Wyrażenia regularne są niezastąpione podczas obróbki różnego rodzaju plików tekstowych. Gdy w tekście chcemy odnaleźć konkretną frazę to nie ma problemu. Potrafi to każdy przyzwoity edytor tekstu. W Unixie mamy program `fgrep`, który to dla nas zrobi. Kłopoty zaczynają się, gdy wyszukiwana fraza nie jest ustalona, a znamy tylko regułę jak ta fraza ma wyglądać. Powiedzmy, że opracowujemy stronę internetową, na której umieściliśmy kilkadziesiąt obrazków w postaci plików JPG i teraz chcemy odnaleźć miejsce wystąpienia tych, których nazwy zaczynają się od `img`, potem jest myślnik (minus, -) a po nim dokładnie 4 cyfry i w końcu rozszerzenie `.jpg`. Możemy szukać frazy `img` albo `.jpg`, ale niekoniecznie znajdziemy to czego szukamy. Takich wyników pewnie będzie więcej bo tag do wstawiania obrazków to właśnie `img`, a poza szukanymi na stronie są inne obrazki JPG. Problem łatwo rozwiążemy korzystając z wyrażeń regularnych. Cokolwiek to teraz znaczy, szukać będziemy wyrażeń postaci:

```
img-[0-9]{4}\.jpg
```

Za chwilę wyjaśni się co oznaczają powyższe magiczne zaklęcia, ale są one dość sugestywne i moża uwierzyć, że to wyrażenie regularne da to czego szukamy.

Już na tym małym przykładzie widać pewne podobieństwo do metaznaków z shella Unixowego:

`*`	zastępuje dowolny ciąg znaków, także pusty
`?`	zastępuje jeden dowolny znak
`[abc]`	zastępuje jeden z wyszczególnionych znaków
`[!abc]`	zastępuje jeden znak spoza wyszczególnionych znaków
`[a-z]`	zastępuje jeden znak z podanego zakresu
`[!a-z]`	zastępuje jeden znak spoza podanego zakresu

Idea jest w zasadzie ta sama - wyrazić pewną regułę jak fraza jest zbudowana. W przypadku shella nie przeszukujemy zawartości plików tekstowych a system plików i fraza oznacza tutaj nazwę pliku. Używane znaki specjalne w wyrażeniach regularnych mają jednak trochę inne znaczenie.

Z powodów historycznych zestaw znaków specjalnych do budowy wyrażeń regularnych dzieli się na 3 kategorie:

- podstawowe,
- ograniczone,
- pełne.

Ze względów technicznych i praktycznych wyrażenia podstawowe i ograniczone wpadają do jednej klasy wyrażeń, a pełne do drugiej. Chodzi o to, że algorytmy związane z przetwarzaniem i kompilacją wyrażeń podstawowych i ograniczonych są znacznie prostsze. Dzięki temu programy korzystające z tej klasy wyrażeń, na przykład `grep`, `sed`, działają szybciej niż `egrep`, `awk`, `perl` korzystające z pełnego zastawu. Wspomniany wcześniej `fgrep` w ogóle nie rozumie wyrażeń regularnych, ale jest najszybszy.

Pomysł wyrażeń regularnych jest bardzo stary. Pochodzi z lat 50-tych XX wieku, czyli zrodził się zanim właściwie pojawiły się komputery. Jest cała matematyczna teoria wyrażeń regularnych. Są one stosowane w językach formalnych i regularnych.

Opracowano wiele algorytmów realizujących wyrażenia regularne i kilka rozszerzeń wzmacniających ich funkcjonalność. Najpopularniejszym rozszerzeniem i standardem zarazem jest Perl Compatible Regular Expressions (PCRE).

