

Language and Machines Theory Projects

Project One and Project Two

زہرا تاکی – Zahra Taki

983663001 – زہرا تاکی – ZAHRA TAAKI

فهرست

2. پروژه اول
2. خلاصه پروژه
2. توضیح کد
4. مثال ها
19. پروژه دوم
19. توضیح کد
19. نحوه پیاده سازی استک
21. نحوه طراحی ماشین پشته ای
22. تشریح پذیرش و عدم پذیرش رشته ها
23. نحوه تبدیل عبارات ریاضی به رشته ای از a و b ها
24. تشریح عملکرد تشخیص خطا در پرانتز گذاری
25. تابع اصلی main
26. اجرای برنامه با ورودی های مختلف

پروژه اول

خلاصه

برنامه تست عبارت های منظم، باید به عنوان یک **pattern**، یک عبارت منظم را دریافت کند، همچنین یک متن نیز دریافت کند و عبارت هایی که مطابق با آن **pattern** یا همان عبارت منظم هستند را پیدا کند. در واقع کار این برنامه پیدا کردن قسمت هایی از متن، که عبارت منظم داده شده میتواند آنرا تولید کند، است. برای مثال برای عبارت `\b\d{3}` برنامه باید عددی که 3 بار عدد پشت سر هم را پیدا کند:



توضیح کد

برنامه با استفاده از زبان جاوا نوشته شده. برنامه شامل سه کلاس `RegexText` و `Test` و `InputForm` است.

کلاس `RegexText`

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexText {
}
```

کار اصلی این کلاس پیدا کردن عبارت های مطابق با عبارت منظم است. این کلاس شامل:

1. دو کلاس `import` شده ی `Pattern` و `Matcher` است.
2. شامل ویژگی هایی با نام `pattern`, `matcher`, `text`, `matchTextPart` هست.
 - `matchTextPart` از نوع `StringBuilder` در جاوا است. این کلاس برای اضافه کردن یک رشته جدید به ادامه ی رشته ی قبلی مناسب است. `matchTextPart` ویژگی خصوصی این کلاس است که توسط کلاس های دیگر با استفاده از متد `getMatchPart()` قابل دستیابی است. در متد اصلی این کلاس، تا زمانی که `matcher` متنی مطابق عبارت منظم ما پیدا کند، آن قسمت از متن به پایان `matchTextPart` اضافه میشود.
 - `text` از نوع `string` و همان متن ورودی است که از بین آن باید عبارت های مطابق با عبارت منظم پیدا شود.
 - `matcher` شیئی ایجاد شده از کلاس `Mathcher` است که متن ورودی با همان `text` را دریافت میکند.
 - ویژگی `pattern` شیئی ایجاد شده از کلاس `Pattern` است که متن مربوط به آن از `constructor` تابع به عنوان ورودی دریافت میشود و به `pattern` داده میشود
3. شامل یک سازنده به صورت `RegexText(String pattern, String text)` است.
4. شامل دو متد `findMatch()` و `getMatchPart()` است.

متد `findMatch()` در این کلاس، عبارت منظم را از ورودی میخواند. همچنین متن ورودی را میخواند و با استفاده از کلاس `matcher` قسمت هایی از متن را که مطابق عبارت منظم هستند، با استفاده از `"\n"` به عنوان ایجاد فاصله بین آنها، در رشته ای به اسم `matchPartText` ذخیره میکند و متد `getMatchPart()` آن را برمیگرداند.

```
private void findMatch() {
    while (matcher.find()) {
        matchTextPart.append(text.substring(matcher.start(),
            matcher.end()) + "\n");
    }

    if (matchTextPart.length() == 0)
        matchTextPart.append("No Matched Found!");
}

public StringBuilder getMatchPart() {
    findMatch();
    return matchTextPart;
}
```

کلاس `InputForm`

این کلاس، پیاده سازی پنجره ی ورودی هاست و کار اصلی آن پیاده سازی عملیات کلید `RUN` و استفاده از کلاس `RegexText` است

