

ნაწილი I

შესავალი GNU/Linux სისტემებში არჩილ ელიზბარაშვილი

სარჩევი

I შესავალი GNU/Linux სისტემებში არჩილ ელიზბარაშვილი	1
1. კომპიუტერის შექმნიდან დღემდე	7
1.1 Unix-ის ეპოქის დასაწყისი	8
1.2 Unix-ის მსგავსი სისტემები	11
1.3 პროგრამები და ლიცენზიები	13
1.4 თავისუფალი პროგრამების განვითარება	17
1.5 ლინუქსის ისტორია	18
2. ბრძანებათა ზაზი	23
2.1 shell	23
2.2 ლინუქსის საბაზისო ბრძანებები	25
3. სისტემის სტრუქტურა	31
4. მოქმედებები ფაილებზე	37
4.1 მაგენერირებელი სიმბოლოები	37
4.2 ფაილების შექმნა	39
4.3 ფაილების ასლების შექმნა, გადატანა, წაშლა	44
5. გადამისამართება	47
5.1 სტანდარტული გამოსასვლელი	48
5.2 შეცდომების გამოსასვლელი	49
5.3 სტანდარტული შესასვლელი	50
6. ბრძანებების გადაბმა	53
6.1 ოპერატორები	53
6.2 მილი	55
7. ტექსტური რედაქტორები	59
7.1 რედაქტორი – vi/vim	60
7.2 რედაქტორი – GNU Emacs	65
8. წვდომის უფლებები	73
8.1 მფლობელი და ჯგუფი	73
8.2 ძირითადი ატრიბუტები	75
8.3 სპეციალური ატრიბუტები	82
9. პროცესები	85
9.1 ფონური რეჟიმი	92

9.2	სიგნალები	95
9.3	პროცესის პრიორიტეტები	97
10.	გარემო, ცვლადები	99
10.1	ცვლადის დამუშავება	108
10.2	მარტივი არითმეტიკული გამოთვლები	116
11.	ტექსტის დამუშავება	119
11.1	tac	119
11.2	head, tail	120
11.3	sort	122
11.4	uniq	124
11.5	cut	126
11.6	paste	128
11.7	join	130
11.8	tr	132
12.	სხვა სასარგებლო ბრძანებები	137
12.1	ტექსტის შედარება	137
12.2	find	139
12.3	split	148
12.4	echo	149
12.5	xargs	153
12.6	expand, unexpand	157
12.7	fmt, nl	160
12.8	od	161
12.9	date, cal	164
13.	რეგულარული გამოსახულება	167
13.1	grep	167
13.2	sed	178
14.	არქივი, შეკუმშვა	197
14.1	gzip	198
14.2	bzip2	200
14.3	tar	202
14.4	cpio	206
15.	ამოცანების გაშვების ავტომატიზაცია	209
15.1	cron	209
15.2	at	214
II	Shell-ის სკრიფტები	217
16.	შელის სკრიფტები	219
16.1	ინტერპრეტატორები	219
16.2	პირველი shell სკრიფტი	220
16.3	გამოსახულების შეფასება	221
16.4	გადაწყვეტილების მიღება	229
16.5	ციკლი	237

16.6	არითმეტიკული გამოთვლები	252
16.7	სპეციალური ცვლადები	268
16.8	ფუნქციები	274
16.9	სკრიფტის ოფციები	282
17.	ტერმინალები	287
17.1	ტერმინალის საკონტროლო კოდები	287
17.2	ANSI კოდები	300
17.3	კურსორის მართვა	310
17.4	ტერმინალის მულტიპლექსერები	317
18.	მასივები	329
18.1	ასოციაციური მასივი	333
სათაური		335

თავი 1

კომპიუტერის შექმნიდან დღემდე

როდის შეიქმნა პირველი კომპიუტერი? ერთ-ერთი ფართოდ გავრცელებული მოსაზრებით, პირველი კომპიუტერი ჩარლს ბებიჯმა (Charles Babbage) 1822 წელს შექმნა. თუმცა, მას დღევანდელ კომპიუტერთან საერთო ბევრი არაფერი აქვს. ეს იყო პირველი მექანიკური ავტომატური გამომთვლელი მანქანა, რომელიც ითვლიდა რიცხვებს. საბოლოოდ, უსახსრობის გამო, ბებიჯმა ვერ შეძლო სრულად ფუნქციონირებადი მანქანის შექმნა.

თავად სიტყვა კომპიუტერი (computer) პირველად 1613 წელს გამოჩნდა და მას უწოდებდნენ ადამიანს, რომელიც გამოთვლას, დათვლას აწარმოებდა. მე-19 საუკუნის ბოლოსკენ, ინდუსტრიული რევოლუციის შემდეგ, ამ ტერმინით უკვე მოწყობილობებს მოიხსენიებდნენ.

1936 წელს ალან ტიურინგმა (Alan Turing) წარმოადგინა ახალი მანქანის იდეა. მას ტიურინგის მანქანა უწოდეს. ის გამომთვლელი მოწყობილობის აბსტრაქტულ მოდელს წარმოადგენდა და დღესაც, თანამედროვე კომპიუტერების მუშაობის პრინციპი ტიურინგის მანქანის კონცეფციასა და დამყარებული.

იმავე წელს, გერმანელმა ინჟინერმა კორნად ცუზემ (Kornad Zuse) დაიწყო პირველი დაპროგრამებადი კომპიუტერის შექმნა და 1941 წელს მან, სრულიად დამოუკიდებლად, ამ კომპიუტერების ხაზის მესამე ვერსია – Z3 დაასრულა. ეს იყო ელექტრული კომპიუტერი, რომელშიც გამოიყენებოდა რელეები.

1937 წელს პროფესორმა ჯონ ვინსენტ ათანასოფმა (John Vincent Atanasoff) და მისმა სტუდენტმა კლიფ ბერიმ (Cliff Berry) დაიწყეს ახალი კომპიუტერის, ABC-ის (Atanasoff-Berry Computer) შექმნა. მას პირველ ელექტრონულ და ციფრულ კომპიუტერად მოიაზრებენ. ელექტრონულად – რადგან მასში გამოიყენებოდა ვაკუუმური მილაკები. ABC კომპიუტერისგან განსხვავებით, წინა თაობის ელექტრულ კომპიუტერებში გამოიყენებოდა ელექტრული ძრავები ან რელეები (ელექტრომექანიკური გადამრთველები). ციფრულად – იმიტომ, რომ მასში შემოტანილი იყო ორობითი სისტემის არითმეტიკისა და ლოგიკური წრედების კონცეფციები.

პირველ ელექტრონულ, დაპროგრამებად კომპიუტერად ზოგიერთი სპეციალისტი კოლოსს (Colossus) მიიჩნევს. ის ტომი ფლაუერსმა (Thomas (Tommy) Harold Flowers) შექმნა 1943 წელს იმ მიზნით, რომ მეორე მსოფლიო ომის მსვლელობისას გერმანელების მიერ გადაცემული დაშიფრული შეტყობინებები, ე.წ. ლორენცის შიფრები¹ გაეტყუათ. ალან ტიურინგის დახმარებით კი, ასევე ბრიტანელმა კრიპტოლოგებმა შექმნეს

¹ შეტყობინებების დაშიფრვა ხდებოდა ლორენცის მანქანით. ბრიტანელმა კრიპტოანალიტიკოსებმა მოახერხეს ამ დაშიფრული ინფორმაციის მოპოვება, მისი ლოგიკური სტრუქტურის შესწავლა და შეტყობინებების წაკითხვა. ამ მოვლენიდან მხოლოდ 3 წლის შემდეგ შეძლეს ფიზიკურად ენახათ ლორენცის მანქანა.

ელექტრომექანიკური მანქანა, სახელად „ბომბი“ (bombe), რომლითაც ენიგმას² (Enigma) კოდები გაშიფრეს.

ამ პერიოდის ცნობილი კომპიუტერი იყო ENIAC (Electronic Numerical Integrator and Calculator). ისიც, როგორც კოლოსი და ბომბი, სამხედრო მიზნებისთვის, კონკრეტულად – ბალისტიკური რაკეტის ტრაექტორიის გამოსათვლელად, შეიქმნა პენსილვანიის უნივერსიტეტში პრესპერ ეკერტისა (Presper Eckert) და ჯონ მოჩლის (John Mauchly) მიერ 1943-1946 წლებში. ENIAC ზომაში ძალიან დიდ მანქანას წარმოადგენდა. ის დიდ ფართზე იყო განთავსებული, შეიცავდა 18 000 ვაკუუმურ მილაკს, მოიხმარდა 180 000 ვატს და იწონიდა 50 ტონას. ამასთან ერთად, ის იყო სრულად ფუნქციონირებადი და 1000-ჯერ სწრაფი კომპიუტერი, ვიდრე მისი წინამორბედი ელექტრომექანიკური მანქანები, რის გამოც ბევრი სპეციალისტი, სწორედ ENIAC-ს მიიჩნევს პირველ ციფრულ კომპიუტერად³.

მას შემდეგ კომპიუტერის უამრავი ვარიანტი შეიქმნა, რომელთაც თან ახლდა ახალი ტექნოლოგიური სიახლეები განვითარების სხვადასხვა ეტაპზე. შესაბამისად, კომპიუტერები შეიძლება რამდენიმე თაობად დავყოთ: პირველი თაობის კომპიუტერები, რომლებიც მუშაობდნენ ელექტრონულ-ვაკუუმურ მილაკებზე (თუ არ ჩავთვლით მექანიკურ და ელექტრომექანიკურ მანქანებს). სწორედ ამ თაობის კომპიუტერებია ABC, კოლოსი და ENIAC. მეორე თაობის კომპიუტერებში (1947 წლიდან) ვაკუუმური მილაკები ტრანზისტორებმა ჩაანაცვლეს. ამ თაობის წარმომადგენელია IBM⁴-ის 701. ეს მოდელი იყო IBM-ის პირველი კომერციული სამეცნიერო კომპიუტერი. მესამე თაობის კომპიუტერებში, რომლებიც 1963 წლიდან დღემდე არსებობს, გამოიყენება ინტეგრირებული მიკროსქემები (ე.წ. მიკროჩიპი ან უბრალოდ ჩიპი). ჩიპებმა შეიძლება დაიტიონ რამდენიმე მილიარდი ტრანზისტორი. სწორედ, მათი წყალობით გახდა კომპიუტერები ზომაში გაცილებით პატარა და მძლავრი. შემდეგი თაობის კომპიუტერი, სავარაუდოდ, კვანტური კომპიუტერი იქნება. ეს ტექნოლოგია ჯერ კიდევ განვითარების სტადიაშია და მას უახლოეს მომავალში ფართო მასშტაბით არ უნდა ველოდოთ.

1.1 Unix-ის ეპოქის დასაწყისი

გამოთვლითი მოწყობილობების განვითარებასთან ერთად პარალელურად ვითარდებოდა პროგრამული უზრუნველყოფაც. ყოველი შექმნილი კომპიუტერისთვის იქმნებოდა შესაბამისი სისტემა მასზე სამუშაოდ. იმდროინდელი კომპიუტერები კონკრეტული ამოცანების დასამუშავებლად იგებოდა და ამ კომპიუტერებზე მხოლოდ მათზე მორგებული პროგრამების გაშვება შეიძლებოდა. პროგრამა, რომელიც ერთი მწარმოებლის მიერ ერთი კომპიუტერისათვის იყო დაწერილი, სხვა მწარმოებლის მიერ შექმნილი კომპიუტერისთვის არათავსებადი იყო. ამავდროულად, ამ კომპიუტერებზე შეუძლებელი იყო ორი ამოცანის ერთდროულად გაშვება. დღევანდელი გადასახედიდან რომ ვიმსჯელოთ, შეუძლებელი იყო ტექსტის აკრეფა და მუსიკის მოსმენა ერთდროულად. აქედან გამომდინარე, 1960-იან წლების ბოლოს, სისტემის შექმნისას აქცენტი გადაიტანეს time-sharing კონცეფციაზე. ასეთ სისტემაში გაზიარებულია კომპიუტერის რესურსები მრავალ მომხმარებელსა და ერთდროულად გაშვებულ მრავალ ამოცანაზე. time-sharing ოპერაციული სისტემის ერთ-ერთი ადრეული ვარიანტის პროექტი იყო MULTICS (Multiplexed Information and

² ენიგმას მანქანები – ელექტრო-მექანიკური დაშიფრვის მანქანები პირველი მსოფლიო ომის შემდეგ შეიქმნა გერმანიაში და ისინი მე-20 საუკუნის დასაწყისში გამოიყენებოდა სამხედრო, დიპლომატიური და კომერციული ინფორმაციული კომუნიკაციების დასაცავად. მეორე მსოფლიო ომის დროს გერმანელი სამხედროები ამ მანქანით შიფრავდნენ შეტყობინებებს და ისე უზგავნიდნენ ერთმანეთს. არსებობდა აგრეთვე ენიგმას იაპონური და იტალიური მოდელებიც.

³ პირველი ელექტრონული კომპიუტერის პატენტი პრესპერ ეკერტსა და ჯონ მოჩლის გადაეცათ, თუმცა, მოგვიანებით, ა.მ.შ.-ს სასამართლოს განჩინებით, მათ ჩამოერთვათ და პატენტი გადაეცა ჯონ ათანასოვს, ABC-ის შემქმნელს.

⁴ IBM (International Business Machines) – კომპიუტერული ტექნიკისა და პროგრამული უზრუნველყოფის ერთ-ერთი უმსხვილესი მწარმოებელი კორპორაცია.

Computing Service). ის დაიწყო 1964 წელს კემბრიჯში, მასაჩუსეტსი და MIT⁵-ის თაოსნობით ზორციელდებოდა General Electric⁶-სა და AT&T Bell Labs⁷-თან თანამშრომლობით. 1969 წელს Bell Labs გამოვეო პროექტს.

1969-1970 წლებში, კენეტ (კენ) ტომპსონმა (Kenneth Thompson) და დენის რიჩი (Dennis Ritchie), რომლებიც MULTICS-ის პროექტის თანამონაწილე იყვნენ, AT&T Bell Labs-ში გადაწყვიტეს მსგავსი ოპერაციული სისტემის დაწერა პატარა მასშტაბით PDP-7-ზე⁸ ასემბლერის⁹ გამოყენებით. სულ მალე ოპერაციულ სისტემას დაარქვეს სახელი Unics (UNiplexed Information and Computing Service), როგორც MULTICS-ის კალამბური, მოგვიანებით კი Unix. ასემბლერზე დაწერილი პროგრამა ნიშნავდა იმას, რომ ეს პროგრამა სხვა არქიტექტურის კომპიუტერზე ვერ გაეშვებოდა. ამასობაში, ტომპსონი და რიჩი ქმნიან დაპროგრამების ენას C, რომლის მიზანაც არის პორტატული პროგრამის დაწერა. 1972-1973 წლებში სისტემა გადაიწერა C ენაზე და დაემატა ბევრი ახალი ფუნქციონალი. 1971-1979 წლებში პროდუქტი გამოდიოდა Unix Time-Sharing System V1, V2, V3, V4, V5, V6 და V7 დასახელებებით.

1975 წელს კენ ტომპსონი Bell Labs-დან გადავიდა ბერკლიში (Berkeley), კალიფორნიის უნივერსიტეტში,¹⁰ როგორც მოწვეული პროფესორი. ის დაეხმარა Unix V6-ის დაყენებაში და ამ სისტემისთვის დაიწყო დაპროგრამების ენა – პასკალის (Pascal) შექმნა. მოგვიანებით ჩაკ ჰალიმ (Chuck Haley) და ბილ ჯოიმ (Bill Joy) გააუმჯობესეს ტომპსონის პასკალი და ამავდროულად, შექმნეს გაუმჯობესებული ტექსტური რედაქტორი სახელად ex. 1978 წელს კალიფორნიის უნივერსიტეტის კომპიუტერული სისტემების კვლევის ჯგუფმა გამოუშვა პირველი პროდუქტი Berkeley Software Distribution (1BSD). 1BSD დამენებული იყო Unix V6-ის საწყის კოდზე. მისი მეორე ვერსია – 2BSD 1979 წელს გამოვიდა. პირველი ვერსიის განახლებების გარდა, იგი შეიცავდა ბილ ჯოის ორ ახალ პროგრამას: vi ტექსტურ რედაქტორს, რომელიც ex-ის ვიზუალურ ვერსიას წარმოადგენდა და C shell-ს.

1977-1979 წლებში, კენ ტომპსონმა და დენის რიჩიმ ხელახლა გადაწერეს Unix, რათა მეტად პორტატული გაეხადათ და AT&T Bell Labs 80-იანი წლების დასაწყისში გამოუშვეს Unix-ის განახლებული ვარიანტი „System III“-ს. ამასობაში Bell Labs გამოვეო AT&T კომპანიას. ამ გამოყოფას უფრო სტრატეგიული დანიშნულება ჰქონდა. საქმე ის იყო, რომ AT&T წარმოადგენდა (და წარმოადგენს) უმსხვილეს სატელეკომუნიკაციო კომპანიას, რომელიც ფლობდა Bell Labs კვლევით ცენტრს. იმ დროისთვის AT&T მონოპოლისტად ითვლებოდა თავის სფეროში და მთავრობა მას მკაცრად აკონტროლებდა. Bell Labs კი მხოლოდ კვლევით ცენტრს წარმოადგენდა, რომელსაც დიდი ფინანსური რესურსი ჰქონდა (AT&T-ს წყალობით). გარკვეული იურიდიული შეზღუდვების გამო AT&T-ს აეკრძალა კომპიუტერულ ბაზარზე გამოსვლა და მას მხოლოდ Unics-ის კოდების დალიცენზირების საშუალება დარჩა. Bell Labs კი იურიდიული შეზღუდვებისგან თავისუფალი იყო. მოგვიანებით AT&T უშვებს Unix-ის კომერციულ ვერსიას „System V“-ის დასახელებით.

⁵MIT – Massachusetts Institute of Technology, მასაჩუსეტსის ტექნოლოგიური ინსტიტუტი არის ძალიან ცნობილი კერძო სასწავლო, კვლევითი უნივერსიტეტი კემბრიჯში, მასაჩუსეტსი, აშშ.

⁶General Electric (GE) – ჯენერალ ელექტრიკი არის ამერიკული კორპორაცია (კემბრიჯი, მასაჩუსეტსი), რომელიც აწარმოებს ტექნიკის ფართო სპექტრს.

⁷AT&T Bell Labs ერთ-ერთი ძველი დასახელებაა დღევანდელი Nokia Bell Labs-ის. მას სხვადასხვა პერიოდებში აგრეთვე ერქვა Bell Telephone Laboratories და Bell Labs. AT&T Bell Labs არის კვლევითი და სამეცნიერო განვითარების ამერიკული კომპანია. მოგვიანებით დაიყო ორად (Bell Labs და AT&T Laboratories). ის მდებარეობს აშშ-ში და ეკუთვნის ფინურ კომპანია Nokia-ს. მისი ლაბორატორიები მსოფლიოს სხვადასხვა ქვეყანაშია განაწილებული. ისტორიული ლაბორატორია (Volta Laboratory and Bureau) ჯერ კიდევ მე-19 საუკუნეში შექმნა ალექსანდრე ბელმა (Alexander Graham Bell). Bell Labs იმითაც არის ცნობილი, რომ იქ შეიქმნა ტრანზისტორი, ლაზერი, განაწილებული ოპერაციული სისტემები – Plan 9 და Inferno, დაპროგრამების ენები C, C++, S და ა.შ.

⁸PDP-7 არის Digital Equipment Corporation-ის მიერ შექმნილი PDP (Programmed Data Processor) სერიის ერთ-ერთი მინიკომპიუტერი. იმ დროისთვის ის ითვლებოდა იაფად, თუმცა იყო მძლავრი. მისი ფასი 1965 წელს იყო 72000\$, რაც დღეს დაახლოებით 500000\$-ის ექვივალენტია.

⁹ასემბლერი (assembly language ან assembler language, შემოკლებით – asm) არის კომპიუტერის დაბალი დონის დაპროგრამების ენა.

¹⁰University of California (UC) არის ძალიან ცნობილი საჯარო უნივერსიტეტი კალიფორნიაში, აშშ.

საწყის ეტაპზე Unix-ის სისტემები AT&T Bell Labs-დან შეიცავდა ოპერაციული სისტემის საწყის კოდს, რაც მკვლევარებს საშუალებას აძლევდა შეეცვალათ, გაეფართოვებინათ და გაეუმჯობესებინათ Unix. ეს კოდები თავისუფლად ვრცელდებოდა უნივერსიტეტებსა თუ კვლევით ცენტრებში. შედეგად, Unix-ის ბევრი სხვადასხვა ვარიანტი განვითარდა. უმეტესი მათგანი გახდა მესაკუთრეობრივი და ისინი მხარდაჭერილები იყვნენ შესაბამისი მწარმოებლისგან (ვენდორებისგან). ამ ყველაფერმა კი გამოიწვია ის, რომ ცალკეულმა მწარმოებლებმა პროდუქციის განვითარებისას საკუთარი ზაზი შექმნა, საკუთარი პროგრამებით, რომლებიც სრულიად არათავსებადი იყო სხვა მწარმოებლის მიერ შექმნილ სისტემებთან.

1984 წელს დაარსდა X/Open ჯგუფი. მისი წევრები იყვნენ ინფორმატიკაში მომუშავე კომპანიები. X/Open ჯგუფის მიზანი იყო Unix-ის სხვადასხვა ვერსიის ნორმალიზება, რათა გაეზარდათ მათ შორის თავსებადობა და შეემცირებინათ პორტირების ზარჯები. ამ ჯგუფის გარდა, აგრეთვე ამავე მიზნებისთვის ჩამოყალიბდა ჯგუფი POSIX¹¹, რომელიც IEEE¹² ასოციაციის ნაწილია. ეს ორი ჯგუფი გახდა სტანდარტიზაციის ნამდვილი ორგანიზმი, თუმცა POSIX-ის მიერ გაწეული სამუშაოები უფრო გადამწყვეტი აღმოჩნდა. თუ მწარმოებლები ამ სტანდარტს გაითვალისწინებდნენ, ოპერაციულ სისტემებს შორის თავსებადობა გარანტირებული იქნებოდა.

ამავე წელს MIT-ში იქმნება X Window – მრავალფანჯრიანი გრაფიკული სისტემა. ეს სისტემა Unix-სგან დამოუკიდებელი იყო, თუმცა მის წარმატებაში დიდი წვლილი შეიტანა.

1987 წელს AT&T (Unix-ის მფლობელი) და Sun (კომპიუტერების მწარმოებელი ერთ-ერთი ლიდერი კომპანია, რომელიც იყენებდა BSD-ს) ქმნიან ალიანსს, რათა მოახდინონ ორი სისტემის შერწყმა. 1988 წელს იქმნება ორი კონსორციუმი:

- OSF (Open Software Foundation) (წევრები: DEC, HP, IBM, ...), სადაც მუშაობენ ახალი Unix-ის ნორმალიზებაზე და რომელიც მონათლეს OSF1 სახელით.
- Unix International (წევრები: AT&T, Sun, ...), რომლის მიზანია Unix system V-ის პოზიციების გამყარება.