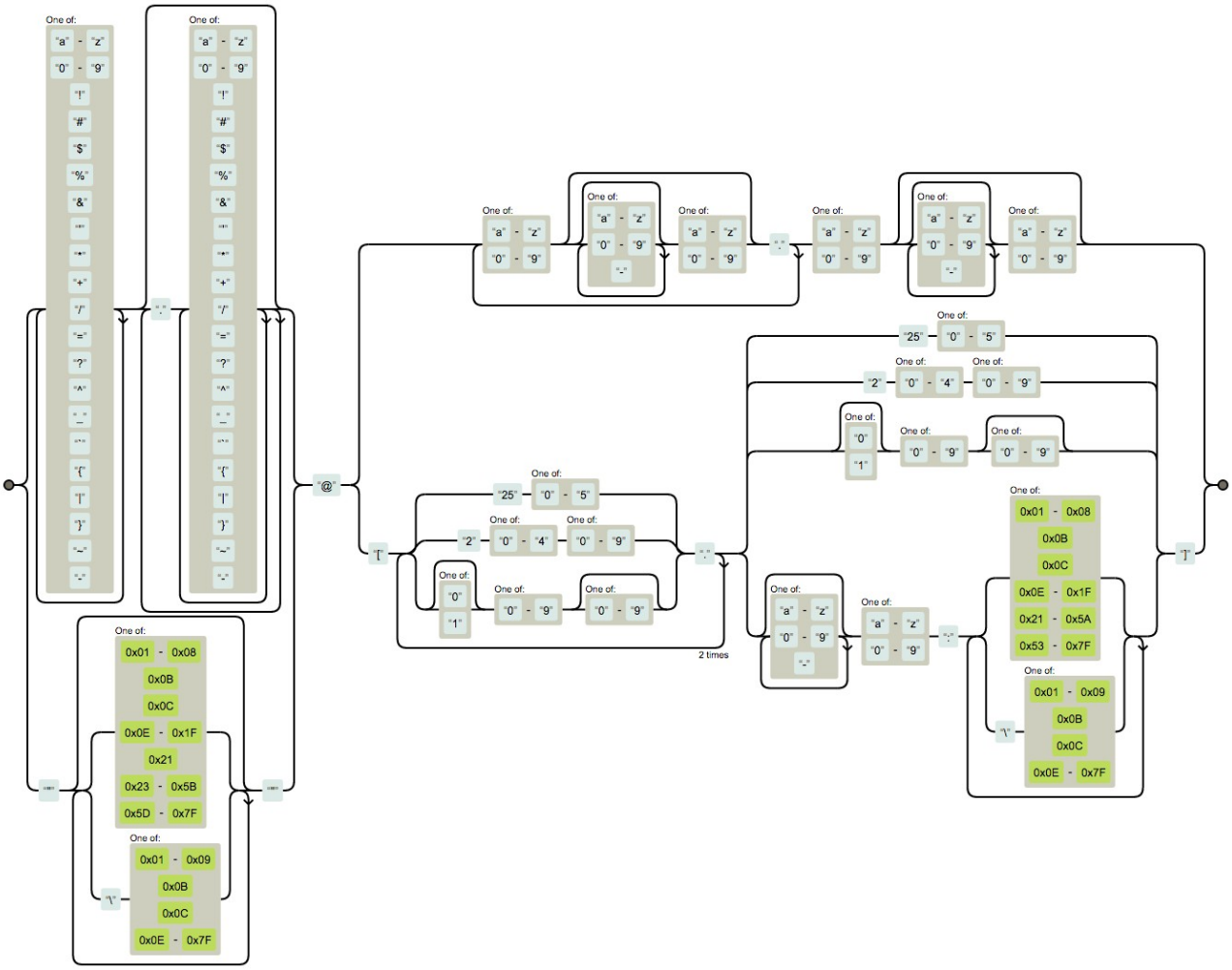
Poniżej znajduje się pełna lista znaków specjalnych do konstrukcji wyrażeń regularnych:

`\`	znak chroniący, pozbawia znak specjalnego znaczenia
`.`	jeden dowolny znak
`[abc]`	jeden z wyszczególnionych znaków
`[^abc]`	jeden znak spoza wyszczególnionych znaków
`[a-z]`	jeden znak z podanego zakresu
`[^a-z]`	jeden znak spoza podanego zakresu
`*`	zero lub więcej wystąpień wyrażenia poprzedzającego
`^`	początek linii
`$`	koniec linii
`{m}`	dokładnie *m* wystąpień wyrażenia poprzedzającego
`{m,}`	co najmniej *m* wystąpień wyrażenia poprzedzającego
`{m,n}`	od *m* do *n* wystąpień wyrażenia poprzedzającego
`(...)`	podwyrażenie regularne, odwołania poprzez \1, \2,. . .
`(...(...)...)`	możliwe zagnieżdżenia podwyrażeń regularnych
`+`	jedno lub więcej wystąpień wyrażenia poprzedzającego
`?`	brak lub jedno wystąpienie wyrażenia poprzedzającego
`|`	alternatywa, jedno z dwóch rozdzielonych wyrażeń