```
button1.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        regex = new RegexText(patternField.getText(), textField.getText());
        machedPartField.setText(regex.getMatchPart().toString());
    }
});
```

کلاس `Test`

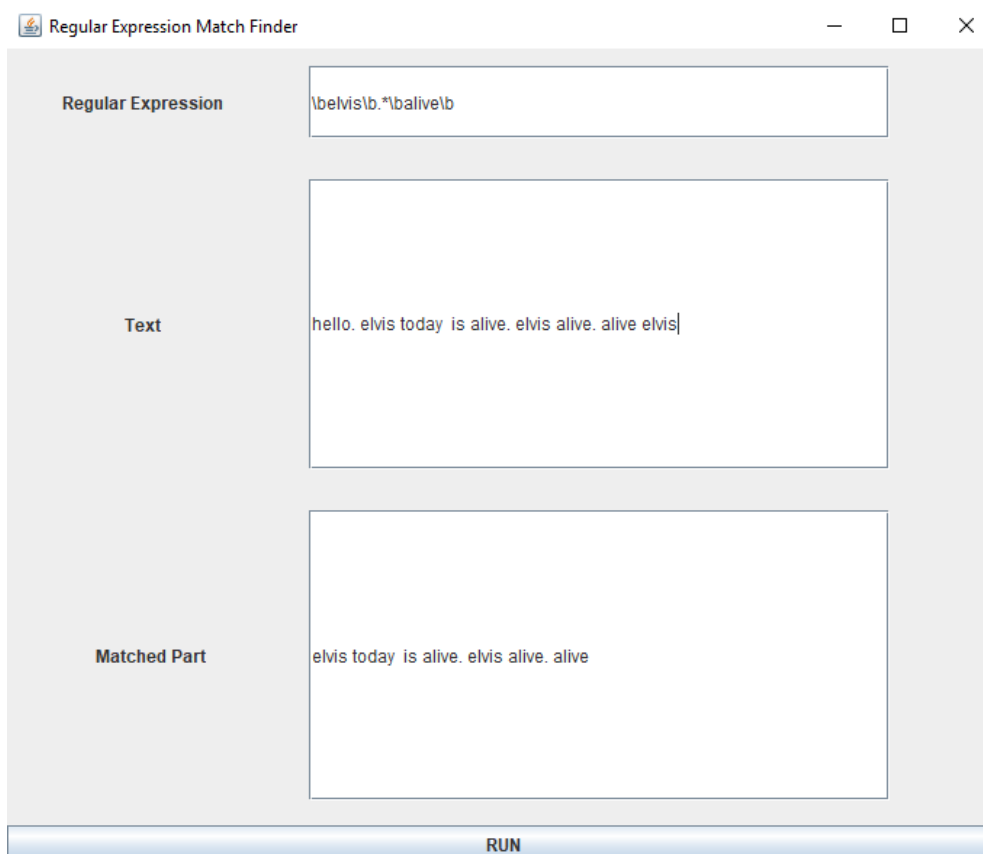
کلاسی است که تابع اصلی در آن قرار دارد و پنجره ی ورودی را میسازد و باز میکند.

1. $\backslash\text{elvis}\backslash\text{b}.*\backslash\text{alive}\backslash\text{b}$

با استفاده از این عبارت منظم برنامه همه ی متن هایی که ابتدا **elvis** و بعد از آن **alive** آمده باشد را باید پیدا کند.

وجود کاراکتر **\b** ابتدای عبارت یعنی **elvis** لازم نیست ابتدای متن باشد و میتواند هر جای متن قرار بگیرد. عبارت **\b.*** به معنی پایان کلمه است و همچنین بیان میکند که بعد از **elvis** میتواند صفر یا بیشتر از کاراکتر های دیگر بیاید. و به دنبال آن واژه ی **alive** که باید بین **\b** قرار بگیرد که به معنی شروع و پایان و جدا بودن این کلمه از باقی عبارت های متن است.

برنامه را برای متن “hello. elvis today is alive. elvis alive. alive elvis” امتحان میکنیم. در اینجا عبارت “elvis today is alive” متنی است که عبارت منظم بالا برای آن بر قرار است. همچنین برای عبارت “elvis alive” و همچنین عبارت “alive” در آخر. چون قبل از آن واژه ی **elvis** آمده بوده است.



2. $\backslash b \backslash d \backslash d \backslash d - \backslash d \backslash d \backslash d \backslash d$

با استفاده از این عبارت منظم برنامه باید قسمت هایی از متن را که به صورت شماره تلفن 7 رقمی نوشته شده است و بین رقم سوم و چهارم آن یک خط تیره وجود دارد پیدا کند.

کarakter $\backslash b$ در ابتدای آن آمده تا نشان دهد که این 7 رقم باید به صورت جدا نوشته شده باشند و به عبارت دیگر، عبارت های قبلی آن باید پایان یافته باشند. karakter $\backslash d$ نیز نشان دهنده عدد است.

برنامه را برای متن "my phone number is 123-4567. please call me tomorrow. 1234567 is wrong. 1234-567 is wrong" امتحان میکنیم. تنها عبارت حاصل "123-4567" است.

Regular Expression Match Finder

Regular Expression	$\backslash b \backslash d \backslash d \backslash d - \backslash d \backslash d \backslash d \backslash d$
Text	3-4567. please call me tomorrow. 1234567 is wrong.
Matched Part	123-4567


RUN

3. $\backslash b\{3\}-\backslash d\{4\}$

این مثال نیز باید نتیجه مثال قبل را بدهد زیرا بیانی دیگر از پیدا کردن عدد 7 رقمی است.

چون کاراکتر $\{n\}$ به معنی n بار تکرار است. هر عبارتی که عبارات قبلی آن به پایان رسیده باشند و 3 عدد آمده باشد و بعد از یک خط، تیره 4 عدد آمده باشند، قسمتی از متن است که عبارت منظم بالا میتواند آن را تولید کند و حاصل برنامه ما است.

عبارت مثال قبل را برای آن امتحان میکنیم و انتظار نتیجه مشابه را داریم.

 Regular Expression Match Finder

Regular Expression

$\backslash b\{3\}-\backslash d\{4\}$

Text

my phone number is 123-4567. please call me tomorrow. 1234567 is wr

Matched Part

123-4567

RUN

.4 $\backslash ba\backslash w^*\backslash b$

با استفاده از این عبارت منظم برنامه باید کلماتی که با حرف **a** شروع میشوند را پیدا کند.

وجود کاراکتر **\b** اول و آخر این عبارت منظم نشان دهنده ی این است که باید یک واژه ی جداگانه از عبارت های قبل و بعد خود باشد و به حرف دیگری چسبیده نباشد. کاراکتر **\w*** به معنی وجود صفر یا بیشتر کاراکتر های چاپی یعنی **[A-Z / a-z / 0-9 / _]** است. به عبارت دیگر حاصل، واژه ای است که با **a** شروع شود و همچنین تمام عبارت های قبل آن تمام شده باشند.

این مثال را برای عبارت **"Grab their ate_0_ntion with one of these solid hooks"** امتحان میکنیم. تنها واژه ی **ate_0_ntion** با **a** شروع شده است و انتظار داریم پاسخ نیز همین باشد.

The screenshot shows a window titled "Regular Expression Match Finder". It contains three main sections: "Regular Expression", "Text", and "Matched Part".

- Regular Expression:** The input field contains the regex `\ba\w*\b`.
- Text:** The input field contains the string "Grab their ate_0_ntion with one of these solid hooks".
- Matched Part:** The output field shows the matched text "ate_0_ntion".


At the bottom of the window, there is a blue button labeled "RUN".

5. $\backslash b \backslash d^+$

با استفاده از این عبارت منظم برنامه باید اعدادی که طول آنها 1 یا بیشتر است را پیدا کند.

$\backslash b$ قبل از $\backslash d$ یعنی باید از عبارات قبلی خود جدا باشد. $\backslash d^+$ نیز یعنی 1 یا بیشتر اعدادی که دنبال یکدیگر قرار دارند.

برای این مثال “i found 1234 letters in 25 rooms with 2 common worlds” را امتحان میکنیم. و انتظار داریم اعداد 1234 و 25 و 2 با عبارت منظم بالا مطابقت داشته باشند.


Regular Expression Match Finder
—
□
×

Regular Expression	<code>\b\d+</code>
Text	i found 1234 letters in 25 rooms with 2 common worlds
Matched Part	1234 25 2

RUN

.6 $\backslash b \backslash w\{6\} \backslash b$

با استفاده از این عبارت منظم برنامه باید همه ی کلمات 6 حرفی را پیدا کند.

وجود $\backslash b$ ابتدا و انتهای عبارت منظم نشان دهنده ی این است که واژه ای جدا که از عبارات بعد و قبل خود است جواب است. $\backslash w$ نیز نشان دهنده ی وجود کاراکترهای چاپی است و $\{6\}$ به معنی 6 بار تکرار کاراکتر $\backslash w$ است.

برای این مثال “i found 123456 letters in 25 rooms with 2 common worlds” را امتحان میکنیم. رشته های 6 حرفی در این مثال 123456, common, worlds هستند و این پاسخ هارو داریم.

Regular Expression Match Finder

Regular Expression: $\backslash b \backslash w\{6\} \backslash b$

Text: i found 123456 letters in 25 rooms with 2 common worlds

Matched Part: 123456 common worlds

RUN

