Edycja tekstu i struktury danych

Paweł Józiak

Wydział Matematyki i Informatyki Politechniki Warszawskiej

12 XII 2022

Plan zajęć

Plan zajęć

Serializacja – ogólnie

- *Serializacją* nazywamy metodę reprezentacji danych w postaci bajtów.
- Celem serializacji może być:
  - transfer danych,
  - zapis danych (w bazach danych, urządzeniach dyskowych etc),
  - zdalne wywoływanie procedur (*RPC*), np. `SOAP` (*Simple Object Access Protocol*),
  - kolportaż obiektów w programowaniu zorientowanym na obiektowe rozproszenie (*COM*, *CORBA* i inne),
  - rozpoznawanie zmienności danych w czasie.
- Kluczową cechą serializacji jest uniezależnienie od platformy sprzętowej, m.in.
  - od systemu operacyjnego,
  - od kolejności reprezentacji bajtów (*Little Endian* vs *Big Endian*),
- Projektowanie serializacji obiektów podlega ryzykom jak:
  - łamanie hermetyzacji obiektów,
  - niejednolita obsługa pomiędzy protokołami (SOAP, RPC itd),
  - niejednolite wsparcie wśród języków programowania (np Python 2.7 vs Python 3.8).

Serializacja a protokoły

- Zagadnienie reprezentacji danych w formacie komputerowym dotyka trzech obszarów:
  - protokółów komunikacyjnych,
  - serializacji,
  - kodowania.
- *Protokoły komunikacyjne* określają reguły, syntaks, semantykę oraz metody synchronizacji agentów, a także sposób naprawy błędów. Na przykład standard protokołu IMAP (poczty przychodzącej) opiera się o metodę serializacji e-maili za pomocą standardu MIME, a także szereg innych reguł pracy aplikacji klienckich tego protokołu.
- *Kodowanie* oznacza odwzorowania przypisujące znakom sekwencje bajtów.
- *Serializacja* oznacza odwzorowanie przypisujące reprezentacje obiektom.
- Istnieją metody serializacji obiektów do bajtów (np. *marshmallow*), albo do tekstu (np. *base64*).

Standardy serializacji

- `XNS` był pierwszym popularnym standardem serializacji (1977), pochodzącym od Xerox, dla celów RPC.
- `XML` to pierwszy standard serializacji oparty o pliki tekstowe (1996); istotny aż do dziś (np. `pom.xml` projektów Java).
- `CSV` (*Comma-Separated Values*) – format dedykowany dla danych tabelarycznych, oparty o pliki tekstowe; pierwsze użycie w latach '60, ale ustandaryzowany dopiero w 2005.
- `JSON` (*Java Script Object Notification*) – stworzony na potrzeby JS dla serializacji list i obiektów JS (odpowiedniki to `dict` w języku Python, `hashmap` w języku Java), lżejsza alternatywa dla `XML` (efektywnie równoważne).
- `YAML` (*Yet Another Markup Language* albo *YAML Ain't Markup Language*) – rozszerzenie standardu JSON o m.in. kotwice (zmienne), komentarze, odpakowywanie list, określenie typów etc.

Serializacja – przykłady

Słownik Pythona

```
{
  'bool': True,
  'string': "Napis",
  'liczby': [
    -2,
    0.5
  ]
}
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <bool type="bool">True</bool>
  <string type="str">Napis</string>
  <liczby type="list">
    <item type="int">-2</item>
    <item type="float">0.5</item>
  </liczby>
</root>
```

JSON

```
{
  "bool": true,
  "string": "Napis",
  "liczby": [
    -2,
    0.5
  ]
}
```

YAML - minimalny

```
bool: true
liczby:
- -2
- 0.5
string: Napis
```

YAML – pełny

```
"bool": !!bool "true"
"liczby":
- !!int "-2"
- !!float "0.5"
"string": "Napis"
```

Serializacja – przykłady

2D tablica (Python)

```
[['float', 'int', 'bool', 'str'],
 [0.1, 1, True, 'jeden'],
 [0.2, 2, False, 'dwa']]
```

CSV

```
"float","int","bool","str"
0.1,1,True,"jeden"
0.2,2,False,"dwa"
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <item type="list">
    <item type="str">float</item>
    <item type="str">int</item>
    <item type="str">bool</item>
    <item type="str">str</item>
  </item>
  <item type="list">
    <item type="float">0.1</item>
    <item type="int">1</item>
    <item type="bool">True</item>
    <item type="str">jeden</item>
  </item>
  <item type="list">
    <item type="float">0.2</item>
    <item type="int">2</item>
    <item type="bool">False</item>
    <item type="str">dwa</item>
  </item>
</root>
```

