable) است که برای تولید شناسه‌های منحصر به فرد برای گره‌ها در گراف استفاده می‌شود. هر بار که یک گره جدید ایجاد می‌شود، این مقدار افزایش می‌یابد تا نام تکراری ساخته نشود.

Variable Methods



adjust_loops

در این متود بررسی میکند اگر قاعده ما شامل یک حلقه بود آن را با یک متغیر جایگزین میکند و سپس یک متغیر به قواعد گرامر میافزاید که قانون دست راست آن همان حلقه باشد.



```
def adjust_loops(self):
    if self.isLoop:
        return

    for z in range(len(self.rules)):
        thisRule = self.rules[z].split(",")
        for i in range(len(thisRule)):
            thisRule[i] = thisRule[i].strip()

            if thisRule[i].startswith("{"):
                if thisRule[i].endswith("}"):
                    rand_name = random.randint(0, 9)
                    self.rules[z] = self.rules[z].replace(thisRule[i], str(rand_name))
                    self.grammar.allVars.append(
                        Variable(self.grammar, f"{rand_name}=: {thisRule[i]}"))
                )
            else:
                if i != len(thisRule) - 1:
                    rand_name = random.randint(0, 9)
                    self.rules[z] = self.rules[z].replace(f"{thisRule[i]},{thisRule[i + 1]}", str(rand_name))
                    self.grammar.allVars.append(
                        Variable(self.grammar, f"(rand_name)=:: {thisRule[i]}, {thisRule[i + 1]}"))
                )
```

Variable Methods



adjust_optionals

در این متود بررسی میکند اگر قاعده ما شامل بخش اختیاری باشد بخش اختیاری آن را به یک متغیر جدید تبدیل میکند که قانون دست راست آن همان بخش اختیاری است. و به صورت کلی همه قوانین اختیاری را تبدیل به یک قانون دو جالته میکند و یک حالت جدید که ایپسیلون است را به قواعد آن متغیر می افزاید.



```
def adjust_optionals(self):
    if self.isOptional:
        self.rules[0] = self.rules[0].strip("[")
        self.rules[0] = self.rules[0].strip("]")
        self.rules.append("\\"\\\"")
        return

    for z in range(len(self.rules)):
        thisRule = self.rules[z].split(",")
        for i in range(len(thisRule)):
            thisRule[i] = thisRule[i].strip()

            if thisRule[i].startswith("["):
                if thisRule[i].endswith("]"):
                    rand_name = random.randint(0, 9)
                    self.rules[z] = self.rules[z].replace(thisRule[i], str(rand_name))
                    thisRule[i] = thisRule[i].strip("[")
                    thisRule[i] = thisRule[i].strip("]")
                    self.grammar.allVars.append(
                        Variable(self.grammar, f"{rand_name}=: {thisRule[i]} | \\\"\\\"")
                )

            else:
                if i != len(thisRule) - 1:
                    rand_name = random.randint(0, 9)
                    self.rules[z] = self.rules[z].replace(f"{thisRule[i]},{thisRule[i + 1]}", str(rand_name))
                    thisRule[i] = thisRule[i].strip("[")
                    thisRule[i + 1] = thisRule[i + 1].strip("]")
                    self.grammar.allVars.append(
                        Variable(self.grammar, f"{rand_name}=: {thisRule[i]},{thisRule[i + 1]} | \\\"\\\"")
                )
```

Variable Methods



find_terminals

این متود ترمینال های موجود در دست راست هر متغیر را پیدا میکند و در مجموعه ترمینال های گرامر قرار میزهد.



```
def find_terminals(self):
    for rule in self.rules:
        this_rule = rule.split(",")
        for token in this_rule:
            token.strip(" ")
            if token.startswith("{") or token.endswith("}"):
                token = token.strip("{")
                token = token.strip("}")
            elif token.startswith("[") or token.endswith("]"):
                token = token.strip("[")
                token = token.strip("]")
            if is_terminal(token) and not self.grammar.terminals.__contains__(token):
                self.grammar.terminals.append(token)
```

