Engineer

Regulært udtryk

et mønster, der beskriver, hvilke tekststrenge der matcher

Hvad det er

Et regulært udtryk er et kort mønster, der beskriver en mængde tekststrenge, så et program kan udvælge dem, der hører til. POSIX-standarden, som The Open Group og IEEE udgiver, indleder sit kapitel om emnet med at beskrive regulære udtryk som en måde at udvælge bestemte strenge fra en mængde af tegnstrenge. I daglig tale og i mange funktions- og biblioteksnavne skrumper betegnelsen til regex eller til regexp, som er stavemåden i JavaScripts RegExp-objekt og i regexp-pakken i Go.

De fleste tegn i et mønster står for sig selv: et `a` i mønsteret matcher et `a` i teksten. Nogle få er særlige. Et punktum matcher et vilkårligt tegn, `*` gentager det foregående nul eller flere gange, et udtryk i kantede parenteser som `[0-9]` matcher ét tegn fra en mængde, `^` og `$` binder mønsteret til starten og slutningen, og et interval som `{4}` kræver et bestemt antal gentagelser. Så `^[0-9]{4}$` beskriver enhver streng, der består af præcis fire cifre, uden andre tegn før eller efter. Standardens eget eksempel på et interval er `c{3}` anvendt på `abababccccccd` : det matcher det syvende til niende tegn, de første tre af de seks c'er.

Der findes mere end én dialekt. POSIX definerer selv to: basale regulære udtryk (BRE), som de fleste af standardens værktøjer bruger, og udvidede regulære udtryk (ERE), som nogle bruger i stedet. C-funktionerne `regcomp()` og `regexec()` tilbyder begge. Forskellen er ikke kosmetisk. I et ERE betyder `a+` ét eller flere `a` . I et BRE er `+` et almindeligt tegn, og standarden overlader det til hver implementering, om `\+` betyder ét eller flere eller et bogstaveligt plustegn.

Den anden store familie nedstammer fra Perl. ECMAScript-specifikationen fastslår, at formen og funktionaliteten af JavaScripts regulære udtryk er modelleret efter Perl 5's faciliteter til regulære udtryk. Regexp-pakken i Go beskriver sin syntaks som den generelle syntaks, som Perl, Python og andre sprog bruger, og Pythons `re` -modul er et tredje medlem. Motorerne deler det meste af notationen, men ikke alle detaljer. Forkortelsesklasserne er et godt eksempel: `\d` (et ciffer) og `\w` (et ordtegn) ser ens ud overalt, men hvad de dækker, afhænger af motoren og dens flag, som afsnittet om praksis nedenfor viser.

Brug begrebet i en beslutning

Brug »Hvad er et regulært udtryk?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. Ét team, der tager en idé fra skitse til noget, jeres kunder kan logge ind på, i stedet for at kaste den mellem tre leverandører.

Se vores ydelser Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. Har ført en GIS-kortapp til 10x omsætningsvækst — 10x omsætningsvækst, drevet af kortproduktet

Har ført en GIS-kortapp til 10x omsætningsvækst Kontakt os →

Hvorfor det betyder noget

Regulære udtryk sidder i en stor del af den software, der ikke ligner bearbejdning af tekst: validering af formularfelter, filtre, der trækker værdier ud af loglinjer, routingregler i webservere, søg og erstat i teksteditorer. Hvert mønster er kort, og derfor er det let at overse, hvad det faktisk accepterer. To spørgsmål afgør resultatet, og svaret på begge afhænger af motoren: hvilket match der vælges, når flere er mulige, og hvor lang tid det må tage at finde et.

