**به نام خدا**



گزارشکار پروژه‌ی کامپایلر – فاز اول

اصول طراحی کامپایلر

استاد: آرش شفیعی

اعضای گروه:

محمدکاظم هرندی (4003623039)

مسعود محمدزاده (4012363147)

بهار 1403

https://github.com/mohammadkahr/Two-M-Compiler.git

فهرست

[مرور کلی از کد: 3](#_Toc165091529)

[پیاده سازی توابع با استفاده از (state machine) 4](#_Toc165091530)

[-پیاده سازی تابع برای شناسایی کامنت ها: 4](#_Toc165091531)

[1-1-توضیحات عبارت منظم: 4](#_Toc165091532)

[2-1-تبدیل عبارت منظم به دیاگرام حالت: 4](#_Toc165091533)

[4](#_Toc165091534)

[3-1-تبدیل دیاگرام حالت به کد: 4](#_Toc165091535)

[-پیاده سازی تابع برای شناسایی شناسه ها: 5](#_Toc165091536)

[با استفاده از عبارت منظم:  **[\_a-zA-Z][\_a-zA-Z0-9]** 5](#_Toc165091537)

[1-1-توضیحات عبارت منظم: 5](#_Toc165091538)

[2-1-تبدیل عبارت منظم به دیاگرام حالت: 5](#_Toc165091539)

[3-1-تبدیل دیاگرام حالت به کد: 6](#_Toc165091540)

[-پیاده سازی تابع برای شناسایی اعداد باینری: 6](#_Toc165091541)

[2-1-تبدیل عبارت منظم به دیاگرام حالت: 6](#_Toc165091542)

[3-1-تبدیل دیاگرام حالت به کد: 7](#_Toc165091543)

[تابع **:** **tokenizer** 7](#_Toc165091544)

# مرور کلی از کد:

1.ثابت‌هایی برای کلیدواژه‌ها، عملگرها و علائم تعریف شده‌اند.

2.یک نگاشت(Mapping) بین توکن‌ها و نام‌های متناظر آنها تعریف شده است.

3.این کد شامل توابعی برای شناسایی انواع مختلف توکن‌ها است:

- کلیدواژه‌ها Keywords

- عملگرها Operators

- شناسه‌ها Identifiers

- علائم Delimiters

- مقادیر عددی (اعشاری و شانزده‌گانه) Numeric literals (decimal and hexadecimal)

- مقادیر رشته‌ای String literals

- توضیحات Comments

- فضاهای خالی Whitespace

4.یک کلاس به نام Token تعریف شده است که توکن‌ها را با ویژگی‌هایی مانند نام، شماره خط، مقدار و تعداد نمایش می‌دهد.

5.تابع tokenizer() تابع اصلی است که مسئول تولید توکن‌ها است. این تابع هر خط را خوانده، هر خط را توکن‌بندی کرده و بر اساس قوانین تعریف شده در کد، توکن‌ها را تولید می‌کند.

6.توکن‌های تولید شده در یک فایل خروجی (output1.txt) ذخیره می‌شوند و توکن‌های فضای خالی حذف می‌شوند.

# پیاده سازی توابع با استفاده از (state machine)

در اینجا به توضیح درباره پیاده سازی چندتا از تابع هایی که با استفاده از (state machine) پیاده سازی شده اند می پردازیم.

## -پیاده سازی تابع برای شناسایی کامنت ها:

**ابتدا عبارت منظم مربوطه را نوشتیم: r'^\/{2}.\*$**

### 1-1-توضیحات عبارت منظم:

**این عبارت منظم به صورت زیر توضیح داده می‌شود:**

**`^`: این نشان دهنده شروع رشته است.**

**`\/{2}`: این بخش با دو علامت `/` مطابقت می‌کند. به علامت `/` اضافه شده است و به دلیل اینکه `/` نیز یک کاراکتر متخصص در عبارت منظم است، باید با `\` متناظر شود. `{2}` نشان دهندهٔ این است که علامت `/` دوبار تکرار می‌شود.**

**`.\*`: این الگو با هر کاراکتری که در ادامه دو علامت `/` آمده باشد مطابقت می‌کند. `.` به معنای هر کاراکتری است و `\*` به معنای صفر یا بیشتر تکرار آن است.**

**`$`: این نشان دهنده پایان رشته است.**

**بنابراین، به طور کلی، این الگو با هر رشته‌ای که با دو علامت `/` شروع شود و سپس دنباله‌ای از هر کاراکتر دیگری داشته باشد تا پایان خط مطابقت می‌کند.**

### 2-1-تبدیل عبارت منظم به دیاگرام حالت:

/

/

## 

Others

### 3-1-تبدیل دیاگرام حالت به کد:

در اینجا، وضعیت state با استفاده از متغیر state برای نشان دادن وضعیت فعلی ماشین حالتی استفاده می‌شود. بر اساس وضعیت فعلی و ورودی، ما به وضعیت جدید منتقل می‌شویم تا در نهایت تشخیص دهیم که آیا رشته ورودی یک توضیحات است یا خیر.

def 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

## -پیاده سازی تابع برای شناسایی شناسه ها:

### با استفاده از عبارت منظم: **[\_a-zA-Z][\_a-zA-Z0-9]**

### 1-1-توضیحات عبارت منظم:

[\_a-zA-Z]: این بخش از عبارت، یک حرف از الفبای انگلیسی (بزرگ یا کوچک) یا یک آندرلاین (\_) را متناظر می‌کند. در اصطلاحات متنی، این بخش به عنوان اولین حرف شناسه استفاده می‌شود و حرف اول شناسه باید یک حرف از الفبا یا آندرلاین باشد.