دريافت و تجزيه گرامر ورودی



در ابتداي برنامه، يك شئ از کلاس گرامر ساخته مي شود که نشان دهنده گرامر ورودی کاربر خواهد بود و ماشين حالت نهايی بر اساس اين شئ ساخته خواهد شد. در ادامه تعداد متغير های گرامر از کاربر گرفته مي شود . سپس هرکدام از متغير ها و قوانينش در يك خط به فرم EBNF از کاربر دريافت مي شود. در اين مرحله يك شئ از کلاس Variable با ورودی دريافت شده از کاربر ساخته مي شود تا پس از بررسی، به گرامر اضافه شود. در ادامه متغير داده شده تحليل مي شود و اطلاعاتی از جمله نام متغير، مجموعه قوانین، لوپ داشتن، قطعیت، معتبر بودن، اختياری بودن و ترمینال هايش از آن استخراج مي شود. سپس منظم بودن متغير داده شده بررسی مي شود و در صورت منظم نبودن، اروى مبتنی بر عدم امكان طراحی برای ماشين حالت برای اين گرامر نمایش داده مي شود و برنامه متوقف مي شود. در صورت منظم بودن، اين متغير به لیست همه قانون های گرامر اصلی ما اضافه مي شود و خواندن ورودی ادامه پيدا مي کند.

- روش تشخيص منظم بودن قانون:

```
● ● ●

def is_regular(self):
    for i in range(len(self.rules)):
        thisRule = self.rules[i].split(",")
        for j in range(len(thisRule)):
            thisRule[j] = thisRule[j].strip(" ")
            if not thisRule[j].startswith("\""):
                if self.name in thisRule[j] and j != 0 and j != len(thisRule) - 1:
                    return False
    return True
```

دريافت و تجزيه گرامر ورودی



طبق کد، برای هر قانون موجود در متغیر فعلی، ترمینال ها و متغیر های موجود در قانون جدا می شوند و اگر خود متغیر در قانون وجود داشته باشد و هم بعد و هم قبل از خودش یک ترمینال یا متغیر دیگر وجود داشته باشد، تشخيص داده می شود که این متغیر منظم نیست و اگر هیچ قانونی این ویژگی را نداشته باشد متغیر منظم است.

در ادامه روند دریافت گرامر ورودی، گرامر اصلاح می شود تا در لیست متغیر هایش متغیر شروع در جایگاه اول قرار گیرد. در مرحله آخر تجزیه گرامر ورودی، بررسی می شود که گرامر داده شده درست است یا خیر و اگر غلط باشد اروری مبتنی بر غلط بودن گرامر داده شده نمایش داده می شود و برنامه متوقف می شود.

- نحوه تشخیص غلط بودن گرامر

```
● ● ●

ans = False
for rule in start.rules:
    ans = ans or start.ends_with_terminal(rule)

if not ans:
    raise Exception("this grammar is not
correct")
```



```

def ends_with_terminal(self, token):
    token = token.strip(" ")
    if token.startswith("{") and token.endswith("}"):
        token = token.strip("{")
        token = token.strip("}")
    elif token.startswith("[") and token.endswith("]"):
        token = token.strip("[")
        token = token.strip("]")
    if is_terminal(token):
        return True
    if "," not in token:
        var = self.grammar.find_var(token.strip(" "))
        if var is None:
            raise Exception("Not all variables are
defined")    return var.isValid
    thisRule = token.split(",")
    ans = True
    for section in thisRule:
        ans = ans and self.ends_with_terminal(section)
    return ans

```

طبق کد، بررسی می شود که با آغاز از متغیر شروع، راهی برای ساخت یک رشته وجود دارد یا نه. اگر به ازای هر انتخاب ممکن همواره فقط متغیر های جدید تولید شوند و درواقع ای برای اتمام این روند وجود نداشته باشد، نتیجه گرفته می شود که گرامر داده شده معتبر نیست و خطای نمایش داده می شود. برای مثال این گرامر معتبر نیست:

$S \rightarrow Sa$

زیرا هیچ قانون قابل دسترسی وجود ندارد که فقط به ترمینال برود و هیچ وقت روند ساخت یک رشته از این گرامر تمام نمی شود و عملاً یک لوب بی نهایت است.