JSON

```
[
  [
    "float",
    "int",
    "bool",
    "str"
  ],
  [
    0.1,
    1,
    true,
    "jeden"
  ],
  [
    0.2,
    2,
    false,
    "dwa"
  ]
]
```

YAML

```
- - float
  - int
  - bool
  - str
- - 0.1
  - 1
  - true
  - jeden
- - 0.2
  - 2
  - false
  - dwa
```

Plan zajęć

Vi IMproved

- `VIM` (*Vi IMproved*) to rozszerzenie starego edytora tekstu `vi` o szereg przydatnych własności:
  - wsparcia dla `vim` na praktycznie wszystkich systemach,
  - interfejs GUI w oparciu o GTK+ (`gvim`),
  - wsparcie dla wielojęzyczności,
  - wsparcie dla zróżnicowanego syntaksu (podświetlanie) oraz korekta słownikowa (*spell-check*),
  - wsparcie dla rozszerzonych wyrażeń regularnych,
  - praktycznie nieskończone *undo* (rejestr 1000 ostatnich modyfikacji),
  - wsparcie dla myszy,
  - obsługa plików skompresowanych,
  - rozszerzalność przez plug-iny
- i wiele innych.

Tryby pracy *VIM*a

- `VIM` pracuje w trzech trybach:
  - *tryb komend* (domyślny, można do niego w każdej chwili wrócić naciskając <ESC>)
    - ★ edycja instrukcji bufora
    - ★ zarządzanie plikiem oraz komendami konfiguracji (dzięki `:`)
    - ★ komendy powłoki (dzięki `:!`)
  - *trybie wprowadzania*, w którym wprowadzamy zmiany w tekście za pomocą klawiatury/schowka,
  - *trybie wizualnym*, w którym zaznaczamy znaki/linie do dalszego procesowania.

Tryb komend *ex*

- Komendy, których nazwa zaczyna się od `:` nazywane są komendami *ex*.
- Większość komend ex tyczy się zarządzania plikiem, VIMem i wyświetlania podpowiedzi.
- Wpisanie `:` i użycie strzełek góra/dół pozwala przewijać historię wpisanych komend ex.
- `%` w komendach ex reprezentuje *bieżący plik*.
- Większość komend ex jest pełnymi słowami języka angielskiego. W przypadku braku niejednoznaczności, niektóre z nich mają formy skrócone – jednoliterowe Na przykład `:q` oraz `:quit` są tą samą komendą.

Komendy *menu plik*

- Części komend w nawiasach kwadratowych są opcjonalne (wariant pełny vs wariant skrócony komedy)
- Ze względów praktycznych często używa się wariantu skróconego.
  - `:q[uit]` – zamknięcie obecnego procesu VIMa,
  - `:w[rite]` – zapisanie bieżącego pliku,
  - `:w[rite]` *`plik`* – zapisanie bieżącego pliku pod nazwą *plik*,
  - `:wq` – zapisz i wyjdź,
  - `:q!` – wyjdź i porzuć zmiany,
  - `:e[dit]` *`plik`* – otwórz plik o nazwie *plik* do edycji,
  - `:e %` – przeładuj bieżący plik.

Uzyskiwanie pomocy

- Dla czytelności, przykłady na tym slajdzie używają wersji pełnej komendy `:help`. Wariant krótki jest równoważny.
- Uzyskiwanie pomocy:
  - `:h[elp]` – wyświetla sekcję pomocy,
  - `:help` *`komenda`* – wyświetla podsekcję pomocy dotyczącą komendy VIMa *komenda*.
- Przykłady:
  - `:help :q` – wyświetla informacje o komendzie `:q`,
  - `:help i` – wyświetla informacje o komendzie `i`,
  - `:help :set` – wyświetla informacje o komendzie `:set`,
  - `:help number` – wyświetla informacje o ustawieniu `number` (zob. następujące slajdy).
- Zamknięcie okna pomocy odbywa się komendą `:q`.