1992 წელს გამოდის OSF-ის პირველი, კომერციულად ხელმისაწვდომი სისტემის ვერსია DEC/OSF1 დასახელებით, ხოლო Sun უშვებს System V-სა და BSD-ს პირველ შერწყმულ კომერციულ ვერსიას. ზოგადად, 1980-იანი წლების ბოლოსა და 1990 წლების დასაწყისში AT&T-სა და Berkeley-ს შორის დიდი „ომი“ იყო გაჩაღებული ბაზარზე პირველობის მოსაპოვებლად. კომერციული თვალსაზრისით შეიძლება ჩავთვალოთ, რომ „ომში“ „System V“-მ იმარჯვა, რადგან ტექნიკის მწარმოებლების (hardware vendors) უმრავლესობა AT&T-ს „System V“-ზე გადაერთო. თუმცა, BSD სისტემაც ფართოდ გამოიყენებოდა მკვლევარებს შორის პერსონალურ კომპიუტერებსა და პირადი გამოყენების სერვერებზე. ერთი კი აღსანიშნავია – მრავალი წლის განმავლობაში System V და BSD ხშირად ითვისებდნენ ერთმანეთისგან მნიშვნელოვან ინოვაციებს და ნერგავდნენ საკუთარ პროდუქტებში. AT&T-მ 1990-იანი წლების დასაწყისში კალიფორნიის უნივერსიტეტს სასამართლოში უჩივლა იმ მიზეზით, რომ BSD შეიცავდა AT&T-ს კუთვნილი Unix-ის საწყის კოდებს. ამან საგრძნობლად შეაფერხა BSD სისტემის განვითარება. 1995 წელს მისი ბოლო ვერსია – 4.BSD-Lite Release 2 გამოვიდა ბერკლიში. შემდეგ კი კომპიუტერული სისტემების კვლევის ჯგუფი დაიშალა და BSD-მაც ბერკლიში შეწყვიტა განვითარება. BSD-ს ლიცენზიის დამთმობმა მხარემ საშუალება მისცა ბევრი სხვა კომპანიის საკუთრებაში მყოფი ოპერაციულ სისტემაში, თავისუფალსა თუ მესაკუთრეობრივში, ჩაენერგა BSD-ს კოდები. მაგალითად, მაიკროსოფტ ვინდოუსმა (Microsoft Windows) გამოიყენა BSD-ს კოდები TCP/IP-ის დანერგვის დროს. ასევე, BSD-ს უამრავი კოდი აქვს ჩაშენებული Apple-ის MacOS-სა და Solaris-ს.

¹¹POSIX – Portable Open System Interface eXchange

¹²IEEE – Institute of Electrical and Electronics Engineers

1.2 Unix-ის მსგავსი სისტემები

ბევრი ოპერაციული სისტემა Unix-დან არის წარმოებული. ასეთ სისტემებს Unix-ის მსგავს სისტემებს (Unix-like) უწოდებენ. ოპერაციული სისტემის ეს კატეგორია მოიცავს ისეთ სისტემებს, რომლებსაც ორიგინალი Unix-ის მსგავსი სტრუქტურა აქვთ. მასში ასევე მოიაზრება UNIX-ის კლონები¹³. ცნობილია Unix-ის მსგავსი შემდეგი სისტემები:

Solaris	1983 წელს კომპანია Sun Microsystems-მა გამოუშვა BSD-ს ვარიანტი SunOS, რომელიც გათვალისწინებული იყო SPARC პლატფორმის სერვერული ტიპის კომპიუტერებისთვის. 1992 წელს კომპანიაში დაიწეს ახალი, უფრო თანამედროვე Unix-ის მსგავსი ვერსიის შექმნა, რომელსაც უწოდეს Solaris. ამჟამად ის ცნობილია როგორც Oracle Solaris, რადგან Sun Microsystems 2010 წელს კომპანია Oracle-მა შეიძინა.
macOS	თანამედროვე მაკინტოშის სისტემების შექმნაც Unix-დან იღებს სათავეს. კომპანია Apple 1997 წელს NeXT-ის შეძენასთან ერთად დაეუფლა ოპერაციული სისტემა NeXTSTEP-ს, რომელიც წარმოებულია BSD-ს ადრეული ვერსიისგან. კომპანიაში 2001 წელს სისტემა სრულად განაახლეს და გამოუშვეს Mac OS X, რომელიც ასევე დაფუძნებულია Unix-ის მსგავს სისტემა Darwin-ზე (მოხალისეების მიერ შექმნილი). 2012 წელს Mac OS X-ს შეუცვალეს დასახელება OS X-ით. 2016 წლიდან მისი ოფიციალური სახელია macOS.
AIX	ეს სისტემა კომპანია IBM-მა განავითარა, რომელიც ძირითადად IBM-ის mainframe-სა და PowerPC-ზე მუშაობს.
HP-UX	კომპანია Hewlett-Packard-ის მიერ მხარდაჭერილი სისტემაა, რომელიც გამოიყენება HP 9000 ზაზის Workstation-სა და სერვერებზე.
Minix	მიკრობირთვიანი ოპერაციული სისტემაა, რომელიც ძირითადად საგანმანათლებლო მიზნებისთვის გამოიყენება.
BlackBerry 10	მესაკუთრებობრივი მობილური ოპერაციული სისტემა BlackBerry-ის ზაზის სმარტფონებისთვის. BlackBerry 10 დაფუძნებულია Unix-ის მსგავს QNX სისტემაზე.
Tru64	კომპანია Compaq-ის მიერ განვითარებული პროდუქტია Alpha პროცესორებისთვის.
FreeBSD	BSD-ს ყველაზე პოპულარული შთამომავალი. შექმნილია მოხალისეების მიერ.
NetBSD	50-ზე მეტ პლატფორმასთან შეთავსებადი BSD-სგან წარმოებული სისტემა (მოხალისეების მიერ განვითარებული).
OpenBSD	ერთერთი ყველაზე უსაფრთხო და დაცული ოპერაციული სისტემა, ასევე BSD-სგან წარმოებული (მოხალისეების მიერ განვითარებული).
Linux	ყველაზე პოპულარული და სწრაფად განვითარებადი სისტემა (მოხალისეების მიერ განვითარებული).

¹³კლონი არის პროგრამა (ოპერაციული სისტემა ან სხვა), რომელსაც ორიგინალის მსგავსი ქცევა და ფუნქციონალი აქვს და არ შეიცავს საწყის კოდებს.

Android	Linux-ზე დაფუძნებული ოპერაციული სისტემა, რომელიც თავდაპირველად გამიზნული იყო სმარტფონებზე. ახლა ის სმარტფონების გარდა, გამოიყენება ტელევიზორებზე, ლეპტოპებზე, პლანშეტებზე. შეიქმნა კომპანია გუგლის (google) მხარდაჭერით, ხოლო 2005 წელს ის თავად გუგლმა შეისყიდა.
---------	---

ოპერაციული სისტემების შესახებ ქრონომეტრული ინფორმაცია შეგიძლიათ ერიკ ლევენეზის (Éric Lévénez) ვებ გვერდზე მოიძიოთ: <https://www.levenez.com>

ოპერაციული სისტემა ყველაზე მნიშვნელოვანი პროგრამაა, რომელიც ეშვება კომპიუტერზე. ოპერაციული სისტემის გარეშე შეუძლებელია სხვა პროგრამების გაშვება. მისი ძირითადი ამოცანებია მომხმარებელთან ურთიერთობა და კომპიუტერული რესურსების მართვა. მაგალითად, მომხმარებლის მიერ კლავიატურიდან (ან შეტანის სხვა მოწყობილობიდან) შეტანილი ინფორმაციის გაგება, მისი დამუშავება და მიღებული შედეგის გამოტანა მონიტორის ეკრანზე (ან სხვა გამომტან მოწყობილობაზე). ოპერაციული სისტემა არის დამაკავშირებელი რგოლი, შუამავალი, ინტერფეისი მომხმარებელსა და კომპიუტერს შორის. ის აგებინებს კომპიუტერს მომხმარებლის „ნათქვამს“ და პირიქით.

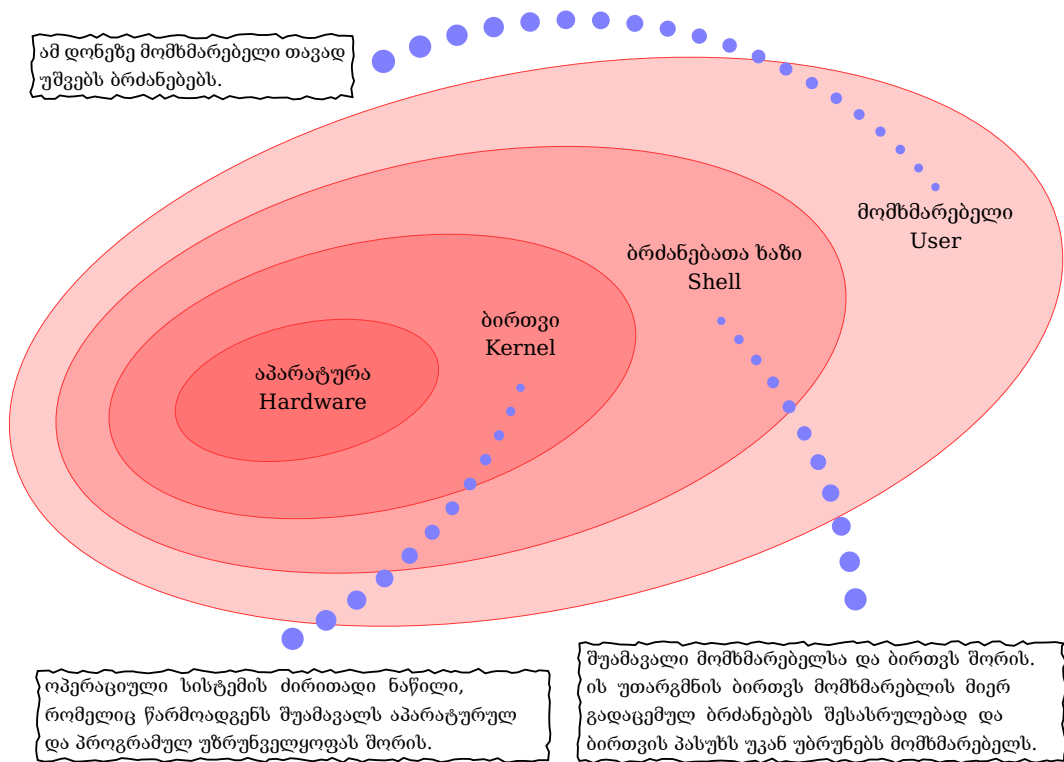
მინიმალისტური ოპერაციული სისტემის მოდელი რომ წარმოვიდგინოთ, ის უნდა შედგებოდეს შემდეგი ნაწილებისგან (იხილეთ სურ. 1.1):

- ბირთვი (kernel) – ეს არის ოპერაციული სისტემის ძირითადი პროგრამა. ის წარმოადგენს დამაკავშირებელ რგოლს კომპიუტერის პროგრამულ (software) და აპარატურულ (hardware) ნაწილებს შორის. თვალსაჩინოებისთვის წარმოვიდგინოთ შემდეგი სცენარი: თქვენ გახსენით ფაილი ერთერთ ტექსტურ რედაქტორში, აკრიფეთ ტექსტი და შემდეგ შეინახეთ ფაილი USB ფლეშ მოწყობილობაზე. პროგრამა, ამ შემთხვევაში ტექსტური რედაქტორი, ამ ოპერაციის შესასრულებლად (ფაილის შესანახად) მიმართავს ბირთვს. ოპერაციის სპეციფიკურობა, თუ როგორ და რანაირად უნდა შეინახოს ფაილი USB ფლეშ მოწყობილობაზე, უცნობი რჩება როგორც ტექსტური რედაქტორისთვის, ასევე მომხმარებლისთვის. ამას აგვარებს ბირთვი.
- ბრძანებათა ზაზი (shell) – ეს არის ის გარსი, სადაც სრულდება მომხმარებლის მიერ მითითებული ბრძანებები. მომხმარებელს პირდაპირი წვდომა არ აქვს ბირთვთან, არამედ მხოლოდ ბრძანებათა ზაზთან. ბირთვი გარშემორტყმულია ბრძანებათა ზაზით როგორც ნიჟარა და მხოლოდ ამ ნიჟარის გავლით შეუძლია მომხმარებელს ბირთვთან ინტერაქცია. Shell ინგლისური სიტყვაა და ქართულად ნიჟარას ნიშნავს. სწორედ ამიტომ უწოდებენ ბრძანებათა ზაზს Shell-ს.
- თავად ბრძანებები – ბინარული ფაილები, გამშვები ფაილები, პროგრამები.

ბინარულ ფაილებს გამშვებ ფაილებსაც უწოდებენ, რადგან ყველა მათგანის გაშვება შესაძლებელია. მათ ასევე შეიძლება ვუწოდოთ აპლიკაცია, პროგრამა ან სკრიპტი. თუმცა, როგორც წესი, სამივეს სხვადასხვა კონოტაცია აქვს და სხვადასხვა სიტუაციაში გამოიყენება.

ბინარული ფაილი იმიტომ ჰქვია, რომ მისი შიგთავსი პირდაპირ კომპიუტერისთვის გასაგებ ფორმატში – ორობით სისტემაშია ჩაწერილი (0-ებითა და 1-ებით). **აპლიკაცია** უფრო მთლიანი ერთეულია, რომელიც შეიძლება შედგებოდეს ერთად მომუშავე ერთი ან რამდენიმე მოდულისგან (ისინი შესაძლებელია იყოს სხვადასხვა დაპროგრამების ენაზე დაწერილი ან/და სხვადასხვა კომპიუტერზე მომუშავე). **პროგრამა** ცალკეული ერთეულია. ეს არის კომპიუტერისთვის გასაგები ინსტრუქციების ერთობლიობა, რომელიც კომპილაციის¹⁴

¹⁴ კომპილაცია არის პროცესი, რომლიც დროსაც მაღალი დონის კომპიუტერულ ენაზე დაწერილი კოდი გარდაიქმნება მანქანურ ენაზე. შედეგად ვღებულობთ გამშვებ ფაილს. ამ პროცესს ასრულებს პროგრამა, რომელსაც კომპილატორი ჰქვია.



სურ 1.1: Unix-ის არქიტექტურა

შედეგად მიიღება. **სკრიპტი** ბრძანებების ერთობლიობაა და მის გასაშვებად ინტერპრეტატორია¹⁵ საჭირო.

1.3 პროგრამები და ლიცენზიები

როდესაც, რომელიმე პროგრამით სარგებლობა გვსურს, მისი გამოყენების ლიცენზია უნდა მოვიპოვოთ. ლიცენზია ე.წ ხელშეკრულებაა, რომელიც ამ ინტელექტუალური ნაშრომის გამოყენების საშუალებას გვაძლევს. ალბათ, ყველა თქვენგანს პროგრამის ინსტალაციის დროს შეგინიშნავთ ფანჯარა, სადაც პროცესის გასაგრძელებლად გარკვეულ პირობებზე დათანხმებაა საჭირო. ლიცენზია სწორედ ამ პირობების ერთობლიობაა და მასში დეტალურადაა გაწერილი თუ რა უფლებებით შეგვიძლია ამა თუ იმ პროდუქტის გამოყენება. სადღეისოდ ლიცენზიების დიდი რაოდენობა არსებობს.

1.3.1 მესაკუთრეობრივი პროგრამა

პროგრამა მაშინ არის მესაკუთრეობრივი, თუ მისი ლიცენზია ამ პროგრამის მხოლოდ გამოყენების უფლებას გვაძლევს. როგორც წესი, მას ახლავს შეზღუდვები. მაგალითად, ხშირად შეუძლებელია პროგრამის ინსტალაცია რამდენიმე კომპიუტერზე. აკრძალულია მისი

¹⁵ინტერპრეტატორი არის პროგრამა, რომელიც პირდაპირ ასრულებს მაღალი დონის კომპიუტერულ ენაზე დაწერილი კოდის ინსტრუქციებს.

გამოკვლევა მასში ცვლილებების შესატანის მიზნით. ამ ტიპის ლიცენზია იმდენად შემზღუდველია, რომ პროგრამაში გაუმართაობის ან ფუნქციონალის არ არსებობისას შემთხვევაში, ისლა დაგვრჩენია, დავუცადოთ, სანამ გამოცემელი თავად არ გააკეთებს ჩასწორებას. ეს კი, როგორც წესი, ხშირად არ ხდება. და თუ მაინც მოხდა, ამ პროცესს დიდი დრო მიაქვს ხოლმე და ბოლოს, პროგრამის განახლება ხშირად ახალ გადასახადთან არის დაკავშირებული. ასეთ შემთხვევებში, პროგრამის მომხმარებელს თითქმის არ აქვს საშუალება დააჩქაროს შესწორების პროცესი.

1.3.2 პროგრამის საცდელი და უფასო ვერსიები

ბევრი ფიქრობს, რომ პროგრამის საცდელი ვერსია (shareware) და უფასო პროგრამული უზრუნველყოფა (freeware) თავისუფალი პროგრამებია, თუმცა ეს ასე არ არის. ორივე მათგანი მესაკუთრეობრივი პროგრამებია, რომელთა გავრცელების საშუალებაც განსხვავდება. Shareware ფასიანი პროგრამული უზრუნველყოფაა, რომელიც გაცნობის მიზნით გარკვეული დროის განმავლობაში უფასოდ მოგვეცემა, ხოლო freeware, როგორც თვითონ სიტყვიდან ჩანს, უფასოა.

ეს კომპიუტერული პროგრამები ისევე დახურულია, როგორც სხვა მესაკუთრეობრივი პროგრამები. თუ ოდესმე ავტორმა შეწყვიტა მისი განვითარება – საფრთხე, რომელიც ბევრ shareware-ს ემუქრება – სხვა ვერაგინ შეძლებს მათ განახლებაზე ზრუნვას.

1.3.3 თავისუფალი პროგრამული უზრუნველყოფა

პროგრამას ვუწოდებთ თავისუფალს, თუ ის რამდენიმე კრიტერიუმს აკმაყოფილებს:

- მოიცემა თავის საწყის კოდთან ერთად, რომელშიც შესაძლებელია ცვლილების შეტანა.
- შესაძლებელია მისი გავრცელება შეზღუდვის გარეშე.
- შესაძლებელია იმ ცვლილებების გავრცელებაც, რომელიც კოდში შევიტანეთ.

რა თქმა უნდა, თავისუფალ პროგრამებსაც აქვთ თავიანთი ლიცენზიები. ასეთ ლიცენზიებს შორის ერთ-ერთი ყველაზე ცნობილია GPL (General Public License). ამ ლიცენზიით გამოდის ბევრი თავისუფალი პროგრამული უზრუნველყოფა, მაგალითად ლინუქსი (Linux), კომპილატორი GCC (GNU Compillr Collection) და სხვა. ეს ლიცენზია იძლევა საშუალებას, რომ მომხმარებელმა საწყისი კოდი შეისწავლოს, მასში ცვლილები შეიტანოს და გააზიაროს. თუმცა, სავალდებულოა, შეცვლილი პროგრამა გავრცელდეს ლიცენზიით თავდაპირველი მინიჭებული უფლებებით. მაშასადამე, შეუძლებელია GPL ლიცენზიის ქვეშ მყოფი პროგრამა გამოვიყენოთ და გავხადოთ მესაკუთრეობრივი. ამგვარად, GPL ლიცენზიის ქვეშ მყოფი პროგრამების განვითარებას საფრთხე ძნელად თუ ემუქრება. თუ ავტორების ინტერესი შეწყდება თავიანთივე პროგრამებისადმი, სხვებს შეეძლება განაგრძონ მათი განვითარება.

GPL ლიცენზიას ლიცენზირების Copyleft ფორმას აკუთვნებენ. Copyleft Copyright¹⁶ სიტყვის თამაშიდან მიღებული ტერმინია. Copyleft ლიცენზიებში მოითხოვება ორიგინალიდან ნაწარმოებ პროდუქტში იმავე უფლებების შენარჩუნების პირობა და აღინიშნება ©სიმბოლოთი. ფართოდ ცნობილია თავისუფალ პროგრამებში გავრცელებული შემდეგი ლიცენზიები და შესაძლებელია მათი დახარისხება (კატეგორიზაცია) თავისუფლების იმ ხარისხის მიხედვით, რასაც თავად გვთავაზობენ:

- ნების დამრთველი (permissive, არა Copyleft) ლიცენზიები:

¹⁶Copyright ანუ საავტორო უფლებები არის უფლებები, რომელიც წარმოემგება ავტორს მის მიერ შექმნილ ნაწარმოებთან დაკავშირებით. საავტორო უფლებების დაცვის ნიშანია ©.

- **BSD** ლიცენზია – ეს არის ბერკლიში კალიფორნიის უნივერსიტეტში შექმნილი ლიცენზია (Berkeley Software Distribution license). BSD ლიცენზია ნაკლებ შემზღუდვებს აწესებს GPL-თან შედარებით. არსებობს მისი 2 ძირითადი ვერსია: „ორიგინალი“ და „New BSD license“ სახელით ცნობილი „მოდულიზირებული“ ვერსია. წლების განმავლობაში ისინი ხშირად იცვლებოდნენ, რამაც ბევრი ახალი ლიცენზიის შექმნას შეუწყო ხელი. მაგალითად, როგორიცაა „ორ-პუნქტიანი BSD ლიცენზია“ (two-clause BSD license), „სამ-პუნქტიანი BSD ლიცენზია“ (three-clause BSD license), „ოთხ-პუნქტიანი BSD ლიცენზია“ (four-clause BSD license). ყველა მათგანი „BSD-ს ტიპის ლიცენზიის“ სახელით არის ცნობილი. ბევრი პროგრამა – ოპერაციული სისტემები FreeBSD, OpenBSD და NetBSD, ვებ სერვერი Nginx, Chromium ვებ ბრაუზერი, დაპროგრამების ენა Ruby, ინტერპრეტატორი tsch, ტექსტური რედაქტორი vi და სხვა სწორედ ამ ლიცენზიით გამოდის.
- **MIT** ლიცენზია – ეს საკმაოდ პატარა და დამთმობი ლიცენზიაა, რომლის ქვეშაც გამოდის X Window System, jQuery და ა.შ. MIT ლიცენზია თავდაპირველად შეიქმნა მასაჩუსეტსის ტექნოლოგიის ინსტიტუტში და ცნობილია X11 და MIT X დასახელებებითაც.
- **Apache** ლიცენზია – Apache Software Foundation (ASF)-ის მიერ შექმნილი ლიცენზიაა. სადღეისოდ, ლიცენზიის ბოლო ვერსიაა 2.0. Apache HTTP Server და OpenOffice სწორედ ამ ლიცენზიით გამოდის.
- **Public domain** – აქ ნაშრომის განთავსება ნიშნავს, რომ ნებისმიერს, ნებისმიერი მიზნებისთვის შეუძლია მისი თავისუფლად გამოყენება.
- ნაკლებ ამკრძალავი ლიცენზიები (ნაკლებ Copyleft ლიცენზიები):
 - **MPL** (Mozilla Public License) – Mozilla ფონდის მიერ შექმნილი ლიცენზია. ლიცენზიის სადღეისოდ ბოლო ვერსიაა 2.0. ამ ლიცენზიით გამოდის Mozilla Firefox, Mozilla Thunderbird და Mozilla-ს სხვა პროგრამები.
 - **GNU LGPL** (GNU LESSER GENERAL PUBLIC LICENSE) – არის თავისუფალი პროგრამების ფონდში (Free Software Foundation – FSF) შექმნილი ლიცენზია. ის საშუალებას იძლევა პროგრამის ინტეგრირება მოხდეს სხვა პროგრამებში (თუნდაც მესაკუთრეობრივში), ისე რომ არ მოითხოვოს სხვა კომპონენტების კოდის გახსნა. ბევრი მესაკუთრეობრივი პროგრამა იყენებს LGPL-ის ლიცენზიების მქონე კოდებს, ძირითადად გაზიარებული ბიბლიოთეკების (shared library) ფორმით.
- ძლიერ ამკრძალავი ლიცენზიები (Copyleft ლიცენზიები):
 - **GNU GPL** – არის Free Software Foundation (FSF)-ში შექმნილი ლიცენზია. ეს იყო პირველი Copyleft ლიცენზია ზოგადი გამოყენებისთვის. თავდაპირველი ვერსიის შემდეგ, რომელიც რიჩარდ სტალმანმა შექმნა, გამოვიდა კიდევ ორი – GNU GPLv2 და GNU GPLv3 ვერსიები.
 - **Affero GPL** – Affero General Public License. ეს ლიცენზია სპეციალურად შეიქმნა ისეთი პროგრამებისთვის, როგორიცაა ვებ-აპლიკაციები. მისი მიზანია, რომ კომპიუტერული ქსელიდან მიედებულ მომხმარებელს, რომელიც იყენებს შეცვლილ პროგრამას, შეეძლოს მისი საწყისი კოდის მიღება. FSF-ის მიერ გამოცემული ბოლო ვერსია არის GNU AGPLv3.