POSIX vælger det match, der står længst til venstre, og blandt dem det længste: af alle de match, der begynder tidligst i teksten, vinder det længste, på engelsk leftmost-longest. Standardens eksempler er BRE'et `bb*` , der matcher andet til fjerde tegn i `abbbc` , og ERE'et `(wee|week)(knights|night)` , der matcher alle ti tegn i `weeknights` . Perl-familien arbejder anderledes. Den prøver alternativerne i den rækkefølge, de står skrevet, og tager det første, der fører til et match, hvad dokumentationen til Go kalder leftmost-first. Det samme mønster på den samme tekst kan derfor give to svar:

```
$ echo ab | grep -oE 'a|ab'
ab
>>> re.search("a|ab", "ab").group()
```

```
'a'
> "ab".match(/a|ab/)[0]
'a'
```

Go tilbyder begge opførsler. Funktionen `Compile` bruger leftmost-first, mens `CompilePOSIX` begrænser syntaksen til POSIX ERE og skifter til leftmost-longest. Selv da beholder pakken sin egen regel for, hvilke delmatch der vælges, og dokumentationen forklarer hvorfor: POSIX-reglen er ifølge den beregningsmæssigt uoverkommelig og ikke engang veldefineret.

Det andet spørgsmål er tid. Pythons `re` backtracker ligesom de fleste motorer i Perl-familien. Når en del af mønsteret ikke kan komme videre, går motoren et skridt tilbage og prøver en anden måde at matche det foregående på. Pythons dokumentation viser idéen med `a*a` mod `aaaa` : `a*` tager først alle fire bogstaver og giver derefter ét tilbage, så det sidste `a` kan matche. For de fleste mønstre er omkostningen usynlig. For nogle vokser antallet af måder at prøve eksponentielt med inputtets længde, og en streng på et par dusin tegn kan holde en processor beskæftiget i minutvis. MITRE katalogiserer dette som CWE-1333, Inefficient Regular Expression Complexity, og nævner ReDoS (regular expression denial of service) og catastrophic backtracking som andre navne for det. Regexp-pakken i Go blev designet omkring det modsatte løfte. Dokumentationen fastslår, at matchningen garanteret kører i tid, der er lineær i inputtets størrelse, og tilføjer, at de fleste open source-implementeringer ikke giver en sådan garanti.

Sådan ser det ud i praksis

Tag en tilmeldingsformular til kunder i Danmark og Ukraine og et fornavnefelt, der kun skal acceptere bogstaver. Det oplagte mønster er `[A-Za-z]+` , og i Python 3.15 opfører det sig sådan her:

```
>>> import re
>>> re.fullmatch(r"[A-Za-z]+", "Mia")
<re.Match object; span=(0, 3), match='Mia'>
>>> re.fullmatch(r"[A-Za-z]+", "Søren")
>>> re.fullmatch(r"[A-Za-z]+", "Олена")
>>> re.fullmatch(r"\w+", "Søren_42")
<re.Match object; span=(0, 8), match='Søren_42'>
>>> re.fullmatch(r"[^\W\d_]+", "Олена")
```

```
<re.Match object; span=(0, 5), match='Олена'>
>>> re.fullmatch(r"[^\W\d_]+", "Søren_42")
>>>
```

Et kald, der returnerer `None`, udskriver slet ingen linje i den interaktive skal, så ASCII-klassen afviser Søren og Олена (Olena), to helt almindelige navne i København og Kyiv. Skifter man til `\w`, kommer de igennem, men i Python dækker `\w` alle tegn, som `str.isalnum()` er sand for, plus understregningen, så klassen lukker også `Søren_42` ind. Klassen `[^\W\d_]` udelukker tegn, der ikke er ordtegn, decimalcifre og understregningen. Den tillader også visse numeriske tegn, som ikke er decimalcifre, og er derfor ikke en streng test for udelukkende bogstaver. Til navne på mere end ét ord accepterer `[^\W\d_]+( [^\W\d_]+)*` både `Anne Sofie Østergaard` og `Олена Коваль`. Mønsteret afviser stadig et `Anne-Marie` med bindestreg, og det er en grund til at holde et navnefelt så rummeligt, som forretningen kan bære. POSIX-værktøjer har samme problem i en anden form: standarden lader betydningen af et tegnområde som `A-Z` være uspecificeret uden for POSIX-localet og tilbyder klassen `[[:alpha:]]`, hvis medlemmer bestemmes af det aktuelle locale.