Zasady tworzenia wyrażeń regularnych są dosyć proste i czytelne. Same wyrażenia zbudowane zgodnie z tymi regułami mogą jednak być dość skomplikowane, jak poniższe wyrażenie opisujące adres e-mail:

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:
[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\
[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+
[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|
[1-9]?[0-9]))\.){3}(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z0-
9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\
[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

Tutaj wprawdzie zastosowano rozszerzenia PCRE, przez co trochę rośnie stopień komplikacji. Może ono służyć do sprawdzenia, czy podany napis jest poprawnie sformułowanym adresem e-mail. Na poniższym obrazku jest diagram automatu skończonego odpowiadającego temu wyrażeniu.



W zależności od tego jakiego programu, czy języka programowania używamy, wyrażenia regularne stosuje się trochę inaczej. Generalnie sama ich konstrukcja jest taka sama, chodzi o obudowę i dodatkowe opcje. Na przykład w Perlu, Javascript i PHP wyrażenia regularne to napisy umieszczone pomiędzy znakami slash `/`, na przykład:

```
/img-[0-9]{4}\.jpg/gi
```

Taki ciąg znaków oznacza, że to co pomiędzy `//` to wyrażenie regularne. Opcja `i` wyłącza rozróżnianie małych i wielkich znaków, a opcja `g` powoduje wyszukiwanie globalne. Normalnie w wyrażeniach regularnych małe i wielkie litery są odróżniane a wyszukiwanie kończy się po pierwszym dopasowaniu wzorca. Podobnie używa się wyrażeń regularnych w programach `sed` i `awk`. W programach `grep` i `egrep` nie ma potrzeby ograniczania wyrażeń znakami slash `/` a opcje przekazuje się jako parametry do programu.

Najprostsze, poprawne wyrażenie regularne to pusty ciąg znaków. Dowolny ciąg znaków nie zawierający znaków specjalnych to także wyrażenie regularne, na przykład napis: `Unix`. Takie wyrażenie pasuje wyłącznie do siebie, to znaczy jedyny napis spełniający reguły tego wyrażenia to ciąg: Unix.

`[Uu]nix` będzie pasować do fraz Unix oraz unix i tylko do nich. Oczywiście jeśli mamy napis zawierający jedną z tych fraz, na przykład Unixem, unixowi, to dopasowany zostanie odpowiedni fragment takiego napisu. Tak więc, gdy potrzebujemy wyłowić w tekście użycie słowa Unix w różnych odmianach, pisanego z wielkiej lub małej litery, to użyjemy takiego wyrażenia.

`Unix|unix` pasuje również wyłącznie do fraz Unix i unix jak wyżej. Widać zatem, że wyrażenia regularne nie są unikalne. Te same reguły możemy sformułować na wiele sposobów. To wyrażenie jest w tym sensie bardziej skomplikowane od poprzedniego, że korzysta z alternatywy

`[Uu][Nn][Ii][Xx]` pasuje do słowa Unix pisanego małymi, wielkimi lub mieszanymi znakami, np. Unix, unix, uniX itp. - razem 2^4=16 kombinacji.

`[Uu]nix[^a-z]` pasuje do słów Unix lub unix bez naszych polskich wariacji językowych. Za takim słowem musi być znak nie będący małą literą (pomijamy tutaj kwestie polskich znaków diakrytycznych ą, ę itp., one nie są w zakresie a-z), czyli spacja, przecinek, kropka itp. To wyrażenie nie będzie jednak pasować do słowa Unix lub unix na końcu linii bo oczekuje jakiegoś znaku za Unix lub unix. Można to jednak naprawić, na przykład jak poniżej.