[\_a-zA-Z0-9]: این بخش نیز به عنوان حرف دوم و بعدی شناسه استفاده می‌شود. این حرف می‌تواند یک حرف از الفبا (بزرگ یا کوچک)، یک عدد (0 تا 9) یا آندرلاین (\_) باشد. به این ترتیب، شناسه می‌تواند شامل هر یک از این حروف و اعداد باشد.

### 2-1-تبدیل عبارت منظم به دیاگرام حالت:

**\_a-zA-Z**

**\_a-zA-Z0-9**

### 3-1-تبدیل دیاگرام حالت به کد:

def identifier(token: str):

    state = 0

    temp = get\_token\_until\_delimiter(token)

    if len(temp) == 0:

        return False, None

    for char in temp:

        if state == 0:

            if char == '\_' or char.isalpha():

                state = 1

            else:

                return False, None

        elif state == 1:

            if char.isalnum() or char == '\_':

                continue

            else:

                return False, None

    return state == 1, temp if state == 1 else None

## -پیاده سازی تابع برای شناسایی اعداد باینری:

عبارت منظم رو به رو را داریم:

0b[01]\*$ ^

^: این علامت نشان دهنده شروع رشته است، بنابراین رشته باید با الگوی تطابق داشته باشد که از اینجا شروع می‌شود.

0b: این بخش از الگو یک الف تطبیق می‌یابد که در اینجا به "0b" می‌ختمد. این الف‌ها مشخص می‌کنند که عبارت دیگری به عنوان "باینری" (دودویی) شناخته می‌شود.

[01]\*: این قسمت الگو به هر تعداد تکرار از اعداد دودویی (0 و 1) تطابق می‌یابد. \* یک متعلقه‌ی کوانتیف است که نشان می‌دهد که الگوی قبلی می‌تواند صفر یا بیشتر تکرار شود.

$: این علامت نشان دهنده پایان رشته است، بنابراین رشته باید در اینجا به پایان برسد.

### 2-1-تبدیل عبارت منظم به دیاگرام حالت:

**01**

b 0

### 3-1-تبدیل دیاگرام حالت به کد:

def bin(s: str) -> bool:

    state = 0

    for c in s.lower():

        if state == 0:

            if c == "0":

                state = 1

            else:

                return False

        elif state == 1:

            if c == "b":

                state = 2

            elif c in "01":

                return False

            else:

                return False

        elif state == 2:

            if c not in "01":

                return False

    return True

# تابع **:** **tokenizer**

def tokenizer():

    count = 0

    for line in read\_file\_line("test.txt"):

        count += 1

        start = 0

        while start < len(line):

            if comment(line[start:]):

                yield Token("T\_Comment", count, line[start + 2:], count)

                start = len(line)

            elif whitespace(line[start:start + 1]):

                yield Token("T\_Whitespace", count, line[start:start + 1], count)

            elif delimiter(line[start:start + 1])[0]:

                yield Token(delimiter(line[start:start + 1])[1], count, line[start:start + 1], count)

            elif litnum(line[start:])[0]:

                \_, token\_name, number = litnum(line[start:])

                yield Token(token\_name, count, number, count)

                start += len(number) - 1

            elif keyword(line[start:])[0]:

                yield Token("T\_" + keyword(line[start:])[1], count, line[start:start + len(keyword(line[start:])[1])],

                            count)

                start += len(keyword(line[start:])[1]) - 1

            elif identifier(line[start:])[0]:

                yield Token("T\_ID", count, identifier(line[start:])[1], count)

                start += len(identifier(line[start:])[1]) - 1

            elif operators(line[start:])[0]:

                operator = operators(line[start:])[1]

                token\_name = get\_token\_name(operator)

                yield Token(token\_name, count, operator, count)

            elif litstring(line[start:])[0]:

                \_, token\_name, word = litstring(line[start:])

                yield Token(token\_name, count, word, count)

                start += len(word) - 1

            start += 1

تابع `tokenizer` در واقع یک ژنراتور است که به عنوان خروجی توکن‌ها را تولید می‌کند. ورودی این تابع یک فایل متنی به نام "test.txt" است. این تابع در هر مرحله یک خط از فایل را می‌خواند و توکن‌های موجود در آن را استخراج می‌کند.

متغیر `count` به عنوان شمارنده‌ای برای شمارش خطوط متن ورودی استفاده می‌شود. این متغیر در ابتدا صفر است و با گذشت هر خط جدید از فایل متنی، یک واحد به آن افزوده می‌شود. این متغیر به عنوان شماره خط مورد استفاده برای هر توکن تولید شده است.

متغیر `start` به عنوان نشانگری برای موقعیت شروع استخراج توکن در هر خط متنی استفاده می‌شود. این متغیر ابتدا به صفر تنظیم می‌شود و سپس به طول متن خط فعلی متن قرار می‌گیرد تا تمامی توکن‌های موجود در هر خط استخراج شوند.

در هر مرحله از حلقه while، توکن جاری استخراج می‌شود. اگر توکنی از نوع کامنت یا فاصله باشد، متغیر `start` به آخرین موقعیت ممکن برای شروع توکن بعدی در همان خط تنظیم می‌شود. در غیر این صورت، متغیر `start` به سمت رو به جلو حرکت کرده و توکن جاری تا زمانی که به نوعی از نماد یا فاصله می‌رسد، استخراج می‌شود. سپس، توکن مربوطه به عنوان خروجی ژنراتور تولید می‌شود و متغیر `start` به عنوان محل شروع توکن بعدی تنظیم می‌شود. این فرآیند تا پایان خط فعلی ادامه پیدا می‌کند.