At få det rigtigt betyder mere end den ene formular. En af dette websteds ydelser, Internationalisering & lokalisering, beskriver målet som software, der læses naturligt på hvert marked, man sælger på, helt ned til den måde, navne sorteres på. Et navnefelt, der afviser Søren, svigter det mål længe før nogen sortering finder sted.

Cifre rejser det modsatte spørgsmål. Et dansk postnummer består af fire cifre, så `\d{4}` ligner en sikker kontrol af `2200`. I Python matcher `\d` i et tekstmønster dog ethvert decimalciffer i Unicode, ikke kun 0 til 9, og accepterer det samme postnummer skrevet med arabisk-indiske cifre:

```
>>> s = "\u0662\u0662\u0660\u0660"
>>> print(s, int(s))
٢٢٠٠ 2200
>>> re.fullmatch(r"\d{4}", s)
<re.Match object; span=(0, 4), match='٢٢٠٠'>
>>> re.fullmatch(r"\d{4}", s, re.ASCII)
>>> re.fullmatch(r"[0-9]{4}", s)
>>>
```

JavaScript står på den anden side. ECMAScript-specifikationen definerer `\d` som præcis de ti tegn 0 til 9, så `/^\d{4}$/.test(s)` returnerer `false` i Node.js for den samme streng, mens `/^\p{Nd}{4}$/u` returnerer `true`. En formular, der kontrolleres i browseren med den ene regel og på serveren med den anden, vil være uenig med sig selv om det samme input. Pythons `int()` læser strengen som 2200, men en databasekolonne, en opslagstabel eller en anden tjeneste gør det måske ikke. Beslut, hvilke cifre der menes: `[0-9]` eller flaget `re.ASCII`, når kun ASCII-cifre duer, og `\p{Nd}` med flaget `u` i JavaScript, når ethvert decimalciffer er velkomment.

Hvor det går galt

Et mønster, der fordobler tiden for hvert bogstav. Et navnemønster, der skal tillade ord adskilt af enkelte mellemrum, skrives let som `([^\W\d_]+ ?)+`. På et gyldigt navn er det hurtigt. På en række bogstaver efterfulgt af et tegn, det ikke kan acceptere, prøver motoren alle måder at dele bogstaverne op i ord, før den giver op, og antallet af måder fordobles for hvert bogstav. På én maskine med Python 3.15 gav den hurtigste af fem kørsler ved hver længde disse tider for `fullmatch` på en streng af bogstaver efterfulgt af `!`:

```
letters   seconds
     18     0.017
     20     0.067
     22     0.27
     24     1.0
     26     4.1
```

Hvert bogstav fordobler tiden, så to bogstaver mere firedobler den. Fremskrevet med den faktor ville tredive bogstaver tage omkring et minut og toogtredive mere end fire minutter. Det er netop det tilfælde, CWE-1333 beskriver: forsøgene på backtracking vokser eksponentielt med inputtet, inputtet behøver ikke at matche, og inputtet kan være langt nok. Omskrivningen `[^\W\d_]+([^\W\d_]+)*` beskriver de samme navne uden en indlejret kvantor og afviser strengen på 24 bogstaver på omkring et mikrosekund og en million bogstaver efterfulgt af `!` på omkring tredive millisekunder. Fjern indlejrede kvantorer, begræns inputtets længde, før det matches, og kør mønstre, der ser input, man ikke kan stole på, på en motor med en garanti for lineær tid, som den i Go.

Søgning, hvor hele værdien skal matche. I Python accepterer `re.search` et match hvor som helst i strengen, og `re.match` forankrer kun i starten; Python 3.15 giver den det tydeligere navn `prefixmatch` og markerer det gamle navn som soft deprecated med den bemærkning, at andre sprog bruger match om det, Python altid har kaldt search. Så `re.prefixmatch(r"\d{4}", "22000")` lykkes på de første fire cifre, og `re.search(r"\d{4}", "DK-2200")` finder postnummeret inde i strengen. Selv `re.search(r"^\d{4}$", "2200\n")` lykkes, fordi `$` også matcher lige før et linjeskift i slutningen. `re.fullmatch` , der har fandtes siden Python 3.4, afviser alle tre. Brug `fullmatch` , når et mønster beskriver hele værdien, eller forankr med `\A` og `\Z` , som Python 3.14 også staver `\z` .