بعد از انجام این مراحل گرامر داده شده توسط کاربر به طور کامل تجزیه و تحلیل شده و آماده تبدیل به ماشین حالت است.

تولید ماشین حالت

برای تولید ماشین حالت یک گرامر از متغیر شروع آن شروع میکنیم.
و تابع generate_var را روی آن متغیر صدا میکنیم.

- متود **generate_var** : عملکرد کلی این متود به این صورت است که به ازای هر متغیر حالت های مورد نیاز آن را به عنوان گره گراف تولید میکند و با رنگ های مختلف بین این حالت ها تفاوت قائل میشود سپس با صدا زدن تابع المسیر های موجود از آن گره را رسم میکند.

پارامتر ها :

- گرافی که ماشین حالت در آن ایجاد میشود (nx.MultiDiGraph)

• یک نشانه اختیاری که برای نامگذاری گرهها استفاده میشود token

مراحل کار :

1. ایجاد شناسه منحصر به فرد هر گره در گراف باید یک شناسه منحصر به فرد داشته باشد. برای این کار، از یک شمارنده (var_counter) استفاده میشود که با هر نار فراخوان، این متده که واحد افزایش میباید

```
● ● ●  
unique_id = Variable.var_counter  
Variable.var_counter += 1
```

2. بررسی اعتبار متغیر:

در حالتی که یک متغیر معتبر نباشد یا به اصطلاح باعث ایجاد حالت مرده شود نام و رنگ آن را مخصوص این موضوع تعیین میکنیم.

```
● ● ●  
  
if not self.isValid:  
    node_name = f"dead_{unique_id}" if token is None else f"dead_{unique_id}{token}"  
    node_color = "blue"  
    graph.add_node(node_name, color=node_color)  
    return f"{node_name}"
```

تولید ماشین حالت



3. ایجاد گره اصلی:

اگر متغیر معتبر باشد، دو گره اصلی ایجاد می‌شود. یکی گره ای است که هنگام ورود به آن متغیر به آن وارد می‌شویم و دیگری گره ای است که برای خروج از آن متغیر داریم در واقع هنگامی که به انتهای همه قانون‌های آن رسیدیم به این می‌رویم. نام گره‌ها ترکیبی از نام متغیر و شناسه منحصر به فرد است. رنگ گره نیز بر اساس نوع متغیر تعیین می‌شود:

- سبز روشن (lightgreen) برای متغیر شروع.
- آبی روشن (lightblue) برای متغیرهای معمولی.
- صورتی (pink) برای حالت پایانی است و حالت پایانی در واقع حالت نهایی (گره نهایی) حالت شروع است.

در صورتی که یک متغیر حلقه باشد گره پایان آن و گره شروع آن یکسان هستند پس در واقع یک گره برای آن ایجاد می‌کنیم و اگر متغیر شروع حلقه بود پس حالت شروع و پایان یکی هستند و در واقع گره شروع را صورتی قرار میدهیم.

4. فراخوانی متدهای iterate_through_rules

پس از ایجاد گره‌ها، متدهای iterate_through_rules فراخوانی می‌شود تا قواعد متغیر تجزیه و تحلیل شوند و یال‌های مربوطه به گراف اضافه شوند.



```
node_name = f"{self.name}_{unique_id}" if token is None else f"{self.name}_{unique_id}{token}"
node_color_s = "lightgreen" if self.name == self.grammar.start else "lightblue"
node_color_s = "pink" if self.name == self.grammar.start and self.isLoop else node_color_s
node_color_f = "pink" if self.name == self.grammar.start else "lightblue"

graph.add_node(node_name, color=node_color_s)

if self.isLoop:
    self.iterate_through_rules(graph, node_name, node_name)
else:
    final_node = f"{node_name}_f"
    graph.add_node(final_node, color=node_color_f)
    self.iterate_through_rules(graph, node_name, final_node)
```