7. $\backslash b \backslash w \{5,6\} \backslash b$

با استفاده از این عبارت منظم برنامه باید همه ی کلمات 6 یا 5 حرفی را پیدا کند.

وجود کاراکتر $\backslash b$ در ابتدا و انتهای عبارت منظم یعنی قسمت از متن که به صورت جداگانه از عبارت های قبل و بعد از آن است جواب است. کاراکتر $\backslash w$ به معنی کاراکتر های چاپی هستند. یعنی اعداد 0 تا 9 و حروف بزرگ و کوچک a تا z. کاراکتر $\{n\}$ بعد از هر کاراکتری به معنی n بار تکرار آن است. و اینجا به معنی 5 یا 6 بار تکرار کاراکتر های چاپی پشت سر هم.

برای این مثال عبارت "this word 12B45 has five. This one boooook has six" امتحان میکنیم و انتظار خروجی 12B45 و boooook را داریم.

Regular Expression	$\backslash b \backslash w \{5,6\} \backslash b$
Text	this word 12B45 has five. This one boooook has six
Matched Part	12B45 boooook
RUN	

8. $\backslash b\{d\{3\}\backslash s\{d\{3\}\}\backslash d\{4\}$

با استفاده از این عبارت منظم برنامه باید قسمت هایی از متن که به فرم شماره تلفن 10 رقمی هستند را بیاید.

وجود کاراکتر $\backslash b$ یعنی باید از عبارت های قبلی جدا باشد. $\{d\{3\}$ یعنی 3 کاراکتر عددی پشت سر هم قرار گرفته باشند. $\backslash s$ به معنی یک فاصله با یک کاراکتر سفید است. سپس باید دوباره 3 کاراکتر عددی پشت سر هم تکرار شوند. بعد از آنها باید یک خط تیره وجود داشته باشد. و سپس باید 4 کاراکتر عددی پشت سر هم تکرار شوند. در کل عبارت معادل باید به فرم 111 222-3333 باشد. این عبارت منظم درست مانند عبارت منظم در شماره 3 عمل میکند و با همان روش به دنبال عدد 10 رقمی به فرم خاص میگردد.

برای این مثال متن "call this 333 333-4444. Don't call this 333-333-4444 or 123 123 1234" تنها کلمه مطابق با عبارت منظم در متن فوق 333 333-4444 است.

Regular Expression	$\backslash b\{d\{3\}\backslash s\{d\{3\}\}\backslash d\{4\}$
Text	call this 333 333-4444. Don't call this 333-333-4444 or 123 123 1234
Matched Part	333 333-4444
RUN	

9. $\wedge \backslash w^*$

با استفاده از این عبارت منظم برنامه باید اولین کلمه سطر را پیدا کند.

کاراکتر \wedge به معنی شروع یک سطر است. کاراکتر $\backslash w$ هم نشان دهنده کاراکترهای چاپی است. و کاراکتر $*$ کنار آن به معنی صفر یا بیشتر تکرار آن است. در واقع برنامه به ازای این عبارت منظم هر کلمه ی واحد که اول سطر قرار گرفته باشد با طول نامشخص (از 1 تا بیشتر) را پیدا میکند.

برای این مثال متن "you \nShould \nPrint \nAll \nThese \nWords" را امتحان میکنم. در اینجا برای مشخص تر بودن ابتدای هر سطر را با $\backslash n$ نشان میدهم. خروجی برنامه باید you should print all these words باشد.

Regular Expression	$\wedge \backslash w^*$
Text	you \nShould \nPrint \nAll \nThese \nWords
Matched Part	you Sould Print All These Words
RUN	

10. $\backslash(?\{d\}[])\backslash s? \{d\}[-]\{d\}4$

با استفاده از این عبارت منظم برنامه باید شماره تلفن 10 رقمی را که با فرم های خاصی نوشته شده اند پیدا کند.

این عبارت منظم مانند عبارت منظم مورد 8 به دنبال شماره 10 رقمی می‌گردد اما فرم های خاص بیشتری را تولید میکند و برای پیدا کردن شماره تلفن مناسب تر است. \backslash نشان دهنده وجود کاراکتر پرانتز باز در اول عبارت است (چون کاراکتر پرانتز باز رزرو شده است، برای چاپ آن باید قبل از آن از \backslash استفاده شود). علامت سوال بعد از آن به معنی صفر یا یک بار تکرار کاراکتر پرانتز است. در نتیجه هم شماره های 10 رقمی ای که اول آنها (هست پذیرش میشوند هم آنهايي که ندارند یعنی هم "111") هم "111". $\{d\}3$ به معنی 3 کاراکتر عددی پشت سر هم است. کاراکتر $[]$ هم به معنی مشخصا وجود " " است. $\{s\}$ نشان دهنده ی وجود صفر یا یک بار کاراکتر سفید است. اگر کاراکتر سفید و (هر دو وجود نداشته باشند یعنی 6 کاراکتر عددی پشت سر هم هستند و این عبارت پذیرش نمیشود. در نتیجه یا باید (باشد کاراکتر سفید نباشد مثل "111)111-1111" یا باید (نباشد کاراکتر سفید باشد "111 111-1111" یا باید هر دو باشند " 111) 111-1111". $\{d\}3$ با همان مفهوم قبلی و $[-]$ به معنی مشخصا وجود خط تیره و در آخر، وجود 4 کاراکتر عددی پشت سر هم. در نتیجه شماره هایی که پذیرش میکند میتواند 6 فرم مختلف داشته باشند.

برنامه را برای عبارت "(111) 111-1111 and 111) 222-7777 and 5555) 444-444" امتحان میکنیم . انتظار داریم خروجی دو عبارت (111) 111-1111 و (111) 222-7777 باشد.

Regular Expression	$\backslash(?\{d\}[])\backslash s? \{d\}[-]\{d\}4$
Text	(111) 111-1111 and 111) 222-7777 and 5555) 444-444
Matched Part	(111) 111-1111 111) 222-7777
RUN	

\S+.11

با استفاده از این عبارت منظم برنامه باید تمام عبارت هایی که شامل **whitespace** نمیشود را بیاید. به عبارت دیگر باید تمام عبارت ها به جز **whitespace** را چاپ کند.

کاراکتر **\S** به معنی کاراکتر غیر سفید و + تعداد تکرار یک یا بیشتر است. هر رشته ای با طول 1 یا بیشتر که شامل **whitespace** نباشد جواب است.

برای این مثال "does this string contain whitespace?" را امتحان میکنیم. انتظار داریم جواب تمام این عبارت باشد. نکته: خروجی این برنامه رشته ای است شامل جواب های پیدا شده در متن که با کاراکتر **\n** از هم جدا شده اند و در خروجی در یک خط نمایش داده میشوند. در واقع فاصله ی بین کلمات حاصل از وجود **\n** در کد برنامه است و خروجی اصلی شامل **whitespace** نیست.

Regular Expression	<input type="text" value="\S+"/>
Text	<input type="text" value="doese this string contain whitespace?"/>
Matched Part	<input type="text" value="doese this string contain whitespace?"/>
<input type="button" value="RUN"/>	

12. $(\backslash\{d\}\backslash\{d\})s?\{d\}[-]\{d\}$

با استفاده از این عبارت منظم برنامه باید شماره تلفن های 10 رقمی را بیاید.