En tilladelsesliste med et løst punktum. En kontrol, der kun lader omdirigeringer gå til virksomhedens eget websted, kunne skrives som `^https://www.example.com` . Den accepterer `https://www.example.com.evil.example/` , fordi mønsteret holder op, før værtsnavnet gør, og hvert punktum uden escape matcher et vilkårligt tegn, så en vært, der adskiller sig fra den rigtige ved ét enkelt tegn i stedet for et punktum, også slipper igennem. At escape punktummerne, som `re.escape("www.example.com")` gør, og afslutte mønsteret med `/` lukker begge huller. Eller bedre: parse URL'en, og sammenlign dens værtsnavn med det forventede som en hel streng.

Et mønster, der flyttes mellem motorer. Et mønster, der er testet med `grep -E` og sat ind i Python, kan vælge et andet match, som eksemplet med `a|ab` viser, og et, der virker i ét BRE-værktøj med `\+` , kan betyde et bogstaveligt plustegn i et andet. Det `\d` , der accepterer arabisk-indiske cifre i Python, afviser dem i JavaScript. Test hvert mønster i den motor, der skal køre det, med de flag, det skal køre med.

Et globalt flag, der husker. I JavaScript gemmer et regulært udtryk med flaget `g` eller `y` positionen, hvor dets seneste match sluttede, i `lastIndex` og starter næste søgning dér; efter et mislykket forsøg nulstiller det `lastIndex` til 0. Et mønster, der ligger i en konstant og genbruges til validering, svarer så forskelligt ved hvert andet kald:

```
> const re = /^\d{4}$/g;
undefined
> re.test("2200")
true
> re.test("2200")
```

```
false
> re.test("2200")
true
```

Udelad flaget `g` på mønstre, der bruges med `test()`, eller sæt `lastIndex` til 0 før hvert kald.

Tekst, der ser ens ud, men ikke er det. Navnet André kan gemmes med é som ét samlet tegn eller med et almindeligt e efterfulgt af en kombinerende accent. De to ser identiske ud på skærmen, men POSIX fastslår, at matchning følger det bitmønster, der bruges til at kode et tegn, ikke dets grafiske fremtræden, og Pythons `re.search("André", "Andre\u0301")` returnerer `None`. Efter `unicodedata.normalize("NFC", ...)` på teksten finder kaldet navnet. Normalisér mønsteret og teksten til samme form før matchning, med `unicodedata.normalize` i Python eller `String.prototype.normalize` i JavaScript.

Læs mere

- The Open Group Base Specifications Issue 8: Regular Expressions — POSIX-definitionen: BRE og ERE, udtryk i kantede parenteser, tegnklasser og locales samt reglen om leftmost-longest.
- regexp package (Go) — garantien for lineær tid, leftmost-first over for leftmost-longest, og hvorfor CompilePOSIX afviger fra POSIX ved delmatch.
- re — Regular expression operations (Python) — syntaksen, Unicode- og ASCII-klasser, backtracking og possessive kvantorer samt search, prefixmatch og fullmatch.
- ECMAScript: Text Processing (living draft, ECMAScript 2027) — JavaScripts regulære udtryk: arven fra Perl 5, definitionen af `\d`, Unicode-egenskabsescapes og hvordan lastIndex bruges.
- CWE-1333: Inefficient Regular Expression Complexity — catastrophic backtracking og ReDoS: de betingelser, der gør et mønster udnytteligt, observerede eksempler og modforholdsregler.

Ordliste

Vælg ud fra jeres faktiske vilkår

Beskriv arbejdsgangen, de eksisterende grænseflader og de brugere, der har brug for ændringen. Vi kan definere en testbar leverance og en overdragelse, der gør teamet i stand til at vedligeholde løsningen.

Se, hvad det koster Kontakt os →

Gør begreberne praktiske

Spørg, hvordan »Hvad er et regulært udtryk?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

Se vores ydelser

welcome@engineer.company +45 50 10 61 84 København, Danmark

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://engineer.company/da/glossary/regular-expression/