دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

طراحی کامپایلر

گزارش پروژه طراحی کامپایلر برای زبان PL

تاریخ:

۱۴۰۳/۰۴/۱۱

ارشیا شفیعی ۴۰۰۳۶۲۳۰۱۹

کیمیا میرمقتدایی ۴۰۱۲۳۶۳۰۹۱

شرح پروژه:

در این پروژه قصد داریم برای یک زبان برنامه‌نویسی مشابه زبان C یک کامپایلر طراحی و پیاده‌سازی کنیم. یک کامپایلر از تحلیل‌گر لغوی، تحلیل‌گر نحوی، تحلیل‌گر معنایی، تولیدکننده و بهنیه‌ساز کد میانی، تولیدکننده و و بهینه‌شاز کد اسمبلی تشکیل شده است. در این مرحله قصد داریم تا تحلیل‌گر لغوی زبان تعریف شده را طراحی و پیاده‌سازی کنیم. به این منظور از زبان پایتون بهره برده‌ایم. همچنین از هیچ کتاب‌خانه‌ی فرعی برای این منظور استفاده نشده است.

بخش اول (Lexical Analyzer):

رویکرد کلی:

رویکرد کلی در این پروژه تعریف توابعی برای بررسی نوع هر توکن (با توجه به انواع توکن‌های تعریف شده در صورت پروژه) تعریف کرده‌ایم. هربار یک خط از برنامه‌ی نوشته شده را می‌خوانیم و در آن کاراکتر به کاراکتر جلو می‌رویم و این توابع را به ترتیب داخل یک حلقه تا پایان برنامه فراخوانی می‌کنیم. همه‌ی این توابع –علاوه‌بر ویژگی‌های لازم برای هر توکن- یک مقدار True یا False برمی‌گردانند که نشان می‌دهد عبارت فرستاده شده از نوع توکن مربوطه هست یا خیر.

هربار مقدار یک تابع False برگرداند، تابع بعدی در دنباله‌ی توابع اجرا خواهد شد. در صورت True بودن یک تابع، توکن ما از نوع مربوط به آن تابع تشخیص داده می‌شود و بنابراین توکن آن را می‌سازیم و در خروجی قرار می‌دهیم.

بررسی یک به یک توابع:

1. تابع get_tokens:

در این تابع به ازای هر خط داخل کد، متغیر beg را تعریف می‌کنیم که در عمل اشاره‌گری به کاراکتری از خط است که هنوز به توکنی تعلق ندارد. در هر مرحله تا پایان هر خط، beg را یکی یکی زیاد می‌کنیم تا به پایان خط برسیم. هربار با شروع از beg تا پایان خط را به یکی از توابع تشخیص‌دهنده‌ی توکن‌ها می‌دهیم تا در صورت امکان توکن درست را استخراج کنیم.

در صورتی که با شروع از beg توکنی پیدا نشود، مقدار آن را یکی زیاد می‌کنیم و دوباره امتحان می‌کنیم.

```
def get_tokens():
    tokens = []
    count = 0
    for line in read_file_line("tests/test5.txt"):
        count += 1
        beg = 0
        while(beg < len(line)):
            if is_comment(line[beg:]):
                yield Token("T_Comment", count, line[beg + 2:])

                beg = len(line)
            elif is_whitespace(line[beg:beg + 1]):
                yield Token("T_Whitespace", count, line[beg:beg + 1])

            elif is_delimiter(line[beg:beg + 1])[0]:
                yield Token(is_delimiter(line[beg:beg + 1])[1], count, line[beg:beg + 1])

            elif is_keyword(line[beg:])[0]:
                yield Token("T_" + is_keyword(line[beg:])[1], count, None)

                beg += len(is_keyword(line[beg:])[1]) - 1
            elif is_identifier(line[beg:])[0]:
                yield Token("T_ID", count, is_identifier(line[beg:])[1])

                beg += len(is_identifier(line[beg:])[1]) - 1
            elif is_operator(line[beg:])[0]:
                operator = is_operator(line[beg:])[1]
                token_name = get_token_name(operator)


                yield Token(token_name, count, operator)

                beg += len(operator) - 1

            elif is_litnum(line[beg:])[0]:
                _ , token_name, number= is_litnum(line[beg:])
                yield Token(token_name, count, number)

                beg += len(number) - 1
            elif is_litstring(line[beg:])[0]:
                _ , token_name, word = is_litstring(line[beg:])
                yield Token(token_name, count, word)
                beg += len(word) - 1
            beg += 1
    return tokens
```

شکل ۱- کد تابع get tokens

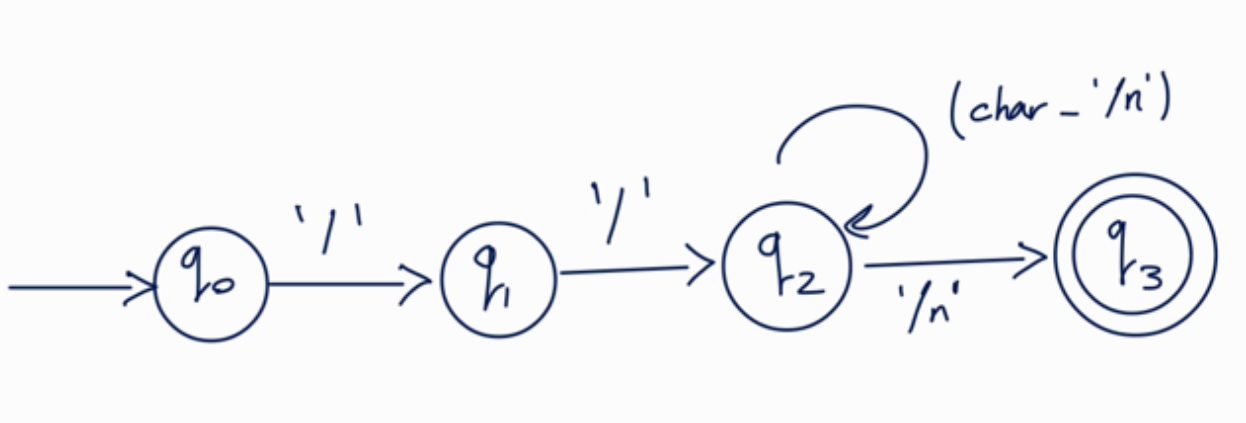
2. تابع is_comment:

- عبارت منظم:

//(char - \n)*\n

در اینجا char تمام کاراکترهای موجود است.

- دیاگرام گذار:



- کد: این تابع وظیفه تشخیص توکن‌های کامنت را دارد. برای این کار رشته داده شده به تابع را کاراکتر به کاراکتر بررسی می‌کنیم و هربار که کاراکتر / را تشخیص می‌دهیم استیت را عوض می‌کنیم. بعد از دو بار به / رسیدن به استیت پایانی می‌رسیم و True برمی‌گردانیم در غیر این‌صورت False برمی‌گردانیم.

```python
def is_comment(token: str):
    state = 0
    for char in token:
        if state == 0:
            if char == "/":
                state = 1
        elif state == 1:
            if char == "/":
                state = 2
        elif state == 2:
            return True
    return False
```

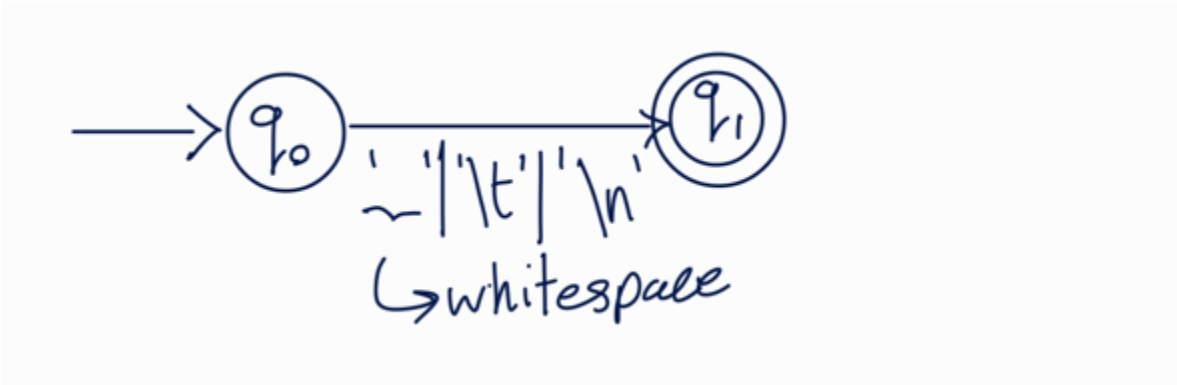
شکل ۲- تابع is_comment

3. تابع is_whitespace:

- عبارت منظم:

(‘ ’|‘\t’|‘\n’)

- دیاگرام گذار:



- **کد:** در این تابع قصد داریم توکن‌های tab ، white space و new line را استخراج کنیم.
برای این‌کار از کد اسکی این کاراکترها استفاده می‌کنیم.

```
def is_whitespace(token: str):
        if ord(token) == 32 or ord(token) == 10 or ord(token) == 9:
            return True
        else:
            return False
```

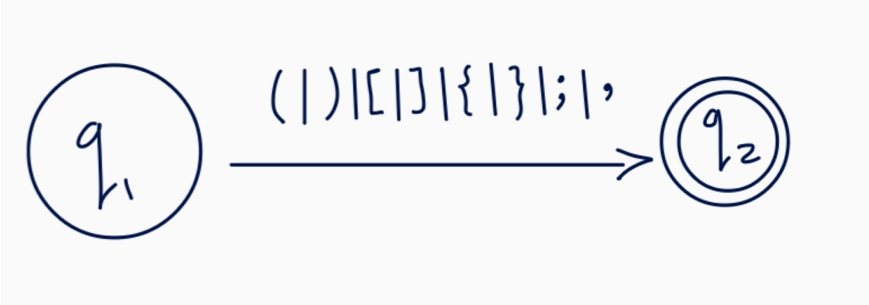
شکل - تابع is_whitespace

4. تابع is_delimeter:

- عبارت منظم:

```
[|]|{|}|(|)|,|;
```

- دیاگرام گذار:



- **کد:** این تابع وظیفه تشخیص توکن‌های جدا کننده semicolon، comma، پرانتزها، براکت‌ها و کروشه‌ها را دارد. همیشه دقیقا یک کاراکتر به این تابع داده می‌شود و بعد از تشخیص وجود توکن، اسم آن را با استفاده از تابع get_token_name می‌گیریم.
  در خروجی تابع در صورت تشخیص عبارت، یک جفت True و نام توکن را برمی‌گردانیم.
  در غیر این‌صورت False و None برمی‌گردانیم.

```python
def is_delimiter(token: str):
    token_name = get_token_name(token)

    if token == '[' or token == ']' or token == '(' or token == ')' \
        or token == '{' or token == '}' or token == ';' or token == ',':
        return True, token_name
    else:
        return False, None
```