تولید ماشین حالت

۵. این تابع نام گره ساخته شده را برمیگرداند تا در مراحل بعدی که مثلا در متود `iterate_through_rules` فراخوانی میشود وقتی میخواهیم گره ها را به یکدیگر وصل کنیم و یا ل تولید کنیم نام متغیر ها را داشته باشیم



```
def generate_var(self, graph, token=None):
    unique_id = Variable.var_counter
    Variable.var_counter += 1
    var = self.grammar.find_var(self.name)
    self.grammar.visitedVars[self.grammar.allVars.index(var)] = True

    if not self.isValid:
        node_name = f"dead_{unique_id}" if token is None else f"dead_{unique_id}{token}"
        node_color = "blue"

        graph.add_node(node_name, color=node_color)
        return f"{node_name}"

    node_name = f"{self.name}_{unique_id}" if token is None else f"{self.name}_{unique_id}{token}"
    node_color_s = "lightgreen" if self.name == self.grammar.start else "lightblue"
    node_color_s = "pink" if self.name == self.grammar.start and self.isLoop else node_color_s
    node_color_f = "pink" if self.name == self.grammar.start else "lightblue"

    graph.add_node(node_name, color=node_color_s)

    if self.isLoop:
        self.iterate_through_rules(graph, node_name, node_name)
    else:
        final_node = f"{node_name}_f"
        graph.add_node(final_node, color=node_color_f)
        self.iterate_through_rules(graph, node_name, final_node)

    return node_name
```

تولید ماشین حالت

• متود **iterate_through_vars** : عملکرد کلی این متود به این صورت است که در یک حلقه تک تک قواعد مختلف هر متغیر را بررسی میکند. در صورتی که متغیری در قواعد سمت راست باشد دوباره گره های مورد نیاز را تولید میکند و با استفاده از ترمینال ها و اپسیلون یال های بین این گره ها را ایجاد میکند.

پارامتر ها :

- گرافی که ماشین حالت در آن ایجاد می شود.
- گره قبلی که قواعد مختلف این متغیر باید از آن سرچشمه بگیرند و مسیر ها را تولید کنند.
- گره پایانی که نقطه پایان مسیر های قواعد متفاوت این متغیر باید باشد.

مراحل کار :

1. تجزیه قوانین:

هر قانون از الحق چند متغیر یا چند ترمینال در یک دیگر ساخته شده است. برای جداسازی این بخش ها هر قانون را نسبت به ، جدا سازی میکنیم

```
● ● ●  
for i in range(len(self.rules)):  
    thisRule = self.rules[i].split(",")
```

2. پردازش هر بخش از قاعده:

برای هر بخش از قاعده، بررسی می شود که آیا ترمینال است یا غیرترمینال:

- اگر ترمینال باشد، به برچسب انتقال (transition) اضافه می شود.
- اگر غیرترمینال باشد، متده `generate_var` برای آن فراخوانی می شود تا گره مربوطه ایجاد شود.

تولید ماشین حالت



```
for j in range(len(thisRule)):
    thisRule[j] = thisRule[j].strip()
    if is_terminal(thisRule[j]):
        transition = f"\'{transition}{thisRule[j].strip('\'')}\'"
    else:
        var = self.grammar.find_var(thisRule[j])
        node_name = var.generate_var(graph, "\'")
```

3. اتصال گره ها:

پس از پردازش هر بخش، یالها به گراف اضافه می شوند. اگر بخشی از قاعده منجر به حالت مرده (dead) شود، یال به گره مرده اضافه می شود. و یک flag تعریف کرده ایم از قبل که در آن بررسی کنیم اگر در حالت مرده رفته بودیم آن گره به گره نهایی که در ورودی داده شده بود نزود.

اگر قاعده منجر به حالت مرده نشود، یال بین گره قبلی و گره جدید ایجاد می شود. سپس گره جدید به عنوان گره قبلی برای بخش بعدی قاعده در نظر گرفته می شود.

```
if "dead" in node_name:
    graph.add_edge(prev_var, node_name, label=transition)
    death_of_rule = True
    continue
graph.add_edge(prev_var, node_name, label=transition)
prev_var = f"{node_name}_f" if f"{node_name}_f" in graph.nodes else f"{node_name}"
```

4. اتصال به گره نهایی:

هنگامی که حلقه تمام می شود و همه بخش های یک قاعده طی می شوند. باید گره نهایی این یک قاعده را به گره آخری که به عنوان ورودی متود داده بودیم وصل کنیم تا این مسیر به پایان خودش برسد.

```
if not death_of_rule:
    graph.add_edge(prev_var, last_var, label=transition)
```