იხილეთ ლიცენზიების შედარებითი ცხრილი შემდეგი კრიტერიუმების მიხედვით:

- ლიცენზიით გათვალისწინებული რა უფლებები გვაქვს მონიჭებული: შესაძლებელია თუ არა, ამ ლიცენზიის ქვეშ არსებული პროგრამის გავრცელება (სხვისთვის გადაცემა)? შეგვიძლია თუ არა,

იმავე ან მისგან წარმოებული პროგრამის გამოყენება კომერციული მიზნებისთვის? ან პროგრამის კოდში ცვლილებების შეტანა? ითვალისწინებს თუ არა პროგრამის ლიცენზია პატენტის უფლებების მინიჭებას კონტრიბუტორებისგან (ავტორებისგან)? ან შეგვიძლია თუ არა, პროგრამა გამოვიყენოთ პირადი მიზნებისთვის?

- ლიცენზიით გათვალისწინებული რა შეზღუდვები გვაქვს დაკისრებული: შეგვიძლია თუ არა, პროგრამის საწყისი კოდი დავხუროთ? აუცილებლად უნდა მოვაყოლოთ თუ არა ლიცენზიის ასლი და საავტორო უფლებები პროგრამას? კომპიუტერული ქსელიდან მიერთებულ მომხმარებელსაც აქვს თუ არა უფლება მიიღოს საწყისი კოდის ასლი? იმავე ან მსგავსი ლიცენზიით უნდა იყოს თუ არა მოდიფიცირებული პროგრამა? აუცილებლად უნდა მივანიშნოთ გაკეთებულ ცვლილებებზე? უნდა აღვწეროთ ისინი დოკუმენტურად?
- ლიცენზიით გათვალისწინებული რა უფლებები გვაქვს მინიჭებული: შეიცავს თუ არა პროგრამის ლიცენზია შეზღუდვებს პასუხისმგებლობაზე? მომხმარებლისთვის არსებობს თუ არა იმის გარანტია, რომ გარკვეული ხარვეზის აღმოჩენის შემთხვევაში, კომპანია ან ავტორი უფასოდ გამოასწორობს ან შეცვლის პროდუქტს? ცალსახად აკრძალული გვაქვს თუ არა პროდუქტის სავაჭრო ნიშნის გამოყენება?

	უფლებები					გავრცელების პირობები					შეზღუდვები		
	გავრცელება	კომერციული მიზნები	კოდის ცვლილება	დაპატენტება	პირადი მიზნები	საწყისი კოდის ლიად დატოვება	ლიცენზიის ასლი/ხაავტორი უფლებები	კოდის ასლი ქსელურ მომხმარებლისთვის	შეცვლილ ვერსიაში იმავე ლიცენზიით	ცვლილებების მინიმუმება	პასუხისმგებლობის არიდება	სავაჭრო ნიშნის აკრძალვა	გარანტიის არარსებობა
BSD 2-clause „Simplified“ License	○	○	○		○		○				○		○
BSD 3-clause Clear License	○	○	○	არა ¹⁷	○		○				○		○
BSD 3-clause „New“ or „Revised“ License	○	○	○		○		○				○		○
MIT License	○	○	○		○		○				○		○
Apache License 2.0	○	○	○	○	○		○			○	○	○	○
GNU LGPLv3	○	○	○	○	○	○	○		○	○	○		○
Mozilla Public License 2.0	○	○	○	○	○	○	○		○		○	○	○
GNU GPLv2	○	○	○		○	○	○		○	○	○		○
GNU GPLv3	○	○	○	○	○	○	○		○	○	○		○
GNU AGPLv3	○	○	○	○	○	○	○	○	○	○	○		○

1.4 თავისუფალი პროგრამების განვითარება

1983 წელს რიჩარდ სტალმანი (Richard Stallman) იწყებს ახალ პროექტს სახელწოდებით GNU, რომლის მთავარი მიზანია სრულიად თავისუფალი პროგრამებისაგან შემდგარი ოპერაციული სისტემის შექმნა. პროექტის ლოგოა ანტილოპა გნუ-ს თავი¹⁸. ამ დროისთვის Unix-სგან წარმოებული ბევრი ოპერაციული სისტემა გადაკეთდა არათავისუფალ სისტემად. ამ ფაქტმა იმდენად დიდი პროტესტის გრძნობა გააჩინა რიჩარდ სტალმანში, რომ მან თავის პროექტს დაარქვა GNU (Gnu's Not Unix). ეს რეკურსიული აკრონიმია და ზაზს

¹⁷ლიცენზია ზაზგახშით გამოირიცხავს დაპატენტების საშუალებას.



GNU-ს ლოგოს ეს გამართივებული ვარიანტი შექმნილია აურელიო ჰეკერტის (Aurelio Heckert) მიერ. ლოგოს თავდაპირველი ვერსია კი შექმნა ეტიენ სუვაზამ (Etienne Suvasa).
www.gnu.org

უსვამს, რომ GNU არ არის Unix. ამ პროექტის ფარგლებში მან შექმნა შემდგომი პროდუქტები: GNU Compiler Collection (GCC), GNU Debugger, ტექსტური რედაქტორი GNU Emacs, ლიცენზია GNU GPL. პირველმა სწორედ სტალმანმა განავითარა Copyleft კონცეფცია.

თავისუფალი Unix!

მადლიერების¹⁹ ამ დღიდან ვაპირებ შევექმნა Unix-თან შეთავსებადი სრული სისტემა GNU (Gnu's Not Unix). ის იქნება ღია და მისი გამოყენების უფლება ექნება ყველას. მივესალმები დახმარებას ნებისმიერი ფორმით: დრო, ფული, პროგრამები, აღჭურვილობა ...

– რიჩარდ სტალმანი; გამოქვეყნებული net.unix-wizards-ზე; 1983 წლის 27 სექტემბერი.

Free Unix!

Starting this Thanksgiving I am going to write a complete Unix-compatible software system called GNU (for Gnu's Not Unix), and give it away free to everyone who can use it. Contributions of time, money, programs and equipment are greatly needed...

– Richard Stallman; Posting to net.unix-wizards; 27 Sept. 1983.

1985 წელს იგი არამომგებიან ორგანიზაციას – თავისუფალი პროგრამების ფონდს (FSF²⁰) აფუძნებს. ფონდის მიზანია კომპიუტერული პროგრამების თავისუფლად გავრცელების, შექმნისა და თავისუფლად გარდაქმნის მხარდაჭერა. ამ ფონდში ის აგრძელებს უკვე არსებულ GNU პროექტებს, თუმცა სრულფასოვანი, Unix-ის მსგავსი, GNU ოპერაციული სისტემის შესაკვრელად (ჩამოსაყალიბებლად) მას აკლია ძირითადი კომპონენტი – ბირთვი. მართალია, 1990 წლიდან GNU პროექტის ფარგლებში დაიწყო ბირთვის შექმნა, სახელად Hurd, თუმცა საბოლოოდ ის ვერ განვითარდა.

1.5 ლინუქსის ისტორია

1990 წელს, ფინეთში, ჰელსინკის უნივერსიტეტის სტუდენტი – ლინუს ტორვალდსი (Linus Torvalds) მიიღებს პერსონალურ კომპიუტერს 386²¹-ის ბაზაზე, 40 მბ ტევადობის ხისტი დისკითა და 4 მბ ზომის ოპერაციული მენსიერებით. ის, როგორც ინფორმატიკის მიმართულების სტუდენტი, თავის კომპიუტერზე ცდის უნივერსიტეტში არსებულ სისტემებს და მალევე იმედგაცრუებული დარჩება. ამის შემდეგ ის შეეცდება ინტერნეტში მოიძიოს მისთვის სასურველი ოპერაციული სისტემა, თუმცა ვერც ამ ჯერზე იპოვის მისთვის დამაკმაყოფილებელ ან/და ფინანსურად ხელსაყრელ ვარიანტს (იმ დროს კომერციული Unix Intel 386 კომპიუტერებისთვის საკმაოდ ძვირი იყო) და გადაწყვეტს ჰობად გაიხადოს საკუთარი სისტემის წერა.

1991 წელს ლინუს ტორვალდსმა დაიწყო ოპერაციული სისტემის ბირთვის წერა. ის მუშაობდა Minix-ზე და იყენებდა C და ასემბლერ დაპროგრამირების ენებს, ასევე GNU C კომპილატორს. საბოლოო პროდუქტს დაერქვა „Linux“.²²

¹⁹მადლიერების დღე (Thanksgiving Day) არის ეროვნული დღესასწაული აშშ-ში, კანადასა და სხვა ქვეყნებში. აშშ-ში ის ნოემბრის მე-4 ხუთშაბათს აღინიშნება.

²⁰FSF – Free Software Foundation

²¹Intel 80386 ცნობილი აგრეთვე, როგორც i386 ან უბრალოდ 386 არის 32 ბიტიანი მიკროპროცესორი. გამოვიდა 1985 წელს.

²²თავდაპირველად ლინუს ტორვალდს სურდა ბირთვისთვის დაერქვია „Freax“, როგორც სამი სიტყვის კომბინაცია: „Free“ (თავისუფალი), „Freak“ (ახირება, უცნაური აჩემება) და „X“ (როგორც Unix-ის ალუზია, მინიშნება). პროექტის ერთ-ერთ დასაკომპილირებელ ფაილსაც სწორედ „Freax“ ერქვა. საწყის ეტაპზე ლინუსს განხილული ჰქონდა „Linux“-ის ვარიანტი მაგრამ, მეტისმეტად ეგოისტურად მიიჩნია და გადაიფიქრა. პროექტის საბოლოო ვერსია „Linux“-ის დასახელებით აიტვირთა ftp სერვერზე. საქმე ის იყო, რომ სერვერის ერთ-ერთმა ადმინისტრატორმა, არი ლემკემ (Ari Lemmke), რომელიც ამავდროულად ლინუს ტორვალდის თანამშრომელი იყო ჰელსინკის ტექნოლოგიურ

გამარჯობა ყველას, ვინც აქ იყენებს minix-ს { მე ვქმნი (თავისუფალ) ოპერაციულ სისტემას (მხოლოდ ჰობია. არ იქნება დიდი და ისეთი პროფესიონალური, როგორიც GNU) 386(486) AT კლონებისთვის... მაინტერესებს ხალხის აზრი, თუ რა შესაძლებლობების ხილვა სურთ ამ სისტემაში. ნებისმიერ წინადადებას მივესალმები, მაგრამ არ გპირდებით, რომ ყველა მათგანს გავითვალისწინებ :-)

– ლინუს ტორვალდსი; გამოქვეყნებული comp.os.minix-ზე; 1991 წლის 25 აგვისტო.

Hello everybody out there using minix - I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386 (486) AT clones... I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

– Linus Torvalds; Posting to comp.os.minix; 25 Aug. 1991.

ბირთვის თავდაპირველი ვერსია 0.01 ლინუს ტორვალდსმა გამოუშვა 1991 წლის 17 სექტემბერს და მისი კოდი განათავსა FTP საიტზე (ftp.funet.fi). კოდი შეიცავდა 10 239 ხაზს. დეკემბრის თვეში გამოვიდა ახალი სტაბილური ვერსია 0.11 ლინუსის გამოგონილი ლიცენზიით. ამავდროულად, ეს ბირთვი გამოიყენებოდა ისეთ პროგრამებთან ერთად, რომლებიც GPL ლიცენზიით GNU პროექტის ნაწილი იყო. ტორვალდსმა, იცნობდა რა GNU პროექტის ფარგლებში შესრულებული სამუშაოების მაღალ ხარისხს, გადაწყვიტა, თავის ნაშრომსაც GPL ლიცენზია მიენიჭებინა და GNU სისტემას მიერთებინა. იგი იმედოვნებდა, რომ ამით მისი სისტემაც დიდ პოპულარობას მოიპოვებდა. 1992 წლის თებერვალში ტორვალდსმა GNU GPL ლიცენზიით გამოუშვა ბირთვის განახლებული ვერსია – 0.12. მოგვიანებით, ტორვალდსმა აღნიშნა – „ლინუქსის GPL-ში განთავსება საუკეთესო რამ იყო, რაც ოდესმე გამიკეთებია“. მას შემდეგ ინტერნეტის საშუალებით ბირთვი საგრძნობლად გაუმჯობესდა ტორვალდსისა და მოხალისე პროგრამისტების მიერ. სადღეისოდ, ლინუქსის განვითარებაში ათასობით პროგრამისტი მონაწილეობს და მისი კოდი ათობით მილიონამდე ხაზს შეიცავს.

დღეს, ლინუქსი შეიცავს ყველა იმ ფუნქციონალს, რომელიც თანამედროვე, სრულფასოვანი სისტემისათვის არის საჭირო. ლინუქსი არის მრავალმოქმედებიანი (შესაძლებელია, მრავალი მომხმარებელი დაუკავშირდეს სისტემას და ერთდროულად იმუშაოს), მრავალამოცანიანი (მომხმარებელს შეუძლია მრავალი ამოცანის, პროცესის გაშვება ერთდროულად), პორტატული (ის შეთავსებადია ბევრ კომპიუტერულ არქიტექტურასთან, როგორიცაა: x86, IA-64, ARM, PowerPC, SPARC, SuperH და ა.შ.). სადღეისოდ (20.2020) ლინუქსის ბოლო სტაბილური ვერსიაა 5.7.4 (პირველი რიცხვი ბირთვის ვერსიას წარმოადგენს, მეორე და მესამე კი მთავარი (major) და მეორეხარისხოვანი (minor) რევიზიის აღმნიშვნელი რიცხვებია). ახლა, როდესაც თქვენ ამ წიგნს კითხულობთ, შესაძლებელია, ბოლო ვერსია უკვე სხვა იყოს. ბირთვის ვერსიების არქივის ნახვა შეგიძლიათ www.kernel.org ვებ-საიტზე.

1996 წელს ტორვალდსმა ლინუქსის თილისმად გამოაცხადა პინგვინი, რომელსაც

უნივერსიტეტში Helsinki University of Technology (HUT), ერთპიროვნულად ჩათვალა, რომ ბირთვისთვის „Freax“-ის დარქმება არ იყო კარგი იდეა და ლინუსთან კონსულტაციის გარეშე პროექტს „Linux“ დაარქვა. საბოლოოდ ლინუსი დათანხმდა.

დაარქვეს TUX²³. სიტყვა „Linux“ არის ლინუს ტორვალდსის მიერ დარეგისტრირებული სავაჭრო ნიშანი და წარმოადგენს მფლობელის კუთვნილებას.

1.5.1 თანამშრომლობითი განვითარება

იშვიათია პროგრამისტი, რომელსაც შეუძლია კომპლექსური პროგრამის ყველა ნაწილის, ყველა ეტაპის ბოლომდე ოპტიმალურ დონეზე მიყვანა. როგორც წესი, ისინი კონკრეტული დარგის სპეციალისტები არიან. თანამშრომლობითი განვითარებით კი, რომელიც თავისუფალი პროგრამების ლიცენზიების წყალობით გვაქვს, თითოეულ პროგრამისტს შეუძლია, თავისი მაღალი კომპეტენციების ფარგლებში, პროგრამის ნაწილი განავითაროს. თავისუფალი პროგრამული უზრუნველყოფის წარმატებაც სწორედ ამ პრინციპზეა დამყარებული. არსებობს თავისუფალი პროგრამების ბევრი საზოგადოება, მაგალითად ლინუსის საზოგადოება (Linux community). ეს საზოგადოებები ინფორმაციას ცვლიან ინტერნეტში ფორუმებისა და დასაგზავნი სიების (mailing lists) საშუალებით. შესაბამისად, სირთულეს არ წარმოადგენს, პრობლემის შემთხვევაში, დახმარება მიიღოთ ამდენი საზოგადოებიდან. თანამშრომლობითი მიდგომა ხშირად მომხმარებლებზეც გადადის და შემდეგ ეს განწყობა მათშიც ვითარდება.

რადგან Linux არის მხოლოდ და მხოლოდ ბირთვი, არასწორია ვთქვათ ოპერაციული სისტემა – Linux. ოპერაციული სისტემა არის GNU/Linux. GNU/Linux ვრცელდება დისტრიბუტივების (distro) ფორმით. დისტრიბუტივი შეიცავს თავად ოპერაციულ სისტემას და დამატებითი პროგრამების დიდ კოლექციას, რაც მომხმარებელს ძალიან კომფორტულ სამუშაო გარემოს უქმნის.

არსებობს სრულიად თავისუფალი GNU/Linux დისტრიბუტივები. ისინი მხოლოდ თავისუფალ პროგრამებს შეიცავენ. ასეთებია: **Dragora**, **Dynebolic**, **Parabola**, **Hyperbola**, **Trisquel**, **gNewSense** და სხვა.

უფრო გავრცელებული და ცნობილი დისტრიბუტიბებია:

დისტრიბუტივი	ვებ-გვერდი
Debian	www.debian.org
Ubuntu	www.ubuntu.com
Red Hat	www.redhat.com
CentOS	www.centos.org
Fedora	getfedora.org
Arch Linux	www.archlinux.org
Linux Mint	linuxmint.com
Gentoo	www.gentoo.org
Manjaro	manjaro.org
Elementary OS	elementary.io
Knoppix	www.knopper.net
Suse	www.suse.com

23



TUX – (Torvalds' UniX). თავდაპირველად Tux-ის რასტრული ვერსია 1996 წელს შექმნა ლარი ევინგმა (Larry Ewing) ალან კოქსის (Alan Cox) რჩევით. შემდეგ კი დაიხვეწა თავად ლინუს ტორვალდსის მიერ.

მათ გარდა კიდევ ბევრი დისტრიბუტიბი არსებობს. ისინი თავისუფალ პროგრამებთან ერთად შეიცავენ არათავისუფალ ნაწილსაც – მაგალითად აპლიკაციას, დრაივერს (driver), ჩაშენებულ პროგრამას (firmware), თამაშსა თუ დოკუმენტაციას.

1.5.2 ლინუქსის დოკუმენტაცია

ლინუქსი არის ერთ-ერთი ყველაზე კარგად დოკუმენტირებული სისტემა. მისი ბრძანებების თითქმის სრულ უმრავლესობას თანდართული აქვს სრულყოფილი დოკუმენტაცია და პროცედურების უმრავლესობას თან ახლავს ყოველი ეტაპის დეტალური აღწერილობა, რომლებსაც HOWTO ჰქვია. გარდა ამისა, ასოციაცია „ლინუქსის დოკუმენტაციის პროექტის“ წეალობით (The Linux Documentation Project – tldp.org), რომელიც მიზანიც არის ლინუქსთან დაკავშირებული სხვადასხვა დოკუმენტაციის მოძიება და შეგროვება, ამ დოკუმენტაციის უმრავლესობა ნათარგმნი არის მსოფლიოს ბევრ ენაზე.

ხანდახან დოკუმენტაციის ასეთი სიმრავლე თავსატეხსაც წარმოადგენს. ზოგჯერ რთულია ასეთი დიდი რაოდენობის დოკუმენტაციიდან ზუსტად მორგებული დახმარების ამოჩვენება. Unix-ის ბრძანებების უმრავლესობას -h ან -help ან --help ოფცია გააჩნია. მათი დახმარებით მომხმარებელს შეუძლია მოკლე დამხმარე ინფორმაციის ნახვა – თუ როგორ შეიძლება სინტაქსურად გამართული ფორმით ამ ბრძანების ჩაწერა. ხშირად მხოლოდ ბრძანების გაშვებაც საკმარისია, რომ ინტერპრეტატორმა თვითონ მიგვითითოს არასრულად აკრეფილი ბრძანების სწორი ჩაწერის შესაძლო ფორმებზე. თუ გვსურს უფრო დეტალური ინფორმაციის მიღება კონკრეტული ბრძანების შესახებ, შეგვიძლია man ბრძანება გამოვიყენოთ. man, იგივე man page ან manual page (სახელმძღვანელო გვერდი) ძირითადად შედგება შემდეგი სექციებისგან:

NAME – ბრძანების დასახელება, რომელსაც თან ახლავს ერთხაზიანი განმარტება
SYNOPSIS – ბრძანების ჩაწერის სწორი ფორმა
DESCRIPTION – დეტალური აღწერა, თუ რას აკეთებს ეს ბრძანება
OPTIONS – ბრძანების ოფციები თავიანთი განმარტებით
EXAMPLES – ბრძანების გამოყენების მაგალითები
SEE ALSO – ამ ბრძანების მსგავსი სხვა ბრძანებების სია

Unix-ის ბრძანების ჩაწერის ზოგადი სტრუქტურა ასეთია:

`command [option(s)] [argument(s)]`

კვადრატული ფრჩხილები, როგორც man page-ის SYNOPSIS სექციაში იგულისხმება, მიუთითებს, რომ ოფციისა და არგუმენტის მითითება არასავალდებულოა. ანუ ასეთი ბრძანება სინტაქსურად შეიძლება ჩაწეროთ ოფციისა და არგუმენტის გარეშე, ან მხოლოდ ოფციით, ან მხოლოდ არგუმენტით, ან ოფციითა და არგუმენტით ერთად, ასე:

- `command`
- `command option(s)`
- `command argument(s)`
- `command option(s) argument(s)`

კონკრეტული ბრძანების სახელმძღვანელო გვერდის გამოსატანად ბრძანებათა ზაზში `man` ბრძანებას არგუმენტად უნდა მივუთითოთ ამ ბრძანების სახელი და ისე გავუშვათ. მაგალითად, თავად `man` ბრძანების სახელმძღვანელო გვერდის გამოტანა ასე შეგვიძლია:

```
achiko@debian:~$ man man
```

დაინსტალირებული პაკეტების დოკუმენტაცია, ძირითადად, `/usr/share/doc` დირექტორიაშია განთავსებული და ის ხელმისაწვდომია რამდენიმე ფორმატით: ტექსტური, html, PostScript და სხვა. ბრძანების ყველაზე დეტალური დოკუმენტაცია HOWTO კი განთავსებული არის `/usr/share/doc/HOWTO` დირექტორიაში.

თავი 2

ბრძანებათა ხაზი

ბრძანებები Unix-ში შეიძლება გაეშვას ბრძანებათა ხაზიდან სხვა ბრძანებებთან ერთად კომბინაციაში. სწორედ ესაა Unix-ის ფილოსოფიის ძლიერ მხარე – უმჯობესია, გამოვიყენოთ რამდენიმე სხვადასხვა სპეციალიზებული ინსტრუმენტების კომპლექტი კონკრეტულ ამოცანაში, ვიდრე გამოვიყენოთ ერთი ისეთი აპლიკაცია, რომლებიც მრავალმხრივი, კომპლექსური და უფრო ფართო დანიშნულების არის.

2.1 shell

ლინუქსში მუშაობისას ხშირად ვსაუბრობთ ხოლმე ბრძანებათა ხაზზე (CLI – Command Line Interface). მისი საშუალებით ვეუბნებით კომპიუტერს თუ რისი გავთვება გვსურს. ბრძანებათა ხაზის ქვეშ, სინამდვილეში, ვგულისხმობთ shell-ს. Shell არის პროგრამა, რომელიც კლავიატურიდან იღებს ჩვენგან გადაცემულ ბრძანებებს და შესასრულებლად გადასცემს სისტემას. ლინუქსის თითქმის ყველა დისტრიბუტივი შეიცავს GNU პროექტის shell-ის ვარიანტს bash-ს. Unix-ის ორიგინალი shell (sh) 1979 წელს Steve Bourne-მა დაწერა Bell Labs-ში. Bash (Bourne Again SHell) კი მისი გაუმჯობესებული ვარიანტია, რომელიც ბრაიან ფოქსმა (Brian Fox) შექმნა. მისი პირველი ვერსია 1989 წელს გამოვიდა და სწრაფადვე ფართოდ გავრცელდა. bash, როგორც ძირითადი shell, გვხვდება არამხოლოდ ლინუქსის დისტრიბუტივებზე, არამედ Apple-ის macOS-ზე. Bash-ის ერთი ვერსია Windows 10-ზეც არის ხელმისაწვდომი.