Konfigurowanie VIMa

- Mieszamy tutaj krótkie i długie nazwy ustawień (jak większość użytkowników VIMa).
  - `:set` – wyświetla dostępne opcje,
  - `:set spell` – włącza korektę słownikową,
  - `:set nospell` – wyłącza korektę słownikową,
  - `:set spelllang=pl` – ustawia język słownika do korekty (na polski),
  - `:set nu[mber]` – włącza numerację linii,
  - `:set nonu[mber]` – wyłącza numerację linii,
  - `:set enc=`*`encoding`* – ustawia kodowanie *encoding* do ładowania pliku; wymaga przeładowania bieżącego pliku, by opcja poskutkowała również na nim,
  - `:set fenc=`*`encoding`* – ustawia kodowanie używane do zapisu pliku (`fileencoding`),
  - `:set ts=`*`n`* – ustawia rozmiar tabulacji na `n` spacji (`tabsize`),
  - `:set sw=`*`n`* – ustawia rozmiar wcięcia na `n` spacji (`shiftwidth`).

Wykonywanie komend Powłoki

- Komendy zaczynające się od `:!` są wykonywane przez Powłokę,
- Przykłady:
  - `:!date` – która teraz godzina?
  - `:!chmod +x %` – dodanie bieżącemu plikowi atrybutu `x` wykonywalności.
  - `:!./%` – wykonanie bieżącego pliku.
  - `:!gcc %` – skompilowanie bieżącego pliku kompilatorem GNU `C`.
  - `:!man 3 printf` – wyświetlenie strony podręcznika o funkcji `printf()` .

Przechodzenie do trybu wprowadzania/edycji

- Każda z poniższych komend zmienia tryb pracy VIMa na wprowadzanie:
  - `i` – wprowadzanie przed kursorem,
  - `I` – wprowadzanie na początku linii kursora,
  - `a` – wprowadzanie za kursorem (*append*)
  - `A` – wprowadzanie na końcu linii kursora,
  - `o` – wprowadzanie w nowej linii za kursorem (*open*),
  - `O` – wprowadzanie w nowej linii przed kursorem (*open*).
- Powrót do trybu komend jest możliwy za pomocą <ESC>.

Podstawowe komendy edycji

- `x` – usunięcie znaku/zaznaczenia,
- `r` – wymiana znaku,
- `.` – powtórzenie ostatniej komendy,
- `u` – *undo*, cofnięcie zmiany,
- `<CTRL>+ <R>` – *redo*, przywrócenie cofniętej zmiany.

- Komendy mogą być poprzedzone liczbą, aby wskazać na ich wielokrotne wykonanie. Np `2u` cofa dwie zmiany, `4x` usuwa 4 znaki itd.

Przeszukiwanie

- Jeśli to możliwe, VIM opiera się o zewnętrzne narzędzia. Przeszukiwanie działa tak samo, jak w `less` :
  - `/wzorzec` – przeszukuje pod kątem wystąpienia wzorca *wzorzec* w tekście w przód,
  - `?wzorzec` – przeszukuje pod kątem wystąpienia wzorca *wzorzec* w tekście w tył,
  - `n` – przejście do kolejnego wystąpienia,
  - `N` – przejście do poprzedniego wystąpienia.
- VIM obsługuje wyrażenia regularne PCRE.
- Znaki specjalne muszą być *eskejpowane* znakiem dla zachowania znaczenia dosłownego,
- Wprowadziwszy `/` albo `?` można używać strzałek w górę/dół dla przeglądania historii używanych wzorców.
- Odnalezione dopasowania są podświetlone. Dla wyłączenia podświetlenia, użyj `:nohl` .

Podstawowe komendy nawigacji

- `1G` – przejdź do pierwszej linii (jak w `less`),
- `G` – przejdź do ostatniej linii (jak w `less`),
- `nG` – przejdź do linii numer *n* (jak w `less`),
- `0` – ustaw kursor na początku linii,
- `^` – ustaw kursor na pierwszym nie-białym znaku linii (jak w *grep*),
- `$` – ustaw kursor na końcu linii (jak w *grep*),
- `%` – znajdź pasujący nawias (`(` , `)` , `[` , `]` , `{` , `}`).
- Zamiast klawiszów strzałek można używać `h` , `j` , `k` , `l` . Te operacje również można poprzedzać liczbą.

Operacje na blokach

- `d` – wytnij (*delete*) bieżący blok tekstu,
- `dd` – wytnij (*delete*) bieżącą linię tekstu,
- `y` – skopiuj (*yank*) bieżący blok tekstu,
- `yy` – skopiuj (*yank*) bieżącą linię tekstu,
- `p` – wklej blok (po kursorze).