مانند عبارت منظم شماره 10 این عبارت منظم نیز شماره تلفن های 10 رقمی را تولید میکند و این روش از روش قبلی بهتر است. زیرا بسیار قابل فهم تر است. این عبارت منظم از دو عبارت $(\backslash\{d\}\backslash\{d\})$ و $s?\{d\}[-]\{d\}$ تشکیل شده. عبارت اول بیان میکند که سه رقم اول شماره تلفن $(\{d\})$ به چه فرمی نمایش داده شود. \backslash یعنی وجود کاراکتر پرانتز باز و $\{d\}$ یعنی وجود یک کاراکتر پرانتز بسته. پس عبارت اول بیان میکند که سه رقم اول یا به فرم "111" هستند یا به فرم "(111)".
 $s?$ در عبارت بعدی یعنی صفر یا یکی کاراکتر سفید. $\{d\}[-]$ یعنی 3 کاراکتر عددی که بعد از آن خط تیره باشد و در آخر 4 کاراکتر عددی بیاید.

برنامه را برای عبارت "(111) 111-1111 and 111) 222-7777 and 5555) 444-444" امتحان میکنیم. انتظار میرود خروجی عبارت (111) 111-1111 را چاپ کند.

Regular Expression	$(\backslash\{d\}\backslash\{d\})s?\{d\}[-]\{d\}$
Text	(111) 111-1111 and 111) 222-7777 and 5555) 444-444
Matched Part	(111) 111-1111
RUN	

13. $(\backslash d\{1,3\}\backslash.){3}\backslash d\{1,3\}$

با استفاده از این عبارت منظم برنامه باید IP address ها را پیدا کند.

این عبارت منظم شامل دو قسمت $(\backslash d\{1,3\}\backslash.){3}$ و $\backslash d\{1,3\}$ است. $\backslash d\{1,3\}$ به معنی وجود حداقل 1 و حداکثر 3 کاراکتر عددی پشت سر هم. $\backslash.$ به معنی وجود کاراکتر نقطه بعد اعداد پشت هم هست. کل این عبارت داخل یک پرانتز است و بعد از آن $\{3\}$ قرار داد به این معنی که این الگو (1 تا 3 عدد پشت سر هم و بعد از آن نقطه) 3 بار تکرار شود. عبارت بعدی نیز نشان دهنده وجود 1 تا 3 کاراکتر عددی پشت هم است. فرم یک IP آدرس به این صورت است که چهار تا عدد در رنج صفر تا 255 که بین نقطه قرار دارد و عبارت منظم فوق میتواند تعدادی از آنها را تولید کند. (البته عبارت فوق عبارت هایی که بهفرم IP نیستند را نیز تولید میکند مانند (900.900.900.900))

برای این مثال متن "0.0.0.0 and 900.0.500.0 and 5000.22.22" را امتحان میکنیم. انتظار داریم خروجی برنامه دو عدد 0.0.0.0 و 900.0.500.0 باشد.

Regular Expression	$(\backslash d\{1,3\}\backslash.){3}\backslash d\{1,3\}$
Text	0.0.0.0 and 900.0.500.0 and 5000.22.22
Matched Part	0.0.0.0 900.0.500.0
<div>RUN</div>	

14. $((2[0-4]\backslash d|25[0-5]||[01]? \backslash d \backslash d?) \backslash .) \{3\} (2[0-4]\backslash d|25[0-5]||[01]? \backslash d \backslash d?)$

با استفاده از این عبارت منظم برنامه باید IP address ها را بیاید. این یک روش بهتر است و محدودیت 0 تا 255 اعداد IP را رعایت میکند و IP address های واقعی را میابد.

این عبارت منظم خود شامل دو قسمت عبارت منظم است که در هر دو، دو الگو تکرار شده است. و آن هم به دلیل وجود کاراکتر نقطه بین آنهاست. آن عبارت منظم به شکل $2[0-4]\backslash d|25[0-5]||[01]? \backslash d \backslash d?$ است. در این عبارت منظم $2[0-4]$ به این معنی است که اگر ابتدا عدد 2 آمده است، کاراکتر بعدی آن باید از بین کاراکترهای 0 تا 4 باشد و بعد از آن $\backslash d$ یعنی هر عددی بین 0 تا 9. در واقع این قسمت اعداد بین 200 تا 249 را تولید میکند. عبارت $25[0-5]$ یعنی اگر دو کاراکتر اول 25 است، کاراکتر بعدی باید از بین کاراکترهای 0 تا 5 باشد. این عبارت اعداد بین 250 تا 255 را تولید میکند. عبارت $[01]? \backslash d \backslash d?$ به این معنی است که اگر کاراکتر 0 یا 1 به تعداد صفر یا یک بار وجود داشتند، کاراکتر بعدی آنها یک عدد بین 0 تا 9 باشد و پس از آن میتواند یک عدد بین 0 تا 9 باشد یا نباشد. درواقع این عبارت اعداد بین 0 تا 199 را تولید میکند. که برای اعداد 1 و 2 رقمی میتواند قبل از آنها صفر هم وجود داشته باشد (هر دو عبارت 001 و 1 قابل قبول هستند). این به دلیل این که اعداد IP باید بین 0 تا 255 باشد. و اجتماع این سه عبارت (همان عملگر | به معنی "یا") پذیرنده اعداد 0 تا 255 هستند. بعد از این سه عبارت باید یک کاراکتر نقطه باشد. و این الگو چهار بار تکرار شود (بار آخر بدون کاراکتر نقطه).

این مثال را با متن "255.127.001.20 and 255.0001.200.3" امتحان میکنیم. انتظار داریم عبارت 255.127.001.20 به عنوان خروجی چاپ شود.

Regular Expression	$((2[0-4]\backslash d 25[0-5] [01]? \backslash d \backslash d?) \backslash .) \{3\} (2[0-4]\backslash d 25[0-5] [01]? \backslash d \backslash d?)$
Text	255.127.001.20 and 255.0001.200.3
Matched Part	255.127.001.20
RUN	

\b(\w+)\b\s*\1\b .15

با استفاده از این عبارت منظم برنامه باید عبارت های تکراری داخل متن را بیاید.

این عبارت شامل دو قسمت عبارت منظم دیگر به ترتیب $\backslash b(\backslash w+)\backslash b\s*\backslash 1\backslash b$ و $\backslash 1\backslash b$ است. عبارت منظم اول، به معنی یک یا بیشتر کاراکتر چاپی که بصورت جداگانه اند و بعد از آنها به صفر یا بیشتر کاراکتر سفید وجود دارد است. عبارت منظم دوم به معنی یاد آوری و تطبیق آخرین متنی که با عبارت منظم اول (یعنی $\backslash b(\backslash w+)\backslash b\s*$) مطابق داشته است.

متن "a a s s a s a" را برای این مثال امتحان میکنیم. انتظار داریم a a و s s در خروجی چاپ شوند.

Regular Expression	<code>\b(\w+)\b\s*\1\b</code>
Text	a a s s a s a
Matched Part	a a s s
RUN	

پروژه دوم

توضیح کد

برنامه با کد جاوا شامل سه کلاس `MyStack`, `PDA`, `Main` می باشد.

- `MyStack` این کلاس استک پیاده سازی شده با استفاده از آرایه ای از اعداد است زیرا این استک تنها عدد 0 و 1 را میپذیرد. دارای دو متد `push`, `pop` است.
- `PDA` این کلاس همان ماشین پشته ای است. ماشین پشته ای شامل یک استک است که در این کلاس قرار دارد. همچنین دارای یک ماشین حالت یا یک گرامر است که در کدنویسی متد های این برنامه این گرامر در نظر گرفته شده و این چنین ایفای نقش میکند. و همچنین یک `input string` که از ورودی خوانده میشود و به متد های این کلاس داده میشود.
- `Main` این کلاس به کارگیری کلاس های فوق است و همچنین رابط با کاربر را از طریق ترمینال ایجاد میکند. ورودی ها را میگیرد و خروجی ها را نشان میدهد.

نحوه پیاده سازی استک