Shell-ში სამუშაოდ საჭიროა ტერმინალის¹ ქონა ან ვირტუალური ტერმინალის გაშვება (ამიერიდან ვირტუალურ ტერმინალს, მოკლედ, ტერმინალს ვუწოდებთ ხოლმე). თუ თქვენ იყენებთ გრაფიკულ გარემოს (GUI – Graphical User Interface), მაშინ shell-ის გასაშვებად დაგჭირდებათ ტერმინალის ემულატორები. ასეთებია, მაგალითად konsole – KDE-ს შემთხვევაში ან gnome-terminal – GNOME-ის შემთხვევაში. არსებობს ბევრი სხვა ტერმინალის ემულატორებიც.

გრაფიკულ რეჟიმის უკან, პარალელურად, სისტემაში რამდენიმე ვირტუალური ტერმინალია გაშვებული, რომელიც ტექსტურ სამუშაო გარემოს გვთავაზობს (TUI – Text-based User Interface). მათ ხშირად ვირტუალურ კონსოლს (console) უწოდებენ.

¹თავდაპირველად ტერმინალს ეძახდნენ მოწყობილობას, რომლის მეშვეობითაც უკავშირდებოდნენ კომპიუტერს. Unix-ის ადრეულ პერიოდში ტერმინალით მოიხსენიებდნენ საბეჭდი მანქანის მსგავს მოწყობილობას teleprinter-type device-ს, იგივე teletypewriter-ს ან მოკლედ tty-ს. ეს იყო ელექტრო მექანიკური მოწყობილობა, რომლის საშუალებითაც იგზავნება ტექსტური შეტყობინება. უფრო ტრადიციული გაგებით ტერმინალი არის მოწყობილობა, რომლითაც შეგვიძლია კომპიუტერთან ინტერაქცია კლავიატურისა და დისპლეის მეშვეობით. სადღესოდ პროგრამები უზრუნველყოფენ ერთი ფიზიკური ტერმინალის ნაცვლად პროგრამულად რამდენიმე ვირტუალური ტერმინალის არსებობის შესაძლებლობას. მათ აღსანიშნავად ხშირად შეგვხვდება სხვადასხვა ტერმინები: „ტერმინალი“, „ვირტუალური ტერმინალი“, „კონსოლი“, „ვირტუალური კონსოლი“.

ვირტუალურ კონსოლზე გადასვლა ხდება **Alt** + **Ctrl** + **F1** , **Alt** + **Ctrl** + **F2** , ... **Alt** + **Ctrl** + **F7** კლავიშების კომბინაციით (აქედან ერთ-ერთი გრაფიკულ გარემოში დაგვაბრუნებს, ზოგ დისტრიბუტივზე ეს **Alt** + **Ctrl** + **F1** -ია, ზოგზე კი **Alt** + **Ctrl** + **F7**). ტერმინალის ემულატორებისგან განსხვავებით, კონსოლი სრულ ეკრანზეა გაშლილი.


ყურადღება!

console არ აგერიოთ **konsole**-ში. ქართულად ორივე ერთნაირად ჟღერს – კონსოლი. **console** ზოგადი ტერმინია. მას ხშირად პირველ ტერმინალად მოიხსენიებენ, რომელზეც სისტემის შეტყობინებები გამოდის, ხოლო **konsole** ვირტუალური ტერმინალის ერთ-ერთი პროგრამული უზრუნველყოფის დასახელებაა.

დავიწყოთ. ამ წიგნში ნაჩვენები ყველა მაგალითი Debian 9 (stretch)/Debian 10 (buster) დისტრიბუტივების ბაზაზეა განხორციელებული, თუმცა ლინუქსის ნებისმიერი დისტრიბუტივისთვისაც იქნება სამართლიანი.

მოდით, გავხსნათ ერთ-ერთი ტერმინალის ემულატორი, მაგალითად `gnome-terminal`, რომელშიც ნაგულისხმევი მნიშვნელობით (by default) გაშვებული არის `bash`. გახსნილ ფანჯარაში ასეთი ან მსგავსი ჩანაწერი დაგვხვდება, რომლის ბოლოშიც კურსორი იციმციმებს.

```
achiko@debian:~$
```

ეს არის shell-ის მოსაწვევი (shell invitation, prompt) და მისი გამოჩენა ნიშნავს, რომ shell მომხმარებლის მიერ ბრძანების შეტანას ელოდება. ვცადოთ და ავკრიფოთ რამე. შევევანის ლილაკით –  შესასრულებლად გავუშვათ ჩვენ მიერ ახლახანს შეტანილი „ბრძანება“. რადგანაც ის უბრალოდ სიმბოლოების ერთობლიობაა და არ წარმოადგენს shell-ის ნამდვილ ბრძანებას, shell მას ვერ გაიგებს და ასეთ პასუხს დაგვიბრუნებს:

```
achiko@debian:~$ asdfasd
-bash: asdfasd: command not found
achiko@debian:~$
```

ერთი ბრძანების დასრულების შემდეგ, მიუხედავად იმისა, ის წარმატებით შესრულდება თუ წარუმატებლად, shell-ში კვლავ მოსაწვევი გამოვა. ეს იმის მანიშნებელია, რომ shell მორიგ ბრძანებას ელის მომხმარებლისგან. კლავიატურაზე მიმართულების კლავიშებით შეგიძლიათ ნახოთ და გაუშვათ წინ აკრეფილი ბრძანებები.

ყურადღება!

არ ეცადოთ **Ctrl-c** და **Ctrl-v** კლავიშების კომბინაციით მაუსით მონიშნული ტექსტის კოპირება და ჩასმა. ლინუქსში გრაფიკული სისტემა მონიშნულ ტექსტს ავტომატურად აკოპირებს ბუფერში და მისი ჩასმა მარტივად შეგიძლიათ მაუსის შუა ლილაკზე დაჭერით.

2.2 ლინუქსის საბაზისო ბრძანებები

ბრძანება `ls` ყველაზე ხშირად გამოყენებადი ბრძანებაა `shell`-ში და მას ეკრანზე გამოაქვს მიმდინარე დირექტორიის² შიგთავსი, მასში არსებული ფაილებისა და დირექტორიების სია. ვცადოთ:

```
achiko@debian:~$ ls
Desktop/   Downloads/ Pictures/  Templates/
Documents/ Music/     Public/   Videos/
```

მშვენიერი!

მოდით, გავიგოთ რომელ დირექტორიაში ვიმყოფებით ახლა. ამაში ბრძანება `pwd` (Print Working Directory) დაგვცხმარება.

```
achiko@debian:~$ pwd
/home/achiko
```

როგორც ჩანს, `/home/achiko`-ში ვიმყოფებით სამუშაოდ. ზოგადად, როდესაც მომხმარებელი სისტემაში შედის ტერმინალით, ის ავტომატურად საკუთარ დირექტორიაში აღმოჩნდება ხოლმე. ეს ის გარემოა, სადაც მომხმარებელი ინახავს საკუთარ ფაილებს, დირექტორიებს, პროგრამებს და სხვა.

Unix-ის მსგავს სისტემებში ფაილები განაწილებულია იერარქიულ, ხისებრ სტრუქტურაში. პირველ დირექტორიას ძირი დირექტორია ჰქვია (root directory) და აღინიშნება `/` -ით, რომელიც შეიცავს ქვედირექტორიებსა და ფაილებს. თავის მხრივ, ეს დირექტორიებიც შეიცავენ ქვედირექტორიებსა და ფაილებს. დირექტორიების სტრუქტურა გრაფიკულად ასე შეიძლება წარმოვიდგინოთ. იხილეთ სურათი 2.1.

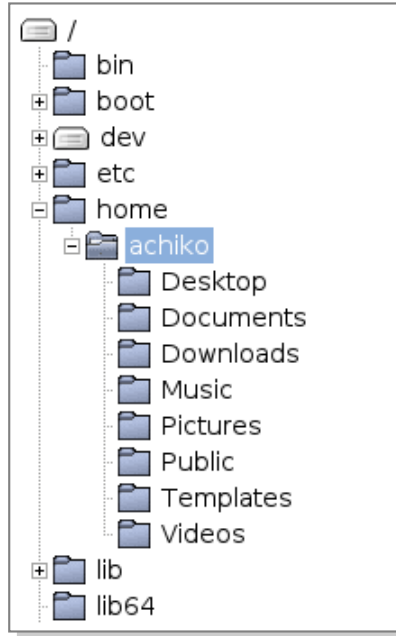
`cd` (Change Directory) ბრძანებით ერთი დირექტორიიდან მეორეში გადავდივართ. ამისთვის `cd`-ს არგუმენტად იმ დირექტორიის გზის სახელი (Pathname, მოკლედ Path) უნდა მივუთითოთ, სადაც გვსურს გადასვლა. Path, ფაქტობრივად, არის ხისებრ სტრუქტურაში გზა ფაილისკენ³. Path შეიძლება ჩაიწეროს ორი ფორმით – როგორც აბსოლუტური გზა (absolute path) ან როგორც ფარდობით გზა (relative path).

აბსოლუტური გზა ძირი დირექტორიიდან იწყება `/`-ით. მაგალითად, `/usr/bin` აბსოლუტური გზაა და აღნიშნავს `bin` დირექტორიას, რომელიც განთავსებულია დირექტორია `usr`-ში, ხოლო `usr` კი თავის მხრივ განთავსებულია `/`-ში. პირველი `/`, როგორც აღვნიშნეთ, ძირი დირექტორიის სახელია, ხოლო დანარჩენი `/`-ები უბრალოდ დირექტორიებს შორის გამყოფ სიმბოლოდ მოიაზრება. ამას გარდა, დირექტორიის დასახელებას, იმის ზაზგასასმელად, რომ საქმე დირექტორიას ეხება და არა სხვა ტიპის ფაილს, ხშირად `/`-ს უმატებენ ხოლმე ბოლოში (როგორც დირექტორიებს შორის გამყოფს).

```
achiko@debian:~$ cd /usr/bin
achiko@debian:/usr/bin$ pwd
```

²დირექტორიას მეორენაირად საქალაქდევაც ეძახიან, თუმცა ისინი ერთსა და იმავეს აღნიშნავს. საქალაქდევს უფრო ხშირად მაშინ ვუწოდებთ, როდესაც გრაფიკულ გარემოში ვმუშაობთ და მას რაიმე პიქტოგრამა, ხატულა შეესაბამება. ბრძანებათა ზაზში მუშაობისას კი უპირატესად დირექტორიას ვიყენებთ.

³ლინუქსში ყველაფერი ფაილია. დირექტორიაც ფაილია, მხოლოდ დირექტორიის ტიპის. არსებობს ფაილის სხვა ტიპებიც. ტრადიციული გაგების ფაილს ლინუქსში ჩვეულებრივი ტიპის ფაილს უწოდებენ. თუმცა ხშირად, უბრალოდ ფაილს ვამბობთ ხოლმე.



სურ 2.1: გრაფიკულ გარემოს ფაილების მენეჯერიდან დანახული დირექტორიების სტრუქტურა.

```
/usr/bin
achiko@debian:/usr/bin$ ls
[
2to3
2to3-2.7
2to3-3.5
411toppm
7z
7za
7zr
aconect
add-apt-repository
addpart
addr2line
alsabat
alsaloop
...
```

ფარდობითი გზა მიმდინარე დირექტორიით იწყება. მიმდინარე დირექტორიას სპეციალური სიმბოლო მეშვეობით, „.“ (წერტილით) აღნიშნავენ. „..“ (ორი წერტილი) კი მიმდინარე დირექტორიის მამა დირექტორიის აღსანიშნად გამოიყენება.

ვნახოთ როგორ მუშაობს. შევიცვალეთ მიმდინარე სამუშაო დირექტორია და გადავიდეთ /usr/bin-ში (თუ არ ვიმყოფებით მასში).

```
achiko@debian:~$ cd /usr/bin
achiko@debian:/usr/bin$ pwd
/usr/bin
```

თუ გვსურს მიმდინარე დირექტორიის მამა დირექტორიაში გადავიდეთ (/usr/bin-ის მამა დირექტორიაა /usr), მასში გადასასვლელად შეგვიძლია გამოვიყენოთ, როგორც ზემოთ აღვნიშნეთ, ორი ხერხი: აბსოლუტური გზა ან ფარდობითი გზა.

აბსოლუტური გზის გამოყენებით:

```
achiko@debian:~$ cd /usr
achiko@debian:/usr$ pwd
/usr
```

ფარდობითი გზის გამოყენებით (ვგულისხმობთ, რომ კვლავ /usr/bin-ში ვიმყოფებით):

```
achiko@debian:/usr/bin$ cd ..
achiko@debian:/usr$ pwd
/usr
```

ორივე ხერხის გამოყენებით ერთსა და იმავე შედეგს მივიღებთ – მიმდინარე დირექტორიიდან მამა დირექტორიაში გადავალთ. მამა დირექტორიიდან უკან დაბრუნებაც (/usr/bin-ში) ამ ორი ხერხით შეგვიძლია.

აბსოლუტური გზით:

```
achiko@debian:/usr$ cd /usr/bin
achiko@debian:/usr/bin$ pwd
/usr/bin
```

ფარდობითი გზით:

```
achiko@debian:/usr$ cd ./bin
achiko@debian:/usr/bin$ pwd
/usr/bin
```

„./“ ავტომატურად იგულისხმება ფარდობითი გზის ჩაწერისას და მისი მიწერა აუცილებელი აღარ არის. შეიძლება პირდაპირ, მარტივად ასე ჩავწეროთ:

```
achiko@debian:/usr$ cd bin
```

რამდენიმე სასარგებლო აღნიშვნა

cd - გვაბრუნებს წინა დირექტორიაში.

<code>cd</code>	გადაგვიყვანს ჩვენს პირად დირექტორიაში (~ პირად დირექტორიას
<code>cd ~</code>	აღნიშნავს).
<code>cd ~user</code>	გადაგვიყვანს user მომხმარებლის პირად დირექტორიაში.

დავუბრუნდეთ `ls` ბრძანებას და უფრო მეტი გავიგოთ მის შესახებ. გაშვებისას მას შეგვიძლია ერთდროულად რამდენიმე ოფცია ან/და არგუმენტი გადავცეთ. ოფციებისა და არგუმენტების გარეშე, მხოლოდ `ls`, როგორც ზემოთ დავრწმუნდით მხოლოდ მიმდინარე დირექტორიის ფაილებისა და ქვედირექტორიების სიას გვანახებს. მოდით, გადავცეთ არგუმენტი, მაგალითად ფაილი `GPL3.txt`. ის ბრძანებისგან გამოტოვების სიმბოლოთი (`space`) უნდა იყოს დაშორებული. ბრძანებათა ზაზში ჩაწერილი `lsGPL3.txt` (გამოტოვების გარეშე) დამოუკიდებელ ბრძანებად აღიქმება და რადგან ასეთი ბრძანება არ არსებობს, ეკრანზე შეცდომის შეტყობინება გამოვა. ამიტომაც აუცილებელი გამოტოვების გამოყენება.

```
achiko@debian:~$ ls GPL3.txt
GPL3.txt
```

```
achiko@debian:~$ lsGPL3.txt
-bash: lsGPL3.txt: command not found
```

`ls` ბრძანების არგუმენტი შეიძლება იყოს ფაილი ან დირექტორია. თუ ფაილი მივუთითეთ, ამით ვამოწმებთ არსებობს თუ არა მიმდინარე დირექტორიაში ამ დასახელების მქონე ფაილი. თუ დირექტორიაა არგუმენტად მითითებული, ამით შევამოწმებთ არსებობს თუ არა ამ დასახელების მქონე დირექტორია, და მისი არსებობის შემთხვევაში, `ls` მის შიგთავსს გამოიტანს ეკრანზე.

ამ მაგალითში `ls` ბრძანებას ერთი არგუმენტი – `GPL3.txt` გადავცეთ. მას შეიძლება გადავცეთ აგრეთვე გამოტოვების სიმბოლოთი დაშორებული რამდენიმე არგუმენტი ერთად, ფაილები ან/და დირექტორიები.

```
achiko@debian:~$ ls GPL3.txt Downloads
GPL3.txt

Downloads/:
Tux.jpg
```

ძალიან ხშირად, ლინუქსში ბრძანებებს ერთი ან რამდენიმე ოფცია გადაეცემა გაშვებისას. ეს ოფციები ბრძანებების ქცევას ცვლიან. ბრძანების ჩაწერის ზოგადი ფორმა ასეთია:

```
achiko@debian:~$ command -options arguments
```

ოფციების უმეტესობას წინ „-“ (ტირე) მიეთითება. მაგალითად `ls -a`. ასეთ შემთხვევაში ოფცია ერთ ასო-ნიშანს წარმოადგენს. მას მოკლე ოფციას ეძახიან. არსებობს ოფციის ჩაწერის გრძელი ფორმაც, რომელიც „--“-თი (ორი ტირე) მოიცემა. მაგალითად `ls --all`. გრძელი ოფცია ინგლისური სიტყვით მოიცემა. ისინი ხშირად გვხვდება GNU

პროექტის ფარგლებში შექმნილ ბრძანებებში. მოკლე ოფციის მითითებას რამდენიმე უპირატესობა აქვს: 1) კლავიატურიდან ნაკლები სიმბოლოს აკრეფა გვიწევს და 2) რამდენიმე ოფციის მითითებისას მათი ტირეთი გაერთიანება შეგვიძლია. მაგალითად `ls -a -t` და `ls -at` ერთმანეთის ექვივალენტურია. გრძელი ოფციის ჩაწერის უპირატესობა ისაა, რომ ის უფრო ინტუიტიურია. ოფციის ეს ფორმა, როგორც წესი, ინგლისურ ენაზე მის მნიშვნელობას წარმოადგენს და ადვილად მისახვედრს ხდის, თუ რას აკეთებს ბრძანება ამ ოფციით. ის მნემონიკის ერთ-ერთი ფორმასაც წარმოადგენს.

ლინუქსში ყველა ბრძანებას აქვს საკუთარი ოფციები. მათი სრული ჩამონათვალი და განმარტება შეგვიძლია `man` ბრძანებით ვნახოთ. `ls`-ის შემთხვევაში – `man ls`.

სისტემის შესწავლისას სასარგებლოა ვიცოდეთ რას შეიცავს ფაილი, რა ფორმატის ფაილთან გვაქვს საქმე. კომპიუტერში ინფორმაცია ბევრი სხვადასხვა სახით არის წარმოდგენილი. ზოგი გამოსახულებაა (სურათი), ზოგი ხმა, ზოგი ვიდეო, ზოგი ტექსტი და ა.შ. თუმცა ყველა ეს ფორმატი საბოლოო ჯამში ნულებად (**0**) და ერთიანებად (**1**) დაიყვანება, რადგან კომპიუტერს ინფორმაცია მხოლოდ ასეთი ციფრების ერთიანობით ესმის. ფაილის ყველაზე მარტივი, ცნობილი ფორმატია ASCII (American Standard Code for Information Interchange) ტექსტი. ეს არის მარტივი კოდირების სქემა, რომელიც პირველად გამოიყენეს Teletype machine-ებზე, რათა კლავიატურის სიმბოლოები გადაეყვანათ რიცხვებში. ამ ფორმატის ტექსტი წარმოადგენს ასო-ნიშნების ერთი ერთში გადაყვანას რიცხვებში. შესაბამისად, ის ძალიან კომპაქტურია. 50 ასო-ნიშანი მონაცემთა 50 ბაიტის ექვივალენტურია. მნიშვნელოვანია იმის გაგება, რომ ტექსტური ფორმატი მხოლოდ ასო-ნიშნების ციფრებში შესაბამისობას შეიცავს. ბევრ ტექსტურ რედაქტორში, მაგალითად LibreOffice, OpenOffice ან Microsoft Word, შექმნილი ფაილი, ASCII ტექსტისგან განსხვავებით, ბევრ სხვა ელემენტს შეიცავს, რომლებიც მის ფორმატსა და სტრუქტურას განსაზღვრავენ. სუფთა ASCII ტექსტი შეიცავს მხოლოდ ასო-ნიშნებს და კონტროლის კოდების ძალიან მცირე რაოდენობას, როგორიცაა ტაბულაცია (რამდენიმე გამოტოვება ერთად), ახალი ხაზზე გადასვლა (`new line`, `\n`), იმავე ხაზის დასაწყისზე გადასვლა (`carriage return`, `\r`) და ა.შ.

სისტემაში ძალიან დიდი დატვირთვა აქვს ტექსტურ ფაილს. ბევრი მნიშვნელოვანი ინფორმაცია შენახული არის ტექსტური ფორმატით. შესაბამისად, ლინუქსში მათ დასამუშავებლად ბევრი ბრძანება არსებობს.

ბრძანება `file` გვეუბნება ფაილის ტიპსა და ფორმატს.

```
achiko@debian:~$ file Downloads/Tux.jpg
Tux.jpg: JPEG image data, JFIF standard 1.01, resolution (DPI), density 72x72, segment length 16, baseline, precision 8, 612x720, frames 3
```

როგორც ჩანს, `Tux.jpg` ფაილი JPEG ფორმატის სურათია. ფაილს `Tux.jpg`-ის ნაცვლად რომ მხოლოდ `Tux` ერქვას, `file Tux` ბრძანება მაინც იგივე პასუხს დაგვიბრუნებდა, რადგან ლინუქსში მნიშვნელოვანია არა ფაილის გაფართოება, არამედ მისი შიგთავსი. გაფართოება ფაილის დასახელების ნაწილია.

```
achiko@debian:~$ file GPL3.txt
GPL3.txt: ASCII text
```

`GPL3.txt` კი ASCII ტექსტური ფაილია. ტექსტური ფაილების შიგთავსის სანახავად შეგვიძლია გამოვიყენოთ ბრძანება `cat`.

```
achiko@debian:~$ cat GPL3.txt
```

```
...
```

```
The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.
```

დიდი შიგთავსის მქონე ფაილის ნახვა თუ გვსურს, უმჯობესია `less` ბრძანება გამოვიყენოთ, რადგან `cat` მხოლოდ იმ პორციას გვაჩვენებს, რაც ეკრანის ბოლოს დაეტევა. `less`-ის შემთხვევაში კი კლავიატურაზე მიმართულების ისრებით შეგვიძლია ზევით/ქვევით გვერდ-გვერდ/ხაზ-ხაზ გადაადგილება და ფაილის შიგთავსის ამგვარად დათვალიერება. ამ პროგრამიდან გამოსასვლელად „q“ კლავიშს უნდა დავაჭიროთ.

```
achiko@debian:~$ less GPL3.txt
```

```
GNU GENERAL PUBLIC LICENSE
Version 3, 29 June 2007
```

```
Copyright (C) 2007 Free Software Foundation, Inc.
<http://fsf.org/>. Everyone is permitted to copy and
distribute verbatim copies of this license document,
but changing it is not allowed.
```

```
Preamble
```

```
The GNU General Public License is a free, copyleft
license for software and other kinds of works.
```

```
The licenses for most software and other practical
works are designed ...
```

`less` ბრძანება `more` ბრძანების გაუმჯობესებული ვარიანტია. `more`-ის შემთხვევაში ფაილის შიგთავსის დათვალიერებისას მხოლოდ დასაწყისიდან ბოლოსკენ ჩამოსვლა შეგვიძლია, უკან ასვლა კი აღარ.

Linux-ში, განსაკუთრებით კი ტექსტურ გარემოში, ფაილის გაფართოვებას დიდი დატვირთვა არ გააჩნია. ის მხოლოდ მისი დასახელების ნაწილია. მნიშვნელოვანია ფაილის შიგთავსი და არა მისი გაფართოება. ფაილის გაფართოებებს ფაილების მენეჯერის პროგრამები იყენებენ და ასე შეუძლია მომხმარებელს განსაზღვროს თუ რომელი პროგრამით გახსნას ესა თუ ის ფაილი.