- Po komendach `d` oraz `y` musi następować jakiś operator nawigacji, np. `d$` albo `dG` tudzież `d%` etc.
- Komendy `dd` oraz `yy` mogą być poprzedzone liczbą.
- Zwróć uwagę, że schowek VIMa (*rejestr*) jest niezależny od schowka systemowego! Można je jednak przemapować za pomocą funkcji VIMa.

Przykłady operacji na blokach

- `d$` – usuń od pozycji kursora do końca linii,
- `d/tekst` – usuń aż do następnego wystąpienia frazy *tekst*,
- `d%` – usuń tekst pomiędzy odpowiadającymi nawiasami,
- `dG` – usuń wszysktie linie od bieżącej aż do końca pliku,
- `ddp` – zamień kolejność linii,
- `yy10p` – zduplikuj bieżącą linię 10-krotnie.

Tryb wizualny

- Bloki mogą być również zaznaczone za pomocą trybu wizualnego:
  - `v` – rozpoczyna zaznaczanie znaków (uruchamia tryb wizualny),
  - `V` – rozpoczyna zaznaczanie linii (uruchamia tryb wizualny),
- Tekst można zaznaczać za pomocą wszelkich komend nawigacji.
- Tekst można zaznaczać lewym przyciskiem myszy.
- W *gvim* podwójny klik myszy zmienia tryb pomiedzy normalnym a wizualnym.
- Jeśli jakiś blok jest zaznaczony, `d` oraz `y` operuje na tym bloku.
- `~` – zmienia wielkość liter w zaznaczeniu,
- `>` – przesuwa zaznaczony blok w prawo (o wielkość `shiftwidth`),
- `<` – przesuwa zaznaczony blok w lewo (o wielkość `shiftwidth`),

Podmiany tekstu

```
:zakress/tekst1/tekst2/opcje
```

- Powyższa komenda wymienia wystąpienia *teskt1* na *tekst2* w zakresie *zakres*, zgodnie z opcjami *opcje*.
- Zakres może być:
  - pusty – operacja tyczy się tylko bieżącej linii,
  - numerem linii,
  - parę numerów `od,do` albo `od;do` ,
  - `%` – całym plikiem.
- Dopuszczalne opcje to kombinacje poniższych:
  - `i` – ignoruj wielkość liter,
  - `g` – zastąp wszystkie wystąpienia (domyślnie: jedynie najbliższe),
  - `c` – pytaj o potwierdzenie przed każdą zamianą (*confirm*).

Przykład podmiany: zamienia "Jan Kowalski" $\rightarrow$ "Kowalski, Jan" itd.

```
:%s/\(\[A-Z\]\[a-z\]\+\) \(\[A-Z\]\[a-z\]\*ski\)/\2, \1/g
```

Uwagi o rejestrze/schowku

- Zanim wkleisz tekst z innego źródła do pliku edytowanego w VIMie, musisz zmienić tryb w tzw. *insert-paste*:

```
:set paste
i
```

- Domyślne ustawienia VIMa w laboratorium wyłączają możliwość kopiowania do terminala z zaznaczenia w VIMie. Zmiana tego wymaga:

```
:set mouse-=a
```

Inne cechy VIMa

- Praca z wieloma plikami jednocześnie.
- Możliwość określenia dodatkowych komend/makr.
- Konfiguracja domyślnych akcji.
- Możliwość pracy z wieloma schowkami (rejestrami).
- Tryb arkusza kalkulacyjnego.
- VIM jako IDE:
  - `:!./%` – wykonuje bieżący plik jako skrypt,
  - `:!gcc -Wall -pedantic %` – kompiluje bieżący plik kompilatorem C,
  - `:!pdflatex %` – kompiluje źródłowy plik LaTeX.

Plan zajęć

AWK

- `awk` to narzędzie pozwalające na sprawną manipulację plikami tekstowymi, interpretowanymi jako tabele bazy danych.
- `awk` , w domyślnej konfiguracji, interpretuje linie jako rekordy, a kolejne słowa jako pola.
- Liczba pól nie musi być jednolita wśród rekordów.
- Pliki procesowane są rekord po rekordzie.
- `awk` oferuje wsparcie dla wyrażeń regularnych oraz podstawowego syntaksu języka `C` .
- W przypadku braku nazwy pliku w argumencie, operuje na `stdin` – można go używać przy *składaniu strumieni*.