تولید ماشین حالت



```
def iterate_through_rules(self, graph, prev_var, last_var):
    og_prev = prev_var
    og_last = last_var
    for i in range(len(self.rules)):
        thisRule = self.rules[i].split(",")
        transition = ""
        prev_var = og_prev
        last_var = og_last
        death_of_rule = False
        for j in range(len(thisRule)):
            thisRule[j] = thisRule[j].strip("{}")
            thisRule[j] = thisRule[j].strip(" ")
            thisRule[j] = thisRule[j].strip(" ")
            if is_terminal(thisRule[j]):
                transition = transition.strip("\\" )
                thisRule[j] = thisRule[j].strip("\\" )
                transition = f"\\"{transition}{thisRule[j]}\\""
            else:
                var = self.grammar.find_var(thisRule[j])
                if var is None:
                    raise Exception("Undefined Variable")
                node_name = var.generate_var(graph, "\\'")
                if "dead" in node_name:
                    transition = transition.strip(" ")
                    transition = transition if transition != "" and transition != "\\"\\\" else "eps"
                    graph.add_edge(prev_var, node_name, label=transition)
                    death_of_rule = True
                    continue
                transition = transition if transition != "" and transition != "\\"\\\" else "eps"
                graph.add_edge(prev_var, node_name, label=transition)
                prev_var = f"\'{node_name}_f" if f"\'{node_name}_f" in graph.nodes else f"\'{node_name}"
                transition = ""
        if not death_of_rule:
            transition = transition if transition != "" and transition != "\\"\\\" else "eps"
            graph.add_edge(prev_var, last_var, label=transition)or_f)
        self.iterate_through_rules(graph, node_name, final_node)

    return node_name
```

نمایش گراف یک ماشین حالت



متدهای generate_state_machine مسئول ایجاد گراف ماشین حالت، نمایش آن به صورت تصویری و ذخیره جزئیات مربوط به ماشین حالت در یک فایل متنی است. این متدها مراحل زیر را انجام می‌دهند:

مراحل کار:

۱. مقدار دهنده حالت‌های بازدید شده:

در شروع کار همه متغیرها را بازدید نشده فرض می‌کنیم تا سپس هنگام طی کردن مسیرهای مختلف از متغیر شروع هر متغیر را که دیدیم (در تابع generate_var) به عنوان بازدید شده علامت می‌زنیم تا با این کار unreachable_states را بیابیم.

۲. شروع تولید ماشین حالت:

با صدا زدن تابع generate_var بر روی متغیر شروع ماشین حالت این گرامر را به صورت یک گراف می‌سازیم.

۳. رسم ماشین حالت:

برای رسم آن از کتابخانه‌های networkx.drawing.nx_pydot و matplotlib.pyplot استفاده کرده‌ایم.

از آنجایی که می‌خواهیم هر گره گراف با همان رنگ تعیین شده و هر یال با همان label تعیین شده که در واقع transition بین دو حالت بود رسم شود پس باید این ها را هنگام رسم تعیین کنیم

رنگ گره‌ها را در یک لیست ذخیره می‌کنیم و سپس گره‌ها و یال‌ها را با ویژگی‌های تعیین شده رسم می‌کنیم.

```
● ● ●  
node_colors = [data["color"] for _, data in self.G.nodes(data=True)]  
pos = nx.spring_layout(self.G)  
nx.draw(self.G, pos, with_labels=True, node_size=2000, node_color=node_colors, font_size=10)  
nx.draw_networkx_edge_labels(self.G, pos, edge_labels=nx.get_edge_attributes(self.G, "label"))
```

نمایش گراف یک ماشین حالت



4. ذخیره گراف به صورت تصویر :

گراف رسم شده به صورت یک فایل تصویری با فرمت PNG ذخیره می شود. برای این کار، گراف به فرمت pydot تبدیل می شود و سپس به تصویر تبدیل می گردد.

```
pydot_graph = to_pydot(self.G)
pydot_graph.set_graph_defaults(dpi=300)
pydot_graph.write_png("state_machine.png")
```