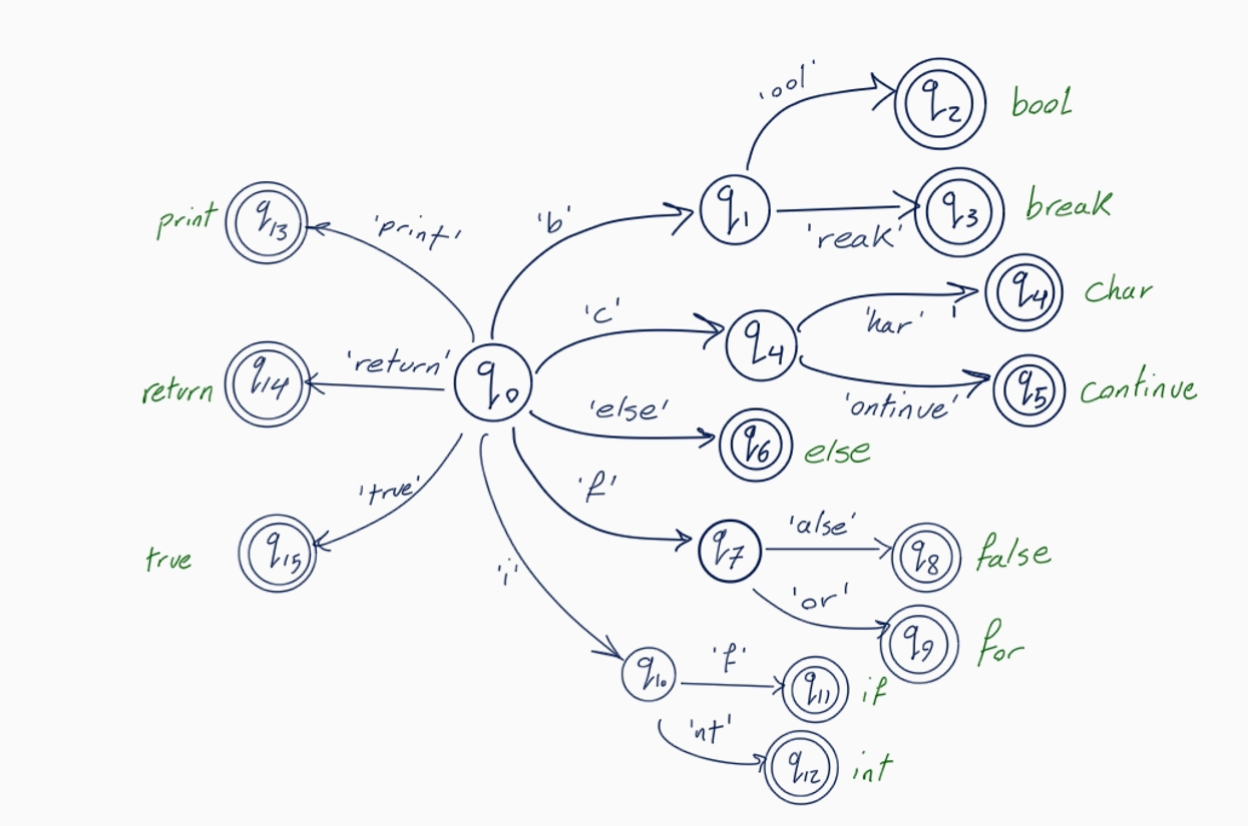
شکل - تابع is_delimiter

5. تابع is_keyword:

- عبارت منظم:

(bool|break|char|continue|else|false|for|if|int|print|return|true)

- دیاگرام گذار:



- کد: در این تابع قصد داریم تا تشخیص دهیم که با شروع از کاراکتر beg ام، آیا یک توکن از نوع keyword داریم یا خیر. به این منظور این مسئله را در شرط‌هایی چک می‌کنیم.

```python
def is_keyword(token: str):
    golabi = get_token_until_delspop(token)

    if golabi == "bool":
        return True, "Bool"
    elif golabi == "break":
        return True, "Break"
    elif golabi == "char":
        return True, "Char"
    elif golabi == "continue":
        return True, "Continue"
    elif golabi == "else":
        return True, "Else"
    elif golabi == "false":
        return True, "False"
    elif golabi == "for":
        return True, "For"
    elif golabi == "if":
        return True, "If"
    elif golabi == "int":
        return True, "Int"
    elif golabi == "print":
        return True, "Print"
    elif golabi == "return":
        return True, "Return"
    elif golabi == "true":
        return True, "True"
    else:
        return False, None
```

شکل - تابع is_keyword

- **تابع get_token_until_delspop:**

کد: این تابع به ما کمک می‌کند تا با شروع از beg تا اولین whitespace یا delimiter و یا operator

را جدا کرده و return می‌کند. با این کار ارزیابی ما راحت‌تر می‌شود.

```python
def get_token_until_delspop(token: str) -> str:
    #print(token)
    index = 0
    for i in range(len(token)):
        if is_whitespace(token[i]) or is_delimiter(token[i])[0] or is_a_operator(token[i]):
            index = i
            break
    return token[0:index]
```