Syntaks AWK – przykłady

Skrypt AWK w linii poleceń

```
$ awk 'tutaj_instrukcje_awka' plik_bazodanowy.txt
```

Skrypt AWK w zewnętrznym pliku

```
$ awk -f skrypt.awk plik_bazodanowy
```

Skrypt AWK jako plik wykonywalny

```
$ head -n1 skrypt.awk
  #!/usr/bin/awk -f
$ chmod +x skrypt.awk
$ ./skrypt.awk plik_bazodanowy.txt
```

Struktura skryptu AWK

```
#!/usr/bin/awk -f

BEGIN {instrukcje_B}
warunek_1 {instrukcje_1}
warunek_2 {instrukcje_2}
...
warunek_n {instrukcje_n}
END {instrukcje_E}
```

- sekcje `BEGIN` oraz `END` są nieobowiązkowe,
- niewypowiedziane warunki są równoważne prawdzie,
- niewypowiedziane instrukcje są równoważne domyślej instrukcji: *wypisz rekord na* `stdout` .

Logika procesowania AWKa

```
if (present(BEGIN)) {instrukcje_B}
while (read(linia))
    for (i=1; i<=n; i++)
        if (warunek_i(linia))
            instrukcje_i(linia)

if (present(END)) {instrukcje_E}
```

Symbole specjalne w AWK

- Dla uproszczenia, będziemy mówić o *słowie* i *wierszu*, zamiast bardziej ogólnych *polu* i *rekordzie*.
- `$0` – bieżący wiersz,
- `NF` – liczba słów w bieżącym wierszu,
- `$1` , `$2` ,..., `$(NF-1)` , `$NF` – pierwsze, drugie, ..., przedostatnie, ostatnie słowo w bieżącym wierszu.
- `NR` – liczba wierszy dotychczas przeprocesowanych (włącznie z bieżącym),
- `FS` , `OFS` – separator pola na wejściu/wyjściu (domyślnie: dowolnie długie sekwencje znaków białych),
- `RS` , `ORS` – separator rekordu na wejściu/wyjściu (domyślnie: znak końca linii),
- `ARGC` , `ARGV` – jak w języku `C`.

Instrukcje i operatory

- Instrukcje AWK zą zaczerpnięte z języka `C` i mają jednolity syntaks. Jedyna instrukcja języka `C`, która nie jest odwzorowana w AWK, to instrukcja switch/case.
- W AWK zaimplementowano wszystkie operatory logiczne, arytmetyczne i porównania (w tym operator trójkowy `?:`),
- Większość podstawowych funkcji arytmetycznych języka `C` jest dostępnych i działa w jednakowy sposób w AWK,
- W AWK zaimplentowano funkcję `printf()`, która używa syntaksu języka `C` oraz jego modyfikatorów.
- AWK zapewnia nieco zmieniony syntaks prostych funkcji do procesownaia tekstu.

Podstawowy/uproszczony syntaks C

- Jeśli nie prowadzi to do niejednoznaczności, można zignorować nawiasy.
- Średniki są opcjonalne, niezbędne wyłącznie w przypadku wielu instrukcji w jednej linii.
- Zmiennych nie trzeba deklarować/definiować przed użyciem.
- Konwersja typu jest przeprowadzana automatycznie, w zależności od kontekstu.
- Nie ma wskaźników.
- Pusty operator jest operatorem konkatenacji sekwencji znakowych.
- Zmienne tekstowe porównujemy operatorami arytmetycznymi (`==` , `!=` , `<` `<=` , `>` , `>=`).
- Jeśli nie ma potrzeby specjalnego formatowania, to zamiast `printf` można użyć uproszczonej metody `print` .
- Istnieją tablice łączne, można ich używać jak słowników o kluczach będących liczbami naturalnymi.
- Komentarze w stylu Powłoki (`#`).