5. نمایش گراف به صورت تصویر :

تصویر ذخیره شده با استفاده از matplotlib نمایش داده می شود. محورها نیز غیرفعال می شوند تا گراف به صورت تمیز نمایش داده شود.

```
plt.clf()
img = plt.imread("state_machine.png")
plt.imshow(img)
plt.axis("off")
plt.show()
```

6. صدا زدن متود : write_results_to_file

که در واقع خود گرامر، متغیر ها، ترمینال ها و ویژگی های تعیین شده در مرحله گزارش کار را در یک فایل با نام state_machine_details.txt ذخیره می کند.

نمایش گراف یک ماشین حالت



```
def generate_state_machine(self):
    self.set_visited()
    start = self.allVars[0]
    start.generate_var(self.G, "")
    node_colors = [data["color"] for _, data in self.G.nodes(data=True)]
    pos = nx.spring_layout(self.G)
    nx.draw(self.G, pos, with_labels=True, node_size=2000, node_color=node_colors, font_size=10)
    nx.draw_networkx_edge_labels(self.G, pos, edge_labels=nx.get_edge_attributes(self.G, "label"))

    pydot_graph = to_pydot(self.G)

    for node in pydot_graph.get_nodes():
        node_name = node.get_name()
        color = self.G.nodes[node_name]["color"]
        node.set_style("filled")
        node.set_fillcolor(color)

    pydot_graph.set_graph_defaults(dpi=300)
    pydot_graph.write_png("state_machine.png")
    plt.clf()
    img = plt.imread("state_machine.png")

    plt.imshow(img)
    plt.axis("off")
    plt.show()
    self.write_results_to_file()
```

تولید خروجی و خطایابی

در این مرحله ماشین حالت تولید شده در قالب یک تصویر PNG ذخیره می شود که در مرحله قبلی توضیح داده شد. همچنین خطایابی ها باید انجام شود که در برنامه ما در مرحله اول هنگام دریافت ورودی ها، بررسی شده بود که گرامر داده شده معتبر و منظم باشد و خطا های مورد نظر در همان مرحله نمایش داده می شد و کاربر مطلع می شد که مشکل در کدام بخش از گرامر است.

تحليل و گزارش ماشین حالت



در این مرحله گزارشی از گرامر داده شده و ماشین حالت تولید شده برای کاربر ساخته می شود که شامل اطلاعاتی مانند ترمینال ها و متغیر های گرامر داده شده و همچنین اطلاعاتی از ماشین حالت مانند تعداد وضعیت ها، تعداد انتقال ها، وجود متغیر غیرقابل دسترس، حالت های مرده و قطعیت ماشین حالت است. یک نمونه از گزارش تولید شده توسط برنامه:

```
Grammar :  
S =:: A|B  
A =:: A  
B =:: "b"  
  
Variables :  
S, A, B  
  
Terminals :  
b  
  
Number of States: 5  
Number of Transitions: 4  
Unreachable Variables: None  
Dead States: dead_1'  
is Grammar Deterministic : True  
is State Machine Deterministic : False
```

نحوه پردازش هرکدام از اطلاعات:

- متغیر ها و ترمینال های گرامر

این گزارش از اطلاعاتی که در مراحل ابتدایی برنامه از گرامر و متغیر ها استخراج شده بود به دست می آید و مستقیماً نمایش داده می شود.

تحليل و گزارش ماشین حالت

- تعداد وضعیت و انتقال ها

تعداد وضعیت ها در واقع همان تعداد راس های گراف تولید شده هستند و تعداد انتقال ها همان تعداد یال های گراف تولید شده هستند که به این صورت استخراج می شوند:

```
● ● ●

def num_of_states(self):
    return
len(self.G.nodes)
def num_of_transitions(self):
    return
len(self.G.edges)
```

- متغیر های غیرقابل دسترس

هر متغیری که راهی وجود نداشته باشد تا با آغاز از متغیر شروع به آن برسیم، متغیر غیرقابل دسترس محسوب می شود و در این بخش نمایش داده می شود. برای پیدا کردن این متغیر ها ما در مراحل ساخت ماشین حالت، هر متغیری که در طول پروسه دیده شده بود را به عنوان راس قابل دسترس علامت گذاری کردیم و در نهایت متغیرهایی که به عنوان دیده شده علامت گذاری نشده بودند را به عنوان متغیر غیرقابل دسترس شناسایی کردیم و در این بخش نمایش دادیم.