თავი 3

სისტემის სტრუქტურა

ღალა, როცა უკვე ვიცით სისტემაში მოძრაობა, დირექტორიიდან დირექტორიაში გადასვლა, მოდით, უკეთ გავიცნოთ, თუ რისგან შედგება Linux სისტემა. როგორც ზემოთ აღვნიშნეთ, ლინუქსში ყველა დირექტორია თუ ფაილი ძირეულ დირექტორიაში, `/`-ში არის განთავსებული. დირექტორიების განაწილების სტრუქტურა და შიგთავსი ლინუქსსა და Unix-ის მსგავს სისტემებში ძალიან ჰგავს ერთმანეთს და ის ფაილების სისტემის იერარქიულ სტანდარტს – (FHS)-ს (Filesystem Hierarchy Standard) მიესადაგება. ეს სტანდარტი შეიმუშავეს Linux Foundation¹-ში. სადღეისოდ FHS-ის მე-3 ვერსია არსებობს, რომელიც გვკარნახობს თუ რა დირექტორიები უნდა იყოს განთავსებული `/`-ში და მათში რა უნდა ინახებოდეს. აღსანიშნავია ის ფაქტი, რომ ამ სტანდარტს მხოლოდ სარეკომენდაციო ხასიათი აქვს. ლინუქსის დისტრიბუციების უმრავლესობა ერთმანეთში თავსებადობის შესანარჩუნებლად იზიარებს ამ რეკომენდაციების უმეტეს ნაწილს. ბევრ შემთხვევაში კი Unix-ის მსგავს სისტემებში შეინიშნება კონკრეტული სისტემისთვის დამახასიათებელი გადანაცვლებები.

ვნახოთ, ჩვენს შემთხვევაში, რა გვაქვს `/`-ში:

```
achiko@debian:~$ ls /  
bin      etc      lib      media    proc     sbin     tmp      vmlinuz  
boot    home     lib64    mnt      root     srv      usr  
dev      initrd.img  lost+found  opt      run      sys      var
```

მიმოვიხილოთ აქ არსებული დირექტორიები და აღვნიშნოთ, თუ რა უნდა ინახებოდეს ყოველ მათგანში FHS-ის სტანდარტის მიხედვით:

`/bin` შეიცავს იმ ძირითად ბრძანებებს, რომლის გამოყენებაც ყველა მომხმარებელს შეუძლია. ამავედროულად, ისინი ხელმისაწვდომი უნდა იყოს სისტემის შეფერხებული მუშაობის დროს, ე.წ. ერთმომხმარებლიან რეჟიმში (single user mode). `bin` მოდის ინგლისური სიტყვა `binary`-დან, რაც ქართულად ორობითს ნიშნავს. გამშვები ფაილები, პროგრამები, ბრძანებები კი, როგორც ვახსენეთ, ორობით კოდშია ჩაწერილი.

მაგალითად, `ls` ბრძანება სწორედ `/bin`-ში არის განთავსებული.

¹Linux Foundation – არამომგებიანი ორგანიზაცია, რომლის მიზანია ლინუქსისა და თანამშრომლობითი განვითარების მხარდაჭერა.

```

achiko@debian:~$ ls /bin
...
bzdifff      loadkeys      sed
bzegrep      login         setfacl
bzexe        loginctl      setfont
bzfgrep      lowntfs-3g    setupcon
bzgrep       ls            sh
bzip2        lsblk        sh.distrib
bzip2recover lsmmod       sleep
bzless       mkdir        ...

```

/boot Boot ინგლისური სიტყვაა და ქართულად ჩატვირთვას ნიშნავს. Boot დირექტორიაში ლინუქსის ბირთვი და სისტემის ჩატვირთვისთვის სხვა აუცილებელი ფაილებია განთავსებული.

მაგალითად: vmlinuz-4.9.0-9-amd64 ფაილი /boot-ში ინახება. ლინუქსში ბირთვს vmlinuz ჰქვია.

```

achiko@debian:~$ ls /boot
config-4.9.0-3-amd64      System.map-4.9.0-3-amd64
config-4.9.0-9-amd64      System.map-4.9.0-9-amd64
grub                      vmlinuz-4.9.0-3-amd64
initrd.img-4.9.0-3-amd64  vmlinuz-4.9.0-9-amd64
initrd.img-4.9.0-9-amd64

```

/dev ყველა მოწყობილობას, რომელიც კომპიუტერთან არის მიერთებული, გააჩნია შესაბამისი ფაილი სისტემაში. სწორედ ამ ფაილების გავლით უკავშირდება მოწყობილობებს სხვადასხვა ოპერაციის გასაკეთებლად მომხმარებელიცა და ბირთვიც. ამიტომ ამბობენ, რომ „ლინუქსში ყველაფერი ფაილია“ (ფაილის სახით არის წარმოდგენილი). ეს ფრაზა ხშირად შეგვხვდებათ ლინუქსთან დაკავშირებულ წასაკითხ მასალებში. /dev დირექტორიაში კომპიუტერზე მიერთებული მოწყობილობების შესაბამისი ფაილებია განთავსებული. dev მოდის ინგლისური სიტყვა device-დან, რაც ქართულად მოწყობილობას ნიშნავს.

მაგალითად: /dev-შია განთავსებული /dev/tty1 – პირველი ვირტუალური ტერმინალი.

```

achiko@debian:~$ ls /dev
...
block      log          shm          tty25  tty49  urandom
bsg        loop-control snapshot    tty26  tty5   vcs
btrfs-control mapper      snd        tty27  tty50  vcs1

```


bus	mcelog	sr0	tty28	tty51	vcs2
...					

/etc შეიცავს ლინუქსში არსებული პროგრამების კონფიგურაციის ფაილებს. ასევე, ჩატვირთვის დროს გასაშვებ სკრიფტებს. etc ლათინური სიტყვა etcetera-დან მოდის და ქართულად იგივეა, რაც „და ასე შემდეგ“.

მაგალითად: `/etc/passwd` – სისტემაში არსებული მომხმარებლების პარამეტრების ფაილი აქ არის შენახული.

```
achiko@debian:~$ ls /etc
...
crontab                parallel
cron.weekly            passwd
cupshelpers            passwd-
dbus-1                 perl
debconf.conf           pm
debian_version         polkit-1
...
```

/home მოიცავს მომხმარებლების პირად დირექტორიებს, რომლებშიც ისინი ინახავენ თავისსავე ფაილებს. როგორც წესი, ამ დირექტორიების დასახელება შესაბამისი მომხმარებლის სახელს ემთხვევა. მასში მომხმარებელი ავტომატურად აღმოჩნდება ხოლმე სისტემაში შესვლის დროს პაროლის აკრეფის შემდეგ.

მაგალითად: `/home/achiko` მომხმარებელი achiko-ს პირადი დირექტორია.

```
achiko@debian:~$ ls /home
achiko
```

/lib შეიცავს სისტემაში არსებული ბრძანებების ფუნქციონირებისთვის საჭირო ბიბლიოთეკებს. ძირითადად მათ, რომლებიც საჭიროა `/bin`-სა და `/sbin`-ში განთავსებული პროგრამებისთვის. ეს ბიბლიოთეკები გაზიარებული ობიექტების ფორმისაა (Shared Objects) და ისინი, გასაშვები პროგრამებისგან დამოუკიდებლად არსებულ ფაილებს წარმოადგენენ (.so გაფართოებით). გაზიარებული ბიბლიოთეკების გარდა, არსებობს სტატიკური ბიბლიოთეკაც (.a გაფართოების ფაილი). ის მთლიანად ჩაშენებულია გასაშვებ პროგრამაში. გაზიარებული ბიბლიოთეკები (.so) Unix-ის მსგავს სისტემებში გამოიყენება, Windows-ში კი მათი ექვივალენტი დინამიკური კავშირის ბიბლიოთეკებია (.dll გაფართოების ფაილები).

მაგალითად: `/lib/x86_64-linux-gnu/libc-2.24.so` – C-ის სტანდარტული ბიბლიოთეკები.

```
achiko@debian:~$ ls /lib
...
libbz2.so.1.0          libnss_nis-2.24.so
libbz2.so.1.0.4       libnss_nisplus-2.24.so
libc-2.24.so          libnss_nisplus.so.2
libcap-ng.so.0        libnss_nis.so.2
libcap-ng.so.0.0.0    libntfs-3g.so.871
...
```

/media ლინუქსის ახალ სისტემებში ამ დირექტორიაში მოხსნადი მოწყობილობების (removable media) მონტაჟის წერტილებია² (mount point) განთავსებული. მაგალითად, როგორცაა CDROM, USB ფლეშ მეხსიერება და ა.შ. როგორც წესი, მათი მონტაჟი /media დირექტორიაში ავტომატურად ხდება კომპიუტერთან მიერთების შემდეგ.

მაგალითად: /media/cdrom – CD-ROM-ის მონტაჟის წერტილია.

```
achiko@debian:~$ ls /media
cdrom  cdrom0
```

/mnt ეს დირექტორია განკუთვნილია დროებით დასამონტაჟებელი მოწყობილობებისთვის. სისტემის ადმინისტრატორები ხშირად იყენებენ მას სხვადასხვა მოწყობილობის ხელით დასამონტაჟებლად.

/opt შეიცავს იმ დაინსტალირებულ პროგრამებს, რომლებიც დისტრიბუტივის შემადგენელი ნაწილი არ არის და ხშირად მესამე მხარის პროგრამულ უზრუნველყოფას წარმოადგენს.

/proc ეს არის სპეციალური დირექტორია. ის არ წარმოადგენს ნამდვილ ფაილურ სისტემას. proc ვირტუალური ფაილური სისტემაა, რომელშიც განთავსებული „ფაილები“ შეიცავს ინფორმაციას გაშვებული პროცესებისა და ბირთვის მდგომარეობის შესახებ.

მაგალითად: /proc/459 დირექტორია შეიცავს იმ პროცესის შესახებ ინფორმაციას, რომლის იდენტიფიკატორიც (PID) არის 459.
/proc/meminfo ფაილში ოპერატიული მეხსიერების მიმდინარე დეტალური ინფორმაციაა მოცემული, ხოლო /proc/cpuinfo ფაილში კი დეტალებს ნახავთ პროცესორის შესახებ.

²მონტაჟის წერტილი იმ დირექტორიას წარმოადგენს, რომელიც მოწყობილობის, მაგალითად დისკის, ფაილს შეესაბამება. კომპიუტერში არსებულ დისკებზე წვდომა მომხმარებელს მხოლოდ ამ დირექტორიების მეშვეობით შეუძლია. მაგალითად: USB ფლეშ მეხსიერებაზე ფაილების კოპირება თუ გსურთ, ეს ფაილები მისი მონტაჟის წერტილში – შესაბამის დირექტორიაში უნდა აკოპირდეთ

```

achiko@debian:~$ ls /proc
1      16      295     432     53565   8          iomem      schedstat
10     164     3       434     53567   9          ioports    self
103    168     30      436     53568   944        irq        slabinfo
105    169     31      43765   53573   96         kallsyms   softirqs
106    171     32      443     53574   99         kcore      stat
107    172     33      45      53575   acpi       keys        swaps
...

```

/root ეს დირექტორია მთავარი ადმინისტრატორის – root მომხმარებლის – პირად დირექტორიას წარმოადგენს.

/sbin /bin-ის მსგავსად ეს დირექტორიაც შეიცავს ძირითად ბრძანებებს. უმრავლეს შემთხვევაში მათ გასაშვებად სპეციალური უფლებების ქონაა საჭირო. ამ პროგრამებს ძირითადად სისტემის ადმინისტრატორები იყენებენ ხოლმე.

```

achiko@debian:~$ ls /sbin
acpi_available      getpcaps            mount.ntfs-3g
agetty              getty               mount.vmhgfs
apm_available       halt                nameif
badblocks           hdparm              ntfsclone
blkdeactivate       hwclock             ntfscp
blkdiscard          ifconfig            ntfslabel
blkid               ifdown              ntfsresize
...

```

/tmp შეიცავს დროებითი ინფორმაციის შემცველ ფაილებს. ძირითადად, მათ სხვადასხვა მომხმარებლის მიერ გაშვებული პროგრამები ავტომატურად ქმნიან. /tmp დირექტორიის შიგთავსი, ხშირ შემთხვევებში, იშლება სისტემის გადატვირთვის შემდეგ. შესაბამისად, არარეკომენდებულია მასში ფაილების შენახვა.

`/usr` ეს დირექტორია თავად წარმოადგენს დიდ იერარქიულ სტრუქტურას. მასში განთავსებულია: `bin`, `sbin`, `lib`, `local` და ა.შ.

`/usr` დირექტორია, როგორც წესი, შეიცავს სისტემის მონაცემების ყველაზე დიდ ნაწილს და შესაბამისად, ერთ-ერთი ყველაზე მნიშვნელოვან დირექტორიას წარმოადგენს. მასში შედის მომხმარებლების ბრძანებები, დოკუმენტაცია, ბიბლიოთეკები და ა.შ. Unix-ის პირველ სისტემებში მომხმარებლების პირადი დირექტორიებიც `/usr`-ში იყო განთავსებული. მისი დასახელებაც აქედან მოდის. სადღეისოდ, Unix-ის მსგავს სისტემებში, ეს ასე აღარ არის, თუმცა დასახელება მაინც დარჩა, რადგან `/usr` მოიცავს მომხმარებელისთვის განკუთვნილ უდიდეს მონაცემს.

- `/usr/bin` – ყველა მომხმარებლისთვის განკუთვნილი დამატებითი ბინარული ფაილები.
- `/usr/sbin` – ადმინისტრატორისთვის განკუთვნილი დამატებითი ბრძანებები.
- `/usr/lib` – სხვადასხვა პროგრამისთვის (ძირითადად `/usr/bin` და `/usr/sbin`-ში განთავსებული) საჭირო გაზიარებული ბიბლიოთეკები.
- `/usr/local` – მხოლოდ მოცემული კომპიუტერში არსებული დამატებითი იერარქიული სტრუქტურა, რომელიც, თავის მხრივ, შეიძლება შეიცავდეს `bin`-ს, `lib`-ს, `sbin`-ს და ა.შ.

```
achiko@debian:~$ ls /usr
bin  games  include  lib  local  sbin  share  src
achiko@debian:~$ ls /usr/bin
[                bc                c++filt
aconnect         bccmd          chacl
add-apt-repository  bdftopcf       chage
...
```

`/var` შეიცავს ისეთ ფაილებს, რომელთა შიგთავსიც ცვალებადია: ბუფერული ფაილები (spool files), ელ.ფოსტის ფაილები, ჟურნალის ფაილები (log files) და ა.შ.

მაგალითად: `/var/log/auth.log` ფაილი შეიცავს აუთენტიკაციის შესახებ ინფორმაციას – რომელი მომხმარებელი შევიდა სისტემაში, როდის, რომელი ტერმინალიდან და ა.შ.

```
achiko@debian:~$ ls /var
backups  cache  lib  local  lock  log  mail  opt  run  spool  tmp
```

ლინუქსის ბევრ დისტრიბუტივში დაინერგა ეს განაწილება, თუმცა, ხშირ შემთხვევებში, მცირე სახეცვლილებებით. კვლავ ხაზი გავუსვათ, რომ FHS-ს მხოლოდ რეკომენდაციის ხასიათი აქვს.

თავი 4

მოქმედებები ფაილებზე

მ თავში განვიხილავთ ფაილებისა და დირექტორიების კოპირების, გადატანის, წაშლის ბრძანებებს. ერთი შეხედვით, ფაილებზე ასეთი მოქმედებების შესრულება გრაფიკულ გარემოში ადვილად არის შესაძლებელი. მაშისთ მათი მონიშვნა და სხვა დირექტორიაში გადატანა ან წაშლა დიდ სირთულესთან არ არის დაკავშირებული. მაშ, რატომ ვირთულებთ საქმეს shell-ში ამ ბრძანებების სწავლით? პასუხი მარტივია: shell-ში ჩაწერილ ბრძანებებს ძალიან ფართო შესაძლებლობები აქვთ. წარმოვიდგინოთ, რომ გვსურს ბევრი jpg ფაილის კოპირება dir1 დირექტორიიდან dir2-ში, მაგრამ მხოლოდ მათი, რომლებიც მეორე დირექტორიაში არ არსებობენ ან ისეთების, რომელთა შიგთავსიც უფრო ახალია. ასეთი ამოცანის შესრულება გრაფიკულ გარემოში საკმაოდ დიდ დროს მოითხოვს, ბრძანებათა ხაზიდან კი წამიერად კეთდება მხოლოდ ერთი ბრძანებით. ასე:

```
achiko@debian:~$ cp -u dir1/*.jpg dir2/
```

4.1 მაგენერირებელი სიმბოლოები

სანამ ფაილებზე მოქმედებების მარტივ ბრძანებებს შევისწავლით, ვთქვათ, რა ხდის shell-ის ბრძანებებს ასე მძლავრს. shell-ში ხშირად გვიწევს დიდი რაოდენობით ფაილებთან მოქმედებები და მათ მოკლედ ჩაწერაში გარკვეული სიმბოლოები გვგზნდება. ასეთ სიმბოლოებს მაგენერირებელი სიმბოლოები ჰქვიათ (wildcard¹, ცნობილია globbing დასახელებითაც).

wildcard	მნიშვნელობა ფაილის დასახელებაში
*	ფიფქი (star wildcard, ცნობილია აგრეთვე როგორც asterisk) აღნიშნავს ნებისმიერ ასო-ნიშანთა ნებისმიერ წყობას.
?	კითხვის ნიშანი აღნიშნავს ერთს და მხოლოდ ერთ ნებისმიერ ასო-ნიშანს.

¹ტერმინი wild card თავდაპირველად გამოიყენებოდა კარტის თამაშის დროს და ასეთი კარტი წარმოადგენდა ე.წ. ჯოკერს, რომელსაც ყველა კარტის მაგივრობის გაწევა შეეძლო. სწორედ აქედან მოდის ეს ტერმინი.

[]	კვადრატული ფრჩხილები აღნიშნავს ნებისმიერ ასო-ნიშნებს იმ ასო-ნიშნებთან რიგიდან, რომელიც მასშია ჩაწერილი. ასო-ნიშნების რიგი შეიძლება „-“-თი (ტირე) ჩაიწეროს ან მოიცეს კლასით და ის ინტერვალის აღმნიშვნელი იქნება. ზშირად გამოყენებადი კლასებია: [:lower:] – ლათინური ანბანის პატარა ასო-ნიშნები, [:upper:] – ლათინური ანბანის დიდი ასო-ნიშნები, [:digit:] – ციფრები, [:alpha:] – ლათინური ანბანის ასო-ნიშნები, [:alnum:] – ლათინური ანბანის სიმბოლოები და ციფრები.
[!] ან [^]	კვადრატულ ფრჩხილებში მოცემული ძახილის ნიშანი ან ქუდი აღნიშნავს იმ ასო-ნიშნებთან რიგს, რომლებიც კვადრატულ ფრჩხილებში არ არის მოცემული.
{ }	ფიგურული ფრჩხილები აღნიშნავს მასში ჩაწერილი, მძიმით გამოყოფილი გამოსახულებების ალტერნატივას. გამოსახულებაში შესაძლებელია რამდენიმე ასო-ნიშანიც იყოს. ასეთი ფრჩხილებში შესაძლებელია აგრეთვე ორი წერტილის (..) გამოყენება. ამით, ასო-ნიშნების ინტერვალის მოცემა შეიძლება.

იხილეთ მაგანერირებული სიმბოლოების გამოყენების რამდენიმე მაგალითი:

ნიშუმი	აღნიშნავს მიმდინარე დირექტორიაში:
*	ყველა ფაილს.
f*.jpg	ყველა ფაილს, რომლების f -ით იწყება და მთავრდება .jpg -ით.
Documents/*	Documents დირექტორიის ყველა ფაილს.
abc	ყველა ფაილს, რომლის დასახელებაშიც abc შედის. აქ თვითონ abc ფაილიც იგულისხმება, რადგან * ასო-ნიშნებთან ნულოვან წყობასაც გულისხმობს.
??	ყველა ფაილს, რომლის დასახელებაც მხოლოდ 2 ასო-ნიშანია.
[abc]*	ყველა ფაილს, რომლებიც იწყება a -თი ან b -თი ან c -თი.
[^abc]*	ყველა ფაილს, რომლებიც იწყება არა a -თი ან არა b -თი ან არა c -თი.
[a-cst]*	ყველა ფაილს, რომლებიც იწყება ან a -დან c -ს ჩათვლით ერთ-ერთი ასო-ნიშნით, ან s -ით და ან t -ით.
[![:digit:]]*	ყველა ფაილს, რომლებიც არ იწყება ციფრით.
{abc,def}*	ყველა ფაილს, რომლებიც იწყება abc -თი ან def -თი.
{1..15}	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15.
{1..15..2}	1, 3, 5, 7, 9, 11, 13, 15.
{b..f}	b, c, d, e, f.
{f..b}	f, e, d, c, b.
{m..a..3}	m, j, g, d, a.
[A-KQ-Z]*.{png, jpg, bmp}	ყველა ფაილს, რომლებიც A -დან K -მდე ან Q -დან Z -მდე ინტერვალით იწყება და აქვს შემდეგი გაფართოება: png , jpg და ან bmp .

*.{[jJ][pP][gG] ყველა ფაილს, რომლის გაფართოებაც არის **jpg** ან **bmp**, დიდი ან
,[bB][mM][pP]} პატარა ან შერეული ასოებით მოცემული.

მაგენერირებელი სიმბოლოებით შექმნილი გამოსახულება შეიძლება ყველა იმ ბრძანებას გადავცეთ, რომელსაც არგუმენტად ფაილები გადაეცემა. მთავარი განსხვავება კვადრატულსა და ფიგურულ ფრჩხილებს შორის უკვე ვიცით – კვადრატულ ფრჩხილებში მოცემული გამოსახულებიდან სათითაო ასო-ნიშნები იგულისხმება, ფიგურულში კი მძიმით გამოყოფილი ასო-ნიშანთა ერთობლიობა, ანუ გამოსახულება. გარდა ამისა, კვადრატული ფრჩხილების გამოყენებით ხდება არსებული ფაილებიდან ამორჩევა და გამოტანა. შესაბამისად, ფაილების შესაქმნელად ეს მიდგომა არ გამოდგება. ფიგურული ფრჩხილების გამოყენებით კი ფაილების შექმნაც შესაძლებელია.

4.2 ფაილების შექმნა

გავაგრძელოთ shell-ის ბრძანებების შესწავლა და ვნახოთ, როგორ შეიძლება შევქმნათ ფაილი. ბრძანება `touch` ქმნის ცარიელ ფაილებს. მისი სინტაქსი მარტივია. მას არგუმენტად უნდა გადავცეთ ფაილ(ებ)ის სახელები.

```
achiko@debian:~$ touch file
```

ამ ბრძანებით შევქმენით ფაილი სახელად `file`. შემდეგი ბრძანება კი შექმნის 3 ფაილს – `file1`, `file2` და `file3`.

```
achiko@debian:~$ touch file1 file2 file3
```

დავრწმუნდეთ:

```
achiko@debian:~$ ls
Desktop/   file1  GPL2.txt  Pictures/  Videos/
Documents/ file2  GPL3.txt  Public/
Downloads/ file3  Music/    Templates/
```

მართლაც, ეს ფაილები შექმნილია მიმდინარე დირექტორიაში.