Operator dopasowania wyrażenia regularnego

Zmienna zawierająca podciąg opisany przez wyrażenie regularne

```
zmienna ~ /wyrazenie_regularne/
```

Równoważne warunki (dzięki automatycznej konwersji typów)

```
zmienna >= 1000 && zmienna < 2000
zmienna ~ /^1[0-9]{3}$/
```

Dziwna sekwencja prawidłowych operacji

```
i=32; j=1 # liczby, nie trzeba deklarować typu zmiennej
k=i j # 321 (automatyczna konwersja do typu tekstowego)
k*=2 # 642 (automatyczna konwersja do typu numerycznego)
k ~ /[246]{3}/
```

Przykład: skrypty AWK a komendy Powłoki

cat plik

```
$ awk '1' plik
```

head plik

```
$ awk 'NR <= 10' plik
```

wc -l plik

```
$ awk 'END {print NR}' plik
```

wc -w plik

```
$ awk 'BEGIN {ctr=0} {ctr+=NF} END {print ctr}' plik
```

grep -E 'wzorzec' plik

```
$ awk '$0~/wzorzec/' plik
```

Przykład: tablice łączne

Sortowanie bąbelkowe

```
#!/usr/bin/awk -f

{tab[NR-1]=$0}

END {
for (i=0;i<NR;i++) # bubble sort
  for (j=0;j<NR;j++)
    if (tab[j+1]<tab[j])
      {temp=tab[j]; tab[j]=tab[j+1]; tab[j+1]=temp}
for (i=0;i<NR;i++) print tab[i]}
```

Zarządzanie formatem pliku

- Separatory pól/rekordów mogą być skonfigurowane w sekcji `BEGIN`.
- Separatory pól mogą być skonfigurowane w argumencie wywołania `awk` za pomocą flagi `-F` .
- Rekordy mogą być modyfikowane w locie.

Modyfikacja rekordów w locie

```
#!/usr/bin/awk -f

BEGIN {FS=":"; OFS=":"}
$3~/^[13579]+$/ {$4+=1000}  # modyfikacja rekordu w locie
```

Zamiana /etc/passwd na CSV

```
$ awk -F':' 'BEGIN {OFS=","} {$1=$1; print $0}' /etc/passwd
```

Czytanie parametrów skryptu

./skrypt.awk plik.txt raz dwa trzy

```
#!/usr/bin/awk -f

# blok BEGIN parsuje parametry pozycyjne skryptu
# i usuwa je z wiersza poleceń
BEGIN {
for (i=2; i<ARGC; i++) {
  params[i-2] = ARGV[i] }
par_no = ARGC - 2
ARGC = 2 }
```

Plan zajęć

JQ

- `jq` jest narzędziem do procesowania danych w formacie JSON.

Syntaks jq

```
$ jq 'komenda' plik.json
```

- `jq` operuje na strumieniach, więc można go używać jako filtr.

jq jako filtr

```
$ cat plik.json | jq 'komenda'
```

- `jq` obsługuje rozbudowany syntaks *komend*

Komendy jq – podstawy

- `.` – ewaluacja całego obiektu,
- `.klucz` – ewaluacja podobiektu zapisanego pod kluczem *klucz*,
- `.[0]` – ewaluacja pierwszego elementu listy,
- `.[]` – zwraca iterator dla listy,
- `.[2:5]` – zwraca iterator dla podlisty,
- `,` – dla wykonania kilku instrukcji niezależnie,
- `|` – dla złożenia kilku instrukcji,
- `length` – wylicza długość obiektu,
- `keys` – zwraca klucze.

jq – przykłady

Operator .

```
$ echo '{"a": true, "b": 0}' | jq '.'
  {"a": true, "b": 0}
$ echo '{"a": true, "b": 0}' | jq '.a'
  true
$ echo '{"a": true, "b": 0}' | jq '.c'
  null
```

Operator [ ]

```
$ echo "[1, 2, 3]" | jq '.[1]'
  2
$ echo "[1, 2, 3]" | jq '.[-1]'
  3
$ echo "[1, 2, 3]" | jq '.[1:3]'
  [2,3]
```

jq – przykłady 2

Operator .[ ]

```
$ echo "[1, 2]" | jq '.[]'
  1
  2
$ echo "[]" | jq '.[]' # pusty
$ echo '{"a": true, "b": 0}' | jq '.[]'
  [true, 0]
```

Operatory |, .[]

```
$ echo '{"a": [1, 2], "b": [1, 2, 3]}' | jq '.[]'
  [1, 2]
  [1, 2, 3]
$ echo '{"a": [1, 2], "b": [2, 3]}' | jq '.a | .[]'
  1
  2
```

jq – przykłady 3

Operatory length, keys

```
$ echo '{"a": [1, 2], "b": [1, 2, 3]}' | jq 'keys'
  ["a", "b"]
$ echo '{"a": [1, 2], "b": [1, 2, 3]}' | jq '.[] | length'
  2
  3
```

Operatory agregacji

```
$ echo '[{"imie": "Jan", "nazwisko": "Kowalski", "wiek":
24}, {"imie": "Antoni", "nazwisko": "Nowak", "wiek": 41}]'
| jq '.[] | {age: .wiek, surname: .nazwisko}'
  {"age": 24, "surname": "Kowalski"}
  {"age": 41, "surname": "Nowak"}
```

Plan zajęć

YQ

- `yq` jest narzędziem do procesowania danych w formacie YAML.