`[Uu]nix([^a-z]|$)` Tutaj po słowie Unix lub unix musi być znak nie będący literą albo to jest koniec linii. Nawiasy `()` ograniczają działanie alternatywy `|`.

`\....` Jak wiemy kropka zastępuje dowolny znak, a backslash `\` chroni znak specjalny, zatem to wyrażenie pasuje do napisu złożonego z kropki i trzech znaków za nią, na przykład:

`.jpg` `.png` `.*+?` `....`

Celowo w dwu ostatnich przykładach zostały użyte znaki, które mają specjalne znaczenie w wyrażeniach regularnych, ale przecież mogą występować w tekście.

Znak specjalny traci swoją moc jeśli go poprzedzimy znakiem backslash `\` albo umieścimy go w nawiasach `[]`. Czyli gdy chcemy dopasować kropkę `.` albo gwiazdkę ***** to należy wpisać

`\.` `\*`

Wyrażenie `[.]` jest poprawne, ale wyciągamy armatę do ustrzelenia muchy bo kompilacja wyrażenia będzie trwała nieco dłużej (są to milisekundy na dzisiejszych komputerach, ale jeśli mamy dużo porównań w pętli to czas może być zauważalny). Gdy chcemy dopuścić kilka znaków specjalnych w dopasowaniu to naturalne jest użycie nawiasów `[]`, na przykład: `[a-z.+*?]` oznacza jeden ze znaków: mała litera, kropka, plus, gwiazdka lub pytajnik. Oczywiście same nawiasy `[]` są znakami specjalnymi. Chronimy je backslashem `\`, ale trzeba uważać na wyrażenia typu `[[]]`. Tutaj mogłoby się wydawać, że chodzi o jeden ze znaków `[]`, ale to nie tak. Pierwszy nawias `[` otwiera zestaw lub zakres. Kolejny nawias `[` jest więc traktowany jako zwykły znak. Teraz nawias `]` to zamknięcie zestawu lub zakresu, a kolejny nawias `]` niczego nie zamyka, więc traktowany jest jako zwykły znak. Ostatecznie to wyrażenie pasuje tylko do napisu `[]`. Jeśli chcemy dopasować jeden z nawiasów `[]` to najprościej tak: `[[\]]`. Tutaj backslash `\` chroni pierwszy nawias domykający `]`.

`\*.\*` Gwiazdka `*` to znak powtórzeń 0 lub więcej razy poprzedzającego znaku, ale tutaj przed `*` stoi backslash \, więc ***** traci swoją moc i staje się zwykłym znakiem. Nie dotyczy to kropki, która zastępuje jeden dowolny znak. Pasujące frazy mogą wyglądać tak:

`*x*` `*#*` `*?*` `*.*`

Znowu ostatnie dwa przykłady zawierają znaki specjalne, ale w przeszukiwanym tekście nie mają one żadnej magicznej mocy i są znakami jak każdy inny.

`2021-[01][1-9]-[0-3][0-9]` Intencją tego wyrażenia pewnie było odszukiwanie daty w formacie RRRR-MM-DD z bieżącego 2021 roku. Ma ono pewne słabości bo nie tylko nie uwzględnia lat przestępnych, ale dopuszcza na przykład 2021-13-32 co nie jest sensowną datą, ale jeśli w tekście mamy daty to tym wyrażeniem odfiltrujemy daty z roku 2021.

`^#.*$` To wyrażenie dla początkujących może wyglądać na kompletną magię, ale rozłóżmy je na czynniki pierwsze. Zaczynamy od znaku `^`, który oznacza początek linii. Pasujące napisy muszą znajdować się więc na początku linijki, nie w jej środku. Znak `#` nie ma specjalnego znaczenia jest więc traktowany dosłownie. Kropka to dowolny znak. Za kropką jest `*`, która mówi, że podwyrażenie przed nią może być powtórzone dowolną ilość razy. Tak więc `.*` to dowolny ciąg znaków, ewentualnie także pusty. Znak `$` to znak końca linii. Ostatecznie zatem nasze wyrażenie opisuje cały wiersz tekstu, w którym `#` jest pierwszym znakiem (nie mogą być przed nim ani spacje ani tabulacje). Takie wiersze to komentarze w skryptach shellowych i bardzo często w plikach konfiguracyjnych.