მოდით, ზემოთაღნიშნული კვადრატულ ფრჩხილებსა და ფიგურულ ფრჩხილებს შორის განსხვავების სანახავად მაგალითი მოვიყვანოთ.

```
achiko@debian:~$ ls file[123]
file1 file2 file3
achiko@debian:~$ ls file{1,2,3}
file1 file2 file3
```

ამ შემთხვევაში შედეგი ერთი და იგივე მივიღეთ. მოდით, ახლა გამოტანისას ისეთი ფაილიც დავუმატოთ, რომელიც ჯერ არ არის შექმნილი.

```
achiko@debian:~$ ls file[1234]
file1 file2 file3
achiko@debian:~$ ls file{1,2,3,4}
ls: cannot access 'file4': No such file or directory
file1 file2 file3
```

მეორე შემთხვევაში შეცდომის შეტყობინებაც გამოვიდა ეკრანზე, რომელიც გვამცნობს, რომ file4 დასახელების ფაილი არ გვაქვს. პირველ შემთხვევაში კი არა. ეს იმიტომ მოხდა, რომ ფიგურული ფრჩხილებში ჩაწერილი მონაცემებიდან ყველა მათგანი განიხილება. ანუ ჩვენი ჩაწერილი ბრძანება `ls file{1,2,3,4}` შემდეგი ბრძანების ექვივალენტია – `ls file1 file2 file3 file4`. კვადრატული ფრჩხილების დროს კი ერთ-ერთი მათგანის არსებობაც კმარა, რომ შეცდომა არ გამოვიდეს. კვადრატული ფრჩხილების შემთხვევაში შეცდომა მხოლოდ მაშინ გამოვიდოდა, თუ არცერთი ფაილი არ იარსებებდა.

ფაილებისგან განსხვავებით დირექტორიები `mkdir` ბრძანებით იქმნება. ასე:

```
achiko@debian:~$ mkdir directory
```

ერთდროულად რამდენიმე დირექტორიის შექმნა კი ასე ხდება:

```
achiko@debian:~$ mkdir dir1 dir2 dir3
```

უკვე შექმნილ დირექტორიაში ფაილის შექმნა ისე, რომ მასში არ გადავიდეთ, ასე ხდება:

```
achiko@debian:~$ touch directory/file
```

მოდით, გამოვიყენოთ ზემოთ ნახსენები მაგენერირებელი სიმბოლოები და მათი გამოყენებითაც შევქმნათ ფაილები და დირექტორიები. დავიწყოთ მარტივი სცენარით – ვთქვათ, გვსურს, გვქონდეს ტექსტური ფაილები შემდეგი დასახელებით: file0.txt, file1.txt ... file9.txt. ამ ამოცანის გადასაჭრელად, რადგან საქმე ფაილების შექმნას ეხება, ფიგურული ფრჩხილები უნდა გამოვიყენოთ. კვადრატული ფრჩხილები ფაილის შექმნისას ჩვეულებრივ სიმბოლოებად აღიქმევა. ფაილების დასახელების გამოტანისას კი შეიძლება მათი გამოყენება:

```
achiko@debian:~$ touch file[0-9].txt
achiko@debian:~$ ls
file[0-9].txt
achiko@debian:~$ touch file{0..9}.txt
achiko@debian:~$ ls file*
file[0-9].txt file1.txt file3.txt file5.txt file7.txt file9.txt
file0.txt     file2.txt file4.txt file6.txt file8.txt
achiko@debian:~$ ls file[0-9]*
file0.txt file2.txt file4.txt file6.txt file8.txt
file1.txt file3.txt file5.txt file7.txt file9.txt
```


ზოგადად, ფიგურულ ფრჩხილებში მიიმე აუცილებლად უნდა გამოვიყენოთ (თუ ორი წერტილით მიმდევრობა არ გვაქვს ჩაწერილი და მხოლოდ ამით არ ვკმაყოფილდებით), რადგან ის ალტერნატივების მოცემის ფორმაა. მიიმის გარეშე ეს ფრჩხილები ჩვეულებრივ სიმბოლოდ აღიქმება შელის მიერ. მოდით, კვლავ მაგალითს მივმართოთ და სასურველ ფაილებს გაფართოება როგორც პატარა, ასევე დიდი ასოებით შევუქმნათ.

```
achiko@debian:~$ touch file{0..9}.{txt,TXT}
achiko@debian:~$ ls file*
file[0-9].txt  file2.txt  file4.TXT  file7.txt  file9.TXT
file0.txt      file2.TXT  file5.txt  file7.TXT
file0.TXT      file3.txt  file5.TXT  file8.txt
file1.txt      file3.TXT  file6.txt  file8.TXT
file1.TXT      file4.txt  file6.TXT  file9.txt
achiko@debian:~$ touch file2.{txt}
achiko@debian:~$ ls file*
file[0-9].txt  file2.txt  file4.TXT  file7.txt  file9.TXT
file0.txt      file2.TXT  file5.txt  file7.TXT  file2.{txt}
file0.TXT      file3.txt  file5.TXT  file8.txt
file1.txt      file3.TXT  file6.txt  file8.TXT
file1.TXT      file4.txt  file6.TXT  file9.txt
```

ოდნავ გავართულოთ ჩვენი ამოცანა და შევქმნათ ორ ასო-ნიშნისანი ყველა ის დირექტორია, რომელის დასახელებაც თვლის თექვსმეტობითი სისტემის სიმბოლოების ყველა კომბინაციას წარმოადგენს. შემდეგ, თითოეულ ამ დირექტორიაში შევქმნათ ფაილები ჩვენთვის ცნობილი დასახელებით, როგორც გაფართოებით, ასევე გაფართოების გარეშე. მხოლოდ გაფართოება ავიღოთ დიდი და პატარა ასოებით შერეულად.

```
achiko@debian:~$ mkdir {{0..9},{A..F}}{{0..9},{A..F}}
achiko@debian:~$ ls
00 10 20 30 40 50 60 70 80 90 A0 B0 C0 D0 E0 F0
01 11 21 31 41 51 61 71 81 91 A1 B1 C1 D1 E1 F1
02 12 22 32 42 52 62 72 82 92 A2 B2 C2 D2 E2 F2
03 13 23 33 43 53 63 73 83 93 A3 B3 C3 D3 E3 F3
04 14 24 34 44 54 64 74 84 94 A4 B4 C4 D4 E4 F4
05 15 25 35 45 55 65 75 85 95 A5 B5 C5 D5 E5 F5
06 16 26 36 46 56 66 76 86 96 A6 B6 C6 D6 E6 F6
07 17 27 37 47 57 67 77 87 97 A7 B7 C7 D7 E7 F7
08 18 28 38 48 58 68 78 88 98 A8 B8 C8 D8 E8 F8
09 19 29 39 49 59 69 79 89 99 A9 B9 C9 D9 E9 F9
0A 1A 2A 3A 4A 5A 6A 7A 8A 9A AA BA CA DA EA FA
0B 1B 2B 3B 4B 5B 6B 7B 8B 9B AB BB CB DB EB FB
0C 1C 2C 3C 4C 5C 6C 7C 8C 9C AC BC CC DC EC FC
0D 1D 2D 3D 4D 5D 6D 7D 8D 9D AD BD CD DD ED FD
0E 1E 2E 3E 4E 5E 6E 7E 8E 9E AE BE CE DE EE FE
0F 1F 2F 3F 4F 5F 6F 7F 8F 9F AF BF CF DF EF FF
$ touch {{0..9},{A..F}}{{0..9},{A..F}}/file{0..9}{,t,T}{x,X}{t,T}
```

შევამოწმოთ. ვნახოთ ერთ-ერთი დირექტორიის შიგთავსი:

```
achiko@debian:~$ ls AO
```

```
file0      file1.Txt  file3.tXt  file5      file6.Txt  file8.tXt
file0.txt  file1.TXT  file3.tXT  file5.txt  file6.TXT  file8.tXT
file0.txT  file1.TXT  file3.Txt  file5.txT  file6.TXT  file8.Txt
file0.tXt  file2      file3.TxT  file5.tXt  file7      file8.TxT
file0.tXT  file2.txt  file3.TXt  file5.tXT  file7.txt  file8.TXt
file0.Txt  file2.txT  file3.TXT  file5.Txt  file7.txT  file8.TXT
file0.TxT  file2.tXt  file4      file5.TxT  file7.tXt  file9
file0.TXT  file2.tXT  file4.txt  file5.TXT  file7.tXT  file9.txt
file0.TXT  file2.Txt  file4.txT  file5.TXT  file7.Txt  file9.txT
file1      file2.Txt  file4.tXt  file6      file7.Txt  file9.tXt
file1.txt  file2.TXT  file4.tXT  file6.txt  file7.TXT  file9.tXT
file1.txT  file2.TXT  file4.Txt  file6.txT  file7.TXT  file9.Txt
file1.tXt  file3      file4.TxT  file6.tXt  file8      file9.TxT
file1.tXT  file3.txt  file4.TXt  file6.tXT  file8.txt  file9.TXT
file1.Txt  file3.txT  file4.TXT  file6.Txt  file8.txT  file9.TXT
```

ახლა შექმნილი ფაილებიდან შევეცადოთ გამოვიტანოთ მხოლოდ ისინი, რომლებიც განთავსებულია დირექტორიებში, რომელთა დასახელებაც არ შეიცავს ციფრს, B, E და F ასოებს და თავად ფაილის გაფართოება კი მხოლოდ დიდ X შეიცავს, პატარას არა.

```
achiko@debian:~$ ls [^0-9BEF][!0-9BEF]/*.*X?
```

```
...
```

```
AC/file5.TXT  CA/file3.TXT  CD/file1.TXT  DA/file9.TXT  DD/file7.TXT
AC/file6.tXt  CA/file4.tXt  CD/file2.tXt  DC/file0.tXt  DD/file8.tXt
AC/file6.tXt  CA/file4.tXT  CD/file2.tXT  DC/file0.tXT  DD/file8.tXT
AC/file6.Txt  CA/file4.TXT  CD/file2.TXT  DC/file0.TXT  DD/file8.TXT
AC/file7.tXt  CA/file5.tXt  CD/file3.tXt  DC/file1.tXt  DD/file9.tXt
AC/file7.tXT  CA/file5.tXT  CD/file3.tXT  DC/file1.tXT  DD/file9.tXT
AC/file7.Txt  CA/file5.TXT  CD/file3.TXT  DC/file1.TXT  DD/file9.TXT
AC/file7.TXT  CA/file5.TXT  CD/file3.TXT  DC/file1.TXT  DD/file9.TXT
```

ზოგადად, ფაილებისა და დირექტორიების შექმნისას, ერთი არგუმენტი მეორისგან, როგორც ზემოთ აღვნიშნეთ, გამოტოვებით არის გამოყოფილი. მაშ, როგორ შეიძლება შევექმნათ ისეთი ფაილი, რომლის დასახელებაშიც დაშორება შედის? მაგალითად, ფაილი სახელად name surname.

```
achiko@debian:~$ touch name surname
```

ამ ბრძანებით shell-ში, როგორც უკვე ვიცით, შეიქმნება ორი დამოუკიდებელი ფაილი: name და surname. ჩვენი კი ერთი ფაილის შექმნა გვსურს სახელად name surname.

ასეთ შემთხვევებში, როდესაც ფაილის დასახელებები სპეციალურ ასო-ნიშნებს (ჩვენს შემთხვევაში ეს გამოტოვების ნიშანია) შეიცავენ, რომელსაც სიმბოლური გამოსახულების მაგივრად სხვა დატვირთვა აქვთ, საჭიროა დავეუკარგოთ თავიანთი მნიშვნელობა. ასე shell მათ აღიქვამს ჩვეულებრივ, მორიგ ასო-ნიშნად და მის ქვეშ აღარ იგულისხმება სხვა რამ.

ასო-ნიშნისთვის მნიშვნელობის დასაკარგად shell-ში 3 ხერხი არსებობს:

1. ” (ბრჭყალი) – ასო-ნიშანი ან ასო-ნიშანთა ერთობლიობა, რომელთა მნიშვნელობების დაკარგვაც გვურს, უნდა მოვათავსოთ ბრჭყალებში. ბრჭყალით ვერ ვუკარგავთ მნიშვნელობას ოთხ სპეციალურ ასო ნიშანს. ესენია:

\$ (დოლარის სიმბოლო)

\ (უკან გადახრილი წილადის ხაზი, ანუ ბექსლექში)

” (თავად ბრჭყალი)

` (მკვეთრი მახვილი²)

2. ’ (აპოსტროფი) – ასო-ნიშანი ან ასო-ნიშანთა ერთობლიობა, რომელთა მნიშვნელობების დაკარგვაც გვსურს, უნდა მოვათავსოთ აპოსტროფებს შორის, როგორც ბრჭყალის გამოყენების შემთხვევაში. ამ დროს ყველა ასო-ნიშანს ეკარგება მნიშვნელობა, გარდა თავად აპოსტროფისა.
3. \ (ბექსლექში (backslash), უკან გადახრილი წილადის ხაზი) – ასო-ნიშანს, რომლის მნიშვნელობის დაკარგვაც გვსურს, წინ უნდა მივუწეროთ ბექსლექში. ეს მეთოდი უნივერსალურია და ის ყველა ასო-ნიშანს უკარგავს თავის მნიშვნელობას.

ახლა უკვე შეგვიძლია შევქმნათ ფაილი სახელად name surname:

```
achiko@debian:~$ touch name" surname
```

ამ მაგალითში ფაილის დასახელებაში ყველა ასო-ნიშანი, გამოტოვების გარდა, ჩვეულებრივი სიმბოლოა, ამიტომ არაფერი დაშავდება, თუ მთლიანად მოვათავსებთ ბრჭყალებში ამ გამოსახულებას. ასეთი ჩანაწერით იმავე შედეგს მივიღებთ:

```
achiko@debian:~$ touch "name surname"
```

აპოსტროფის გამოყენებით შემდეგნაირი ჩანაწერი გვექნება:

```
achiko@debian:~$ touch 'name surname'
```

ბექსლექსით კი ასეთი:

```
achiko@debian:~$ touch name\ surname
```

²მკვეთრი მახვილი (grave accent) qwerty კლავიატურაზე ტაბულაციის კლავიშის მალაა განთავსებული.

ყურადღება

ფაილის დასახელებები Linux-ში, ისევე როგორც Unix-ში, რეგისტრზეა დამოკიდებული, ანუ დიდი ასოებითა და პატარა ასოებით დასახელებული ფაილები სხვადასხვაა. ასე, მაგალითად, File1 და file1 ორი განსხვავებული ფაილია.

Linux-ში შეგვიძლია ფაილებისთვის გრძელი დასახელება ავირჩიოთ. ფაილის დასახელებაში ნებისმიერი სიმბოლოს გამოყენებაა შესაძლებელი გარდა / სიმბოლოსი.

ფაილი, რომლის სახელიც წერტილით (.) იწყება ითვლება დაფარულად. დაფარული იმ გაგებით, რომ `ls` ბრძანებით ის ეკრანზე არ გამოჩნდება. ამისთვის `ls` ბრძანება `a` ოფციით უნდა გავუშვათ, ასე: `ls -a`. მომხმარებლის ანგარიშის შექმნისას პირად დირექტორიაში შეგვხდებათ რამდენიმე დაფარული ფაილი და დირექტორია. ისინი საჭიროა თქვენი სამუშაო გარემოს დასაკონფიგურირებლად. დეტალურად მათ მოგვიანებით დავუბრუნდებით.

4.3 ფაილების ასლების შექმნა, გადატანა, წაშლა

ჩვენ უკვე ვიცით ფაილებისა და დირექტორიების შექმნა. ახლა ვნახოთ, როგორ უნდა მათი კოპირება, გადატანა და წაშლა.

`cp` ბრძანება ქმნის ფაილების ასლს. თუ გვსურს ფაილები ვაკოპიროთ მიმდინარე დირექტორიიდან სხვა დირექტორიაში, უნდა გავუშვათ ეს ბრძანება:

```
achiko@debian:~$ cp file1 file2 ... directory
```

თუ დირექტორიის კოპირებაც გვსურს, მაშინ `-r` ოფცია უნდა მივუთითოთ:

```
achiko@debian:~$ cp -r file1 file2 ... dir1 dir2 ... directory
```

ერთი ფაილის კოპირება მეორე ფაილში კი ასე ხდება:

```
achiko@debian:~$ cp file1 file2
```

`cp` ბრძანებას შეიძლება შემდეგი ოფციები გადაეცეს:

ოფციები	მნიშვნელობა
<code>-i</code> , <code>--interactive</code>	თუ ფაილის კოპირებისას არსებულ ფაილზე გადაწერა უნდა მოხდეს, კოპირების პროცესი ამ ოფციის მითითებით ინტერაქტიული ხდება. ანუ მომხმარებლის მხრიდან დათანხმებაა საჭირო კოპირებისთვის. თუ ეს ოფცია არ არის მითითებული, <code>cp</code> ბრძანება უთქმელად გადააწერს ფაილს სხვა ფაილის შიგთავსს და ფაილში ძველი ინფორმაცია დაიკარგება.

-u, --update	ერთი დირექტორიდან მეორეში ფაილების გადაწერისას მხოლოდ იმ ფაილებს აკოპირებს, რომლებიც მეორე დირექტორიაში არ არსებობს. ხოლო თუ არსებობს, მათი განახლება მხოლოდ იმ შემთხვევაში მოხდება, თუ პირველ დირექტორიაში დასაკოპირებელი ფაილი უფრო ახალია, ვიდრე მეორე დირექტორიაში არსებული ფაილი.
-a, --archive	აკოპირებს ფაილებსა და დირექტორიებს და თან უნარჩუნებს მათ ატრიბუტებს (წვდომის უფლებებზე მოგვიანებით გვექნება საუბარი).
-v, --verbose	კოპირების პროცესის დეტალები ჩანს ეკრანზე.

სხვა ოფციების სანახავად იხილეთ `cp` ბრძანების სახელმძღვანელო გვერდი: `man cp`.

`mv` ბრძანებით, `cp`-სგან განსხვავებით, არ ვაკოპირებთ, არამედ ერთი ადგილიდან მეორეში გადაგვაცქვს ფაილები და დირექტორიები. მოვიყვანოთ მაგალითი:

```
achiko@debian:~$ mv file1 file2 ... dir1 dir2 ... directory
```

ამ ბრძანებით კი შესაძლებელია ფაილის სახელის გადარქმევა:

```
achiko@debian:~$ mv file1 file2
```

`-i`, `-u` და ისინი `-v` ოფციები `mv` ბრძანებასაც შეგვიძლია გადავცეთ და მათი მნიშვნელობა ამ ბრძანებისთვისაც იგივე იქნება, რაც `cp`-ს შემთხვევაში.

`rm` ბრძანებით შეგვიძლია წავშალოთ ფაილები და დირექტორიები. ასე:

```
achiko@debian:~$ rm file1
```

ეს ბრძანება შლის `file1`-ს. იხილეთ `rm` ბრძანების ხშირად გამოყენებადი ოფციები.

ოფციები	მნიშვნელობა
-i	წაშლამდე თანხმობას ელოდება მომხმარებლისგან.
-r, -R, --recursive	შლის დირექტორიებსაც.
-f, --force	თანხმობის გარეშე შლის. (უგულებელყოფს <code>-i</code> ოფციას).

ამ ბრძანებასთან სიფრთხილე გვმართებს. ლინუქსში, ისევ როგორც `unix`-ის მსგავს სხვა სისტემებში, ტერმინალიდან წაშლილი ფაილის აღდგენა ძნელად თუ მოხერხდება, რადგან იგულისხმება, რომ თუ ფაილები წაშალე ე.ი. ეს შეგნებული ნაბიჯი იყო.

```
achiko@debian:~$ rm -r file1 file2 ... dir1 dir2 ...
```

მოდიტ, წავშალოთ ზემოთ შექმნილი ორ ასო-ნიშნის დირექტორიები და ფაილები:

```
achiko@debian:~$ rm -rf file* ??
```

ყურადღება

არ აგერიოთ ეს ბრძანებები ერთმანეთში: `rm -rf ./*` და `rm -rf /*`. პირველი ბრძანებით მიმდინარე დირექტორიაში წაიშლება ყველაფერი, ხოლო მეორე ბრძანება კი სისტემის ძირეული დირექტორიიდან დაწყებული ყველა ფაილს წაშლის!

თავი 5

გადამისამართება

8 ადამისამართების შესაძლებლობა ლინუქსში ამოცანების გადაწყვეტას ძალიან ამარტივებს. როდესაც shell-ში ფაილების ბრძანებებით მართვა გვსურს, შეტან/გამოტანის (Input/Output) ნაკადების ცოდნა მნიშვნელოვნად ამაღლებს პროდუქტიულობას.

ლინუქსში ბევრი ბრძანება ეკრანზე გვაჩვენებს შედეგს. უფრო ტექნიკურად რომ ავხსნათ, ბრძანების ეს შედეგი მიმართულია სტანდარტული გამოსასვლელისკენ (standart output – stdout). თუ ბრძანება წარუმატებლად დასრულდა, ამ შემთხვევაში შედეგი მიმართული იქნება შეცდომების სტანდარტული გამოსასვლელისკენ (standart error – stderr). ორივე შემთხვევაში, სტანდარტული გამოსასვლელიცა და შეცდომების გამოსასვლელიც შეესაბამება ეკრანს, რადგან აქ ვხედავთ ყოველგვარ შედეგს.

Unix-ში, როგორც აღვნიშნეთ, ყველაფერი ფაილია და ბრძანებები, მაგალითად `ls`, თავის შედეგს, ანუ მიმდინარე დირექტორიაში არსებულ ფაილების სიას, აგზავნის სპეციალური ფაილისკენ, რომელსაც სტანდარტულ გამოსასვლელს ვუწოდებთ, ხოლო შეცდომის შედეგს კი სხვა სპეციალური ფაილისკენ, რომელიც შეცდომების გამოსასვლელია. ეს სპეციალური ფაილები, FHS-ით მიხედვით, `/dev` დირექტორიაშია განთავსებული და ისინი, ნაგულისხმევი მნიშვნელობით, ეკრანთან არიან მიბმულნი.

ამას გარდა, ბევრ ბრძანებას სჭირდება არგუმენტის გადაცემა. მომხმარებელს ის შეეავს სტანდარტული შესასვლელიდან (standart input – stdin). ნაგულისხმევი მნიშვნელობით, სტანდარტული შესასვლელიც ეკრანია. მომხმარებელს ეს ინფორმაცია ზომ კვლავ ეკრანზე შეჰყავს, როგორც წესი, კლავიატურის გამოყენებით.

არ აგერიოთ ეს ტერმინები ერთმანეთში!

დისპლეი (Display) არის ფიზიკური მოწყობილობა, სადაც მომხმარებლისთვის ტექსტი ან/და გრაფიკული გამოსახულება გამოდის.

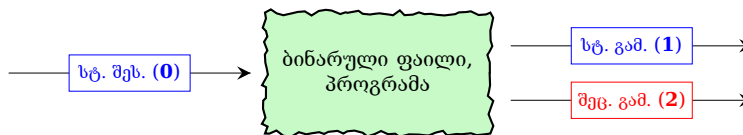
მონიტორი (Monitor) კომპიუტერის დისპლეია და წარმოადგენს პერსონალური კომპიუტერის კომპლექტის ერთ-ერთ დამოუკიდებელ ნაწილს, რომელიც შეერთებულია სისტემურ ბლოკთან.

ეკრანი (Screen) არის მონიტორისა და კომპიუტერის დისპლეის წინა ნაწილია, რომელზეც გამოსახულებას ვიზუალურად ვხედავთ. ის, ძირითადად, მინისა და თხევად-კრისტალური მასალისგან შედგება.

ტელეფონს, ტელევიზორს, პერსონალურ კომპიუტერს – სამივეს აქვს ეკრანი, თუმცა მათგან მხოლოდ პერსონალურ კომპიუტერს აქვს მონიტორი.

ლინუქსის გარემოში შეტან გამოტანა ხორციელდება სამი ნაკადით და ეს ნაკადები შემდეგნაირადაა დანომრილი (სურ. 5.1):

- სტანდარტული შესასვლელს (stdin) შეესაბამება ნომერი – **0**
- სტანდარტული გამოსასვლელს (stdout) შეესაბამება ნომერი – **1**
- შეცდომების გამოსასვლელს (stderr) შეესაბამება ნომერი – **2**



სურ 5.1: ბრძანების შესრულების ორგანიზაცია

0, **1**-სა და **2**-ს ფაილის დესკრიფტორები (file descriptor, მოკლედ fd) ეწოდება და, როგორც უკვე აღვნიშნეთ, ნაგულისხმევი მნიშვნელობით, ეკრანისკენაა მიმართული. ლინუქსში ჩვენ გვაქვს იმის საშუალება, რომ შეტან/გამოტანის მიმართულება შევცვალოთ.