Syntaks yq

```
$ yq [tryb] 'komenda' plik.yaml
```

- `yq` operuje na strumieniach, więc można go używać jako filtr.

yq jako filtr

```
$ cat plik.json | yq 'komenda'
```

- `yq` obsługuje rozbudowany syntaks *komend*, operuje również w kilku trybach: (domyślnym) `eval` , `eval-all` itd, inspirowany w dużej mierze `jq` .
- Ponieważ JSON jest podzbiorem standardu YAML, `yq` jest w stanie również procesować JSONy.

yq – przykłady

Operatory ., [ ]

```
$ cat plik.yaml
  a:
    b:
    - c: json
    - c: yaml
$ yq '.a.b[0].c' plik.yaml
  json
$ cat plik.yaml | yq '.a.b[1].c'
  yaml
```

Modyfikacja struktury pliku YAML

```
$ yq -i '.a.b[0].c = "YAML"' plik.yaml
$ cat plik.yaml
  a:
    b:
    - c: YAML
    - c: yaml
```

yq – przykłady 2

Aliasy/kotwice

```
$ echo '- &kowalski
  nazwisko: Kowalski
  imie: Piotr
  ocena: 4.0
- nazwisko: Nowak
  imie: Antoni
  ocena: 4.5
- <<: *kowalski #pobiera klucze/wartości z obiektu "&kowalski"
  imie: Jan #nadpisuje wartość pola "imie"' | yq 'explode(.)' -
  - nazwisko: Kowalski
    imie: Piotr
    ocena: 4.0
  - nazwisko: Nowak
    imie: Antoni
    ocena: 4.5
  - nazwisko: Kowalski
    ocena: 4.0
    imie: Jan
```

Zaawansowane instrukcje yq

Wybór elementu listy

```
$ echo '- &kowalski
  nazwisko: Kowalski
  imie: Piotr
  ocena: 4.0
- nazwisko: Nowak
  imie: Antoni
  ocena: 4.5
- <<: *kowalski
  imie: Jan
- nazwisko: Kowal
  imie: Adam
  ocena: 3.5' >> plik.yaml
$ yq '.[] | select(.ocena == "4.5") | explode(.)' plik.yaml
$ yq 'explode(.)[] | select (.nazwisko == "Nowak")' plik.yaml
$ yq 'explode(.)[] | select (.nazwisko == "Kowal*")' plik.yaml
$ yq 'explode(.)[] | select (.ocena == "4.*")' plik.yaml
```

Plan zajęć

SED – stream editor

- Program dedykowany do edycji treści tekstowych.
- Pracuje na strumieniach, może być używany jako filtr.
- Ideologicznie zbliżony do `awk` .
- Część projektu GNU.
- Prosty syntaks, pozwalający na kwerendy pliku tekstowego.
- Obszerna dokumentacja:
  `https://www.gnu.org/software/sed/manual/sed.html`
- Domyślnie: wypisuje postać zmodyfikowaną na *stdout*. Dla edycji pliki w locie, używamy opcji `-i`

Syntaks

```
$ sed 'skrypt_tutaj' plik.txt
$ sed -f skrypt.sed plik.txt
```

Przykłady

Wyświetlanie

```
$ sed -n '8,12p' plik.txt # Wypisz tylko linie 8-12
$ sed '8,12d' plik.txt # Wypisz wszystko poza liniami 8-12
```

Zastępowanie wystąpień

```
$ sed 's/foo/bar/' plik.txt # Zastąp wszystkie wystąpienia foo na bar
$ sed -i 's/foo/bar/' plik.txt # Zamień w pliku wszystkie wystąpienia foo na bar
$ sed 's/([A-Z][a-z]+) ([A-Z][a-z]*ski)/\2, \1/' plik.txt # Zamiana regexowa na wyższym poziomie
$ sed 'y/ąćęłńóśżź/acelnoszz/' plik.txt # Usuwanie polskich diakrytyków
```