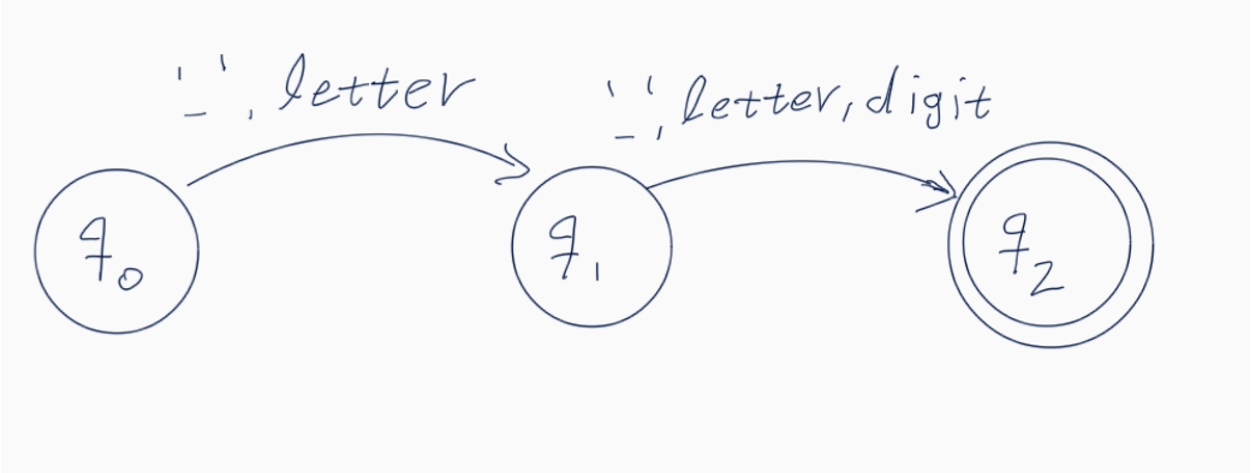
شکل - تابع get_token_until_delspop

6. **تابع is_identifier:**

- عبارت منظم:

('_' | [a-z, A-Z])('_' | [a-z, A-Z] | [0-9])*

- دیاگرام گذار:



- **کد:** این تابع وظیفه تشخیص شناسه‌ها را دارد. یعنی تشخیص ترکیبی از حروف، اعداد و آندرلاین که البته نمی‌تواند با عدد شروع شود. ابتدا با تابع get_token_until_delspop که پیش‌تر توضیح داده شد توکن را تا یک کاراکتر جداکننده، فاصله و یا عملگر جدا می‌کنیم. سپس درصورتی که اولین کاراکتر کلمه _ یا حرف الفبا بود، بقیه کاراکترهای توکن را تک به تک بررسی می‌کنیم. اگر کاراکترها عدد، الفبا یا آندرلاین بود. True و خود توکن را برمی‌گردانیم. در غیر این‌صورت False برمی‌گردانیم. با توجه به اینکه شرط بررسی keyword قبل از این شرط اجرا می‌شود درنتیجه توکن‌های ورودی حتما کلیدواژه از قبل تعریف شده نیستند.

```python
def is_identifier(token: str):
    golabi = get_token_until_delspop(token)
    if len(golabi) == 0:
        return False, None
    if golabi[0] == '_' or golabi[0].isalpha():
        for i in range(len(golabi)):
            if golabi[i] =='_' or golabi[i].isalnum():
                continue
            else:
                return False, None

        return True, golabi
    return False, None
```

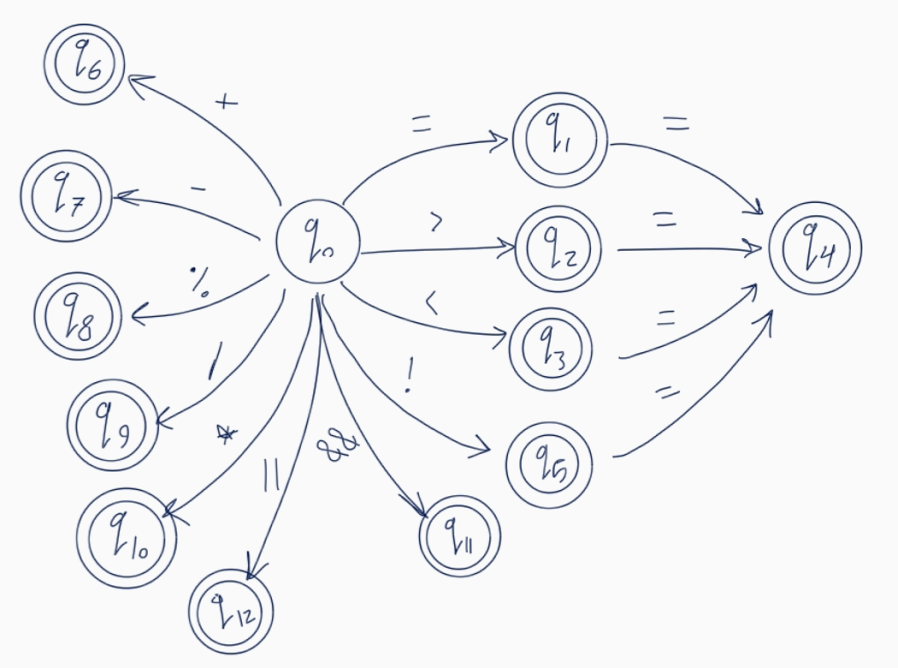
شکل ۷ - تابع is_identifier

7. تابع is_operator:

- عبارت منظم:

(=(=|∈)|>(=|∈)|<(=|∈)|!(=|∈)|\+|-|*|%|/|&&|\|\|)

- دیاگرام گذار:



- **کد:** در این تابع 1 یا 2 توکن پشت سرهم را درنظر می‌گیریم و طبق دیاگرام شرط‌های مربوط به چک کردن عمل‌گر را به ترتیب بررسی می‌کنیم.

```
def is_operator(token: str):
    if token[0] == '=':
        if token[1] == '=':
            return True, "=="
        return True, "="
    elif token[0] == '<':
        if token[1] == '=':
            return True, "<="
        return True, "<"
    elif token[0] == '>':
        if token[1] == '=':
            return True, ">="
        return True, ">"
    elif token[0] == '!':
        if token[1] == '=':
            return True, "!="
        return True, "!"
    elif token[0] == '+':
        return True, "+"
    elif token[0] == '-':
        return True, "-"
    elif token[0] == '*':
        return True, "*"
    elif token[0] == '/':
        return True, "/"
    elif token[0] == '%':
        return True, "%"
    elif token[0:1] == "&&":
        return True, "&&"
    elif ord(token[0]) == 124 and ord(token[1]) == 124:
        return True, "||"
    return False, None
```

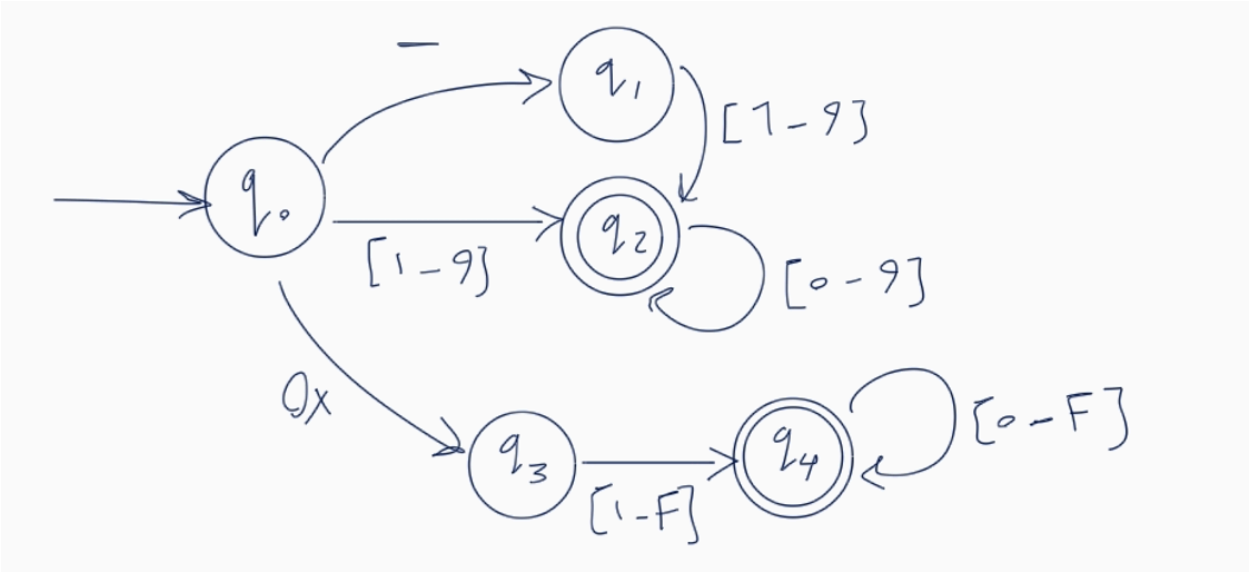
شکل ۸ - تابع is_operator

8. تابع is_litnum:

- عبارت منظم:

(-|∈) [1-9] ([0-9])* | 0X[1-F]([0-F])*

- دیاگرام گذار:



- **کد:** در این تابع قصد داریم اعداد را تشخیص دهیم. اعداد ما یا از نوع hex هستند و یا از نوع int یا عدد صحیح. درصورتی که کاراکتر اول رشته‌ی دریافتی ما از نوع عدد نباشد، می‌دانیم که هیچ‌کدام از حالات بالا برقرار نیستند. اما در غیر این‌صورت می‌توان شرط‌های جلوتر را بررسی کرد. ابتدا تا جایی که به delimiter یا whitespace و یا operator برخورد می‌کنیم را جدا می‌کنیم. برای تشخیص hex بودن عدد از تابع is_hex کمک می‌گیریم

برای اعداد صحیح، درصورت منفی بودن، کاراکتر اول رشته باید معادل – و مابقی کاراکترها باید همگی عددی باشند. اما برای اعداد مثبت عددی بودن همگی کاراکترها شرطی کافی است.

```
def is_litnum(token: str):
    if token[0].isnumeric() or token[0] == '-':
        golabi = get_token_until_delspop(token)

        if is_hex(golabi):
            return True, "T_HexaDecimal", golabi
        elif golabi[0] == '-' and golabi[1:].isnumeric():
            return True, "T_Decimal", golabi
        elif golabi.isnumeric():
            return True, "T_Decimal", golabi
    else:
        return False, None, None
```