5.1 სტანდარტული გამოსასვლელი

სტანდარტული გამოსასვლელში არსებულ ნაკადს მიმართულება რომ შევუცვალოთ და ეკრანის ნაცვლად სხვა ფაილისკენ გადავამისამართოთ, ვიყენებთ გადამისამართების ოპერატორს **>** (მეტობის ნიშანი). ხშირად სასარგებლოა, გამოსული ინფორმაცია ფაილში დავიმასხოვროთ მისი მომავალში გამოყენების მიზნით, რადგან ეკრანზე გამოსული ინფორმაცია მუდმივად არ ინახება კომპიუტერში. სწორედ ასეთ დროსაა სასარგებლო სტანდარტული გამოსასვლელის გადამისამართება.

```
achiko@debian:~$ ls 1>file
```

„1>“ ჩანაწერი ხაზგასმით აღნიშნავს, რომ სტანდარტული გამოსასვლელის (stdout) გადამისამართებას ვასრულებთ, რადგან ფაილის დესკრიფტორი – „1“ stdout -ს შეესაბამება.

თუმცა, მხოლოდ „>“ ოპერატორის ჩაწერა, „1“-ის გარეშეც ავტომატურად გულისხმობს სტანდარტული გამოსასვლელის გადამისამართებას. შესაბამისად, ეს ბრძანება შემდეგნაირადც შეიძლება დავწეროთ:

```
achiko@debian:~$ ls >file
```

„>“ ოპერატორის გამოყენებით file-ის შიგთავსს ბრძანების stdout გადაეწერება და მასში დამახსოვრებული ინფორმაცია დაიკარგება. თუ გვსურს, რომ stdout არ გადაეწეროს ფაილს, არამედ ამ უკანასკნელის შიგთავსის ბოლოს დავმატოვო, მაშინ „>>“ (ორი მეტობის ნიშანი) ოპერატორი უნდა გამოვიყენოთ, ასე:

```
achiko@debian:~$ ls 1>>file
```

ამ შემთხვევაში file-ის შიგთავსი შენარჩუნდება და მას ბოლოში მიემატება ls-ის სტანდარტულ გამოსასვლელზე გამოსული ინფორმაცია. მისი ექვივალენტური, გამარტივებული ჩანაწერი შემდეგნაირია:

```
achiko@debian:~$ ls >>file
```

5.2 შეცდომების გამოსასვლელი

ზშირად გაშვებული ბრძანება წარმატებით არ სრულდება. მიზეზი შეიძლება ბევრი იყოს: შეიძლება ბრძანებაზე გადაცემული არგუმენტი არ არსებობდეს, ან ბრძანება სინტაქსურად არასწორად იყოს ჩაწერილი, თუნდაც კლავიატურიდან შეცდომით აკრეფის ან სხვა მიზეზის გამო.

ასეთ დროს ეკრანზე გამოდის შეცდომის მესიჯი და შეცდომის ეს ტექსტი ეკრანზე ბრძანების შეცდომების გამოსასვლელიდან ხვდება.

```
achiko@debian:~$ lss
-bash: lss: command not found
```

თუ გვსურს ეს შეტყობინება ეკრანზე აღარ დაიწეროს და ის გადამისამართდეს სხვა ფაილისკენ, მაშინ გადამისამართების ოპერატორი უნდა გამოვიყენოთ, ასე:

```
achiko@debian:~$ lss 2>file_error
```

„>>“ (ორი მეტობის ნიშანი) გამოყენების შემთხვევაში შეცდომის ტექსტი file_error-ს ბოლოში მიემატება.

ზშირ შემთხვევაში, სასურველია, ყველა გამოსასვლელი ერთ ფაილში გადამისამართდეს. ტრადიციული მეთოდით, bash-ის ძველ ვერსიებში, ამისთვის შემდეგი ჩანაწერი გამოიყენება:

```
achiko@debian:~$ command >file 2>&1
```

ამ ბრძანებით ორ გადამისამართებას ვაკეთებთ. ჯერ სტანდარტული გამოსასვლელი

გადაგვაქვს file-სკენ და შემდეგ fd2-ს (შეცდომების გამოსასვლელს) ვამისამართებთ fd1-სკენ, ანუ სტანდარტული გამოსასვლელისკენ 2>&1 ჩანაწერით. სტანდარტული გამოსასვლელი კი file-ში მიდის.

bash-ის ახალ ვერსიაში უფრო გამარტივებული ხერხი არსებობს. მასში ძველი ვარიანტიც მუშაობს.

```
achiko@debian:~$ command &>file
```

ჩვენ შეგვიძლია, აგრეთვე, stout დან stderr მივამატოთ ფაილს ბოლოში. ასე:

```
achiko@debian:~$ command &>>file
```

ნათქვამია, სიტყვა ვერცხლია, სიჩუმე კი ოქრო და იმ შემთხვევებში, როდესაც გვსურს ეკრანზე საერთოდ არაფერი გამოვიდეს, გამოსასვლელი ტექსტი ე.წ. „ნაგვის ყუთში“ შეგვიძლია გადავამისამართოთ. shell-ში ამ ე.წ. „ნაგვის ყუთს“ /dev/null ფაილი შეესაბამება. რაც იქ მოხვდება ველარასოდეს აღდგება! ძირითადად აქ შეცდომების მესიჯებს ამისამართებენ.

```
achiko@debian:~$ command 2>/dev/null
```

5.3 სტანდარტული შესასვლელი

აქამდე ჯერ არ შეგვხვედრია ბრძანება, რომელშიც მისი სტანდარტული შესასვლელი გამოგვეყენებინოს. მოდით, ერთ-ერთი მათგანი განვიხილოთ: cat ბრძანება კითხულობს ერთ ან რამდენიმე ფაილს და მათი შიგთავსი გამოაქვს სტანდარტულ გამოსასვლელზე, ასე:

```
achiko@debian:~$ cat [file1 file2 ...]
```

cat ბრძანებით ვნახულობთ რა წერია ფაილში. რამდენიმე ფაილის შემთხვევაში, ეს ბრძანება მათ შიგთავსებს მიაჯრის ერთანეთს და ისე გვანახებს ეკრანზე. ანუ cat შიგთავსების მიჯრას, კონკატენაციას აკეთებს.

რა მოხდება თუ cat ბრძანებას არგუმენტს არ მივუწერთ?

```
achiko@debian:~$ cat
```

არც არაფერი. ერთი შეხედვით, იქმნება შთაბეჭდილება, რომ ბრძანება „დაეკიდა“, აღარაფერს ასრულებს, თუმცა ეს ასე არ არის. ამ დროს ის კითხულობს მის სტანდარტულ შესასვლელს და რადგან სტანდარტული შესასვლელი, ნაგულისხმევი მნიშვნელობით, კლავიატურასთანაა მიბმული, cat ელოდება ჩვენგან კლავიატურაზე რაიმეს აკრეფას და შეეყვანას. მოდით ვცადოთ, ავკრიფოთ ტექსტი და ჯერ **[Enter]** -ს, შემდეგ კი **[Ctrl-d]** -ს დავაჭიროთ (**[Ctrl]** და **[d]** კლავიშებს ერთად). კლავიშების ეს კომბინაცია cat ბრძანებას დასასრულისკენ, end of file-სკენ (EOF) მიუთითებს და ეუბნება, რომ მის სტანდარტულ შესასვლელზე მეტი ტექსტის შეყვანა აღარ გვსურს.

შედეგი ის იქნება, რომ cat დააკოპირებს stdin-ს stdout-ში, და, შესაბამისად, ჩვენი შეყვანილი ტექსტი ხელმეორედ დაიწერება ეკრანზე.

```
achiko@debian:~$ cat
text
text
```

cat ბრძანების ასეთი ქცევა საშუალებას გვაძლევს, შევქმნათ მარტივი ტექსტური ფაილები, ასე:

```
achiko@debian:~$ cat >file.txt
ხაზი 1 [enter]
ხაზი 1
ხაზი 2 [enter]
ხაზი 2
```

არ დაგავიწყდეთ **Ctrl-d** ბოლოს. ვნახოთ, რა გამოვიდა და რა ჩაიწერა file.txt-ში.

```
achiko@debian:~$ cat file.txt
ხაზი 1
ხაზი 2
```

ბუნებრივია! ფაილში დაგვხვდა ის, რაც ჩვენ თვითონ შევიტანეთ.

ახლა უკვე ვიცით, რომ cat ბრძანება არგუმენტის გარდა, კითხულობს სტანდარტულ შესასვლელსაც.

„<“ ოპერატორით (ნაკლებობის ნიშანი) ჩვენ შეგვიძლია ბრძანების სტანდარტული შესასვლელზე, კლავიატურიდან შეყვანილი მონაცემების ნაცვლად, მონაცემები ფაილიდან შევიყვანოთ.

```
achiko@debian:~$ cat <file.txt
ხაზი 1
ხაზი 2
```

შედეგი იგივე იქნება. ეს ბრძანებაც იმას გვანახებს, რაც file.txt-ში წერია. პრაქტიკაში ასეთ ჩანაწერს დიდი გამოყენება არ აქვს, თუმცა ეს მაგალითი კარგი დემონსტრირებაა იმისა, თუ როგორ შეგვიძლია სტანდარტული შესასვლელი ფაილიდან წავიკითხოთ. სხვა ბრძანებებზე მისი გამოყენება გაცილებით სასარგებლოა. მათ მოგვიანებით დავუბრუნდებით.

თავი 6

ბრძანებების გადაბმა

ლინუქსში, ბრძანებათა ხაზში მუშაობისას, შესაძლებელია რამდენიმე ბრძანების ერთ ხაზში ჩაწერა. ბრძანებების შესასრულებლად მათი დაჯგუფების ასეთი ფორმა უფრო კომპაქტურია, ვიდრე ტრადიციული მეთოდით მათი რამდენიმე ხაზში განაწილება. აგრეთვე, ბრძანებების ერთ ხაზში ჩაწერას ის უპირატესობა აქვს, რომ ცალკეული ბრძანების შესრულებისთვის საჭირო დროს ლოდინში არ დაგვარგავთ, არამედ მელი მათ ავტომატურად გაუშვებს.

6.1 ოპერატორები

ბრძანებების შეერთებისთვის, ერთ ხაზში ჩაწერისა და გაშვებისთვის, ლინუქსში რამდენიმე ოპერატორი გამოიყენება. ესენია:

- ; წერტილ-მძიმე
- && ლოგიკური და
- || ლოგიკური ან

წერტილ-მძიმით გადაბმული ბრძანებები გამართული სინტაქსით ასე გამოიყურება:

```
achiko@debian:~$ cmd1; comd2; cmd3;
```

აქ `cmd` ჰიპოთეტური ბრძანებაა. მასში ნებისმიერი ჩვენთვის ნაცნობი ბრძანება შეგვიძლია ვიგულისხმოთ. ეს ჩანაწერი ნიშნავს იმას, რომ ჯერ პირველი ბრძანება (`cmd1`) ეშვება და სრულდება, შემდეგ მეორე ბრძანება (`cmd2`) ეშვება და სრულდება და შემდეგ მესამე (`cmd3`).

დავუშვათ, გვსურს შევქმნათ დირექტორია, გადავიდეთ მასში და იქ შევქმნათ ფაილი. ამის გასახორციელებლად ასეთი ჩანაწერი უნდა გავაკეთოთ.

```
achiko@debian:~$ mkdir dir; cd dir; touch file
```

ამ მაგალითში ჩვენ დარწმუნებულები ვართ, რომ თითოეული ბრძანება წარმატებით შესრულდება, თუმცა ეს „ნაკლებად სწორი“ მიდგომაა ლინუქსში.

ეს სამი თანმიმდევრული ბრძანება, ფორმალურად, ერთმანეთისგან დამოუკიდებელია, თუმცა ლოკურურად – არა. მაგალითად, თუ დირექტორია `dir` გარკვეული

მიზეზების გამო არ შეიქმნა, მასში ვერ გადავალთ `cd` ბრძანებით, და შესაბამისად, ვერც ფაილის შექმნას შევძლებთ.

ამ შემთხვევაში, სასურველი იქნება, რომ ყოველი შემდეგი ბრძანება მხოლოდ მას შემდეგ გაეშვას, თუ წინა წარმატებით დასრულდება. ამაში კი დაგვეხმარება პირობითი ოპერატორი `&&` და ბრძანებების ერთ ხაზში ჩაწერაც ასე მოხდება:

```
achiko@debian:~$ mkdir dir && cd dir && touch file
```

აქ, თუ პირველი ბრძანება (დირექტორიის შექმნა) წარმატებით არ შესრულდა, `shell` შემდეგი ბრძანების (დირექტორიაში გადასვლა) შესრულებას აღარც ეცდება. ამ ბრძანების შესრულება, მხოლოდ მაშინ დაიწყება თუ პირველი წარმატებით დასრულდება. შესაბამისად, მესამე ბრძანება (ფაილის შექმნა) მხოლოდ მაშინ გაეშვება, თუ მეორეც წარმატებით დასრულდება.

შეიძლება საინტერესო აღმოჩნდეს ბრძანების შესრულება იმ შემთხვევაში, თუ წინა ბრძანება წარუმატებლად შესრულდა. ასეთი ქმედება `||` პირობითი ოპერატორით შეიძლება განხორციელდეს.

ასე, მაგალითად:

```
achiko@debian:~$ ls file || touch file
```

ამ ბრძანებით მხოლოდ იმ შემთხვევაში ვქმნით ფაილს, თუ ფაილი `file` არ არსებობს. ხოლო თუ ეს ფაილი არსებობს, ანუ `ls file` წარმატებით დასრულდა, მაშინ შემდეგი ბრძანება აღარ შესრულდება.

ჩვენი წინა მაგალითი ამ ოპერატორით ასეც შეგვიძლია წარმოვიდგინოთ:

```
achiko@debian:~$ ls dir || mkdir dir && touch dir/file
ls: cannot access 'dir': No such file or directory
```

აქ, თუ `ls dir` ბრძანება წარუმატებლად არ დასრულდა, (მაგალითად, თუ ეს დირექტორია რომ არ არსებობს), მხოლოდ ამ შემთხვევაში შეიქმნება `dir` დირექტორია და ამის შემდეგ, თუ წარმატებით მოხდება მისი შექმნაც, მაშინ მასში შეიქმნება ფაილი `file`. ეკრანზე შეცდომის შეტყობინების გამოტანაც ბუნებრივია, რადგან ის პირველ ბრძანებას ეხება (ის ხომ წარმატებით არ შესრულდა). შეტყობინებაც სწორედ იმას გვამცნობს, რომ `dir` დასახელების მქონე ფაილი ან დირექტორია არ არსებობს.

ეს შეტყობინება ეკრანზე რომ არ გამოჩნდეს, შეგვიძლია, შესაბამისი ბრძანების შეცდომების გამოსასვლელი უბრალოდ ნაგვის ურნაში გადავამისამართოთ, ასე:

```
achiko@debian:~$ ls dir 2>/dev/null || mkdir dir && touch dir/file
```

თუ `ls dir` წარმატებით დასრულდა, ანუ თუ ეს დირექტორია უკვე არსებობს, მაშინ, პირდაპირ მესამე ბრძანება შესრულდება, მეორე ბრძანება (`mkdir dir`) კი აღარ. ჩვენს ბრძანებას თუ მეორედ გავუშვებთ, სწორედ ეს სცენარი განხორციელდება (პირველ ჯერზე გაშვებისას, ხომ უკვე შეიქმნა `dir` დირექტორია).

```
achiko@debian:~$ ls dir || mkdir dir && touch dir/file
```

ახლა კი აღარაფერი შეიქმნება, რადგან `dir` დირექტორიაში `file` ფაილი უკვე არსებობს. არც გადაწერება, რადგან `touch` ბრძანება უკვე შექმნილი ფაილის შიგთავსს არ ეხება.

6.2 მილი

როგორც ვნახეთ, ზემოთ მოყვანილი ოპერატორების წყალობით, შეგვიძლია, სხვადასხვა ბრძანების გაშვების თანმიმდევრობა განვსაზღვროთ გარკვეული პირობით ან პირობის გარეშე.

მათგან განსხვავებით, ლინუქსში გვაქვს ბრძანებების ერთმანეთთან გადაბმის სხვა საშუალებაც. მაგალითად, შეგვიძლია ერთი ბრძანების შედეგი დასამუშავებლად სხვა ბრძანებას გადავცეთ.

ბრძანების უნარი გადასცეს მისი სტანდარტული გამოსასვლელი სხვა ბრძანებას სტანდარტულ შესასვლელზე, `shell`-ში ხორციელდება მილით (`pipeline`, მოკლედ `pipe`). მისი შესაბამისი ოპერატორია `|` (ვერტიკალური ხაზი).

```
achiko@debian:~$ cmd1 | cmd2
```

ამ ჩანაწერით `cmd1`-ის სტანდარტული გამოსასვლელი შევა `cmd2`-ის სტანდარტულ შესასვლელზე და შესრულდება.

ჩვენთვის უკვე ნაცნობ `less` ბრძანებას შეუძლია სტანდარტულ შესასვლელზე მონაცემების მიღება. შესაბამისად, ნებისმიერი ბრძანების გამოსასვლელი გრძელი ტექსტი შეგვიძლია გვერდ-გვერდად დავათვალიეროთ `pipe`-ის საშუალებით ასე:

```
achiko@debian:~$ ls /bin /usr/bin | less
```

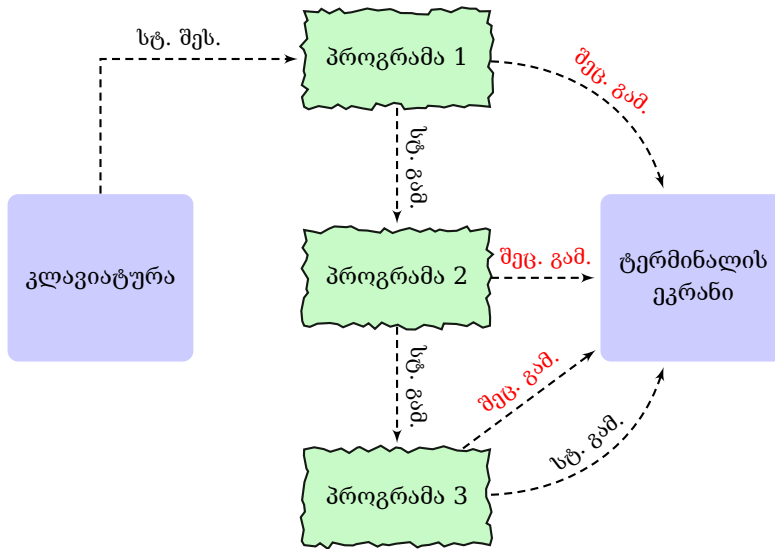
`/bin` და `/usr/bin` დირექტორიების დათვალიერება ახლა გაცილებით მოსახერხებელი გახდა.

`Pipe` ხშირად რთული ოპერაციებისთვის გამოიყენება, როდესაც ერთი ბრძანების გამოსასვლელს სტანდარტულ შესასვლელზე სხვა ბრძანებას ვაწვდით დასამუშავებლად. თავის მხრივ, ამ ბრძანების სტანდარტული გამოსასვლელი მესამე ბრძანების სტანდარტულ შესასვლელზე შედის და ა.შ. შესაბამისად, ასეთ კონსტრუქციაში გამოიყენებული ბრძანებები ე.წ. ფილტრებს წარმოადგენს და ყველა მათგანი წინა ბრძანების გამოტანილ შედეგს თავისებურად ფილტრავს. `Pipe`-ით შეიძლება დიდი რაოდენობით ბრძანებები გადავაბათ ერთმანეთს:

```
achiko@debian:~$ cmd1 | cmd2 | cmd3 ...
```

ასე შეიძლება წარმოვიდგინოთ სამი ბრძანების `pipe`-ით გადაბმის პროცესი გრაფიკულად (სურ. 6.1):

ფილტრი ბრძანებები (ფილტრები) ხშირად გამოიყენება მონაცემებზე რთული ოპერაციების განსახორციელებლად მისი რამდენჯერმე გამოყენებით. მოდით, `sort` ბრძანების გამოყენება ვცადოთ. ვთქვათ, გვსურს `/bin` და `/usr/bin` დირექტორიებში არსებული ბრძანებების გამოტანა, მათი დახარისხება ანბანის მიხედვით (სორტირება) და ისე ნახვა:



სურ 6.1: სამი ბრძანების მილით გადაბმის ორგანიზაცია

```
achiko@debian:~$ ls /bin /usr/bin | sort | less
```

ბრძანება `uniq` ხშირად გამოიყენება `sort`-თან კომბინაციაში და დახარისხებული ხაზებიდან მას გამოაქვს უნიკალური (ყველასგან განსხვავებული) ხაზები.

```
achiko@debian:~$ ls /bin /usr/bin | sort | uniq | less
```

ბრძანება `wc`-ს საშუალებით ვითვლით თუ რამდენი ხაზი, სიტყვა და ბაიტია (ლათინური ტექსტის შემთხვევაში რამდენი ასო-ნიშანია) ფაილში.

```
achiko@debian:~$ wc LICENSE.txt
255  1848 12767 LICENSE.txt
```

ამ შემთხვევაში სამივე მონაცემი გამოვა ეკრანზე: ხაზების, სიტყვებისა და ბაიტების რაოდენობა. თუ მხოლოდ ერთ-ერთი მონაცემის ნახვა გვსურს, მაშინ შესაბამისი ოფცია უნდა მივუთითოთ `wc` ბრძანებას.

ოფციები	მნიშვნელობა
<code>-l,--lines</code>	ხაზების რაოდენობა.
<code>-w,--words</code>	სიტყვების რაოდენობა.
<code>-c,--bytes</code>	ბაიტების რაოდენობა.
<code>-m,--chars</code>	ასო-ნიშნების რაოდენობა (სასარგებლოა არა ლათინური ასო-ნიშნების დასათვლელად).

შეგვიძლია ფაილის ნაცვლად ბრძანება `wc`-ს სხვა ბრძანების გამოსასვლელი გადავცეთ მილით და ვნახოთ, რამდენი უნიკალური ხაზი, ანუ უნიკალური ბრძანება გვაქვს `/bin` და `/usr/bin` დირექტორიებში.


```
achiko@debian:~$ ls /bin /usr/bin | sort | uniq | wc -l
```

grep ბრძანებით შეგვიძლია დამატებით გავფილტროთ ჩვენი სია და ეკრანზე მხოლოდ ის საზეზი გამოვიტანოთ, რომლებიც დასახელებაში გარკვეულ გამოსახულებას შეიცავს, მაგალითად zip-ს.

```
achiko@debian:~$ ls /bin /usr/bin | sort | uniq | grep "zip"
bunzip2
bzip2
bzip2recover
funzip
gpg-zip
gunzip
gzip
p7zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
zipdetails
zipgrep
zipinfo
```

ბრძანება tee კითხულობს სტანდარტულ შესასვლელს, აკოპირებს მას სტანდარტულ გამოსასვლელზე და, ამავდროულად, ჩაწერს არგუმენტად გადაცემულ ერთ ან მეტ ფაილში.