```
public class MyStack {
    static final int MAX = 1000;
    int top;

    int a[] = new int[MAX]; // Maximum size of Stack
}
```

کلاس `myStack` شامل یک متغیر تعبیر ناپذیر `MAX` است که تعداد عناصر استک را مشخص میکند. همچنین دارای یک متغیر `top` که اشاره گر به بالاترین عنصر استک است. استک بر پایه یک آرایه به اسم `a` ساخته شده است که از نوع `integer` است زیرا این استک تنها مقادیر 0 و 1 را در خود قرار میدهد.

```
public class MyStack {
    static final int MAX = 1000;
    int top;

    int a[] = new int[MAX]; // Maximum size of Stack

    MyStack() {
        top = 0;
        a[top] = -1;
    }
}
```

سازنده ی این کلاس بدون پارامتر ورودی است و فقط مقدار اولیه ای به اشاره گر و آخرین خانه ی استک میدهد. در ماشین های پشته ای آخرین خانه ی استک با `Z` یا `$` نشان داده میشود اما اینجا برای اینکه آرایه فقط اعداد را میپذیرد از `-1` استفاده میکنیم.

```
boolean isEmpty() {
    return (a[top] == -1);
}
```

متد `isEmpty()` بدون ورودی و با خروجی `true` یا `false` است. زمانی که اشاره گر به سر استک به خانه ای اشاره کند که در آن `-1` است، یعنی استک خالی است.