`^class` Słowo class (albo napis od niego zaczynający się) na początku wiersza. W PHP i w C++ tak zaczynają się definicje klas obiektów.

`[aiowz]$` Na początku tego wyrażenia jest spacja, tylko jej nie widać. To są tak zwane sieroty w języku polskim, czyli spójniki a, i, o, w, z jako ostatnie w wierszu. W naszym języku to błąd i nie powinno zostawiać się sierot na końcu wiersza tekstu.

`img[^0-9]\.png` Wygląda na wzorzec nazwy pliku z bitmapą PNG. Częścią nazwy pliku jest fraza `img`, za którą jest znak nie będący cyfrą a potem kropka i rozszerzenie `png`. Pasujące napisy:

`imgA.png` `imgB.png` `imgC.png` `imgD.png`

`[A-Z][A-Z]*` Słowo długości co najmniej 1 zbudowane z wielkich liter, na przykład:

`UNIX`

`[a-z- .]*@[a-z- .]*\.[a-z]*` Poprawne adresy e-mail będą pasować do tego wyrażenia. Ale pasować będzie także takie coś:

`jack@domain.` `@domain.` `@.`

co nie jest poprawnym adresem e-mail.

`[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}` Adres IP w wersji IPv4. Poprawne adresy będą pasować, ale będą pasować także takie dziwactwa:

`256.256.256.256` `999.999.999.999`

Dwa powyższe przykłady pokazują, że świat jest skomplikowany i trudno go opisać prostymi regułami. To nie są słabości wyrażeń regularnych bynajmniej. Wszystko zależy od tego co chcemy uzyskać. Jeśli mamy w tekście poprawne adresy e-mail, czy adresy IP to dwa powyższe wyrażenia z powodzeniem je wszystkie wyłapią. Gdy chcemy jednak za pomocą takiego wyrażenia zweryfikować, czy na przykład podany w formularzu HTML adres e-mail albo IP są poprawne to te wyrażenia są za słabe bo dopuszczają adresy niepoprawne.

`[a-z-_.]{3,}@[a-z-_.]{2,}\.[a-z]{2,}` Tutaj mamy nieco poprawione wyrażenie opisujące adres e-mail. Muszą być co najmniej trzy znaki przed `@`, dwa znaki nazwy domenowej po `@` i przynajmniej dwie małe litery w TLD (Top Level Domain). To jednak tylko małe usprawnienie bo 100% poprawności daje nam wyrażenie, które widzieliśmy na początku wykładu.

`[0-9]{3}-[0-9]{3}-[0-9]{3}` Numer telefoniczny, w każdym razie wyrażenie to pasuje do napisów postaci:

`604-065-599` `501-123-456`

`[1-9][0-9]{2}-[0-9]{3}-[0-9]{3}` Praktycznie to samo co wyżej, tylko na samym początku nie może być cyfra 0.

`(Unix) \1` Nawiasy okrągłe `()` ograniczają podwyrażenia, które możemy potem wykorzystać przy podstawieniach. Tutaj słowo Unix ma się pojawić w dwóch kopiach, najpierw literalnie, a po spacji jego kopia. W tym wypadku łatwo możemy to wyrażenie uprościć: `Unix Unix`, bo wiemy co kopiujemy, ale gdy chcemy dopasować powtórzenie dowolnego znaku to zastosowanie nawiasów i podstawień jest jedynym sposobem: `(.) \1` To wyrażenie pasuje do dowolnego znaku powtórzonego po spacji. Powtórzone spójniki: `[ ]([aiowz]) \1[ ]` Tutaj pierwsza i ostatnia spacja zostały ujęte celowo w nawiasy `[]` bo inaczej nie byłoby ich widać. Nawiasy te nie są konieczne jednak.