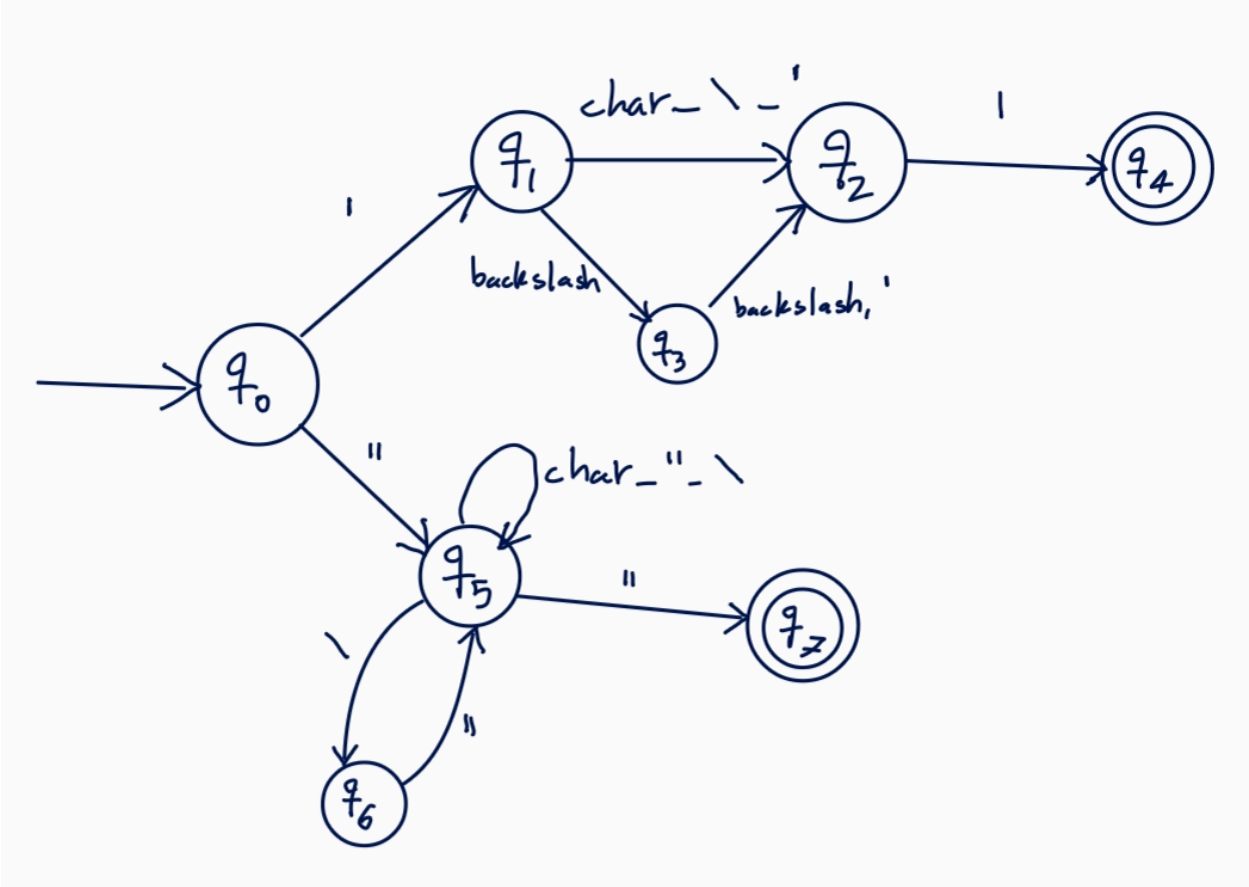
شکل ۹ - تابع is_litnum

9. تابع is_litstring:

- عبارت منظم:

‘(char – (‘,\)| \\ | \’)’ | “((char – (“,\))*| \”)”

- دیاگرام گذار:



- کد: این تابع وظیفه تشخیص توکن‌های استرینگ و تک کاراکتر یعنی عبارت‌هایی که بین دو ‘ یا “ قرار می‌گیرند را دارد. در شرط اول توکن کاراکتر تشخیص داده می‌شود. اگر کاراکتر اول و سوم برابر با ‘ باشد یا کاراکتر اول و چهارم برابر ‘ و کاراکتر دوم \ باشد و کاراکتر سوم یکی از دو کاراکتر \ یا ‘ باشد، توکن تشخیص داده می‌شود و خروجی آن نوع توکن و خود توکن است. در شرط دوم نیز توکن استرینگ تشخیص داده می‌شود. اگر کاراکتر اول برابر با “ باشد، کاراکترهای توکن را تک به تک جلو می‌رویم و هر وقت که دوباره به “ رسیدیم یک کاراکتر عقب‌تر را بررسی می‌کنیم که اگر برابر با \ بود آنگاه توقف نمی‌کنیم و جلوتر می‌رویم.

```python
def is_litstring(token: str):
    if token[0] == "'":
        if token[1] == "\\" and token[3] == "'":
            if token[2] == "'":
                return True, "T_Char", token[:4]
            elif token[2] == "\\":
                return True, "T_Char", token[:4]
        elif token[2] == "'":
            return True, "T_Char", token[:3]

    elif token[0] == '"':
        index = 0
        for i in range(1, len(token)):
            if token[i] == '"':
                if token[i - 1] == '\\':
                    continue
                index = i
                break
        golabi = token[:index + 1]

        return True, "T_String", golabi


    else:
        return False, None, None
```

شکل ۱۰ - تابع is_litstring

در پایان با توجه به مقدار استخراج شده، توکن مربوطه از نوع کلاس توکن ساخته می‌شود و در خروجی قرار می‌گیرد.

بخش دوم (Syntax Analyzer):

پروژه‌ی ما از 9 قسمت اصلی تشکیل شده است.
در فایل Grammar گرامر نوشته شده است. برای نوشتن گرامر از این لینک کمک گرفته‌ایم. هرچند که در مسیر نوشتن هرکجا که احساس کردیم قانون بیشتری نیاز است آن را اضافه کرده‌ایم و قوانین با لینک پیوست شده بسیار متفاوت هستند.

در فایل‌های LeftRecursion و LeftFactoring الگوریتم‌های فاکتورگیری چپ و حذف بازگشت چپ پیاده‌سازی شده‌اند. هردوی این الگوریتم‌ها براساس شبه‌کدهای داخل اسلایدها و کاملا از صفر پیاده‌سازی شده‌اند. الگوریتم فاکتورگیری چپ برای سادگی بیشتر، به جای پیدا کردن بلندترین زیرقانون

مشترک، آن‌ها را یکی یکی فاکتور می‌گیرد. این مسئله ظاهر خروجی را کمی بهم می‌ریزد اما خروجی کاملاً درست است.