```
boolean push(int x) {  
    if (top >= (MAX - 1)) {  
        return false;  
    }  
    else {  
        a[++top] = x;  
        return true;  
    }  
}
```

متد push یک عدد دریافت میکند و اگر استک پر نبود، اشاره گر به سر را یک خانه به جلو میبرد و آن را به استک اضافه میکند. اگر عملیات اضافه کردن موفقیت آمیز بود، true و در غیر اینصورت false میدهد.

```
int pop() {  
    if (top < 1) {  
        return 1-;  
    }  
    else {  
        int x = a[top--];  
        return x;  
    }  
}
```

متد pop بدون پارامتر ورودی و با خروجی عددی، آخرین عنصر در استک را برمیگرداند و اشاره گر به سر استک را یک خانه به عقب میبرد. اگر استک خالی بود، منفی یک برمیگرداند و میتوان فهمید که استک خالی است.

نحوه طراحی ماشین پشته ای

ماشین پشته ای شامل 3 مشخصه ی زیر است که به نحوی در کلاس PDA حضور دارند.

- **states** یا کاشین حالت است که در این کلاس همان گرامری داده شده در کلاس است و از آن در نحوه پیداسازی متد ها استفاده میشود. به عنوان مثال میدانیم که این گرامر رشته ای را تولید میکند که تعداد **a** و **b** های آن یکی باشد در نتیجه در توابع طبق همین قانون عمل میکنیم.
- **Input string** یا همان نوار ورودی است که در این کلاس وجود ندارد اما توسط کلاس اصلی از ترمینال گرفته میشود و به متد های این کلاس داده میشود.
- **استک** که در این کلاس وجود دارد و از کلاس **MyStack** یک **object** ساخته شده است.

```
public class PDA {
    private MyStack stack;

    PDA() {
        stack = new MyStack();
    }
}
```

در نوشته بالا استک مربوط به ماشین و سازنده ی این کلاس مشاهده میشود.

این کلاس شامل متد های **textMatchGrammar()**, **arithmeticToText()**, **notMatchReason()** است.

1. **textMatchGrammar()** عملکرد پذیرش و عدم پذیرش رشته ها
2. **arithmeticToText()** تبدیل عبارات ریاضی به رشته ای از **a** و **b** ها
3. **notMatchReason()** عملکرد تشخیص خطا در پرانتز گذاری

تشریح عملکرد پذیرش و عدم پذیرش رشته ها

در PDA تابع `textMatchGrammar` کار تست کردن رشته های ورودی با زبان این ماشین را انجام میدهد.

```
/**just 0 and 1 is allowed is stack. we consider a as 0 and b as 1*/
public boolean textMatchGrammar(String text){ //text includes a and b

    char[] ab = text.toCharArray();
    boolean isMatch = true;

    for (int i = 0; i<ab.length; ++i) {

        if (stack.isEmpty() && i == ab.length-1) { //if there is any extra closed parentheses
            isMatch = false;
            System.out.println(i);
            break;
        }
        if (ab[i] == 'a')
            stack.push(0);
        else stack.pop();
    }

    if (!stack.isEmpty()) //if there any extra opened parentheses
        isMatch = false;

    return isMatch;
}
```

این تابع با یک ورودی رشته و خروجی `Boolean` است. ابتدا رشته را به آرایه ای از کاراکترهای داخل رشته تبدیل میکند. سپس روی تمام کاراکترهای آن پیمایش میکند. هر زمان که به کاراکتر `a` برسد عدد `0` را وارد استک میکند، و هر زمان که به کاراکتر `b` برسد یکی از صفرها را از استک بر میدارد. همچنین در هر بار پیمایش شروطی را چک میکند.

- عملیات استک در این متد: با توجه به گرامری که داریم میدانیم زبانی با ویژگی زیر تولید میکند.

$$L(G) = \{w : n_a(w) = n_b(w), \\ \text{and } n_a(v) \geq n_b(v) \\ \text{in any prefix } v\}$$

و به عنوان مثال رشته ای مانند `aababb` را پذیرش میکند. اگر به ازای هر کدام از حروف `a` یک صفر وارد استک کنیم و هر گاه به `b` رسیدیم یکی از صفرها را از استک خارج کنیم و در پایان این عملیات رشته تمام شود و استک نیز خالی باشد میتوانیم تمام رشته هایی که این گرامر تولید میکند یا تشخیص دهیم. این کد در واقع پیاده سازی ماشین حالت است.

- شروط داخل متد: اگر در پایان عملیات، استک خالی باشد اما همچنان کاراکتر داشته باشیم، یا اینکه کاراکترها تمام شده باشند اما استک خالی نباشد، این گرامر این رشته را تولید نمیکند. شرط اول داخل حلقه برای رشته ای مانند `aababbbb` است. این رشته یک `b` اضافه دارد در نتیجه به ازای آن میخواهد از استکی که خالی است عدد `0` را بیرون بیاورد و این نشان میدهد که این زبان این رشته را تولید نمیکند. در نتیجه خروجی `false` میدهد. شرط دوم خارج از حلقه برای رشته ای مانند `aabab` است. این رشته به ازای دو کاراکتر `b`، دو تا از سه تا صفرهای داخل استک را `pop` کرده است و رشته به پایان رسیده اما استک هنوز خالی نشده است. در نتیجه برنامه وارد این شرط میشود م خروجی `false` میدهد.

نحوه تبدیل عبارات ریاضی به رشته ای از a و b ها

در کلاس PDA متد `arithmeticToText()` کار تبدیل عبارت ریاضی به رشته ای از a و b ها را انجام میدهد. از آنجایی که گرامر PDA گرامری برای تشخیص پرانتزگذاری ها است، پس قرار است تنها پرانتزگذاری عبارت ریاضی مورد بررسی قرار بگیرد. اگر هر پرانتز بازی را a و پرانتز بسته ای را b در نظر بگیریم، میتوانیم گرامر را بر روی آن امتحان کنیم (مانند یک تابع هم ریختی که حروف آن a و b است). پس کاری که باید انجام دهیم در نظر نگرفتن تمام عمگر ها و عملوند ها به غیر از پرانتز است و مطابق چیزی که گفته شد، پرانتز ها با حروف a و b نشان داده میشوند.