`([ABCDE]{3}) ([abcde]{3}) \2 \1` Nie widać jakiegoś sensownego zastosowania tego wyrażenia, ale podstawienia można używać w dowolnej kolejności i można ich używać kilka razy. Tutaj mają być 3 wielkie litery spacja 3 małe litery a potem ich powtórzenia w odwrotnej kolejności, na przykład:

```
ABC xyz xyz ABC
```

`([0-9]{4})-([0-9]{2})-([0-9]{2})` Tutaj nawiasy `()` zostały użyte trochę na zapas. W tej postaci to wyrażenie jest równoważne z:

```
[0-9]{4}-[0-9]{2}-[0-9]{2}
```

To wyrażenie może opisywać daty w formacie RRRR-MM-DD. Po co nam te nawiasy? Gdy chcemy w tekście daty w tym formacie przekształcić do formatu DD-MM-RRRR to w podstawieniu wyników przeszukiwania użyjemy: `\3-\2-\1`. Dosyć prosta operacja, ale spróbujmy to zrobić bez wyrażeń regularnych! W niektórych aplikacjach, jak LibreOffice, zamiast \n używa się $n.

`(([0-9]{4})-[0-9]{2}-[0-9]{2})` To praktycznie to samo wyrażenie, tylko nawiasy `()` zostały inaczej ustawione. Jako \1 zostanie podstawiona cała, zewnętrzna fraza, a za \2 fraza wewnętrzna. Kolejność numerowania podwyrażeń jest taka, że najpierw numerowane są podwyrażenia z lewej potem ich ewentualne podwyrażenia, a potem wyrażenia dalej na prawo na pierwszym poziomie. W tym przykładzie mamy datę w formacie RRRR-MM-DD i za \1 podstawiona zostanie cała data, a za \2 tylko sam rok (4 pierwsze cyfry).

`Unix files?` Na pierwszy rzut oka to nie musi być wyrażenie regularne, ale jeśli już nim jest to znak (podwyrażenie) przed znakiem `?` jest opcjonalny, może być obecny lub nie. Zatem pasują tutaj dwa napisy:

```
Unix files                    Unix file
```

Podwyrażeniem przed `?` jest znak `s` i to on jest opcjonalny.

`Unix (file)?` Tutaj opcjonalny jest cały napis `files`. Nawiasy `()` ogarniczają podwyrażnie przed znakiem specjalnym `?`.

`(Unix|DOS) file` Mamy tu przykład alternatywy. Dotyczy ona tutaj słów Unix i DOS, które mogą wystąpić zamiennie - jedno lub drugie przed spacją i słowem `files`:

```
Unix file                    DOS file
```

Pamiętajmy, że `UNIX file`, czy `Dos file` nie zostaną dopasowane. Małe i wielkie litery są rozróżniane.

`[ ]+$` Bardzo wygodne wyrażenie do wyłapywania spacji pozostawionych na końcu linii tekstu.

`"[^"]+"` Przykład częstego tricku stosowanego, aby dopasować napisy ograniczone jakimiś nawiasami, albo jak tutaj, cudzysłowami. To wyrażenie pasuje do dowolnych, niepustych napisów umieszczonych w cudzysłowach razem z nimi, to znaczy, na przykład:

```
"I hate Unix"                    "I love regular expressions"
```

`img-[0-9]{3}\.(jpg|png|tiff|gif)` Gdybyśmy chcieli coś zrobić z nawami plików graficznych JPG, PNG, TIFF lub GIF, których nazwy zaczynają się od `img` potem jest myślnik i 3 cyfry, to wyrażenie może się przydać. Pasuje ono do takich właśnie nazw plików o jednym z wymienionych rozszerzeń.

`[0-9]+(EUR|PLN)` To może być cena zaokrąglona do wartości całkowitej w zł lub euro.

`[-+]?[0-9]+\.[0-9]+` Na początku może być opcjonalnie znak – lub `+`. Potem ma być przynajmniej jedna cyfra, za nią kropka i po niej znowu przynajmniej jedna cyfra. To jest wzorzec liczby rzeczywistej, gdzie kropka rozdziela część dziesiętną.

`[-+]?[0-9]*\.?[0-9]+` To jest nieco bardziej ogólny wzorzec wartości numerycznej dopuszczający liczby rzeczywiste z kropką rozdzielającą część dziesiętną i liczby całkowite.

`\$[A-Za-z0-9_-]+` Znak `$` jest specjalny, ale tutaj został ochroniony więc ma się pojawić jako pierwszy w dopasowywanym napisie. Mogą to być nazwy zmiennych shellowych, w PHP albo w Perlu.