در ادامه فایل Fcal قرار دارد که مجدداً طبق شبه‌کدهای داخل اسلایدهای درست به محاسبه‌ی First و Follow ها می‌پردازد. هیچ‌کدام از این مراحل برای تهیه‌ی جدول تجزیه به صورت دستی محاسبه‌نشده‌اند و همگی مستقیماً از طریق کد به دست آمده‌اند.

با استفاده از گرامر خروجی پس از حذف بازگشت از چپ‌ها و گرفتن فاکتورها و همچنین نتایج حاصل از محاسبات First و Follow اطلاعات بدست آمده را به فایل ChackLL1 می‌دهیم تا سه شرط مربوط به LL1 بودن بررسی شود.

نتیجه نشان می‌دهد که گرامر تهیه شده توسط تیم ما LL1 است و بنابراین می‌توانیم First و Follow ها را به فایل ParsingTable بدهیم تا جدول تجزیه‌ی مربوط به آن ساخته شود. در این قسمت بازیابی خطا توسط توکن همگام‌کننده نیز پیاده‌سازی شده است.

درنهایت جدول تجزیه‌ی ثابت ما با توجه به گرامرمان، به شکل زیر خواهد بود:

Certainly! Here is the converted text for you to copy:

Parsing Table:
M[Program, t_int]: Program -> ['Declist']
M[Program, t_char]: Program -> ['Declist']
M[Program, t_bool]: Program -> ['Declist']
M[Program, $]: synch
==========
M[Declist, t_int]: Declist -> ['Dec', "Declist"]
M[Declist, t_char]: Declist -> ['Dec', "Declist"]
M[Declist, t_bool]: Declist -> ['Dec', "Declist"]

M[Declist, $]: synch

==========

M[Declist, t_int]: Declist -> ['Dec', "Declist"]

M[Declist, t_char]: Declist -> ['Dec', "Declist"]

M[Declist, t_bool]: Declist -> ['Dec', "Declist"]

M[Declist, $]: Declist -> ['e']

==========

M[Dec, t_int]: Dec -> ['Type', 't_id', 'Declaration']

M[Dec, t_char]: Dec -> ['Type', 't_id', 'Declaration']

M[Dec, t_bool]: Dec -> ['Type', 't_id', 'Declaration']

M[Dec, $]: synch

==========

M[Declaration, t_assign]: Declaration -> ['Vardec']

M[Declaration, t_lb]: Declaration -> ['Vardec']

M[Declaration, t_comma]: Declaration -> ['Vardec']

M[Declaration, t_semicolon]: Declaration -> ['Vardec']

M[Declaration, t_lp]: Declaration -> ['Funcdec']

M[Declaration, t_char]: synch

M[Declaration, t_bool]: synch

M[Declaration, $]: synch

M[Declaration, t_int]: synch

==========

M[Type, t_int]: Type -> ['t_int']

M[Type, t_bool]: Type -> ['t_bool']

M[Type, t_char]: Type -> ['t_char']

M[Type, t_id]: synch

```
====================================
M[Vardec, t_assign]: Vardec -> ['Vardeclist', 't_semicolon']
M[Vardec, t_lb]: Vardec -> ['Vardeclist', 't_semicolon']
M[Vardec, t_comma]: Vardec -> ['Vardeclist', 't_semicolon']
M[Vardec, $]: Vardec -> ['Vardeclist', 't_semicolon']

M[Vardec, $]: Vardec -> ['Vardeclist', 't_semicolon']
M[Vardec, t_bool]: Vardec -> ['Vardeclist', 't_semicolon']
M[Vardec, t_int]: Vardec -> ['Vardeclist', 't_semicolon']
M[Vardec, t_semicolon]: Vardec -> ['Vardeclist', 't_semicolon']
M[Vardec, t_comma]: Vardec -> ['Vardeclist', 't_semicolon']
```

این قسمتی از جدول تجزیه‌ی تولید شده توسط فایل مضمون می‌باشد. مابقی جدول نیز داخل ترمینال قابل بررسی است.

کد مربوط به تجزیه‌ی پیش‌بینی کننده، بالا به پایین، در فایل PredictiveParser پیاده‌سازی شده است. درخت تجزیه با استفاده از anytree نیز در همین فایل ساخته و چاپ می‌شود.

جدول تجزیه را به این فایل می‌دهیم تا طبق الگوریتم تجزیه شود و درخت تجزیه تولید شود.

خروجی Tokenizer به علاوه‌ی جدول تجزیه، تا طبق الگوریتم تجزیه شود و درخت تجزیه تولید شود.

در صورتی که با خطای نحوی مواجه شدیم، عبارت syntax error با رنگ قرمز داخل ترمینال پرینت می‌شود و تجزیه ادامه پیدا می‌کند.

با توجه به این که گفته شد برای این قسمت به داک جامعی احتیاج نداریم، این داک صرفاً جهت توضیح عملکرد قسمت‌های مختلف تهیه شده‌است. در صورتی که سوالات بیشتری در زمینه‌ی کد مطرح است داک کامل‌تری قابل ارائه خواهد بود.

بخش سوم (Semantic Analyzer):

در این فاز ابتدا با پیمایش درخت پارسر به صورت Preorder، قانون‌های مورد نظر را شناسایی می‌کنیم و بعد از شناسایی آن‌ها قانون معنایی مربوط به آن‌ها را انجام می‌دهیم. مثلا آن‌ها را در سیمبل تیبل اد می‌کنیم.