```
/**this method find the equivalent text with the special format which is possible to check it with
the grammar of PDA
* therefore this method put letter "a" for each "(" and letter "b" for each ")" */
public StringBuilder arithmeticToText(String arithmetic){
    StringBuilder text = new StringBuilder();

    for (char ch:arithmetic.toCharArray()) {
        if (ch == '(')
            text.append("a");
        else if (ch == ')')
            text.append("b");
    }
    return text;
}
```

این متد با ورودی رشته و خروجی از نوع `StringBuilder` است. خروجی از نوع کلاس `StringBuilder` میباشد زیرا در جاوا هنگام کار با `string` هایی که قرار است مقداری به آن اضافه یا کم شود، این کلاس سرعت بالاتر و حافظه ی کمتر و در کل کاربردی تر است.

متغیری از `StringBuilder` میسازیم. سپس رشته ی عبارات ریاضی را به آرایه ای از کاراکتر ها تبدیل میکنیم و روی آنها پیمایش میکنیم. همانطور که قبلا گفته شد باید دو شرط روی هر کاراکتر بررسی شود. اگر پرانتز باز است، رشته ی a را و اگر پرانتز بسته است رشته ی b را به متغیر `text` اضافه میکنیم. در غیر این دو صورت هیچ رشته ای به `text` اضافه نمیشود.

در پایان رشته ی `text` برگردانده میشود که شامل a و b هاست.

تشریح عملکرد تشخیص خطا در پرانتزگذاری

در کلاس PDA متد `notMatchReason()` وظیفه ی تشخیص نوع خطا در پرانتزگذاری را بر عهده دارد. پرانتز گذاری ها در دو حالت دچار خطا هستند. حالت اول؛ پرانتزی که باز شده بسته نشده باشد (پرانتز باز اضافی) و حالت دوم؛ پرانتزی که باز نشده، بسته شده باشد (پرانتز بسته ی اضافی). هر کدام از این دو حالت پیش بیاید، استک را از حالت ایده آل ما؛ یعنی زمانی که رشته کامل پیمایش شده و چیزی هم درون استک قرار ندارد، خارج میکند.

این متد تنها زمانی که مطمئن هستیم عبارت ریاضیاتی مورد نظر با گرامر ما منطبق نبوده صدا زده میشود. به عبارتی زمانی که عبارت ریاضیاتی با تابع `arithmeticToText()` تبدیل به رشته ای از `a` و `b` ها شد و سپس آن رشته توسط تابع `textMatchGrammar()` تست میشود تا مشخص شود که با گرامر ماشین پشته ای منطبق است یا خیر. اگر خروجی این تابع `false` باشد، در این مرحله متد `notMatchReason()` را صدا میزنیم.

به همین دلیل است که در این تابع لازم نداریم تا عبارت را دوباره تسس کنیم و باید تنها از روی شرایط پیش آمده برای استک، متوجه شویم که کدام یک از دو حالت بالا برای عبارت ریاضی وجود دارد.

```
/**this method will be called when the textMatchGrammar method returns false in the Main Class
 * therefore we are sure that the text does Not match the grammar of the PDA so we dont check it
 again
 * and we just review the stack to see in what state it is*/
public String notMatchReason(String text){
    if (stack.isEmpty())
        return "there is extra unopened closed parentheses [] in the string";
    else
        return "there is extra opened unclosed parentheses [(] in the string";
}
```

دو حالت به قرار زیر است:

- حالت اول؛ پرانتزی که باز شده بسته نشده باشد (پرانتز باز اضافی)

با توجه به سازوکار ماشین پشته ای ما برای این گرامر؛ یعنی `push` کردن پرانتز باز ها (`a = 0`) و `pop` کردن یکی از آنها در صورت مشاهده ی پرانتز بسته (`b = 1`)، پس در صورتی که یک پرانتز باز اضافی وجد داشته باشد، یعنی تمام کاراکتر های رشته پیمایش شده اما در آخر استک خالی نشده و داخل آن هنوز عنصر 0 ای وجود دارد. در این صورت متد بالا رشته ی `"there is extra opened unclosed parentheses [(] in the string"` را بر میگرداند.

- حالت دوم؛ پرانتزی که باز نشده، بسته شده باشد (پرانتز بسته ی اضافی)

با توجه به سازوکار ماشین پشته ای و چیزی که قبلا گفته شد، زمانی که پرانتز بسته ی اضافی وجود دارد یعنی تمام پرانتز باز ها از استک بیرون آمده اند و استک خالی است اما هنوز رشته کامل پیموده نشده است. البته این حالت زمانی خطا محسوب میشود که کاراکتر بعدی درون رشته پرانتز بسته باشد. به عنوان مثال برای رشته ی `((()))` بعد از `((())` استک خالی میشود اما کاراکتر بعدی پرانتز بار است در نتیجه شائل خطا نمیشود اما رشته ی `((()))` شامل خطای حالت دوم میشود.

در نتیجه اگر استک خالی باشد این متد عبارت `"there is extra unopened closed parentheses [] in the string"` را برمیگرداند.

تابع اصلی Main

فراخوانی ها در این کلاس و توسط تابع main انجام میگردد. ابتدا از کلاس PDA شیء ساخته میشود تا از متد های ان استفاده شود.

```
private static PDA pda = new PDA();
```

سپس در حلقه ای بینهایت با flag به نام exit که فقط کاربر میتواند با وارد کردن عدد 3 آن را false کند منو نشان داده میشود. بدین صورت:

options:

input a string including a and b..1

input an arithmetic expression..2

Exit!..3

- اگر کاربر شماره 1 را وارد کند وارد case 1 میشود.

```
switch (input.nextInt()) {
    case 1: //the case we want to test a string including a and b
        ABexpression = input.next();
        output = pda.textMatchGrammar(ABexpression) ? MATCH : NOT_MATCH; //does text matches?

        ///////////////OUTPUT////////////////////////////////////
        System.out.println(output);
        break;
```

در این حالت ورودی شامل رشته ای از a و b را دریافت میکنیم. سپس با استفاده از متد () textMatchGrammar انطباق یا عدم انطباق آن را گزارش میدهیم.

- اگر کاربر شماره 2 را وارد کند وارد case 2 میشود.