تحليل و گزارش ماشین حالت



- **حالت های مرده**

در این بخش از گزارش نام وضعیت های مرده ماشین حالت نمایش داده می شود. وضعیت های مرده وضعیت هایی هستند که حالت نهایی نیستند و در صورت ورود به آنها راه خروجی وجود ندارد و اصطلاحاً ماشین fail می شود. برای پیدا کردن این حالت ها در کد ما بین همه راس های گراف، راس هایی را که حالت نهایی نبودند و هیچ یالی از آنها خارج نمیشد را پیدا کردیم و به عنوان وضعیت مرده علامت گذاری کردیم.

```
def dead_states(self):
    deadStates = [node for node in self.G.nodes()
                  if self.G.out_degree(node) == 0
                  and self.G.nodes[node].get("color") != "pink"]
    return deadStates
```

- **قطعیت ماشین حالت**

در این بخش از گزارش نام وضعیت های مرده ماشین حالت نمایش داده می شود. یک ماشین حالت در حالتی قطعی نیست که بتواند با خواندن یک کاراکتر ثابت از یک حالت مشخص به دو حالت برود. برای بررسی این موضوع باید همه حالت های این ماشین را بررسی کنیم و در صورت وجود دو یال با نام یکسان این ماشین قطعی نیست.

تحلیل و گزارش ماشین حالت



```
def has_duplicate_edge_labels(self):
    for node in self.G.nodes():
        label_to_targets = {}

        for _, target, data in self.G.out_edges(node, data=True):
            label = data.get("label")

            if label in label_to_targets:

                if target not in label_to_targets[label]:
                    return True
                else:
                    label_to_targets[label] = set()

            label_to_targets[label].add(target)

    return False
```

```
def is_machine_deterministic(self):
    return not self.has_duplicate_edge_labels()
```

باید این را نیز در نظر بگیریم که قطعیت ماشین حالت و قطعیت گرامر دو بحث متفاوت هستند یک گرامر ممکن است غیر قطعی نباشد ولی ماشین حالت تولید شده برای آن غیر قطعی باشد.

برای بررسی قطعیت یک گرامر باید حالت های مختلف هر قانون (هر متغیر) که با "ا" از هم جدا شده اند را بررسی کنیم. در صورتی که با کاراکتر اول ترمینال یکسان شروع شوند قطعی نیست در صورتی که حداقل یکی از آن ها با یک متغیر شروع شود باید کاراکتر اول حالت های مختلف قانون دست راست آن متغیر را بررسی کنیم و اگر کاراکتر اول یکی از این حالت ها با کاراکتر اول حالت های قبلی برابر بود یعنی گرامر نیز قطعی نیست.

تحلیل و گزارش ماشین حالت



```
def is_deterministic(self, first_chars):
    for i in range(len(self.rules)):
        if self.rules[i].startswith("\""):
            if first_chars.__contains__(self.rules[i][1]):
                return first_chars, False
            first_chars.append(self.rules[i][1])
        else:
            rule_sections = self.rules[i].split(",")
            rule_sections[0] = rule_sections[0].strip(" ")
            rule_sections[0] = rule_sections[0].strip("{")
            rule_sections[0] = rule_sections[0].strip("}")
            rule_sections[0] = rule_sections[0].strip(" ")
            rule_sections[0] = rule_sections[0].strip("[")
            rule_sections[0] = rule_sections[0].strip("]")
            rule_sections[0] = rule_sections[0].strip(" ")
            var = self.grammar.find_var(rule_sections[0])
            first_chars, status = var.is_deterministic(first_chars)
            if not status:
                return first_chars, False
    return first_chars, True
```



```
def is_deterministic(self):
    for i in range(len(self.allVars)):
        first_chars = []
        if not self.allVars[i].is_deterministic(first_chars)[1]:
            return False
    return True
```