پیاده‌سازی Symbol Table:

این کلاس برای مدیریت جدول نمادها (Symbol Table) در تحلیل معنایی استفاده می‌شود. جدول نمادها برای نگهداری اطلاعات مربوط به نمادها (متغیرها، توابع و ...) در طول تحلیل معنایی مورد استفاده قرار می‌گیرد.

متدها:

- `enter_scope`: ورود به یک دامنه جدید و افزایش سطح دامنه.
- `exit_scope`: خروج از دامنه فعلی و کاهش سطح دامنه. همچنین نمادهایی که در این دامنه تعریف شده‌اند از جدول نمادها حذف می‌شوند.
- `add`: اضافه کردن یک نماد جدید به جدول نمادها.
- `lookup`: جستجوی یک نماد در جدول نمادها.

```python
class SymbolTable:
    def __init__(self):
        self.table = {}
        self.scope_level = 0

    def enter_scope(self):
        self.scope_level += 1

    def exit_scope(self):
        # Remove all entries that have the current scope level
        self.table = {k: v for k, v in self.table.items() if v[3] < self.scope_level}
        self.scope_level -= 1

    def add(self, name, category, type, attributes=None):
        entry = [name, category, type, self.scope_level, attributes]
        self.table[name] = entry

    def lookup(self, name):
        return self.table.get(name, None)
```

تابع `semantic_analyzer`

این تابع وظیفه تحلیل معنایی درخت نحوی را بر عهده دارد و جدول نمادها را بر اساس اطلاعات موجود در درخت نحوی به‌روزرسانی می‌کند.

مراحل:

1. ایجاد یک نمونه از `SymbolTable`.
2. پیمایش درخت نحوی به صورت پیش‌سفری (PreOrder).
3. بررسی و اضافه کردن نمادها (توابع و متغیرها) به جدول نمادها.
4. مدیریت دامنه‌ها (ورود و خروج از دامنه‌ها).

بعد از پیاده‌سازی سیمبول تیبل در قانون‌های به خصوصی، آن‌ها را به تیبل اد می‌کنیم. در قسمت زیر نیز برای کنترل اسکوپ برنامه است.

```
#add decelerations to symbol table
for i in range(len(order)):
    if order[i] == "COMPOUNDSTMT":
        print(symbol_table.table)
        symbol_table.enter_scope()
    if order[i] == "}":
        print(symbol_table.table)
        symbol_table.exit_scope()
```

در تابع زیر در برخورد با قوانین معنایی تعریف متغیر و تابع آن‌ها را به سیمبل تیبل اضافه می‌کنیم.

```
if(order[i] == "T_ID"):
    if(order[i + 3] == "FUNCDEC"):
        if order[i + 1] in symbol_table.table.keys():
            print(f"{order[i + 1]} already defined at line {line[i + 1]}!")
            continue
        symbol_table.add(order[i + 1],"Function",order[i - 1], None)
    if(order[i - 2] == "VARDECSTMT"):
        if order[i + 1] in symbol_table.table.keys():
            print(f"{order[i + 1]} already defined at line {line[i + 1]}!")
            continue
        symbol_table.add(order[i + 1],"Variable",order[i - 1], None)
    if(order[i + 3] == "VARDEC"):
        if order[i + 1] in symbol_table.table.keys():
            print(f"{order[i + 1]} already defined at line {line[i + 1]}!")
            continue
        symbol_table.add(order[i + 1],"Variable",order[i - 1], None)
    if(order[i - 1] == "," and order[i + 2] == "VARDECINIT"):
        if order[i + 1] in symbol_table.table.keys():
            print(f"{order[i + 1]} already defined at line {line[i + 1]}!")
            continue
        j = i
        while(order[j]!= "VARDECSTMT"):
            j -= 1
        symbol_table.add(order[i + 1],"Variable",order[j + 1], None)
    else:
        if order[i + 1] not in symbol_table.table.keys():
            print(f"{order[i + 1]} variable or function not defined at line {line[i + 1]}!")
```

تابع expression_evaluasion:

```
def expression_evaluation(order: list, table: dict, line: list):
        j = 0
        actual_type = ""
        types = []
        aop_operators = []
        rop_operators = []
        lop_operators = []      You, 8 minutes ago • Uncommitted changes
        while order[j] != "T_SEMICOLON":
            if order[j] == "T_RP":
                if order[j + 1] == "COMPOUNDSTMT":
                    break
            j += 1
            if order[j] == "T_AOP_PL" or order[j] == "T_AOP_MN" or order[j] == "T_AOP_ML" or order[
                aop_operators.append(order[j])
            elif order[j] == "T_ROP_L" or order[j] =="T_ROP_G" or order[j] == "T_ROP_LE" or order[j
                rop_operators.append(order[j])
            elif order[j] == "T_LOP_AND" or order[j] == "T_LOP_OR" or order[j] == "T_LOP_NOT":
                lop_operators.append(order[j])
```

تابع `expression_evaluation`

این تابع برای ارزیابی نوع یک عبارت استفاده می‌شود. این تابع لیستی از توکن‌ها و جدول نمادها را می‌گیرد و نوع واقعی عبارت را تعیین می‌کند.

منابع:

- Compilers: Principles, Techniques, and Tools by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
- Compiler Design By Neso Academy: https://youtube.com/playlist?list=PLBlnK6fEyqRjT3oJxFXRgjPNzeS-LFY-q&si=kleVlbQJ4-WNKkY1
- https://github.com/FelipeTomazEC/Lexical-Analyzer
- https://docs.python.org/3/library/stdtypes.html#str.startswith
- https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files
- https://github.com/python/cpython/blob/5505b91a684b0fc7ffcb3a5b325302671d74fb15/Grammar/Grammar#L152