```
case 2: //the case we want to test a arithmetic string
    ArithmeticExpression = input.next();
    String text = String.valueOf(pda.arithmeticToText(ArithmeticExpression));
    output = EQUIVALENT + text;
    extraOutput = pda.textMatchGrammar(text) ? MATCH : pda.notMatchReason(text);

    ///////////////OUTPUT////////////////////////////////////
    System.out.println(output);
    System.out.println(extraOutput);
    break;
```

در این حالت عبارت ریاضیاتی را از ورودی دریافت میکنیم. آن را با ايسفاده از متد () arithmeticToText به رشته ای قابل قبول تبدیل میکنیم و آن را به عنوان output چاپ میکنیم. سپس اگر match بود، منطبق بودن آن و در غیر این صورت، دلیل منطبق نبودن و خطای پرانزگذاری را با عنوان extraOutput در خروجی چاپ میکنیم.

- با وارد کردن 3 برنامه خاتمه میابد.

اجرای برنامه با ورودی های مختلف

برنامه را برای 4 رشته ی `aabbab`, `aababbaabb`, `aabbba`, `aaabbaabb` امتحان میکنیم.

خروجی برنامه برای `aabbab`:

اگر مراحل استک را برای این رشته طی کنیم میتوان دید که در آخر استک خالی و رشته تمام میشود در نتیجه این رشته توسط این ماشین پذیرفته میشود.



```
C:\java\jdk1.8.0_202\bin\java.exe ...
options:
1.input a string including a and b.
2.input an arithmetic expression.
3.Exit!
1
aabbab
[this grammar produces this string]
```

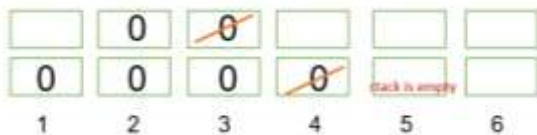
خروجی برنامه برای `aababbaabb`:

برای این مثال نیز اگر مراحل استک را طی کنیم میبینیم که پس از پایان عبارت استک خالی است در نتیجه گرامر این ماشین، این عبارت را تولید میکند.

```
options:
1.input a string including a and b.
2.input an arithmetic expression.
3.Exit!
1
aababbaabb
[this grammar produces this string]
```

خروجی برنامه برای `aabbba`:

اگر مراحل استک را طی کنیم میبینیم که رشته به پایان نمیرسد اما استک خالی است و الگوریتم دیگر نمیتواند به ازای یک `b` یک `a` را از استک بیرون بیاورد.



```
C:\java\jdk1.8.0_202\bin\java.exe ...
options:
1.input a string including a and b.
2.input an arithmetic expression.
3.Exit!
1
aabbba
[this grammar doesn't produce this string]
```

خروجی برنامه برای `aaabbaabb`:

اگر مراحل استک را برای این رشته طی کنیم مشاهده میکنیم که بعد از تمام شدن رشته، استک هنوز دارای یک `0` است که خارج نشده است و در نتیجه این گرامر این رشته را تولید نمیکند.

```
C:\java\jdk1.8.0_202\bin\java.exe ...
options:
1.input a string including a and b.
2.input an arithmetic expression.
3.Exit!
1
aaabbaabb
[this grammar doesn't produce this string]
```

برنامه را برای 4 عبارت ریاضی $a(b+c)+b(a+c)$, $a-(d/(a+b))$, $a-(d/(a+b)))$, $(a(c+b)+b(a+c))$ امتحان میکنیم.

خروجی برای $a(b+c)+b(a+c)$

تبدیل شده ی عبارت ریاضیاتی به رشته ای قابل قبول به صورت **abab** است. که نحوه ی پیاده سازی متد آن قبلا بیان شده است. همچنین از روی رشته ی معادل آن میتوان مشاهده کر که با زبان ماشین پشته ای منطبق است و اگر استک را در آن طی کنیم مشاهده میکنیم که این رشته اوسط این گرامر تولید میشود.

```
options:
1.input a string including a and b.
2.input an arithmetic expression.
3.Exit!
2
a(b+c)+b(a+c)
the equivalent string is:
abab
[this grammar produces this string]
```

خروجی برای $a-(d/(a+b))$:

این رشته ریاضیاتی نیز با همان شرایط مثال قبل، قابل قبول است.

```
options:
1.input a string including a and b.
2.input an arithmetic expression.
3.Exit!
2
a-(d/(a+b))
the equivalent string is:
aabb
[this grammar produces this string]
```

خروجی برای $a-(d/(a+b)))$:

عبارت معادل این رشته ریاضیاتی را در تصویر مشاهده میکنید که توسط متد مبرطه و به طریقی که توضیح داده شده، به دست آمده است. این رشته مطابعت زبان تولیدی توسط گرامر این ماشین نیست زیرا تعداد a و b ها یکی نیست. هم چنین با توجه به موقعیت استک در پایان کار میتوان دید که استک خالی است اما عبارت هنوز دارای b است در نتیجه یک پرانتز بار اضافی وجود دارد.

```
options:
1.input a string including a and b.
2.input an arithmetic expression.
3.Exit!
2
a-(d/(a+b)))
4
the equivalent string is:
aabbbb
there is extra unopened closed parentheses [)] in the string
```

خروجی برای $(a(c+b)+b(a+c))$:

رشته ی معادل عبارت محاسباتی را در تصویر مشاهده میکنید که تعداد b و a های آنها برابر نیستند در نتیجه میتوان گفت این ماشین این رشته را تولید نمیکند. همچنین با توجه به وضعیت استک در آخر کار، چون استک خالی نبوده و رشته تمام شده است پس استک دارای یک 0 اضافی به معنی یک پرانتز باز اضافیست.

```
options:
1.input a string including a and b.
2.input an arithmetic expression.
3.Exit!
2
(a(c+b)+b(a+c))
the equivalent string is:
aabab
there is extra opened unclosed parentheses [(] in the string
```