„>“ ოპერატორით გადამისამართებისას მონაცემები მხოლოდ ფაილის შიგთავსში იწერება და მათ ეკრანზე ვეღარ ვხედავთ. ხშირად სასურველია, ეს მონაცემები ეკრანზეც დავინახოთ. სწორედ ასეთ შემთხვევებში გამოიყენება tee ბრძანება. ის ერთ-ერთი სასარგებლო და იშვიათი ბრძანებაა, რადგან სხვა ბრძანებებისგან განსხვავებით, ერთზე მეტი მიმართულებით შეუძლია ინფორმაციის გადამისამართება.

```
achiko@debian:~$ ls /bin /usr/bin >ls.txt
```

ასე ls.txt-ში ჩაიწერება „ls /bin /usr/bin“ ბრძანების შედეგი და მას ეკრანზე ვეღარ დავინახავთ. tee ბრძანებით კი მას ეკრანზეც ვიხილავთ:

```
achiko@debian:~$ ls /bin /usr/bin | tee ls.txt
...
zenity
zipdetails
zipgrep
zipinfo
```

ზოგადად, tee შეგვიძლია გამოვიყენოთ როგორც მარტივი ტექსტური რედაქტორი. კლავიატურიდან რასაც აკრეფთ file.txt-ში ჩაიწერება, და ამავდროულად, ეკრანზე

დუბლირდება. Ctrl-d კლავიშების კომბინაციით დავასრულებთ პროცესს.

```
achiko@debian:~$ tee file.txt
ხაზი 1 [enter]
ხაზი 1
ხაზი 2 [enter]
ხაზი 2
[Ctrl-d]
```

ასე, ჩვენ მიერ კლავიატურიდან შეტანილი ტექსტი file.txt ფაილის შიგთავსს თავზე გადაწერება. -a ოფციით კი ტექსტი ფაილის ბოლოში დაემატება.

```
achiko@debian:~$ tee -a file.txt
ხაზი 3 [enter]
ხაზი 3
ხაზი 4 [enter]
ხაზი 4
[Ctrl-d]
achiko@debian:~$ cat file.txt
ხაზი 1
ხაზი 2
ხაზი 3
ხაზი 4
```

ამ თავში განხილული ბრძანებებით წარმოდგენა შეგვექმნა ფილტრებზე. გაცილებით მძლავრ და მეტ ბრძანება-ფილტრებს მოგვიანებით დავუბრუნდებით.

თავი 7

ტექსტური რედაქტორები

ქამდე ფაილში სასურველი შიგთავსის განთავსებას გადამისამართებით ან **tee** ბრძანებით ვახერხებდით. მოდით, ახლა სრულფასოვანი ტექსტური რედაქტორები განვიხილოთ. ზოგადად, ოპერაციულ სისტემებში ტექსტური რედაქტორი ძალიან მნიშვნელოვან კომპონენტს წარმოადგენს. მისი საშუალებით ხდება ტექსტური ფაილების შექმნა, კოდების დაწერა, კონფიგურაციის ფაილების რედაქტირება და სხვა. ამ მხრივ, ლინუქსში საკმაოდ ფართო არჩევანი გვაქვს. ტექსტური რედაქტორები არსებობს როგორც გრაფიკულ გარემოში (GUI), ისე ბრძანებათა ხაზში (CLI) სამუშაოდ.

ჩამოვთვალოთ რამდენიმე ცნობილი, ღია კოდის მქონე ტექსტური რედაქტორი:

vi/vim	მძლავრი CLI ტექსტური რედაქტორი. vim Unix-ის ძველი ტექსტური რედაქტორის - vi -ს გაძლიერებულ ვერსიას წარმოადგენს. vim ერთ-ერთი ყველაზე პოპულარული და ფართოდ გამოყენებადი პროგრამაა სისტემის ადმინისტრატორებსა და პროგრამისტებს შორის.
nano	გამოსაყენებლად მარტივი პროგრამა როგორც დამწყები, ასევე გამოცდილი მომხმარებლებისთვის.
emacs (GNU emacs)	განვრცობადი (highly extensible), კარგად მორგებადი (customizable) ერთ-ერთი პირველი ტექსტური რედაქტორი, რომელიც დღესაც არ კარგავს აქტუალობას.
gedit	ზოგადი (საერთო) დანიშნულების მქონე GUI პროგრამა. ნაგულისხმევი მნიშვნელობით, ის დაინსტალირებულია GNOME გრაფიკულ გარემოში.
kate/kwrite	ზოგადი (საერთო) დანიშნულების მქონე GUI პროგრამა და გვხვდება KDE გრაფიკულ გარემოში.
gVim	vim -ის GUI ვერსია.
libreoffice, openoffice	პროფესიონალური საოფისე პროგრამების ნაკრები.

LaTeX (L^AT_EX)

L^AT_EX სისტემები გამოიყენება პროფესიონალური, სამეცნიერო სტატიებისა და წიგნების შესაქმნელად. ცნობილია T_EX - ის შემდეგი დისტრიბუტივები: TeX Live, teTeX, MikTeX და ა.შ. ეს დისტრიბუტივები, თავის მხრივ, შეიცავენ ტექსტურ რედაქტორებს, რომლებიც ჩვენთვის ტექსტთან სამუშაოდ მიჩვეული პროგრამებისგან განსხვავებით, არ წარმოადგენს WYSIWYG („What You See Is What You Get“) სისტემას. WYSIWYG პროგრამების ის კატეგორიაა, სადაც დარედაქტირებული ფაილის საბოლოო სახე (ამობეჭდვის შემდეგ) იგივე იქნება, რაც ვიზუალურად ჩანს მისი შექმნისას ეკრანზე. პირველი ნამდვილი WYSIWYG ტექსტური რედაქტორი, სახელად **bravo**, 1974 წელს შეიქმნა Xerox PARC კომპანიაში და შემდეგ, მაიკროსოფტის ორი პროდუქტის, ასევე WYSIWYG ტიპის – **word** და **excel**-ის საძირკველი გახდა. L^AT_EX -ის სისტემებში ცნობილი ტექსტური რედაქტორებია – **TeXmaker**, **TeXstudio**, **TeXworks** და სხვა.

7.1 რედაქტორი – vi/vim

მოდით, გავიცნოთ ერთ-ერთი ყველაზე გავრცელებული, ეკრანზე ორიენტირებული ტექსტური რედაქტორი **vi** და მისი შესაძლებლობები. ის ყველა Linux სისტემაშია წარმოდგენილი.

რატომ **vi**? თანამედროვე ეპოქაში, როდესაც უამრავ გრაფიკულ რედაქტორთან ერთად, უმარტივესი, მოსახმარად ადვილი არაგრაფიკული რედაქტორები არსებობს, რატომ გვესაჭიროება, ვიცოდეთ **vi**?

ამისთვის რამდენიმე მიზეზი არსებობს: **vi** ყოველთვისაა ხელმისაწვდომი. ის არის თავისუფალი პროგრამა და მისი კოდი ღიაა. **vi** სწრაფია და კომპიუტერის ძალიან ცოტა რესურსს მოიხმარს. ამავდროულად, **vi** მორგებულია სწრაფი ბეჭდვისთვის. გამოცდილ მომხმარებელს ტექსტის აკრეფისას თითქმის არასოდეს მოუწევს თითების აწევა კლავიატურიდან. გარდა ამისა, **vi**-ს ყოველი ინსტრუქცია ძალან კარგად არის დოკუმენტირებული და მისი მრავალმხრივი საზოგადოება ძალიან დიდია. **vi** განვრცობადია, მისი ფუნქციონალი უამრავი საინტერესო დამატებითი მოდულებით შეიძლება გავამდიდროთ. **vi**-ში ფაილის ბევრი ფორმატი და დაპროგრამების ბევრი ენის სინტაქსია მხარდაჭერილი. **nano** რედაქტორი, რომლის პოპულარობაც დღითიდღე იზრდება, ჯერჯერობით, ბოლომდე უნივერსალური არ არის. POSIX-თან თავსებადობა მოითხოვს **vi**-ს არსებობას სისტემაში. სწორედ ამიტომ **vi** რედაქტორი Unix-ის მსგავს სისტემებში დე ფაქტო სტანდარტს წარმოადგენს.

vi-ს გაუმჯობესებულ ვერსიას ჰქვია **vim** (Vim IMproved). ბევრ სისტემაში **vim**-ის დაყენების შემდეგ, ის **vi**-ს მალსახმობი ხდება და **vi**-ს გამოძახება, სინამდვილეში, **vim**-ის გაშვებას ნიშნავს. მოდით, გავუშვათ **vi**. არგუმენტად ფაილი გადაცემთ:

```
achiko@debian:~$ vi file.txt
~
~
~
~
~
~
```

~
~

"file.txt" [New File]

0,0-1

All

თუ file.txt არსებობს, მისი შიგთავსი გაიხსნება რედაქტირებისთვის, თუ არა არსებობს, ის შეიქმნება. ხაზების წინ ტილდა (~) მოუთითებს იმაზე, რომ ეს ხაზი ჯერ გამოყენებული არ არის, ანუ ცარიელია. ბოლო ხაზზე, რომელიც სტატუსის ხაზს (statusline) წარმოადგენს, დეტალური ინფორმაციაა ნაჩვენები (რომელ ხაზზე და მერამდენე სიმბოლოზე დგას კურსორი და ეკრანზე შიგთავსის რა ნაწილი ჩანს). vi რედაქტორში, ფაილის გახსნის შემდეგ, ავტომატურად აღმოვჩნდებით ე.წ. ბრძანებათა რეჟიმში (ე.წ. Command mode). ამ რეჟიმში კლავიშზე დაჭერისას მისი შესაბამისი გრაფიკული გამოსახულება ეკრანზე არ გამოჩნდება. სამაგიეროდ, შესრულება ბრძანება, რომელიც vi-ში ამ კლავიშის ქვეშ იგულისხმება. სწორედ ამიტომ ჰქვია ამ რეჟიმს ბრძანების რეჟიმი. ტექსტის შესაყვანად კი აკრეფის რეჟიმში (ე.წ. Insert mode) უნდა გადავიდეთ. იქ უკვე კლავიატურის კლავიშებით შესაბამისი ასო-ნიშნების ჩაწერას შევძლებთ ფაილში. ამ რეჟიმში გადასასვლელად საკმარისია **[i]** -ს დავაჭიროთ. შედეგად, სტატუსის ხაზზე გაჩნდება ჩანაწერი INSERT, რომელიც მიგვითითებს იმაზე, რომ აკრეფის რეჟიმში ვიმყოფებით. აქ უკვე შეგვიძლია ტექსტის აკრეფა.

achiko@debian:~\$ vi file.txt

The word "pheasant" (ხოხობი) ultimately comes from Phasis, the ancient name of what is now called the Rioni River in Georgia. It passed from Greek to Latin to French (spelled with an initial "f") then to English, appearing for the first time in English around the year 1299.

~
~
~
~
~
~
~
~

-- INSERT --

1,55-43

All

თავდაპირველად vi ყოველთვის ბრძანების რეჟიმში ირთვება. თუ, გარკვეული ქმედებების შემდეგ, ისეთ სიტუაციაში აღმოჩნდით, რომ დაიბენით და ვერ ხვდებით რომელ რეჟიმში ხართ, მაშინ გამოსვლის ლილაკს, **[Esc]** -ს, ორჯერ დააჭირეთ და, ავტომატურად, ბრძანების რეჟიმში აღმოჩნდებით.

vi-დან გამოსასვლელად ბრძანებათა რეჟიმიდან **[:q]** (quit) ბრძანება უნდა გავუშვათ. თუ ფაილის შიგთავსი შევცვალეთ და მისი დამახსოვრებაც გვსურს გამოსვლისას, მაშინ **[:wq]** (write,quit) ბრძანებები უნდა გავუშვათ, ხოლო დამახსოვრების გარეშე თუ გვსურს გამოსვლა, მაშინ **[:q!]**.

მოდით ვნახოთ vi რედაქტორის სხვა ბრძანებებიც:

ფაილის შიგთავსში მოძრაობის ბრძანებები

[h]

კურსორის გადაადგილება მარცხნივ. **[←]**

j	კურსორის გადაადგილება ქვედა ხაზზე. (↓)
k	კურსორის გადაადგილება ზედა ხაზზე. (↑)
l	კურსორის გადაადგილება მარჯვნივ. (→)
w	კურსორის გადაადგილება შემდეგ სიტყვაზე მარჯვნივ. (3w) – ასე, კურსორი გადავა მარჯვნივ, მე-3 სიტყვაზე.
b	კურსორის გადაადგილება წინა სიტყვაზე მარცხნივ. (2b) – კურსორი გადავა მარცხნივ, მე-2 სიტყვაზე.
\$	კურსორის გადაადგილება მიმდინარე ხაზის ბოლოს.
0	კურსორის გადაადგილება მიმდინარე ხაზის დასაწყისში.
^	კურსორის გადაადგილება მიმდინარე ხაზის დასაწყისში პირველ არაცარიელ სიმბოლოზე.
gg	კურსორის გადაადგილება პირველი ხაზის დასაწყისში. (3gg) – კურსორი გადავა მე-3 ხაზის დასაწყისში.
G	კურსორის გადაადგილება ბოლო ხაზის დასაწყისში. (3G) – კურსორი გადავა მე-3 ხაზის დასაწყისში. (1G) – ასე კი პირველი ხაზის დასაწყისში.
Ctrl^f	შემდეგ გვერდზე გადასვლა.
Ctrl^b	წინა გვერდზე გადასვლა.

ფაილის შიგთავსის რედაქტირების ბრძანებები

x	ერთი ასო-ნიშნის წაშლა, იქ, სადაც კურსორი დგას. (5x) – 5 ასო-ნიშნის წაშლა კურსორიდან მარჯვნივ.
X	ერთი ასო-ნიშნის წაშლა კურსორამდე. (5X) – 5 ასო-ნიშნის წაშლა კურსორამდე მარცხნივ.
d	წაშლა. (dw) – შლის სიტყვას მარჯვნივ. (10dw) – შლის 10 სიტყვას მარჯვნივ. (d4h) – შლის 4 სიმბოლოს მარცხნივ. (h) მიმართულებით). (d4k) – შლის მიმდინარე და ზედა 4 ხაზს. (k) მიმართულებით). (d^) – შლის კურსორის პოზიციიდან ხაზის დასაწყისამდე ყველა სიმბოლოს. (d\$) – შლის ყველა სიმბოლოს კურსორის პოზიციიდან ხაზის ბოლომდე. თუ ფაილის მთლიანი შიგთავსის წაშლა გვსურს, მაშინ ჯერ კურსორი პირველ ხაზზე უნდა გადავიყვანოთ და შემდეგ წავშალოთ ბოლო ხაზის ჩათვლით ყველა ხაზი, ასე: (gg) + (dG)
dd	შლის მიმდინარე ხაზს. (7dd) – შლის 7 ხაზს (მიმდინარე ხაზიდან დაწყებული).
D	მიმდინარე ხაზზე შლის კურსორიდან ხაზის ბოლომდე ყველა სიმბოლოს.
y	კოპირება. (y4h) – აკოპირებს 4 სიმბოლოს მარცხნივ (h მიმართულებით).

yy	აკოპირებს მიმდინარე ხაზს. 3yy – აკოპირებს 3 ხაზს მიმდინარე ხაზის ჩათვლით.
p/P	ჩასვამს კოპირებულ ტექსტს მომდევნო/წინა ახალ ხაზზე. ჩასმა ეხება აგრეთვე x და d ბრძანებით წაშლილ ბოლო ტექსტებს, რადგან ეს ტექსტები ბუფერში ინახება.
u	ბოლო შესრულებული ბრძანების გაუქმება (ე.წ. undo).
Ctrl^r	გაუქმებამდე დაბრუნება. ანუ გაუქმების გაუქმება. (ე.წ. redo).
r	კურსორის პოზიციაზე მდგომ ერთ ასო-ნიშანს გადააწერს ჩვენს მიერ აკრეფილ ერთ სიმბოლოს.
R	გადაეწერება კურსორის პოზიციიდან მომდევნო ყველა სიმბოლოს მანამ, სანამ Esc კლავიშს არ დავაჭერთ.
~	პატარა ასო-ნიშანს დიდად გადააკეთებს და დიდს პატარად.
J	შემდეგი ხაზის შეერთება მიმდინარესთან.
/სიტყვა	ტექსტში კურსორის შემდეგ მოძებნის „სიტყვას“. შემდეგ, „სიტყვაზე“ გადასასვლელად, უნდა დავაჭიროთ n -ს (next). წინაზე გადასასვლელად – N -ს.
?სიტყვა	„სიტყვის“ ძიება კურსორის წინ.

აკრეფის რეჟიმში გადასვლა

i	გადავა აკრეფის რეჟიმში და ტექსტის აკრეფა დაიწყება კურსორის მიმდინარე პოზიციის წინ.
I	ტექსტის აკრეფა დაიწყება მიმდინარე ხაზის დასაწყისიდან.
a	ტექსტის აკრეფა დაიწყება კურსორის შემდეგ.
A	ტექსტის აკრეფა დაიწყება მიმდინარე ხაზის ბოლოდან.
o	მიმდინარე ხაზის შემდეგ ახალი ხაზი შეიქმება და იქ დაიწყება ტექსტის აკრეფა.
O	მიმდინარე ხაზის წინ ჩაჯდება ახალი ხაზი და იქ დაიწყება ტექსტის აკრეფა.
cc	მიმდინარე ხაზი წაიშლება, და მის ადგილზე, დაიწყება ტექსტის აკრეფა.

სხვა ბრძანებები

:w	ცვლილებების დამახსოვრება ფაილში.
:wq	შენახვა და გამოსვლა. იგივეა, რაც ბრძანება ZZ .
:q!	ცვლილებების არ შენახვა და ისე გამოსვლა.

<code>:s/სიტყვა1/სიტყვა2/</code>	მიმდინარე ხაზზე, პირველი შემხვედრი „სიტყვა1“ შეიცვლება „სიტყვა2“-ით.
<code>:s/სიტყვა1/სიტყვა2/g</code>	– შეიცვლება მიმდინარე ხაზსა და მის შემდგომ ყველა ხაზში.
<code>:i,s/სიტყვა1/სიტყვა2/g</code>	– შეცვლა მოხდება i-ური ხაზიდან j-იურ ხაზის ჩათვლით.
<code>:%:s/სიტყვა1/სიტყვა2/</code>	– შეიცვლება ყველაგან.
<code>:r newfile.txt</code>	შემდეგ ხაზზე newfile.txt ფაილის შიგთავსის ჩასმა.
<code>!:cmd</code>	vi-დან გამოუსვლელად, შელის ბრძანების (cmd) გაშვება და შედეგის ნახვა. <code>!:ls</code> – ასე ვნახავთ მიმდინარე დირექტორიაში არსებულ ფაილებს.
<code>:n newfile.txt</code>	ასე, vi-ში ახალ ფაილს (newfile.txt) ვხსნით. თუ რამდენიმე ფაილი გვაქვს გახსნილი, მაშინ წინა ან შემდეგ ფაილზე გადასვლა შეგვიძლია <code>:bp</code> და <code>:bn</code> ბრძანებებით. <code>:b</code> ბრძანების აკრეფა და შემდეგ ტაბულაციის ღილაკზე დაჭერა დაგვხმარება გახსნილი ფაილების იდენტიფიცირებასა და არჩევაში.
<code>:set nu</code>	ხაზების ნუმერაციის გამოჩენა.
<code>:set nonu</code>	ხაზების ნუმერაციის გაქრობა.
<code>:set hlsearch</code>	ტექსტში ძებნისას სიტყვის სხვა ფერით მონიშვნა (ე.წ. highlight).
<code>:noh</code>	highlight-ის გამორთვა.

ბრძანებისა და აკრეფის რეჟიმების გარდა, vi-ში არსებობს ვიზუალური რეჟიმიც (ე.წ. Visual mode). ამ რეჟიმში ხდება ტექსტის მონიშვნა. მონიშნული ტექსტი ვიზუალურად გამოჩნდება და მასზე შეგვიძლია ვიმოქმედოთ vi-ს სხვადასხვა ბრძანებებით – წაშლა (d), კოპირება (y) ან სხვა. ვიზუალურ რეჟიმში გადასვლა შესაძლებელია `[v]` ბრძანებით, `[V]` ბრძანებით ან `[Ctrl^v]` კლავიშების კომბინაციით. შემდეგ, ტექსტის მოსანიშნად `[e]` ბრძანება და სხვადასხვა მიმართულებების ღილაკები უნდა გამოვიყენოთ. ვიზუალურ რეჟიმი `[v]` ბრძანებით გადასვლა საშუალებას მოგვცემს მოვნიშნოთ ტექსტის ფრაგმენტი ერთი ასო-ნიშნის სიზუსტით, `[V]` -ს შემთხვევაში – ხაზების სიზუსტით, ხოლო `[Ctrl^v]` -ს გამოყენებისას – ტექსტის ბლოკების სიზუსტით. ასევე, მაუსით ტექსტის მონიშვნა ავტომატურად გამოიწვევს ვიზუალურ რეჟიმში გადასვლას.

ფაილების გახსნა vi ტექსტურ რედაქტორში მხოლოდ კითხვის რეჟიმშიც შეგვიძლია. ეს შესაძლებლობას გვაძლევს თავი დავიზღვიოთ რაიმე შემთხვევითი ცვლილებისგან ფაილის შიგთავსში. მხოლოდ კითხვით რეჟიმში ფაილის გახსნა ასე ხდება:

```
achiko@debian:~$ vi -R file.txt      # ან
achiko@debian:~$ view file.txt
```

საინტერესო და, ამავდროულად, მარტივი, სადემონსტრაციო, ონლაინ ინტერაქტიული ინსტრუქციები მოცემულია ვებ-გვერდზე შემდეგ: <https://www.openvim.com/>

დელტალური დოკუმენტაციის სახანავედ კი შეგიძლიათ, შელში vimtutor ბრძანება გაუშვათ.


```
achiko@debian:~$ vimtutor
```

```
=====
= Welcome to the VIM Tutor - Version 1.7 =
=====
```

```
Vim is a very powerful editor that has many commands, too many to
explain in a tutor such as this. This tutor is designed to descri-
be enough of the commands that you will be able to easily use Vim
as an all-purpose editor.
```

```
...
```

```
~~~~~
Lesson 1.1: MOVING THE CURSOR
```

```
...
```

7.2 რედაქტორი – GNU Emacs

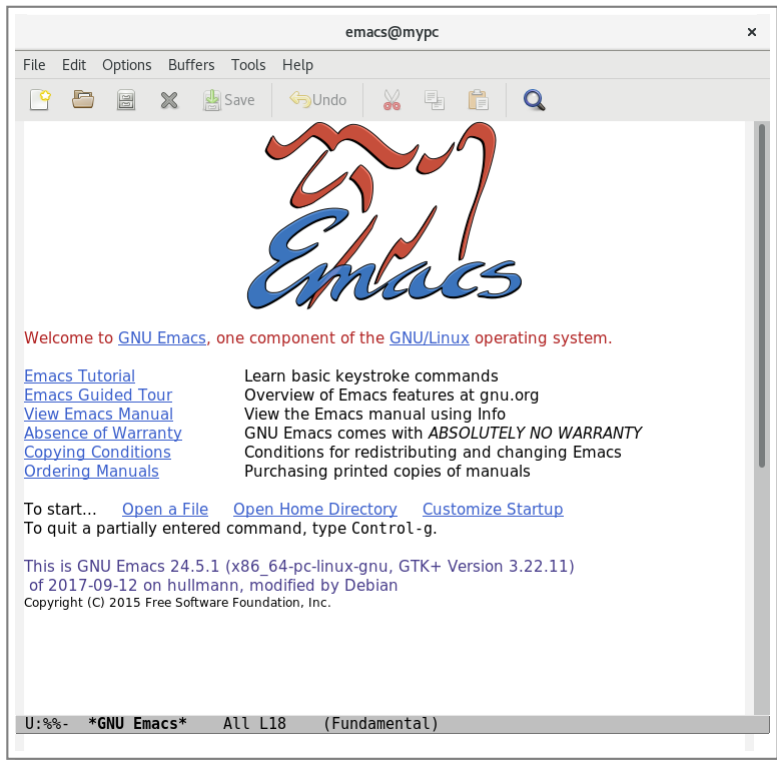
GNU Emacs ერთ-ერთი ყველაზე მძლავრი ტექსტური რედაქტორია მათ შორის, რომლებსაც, პირადად, ვიცნობ. მას აქვს ძალიან მდიდარი დოკუმენტაცია და უამრავი ფუნქცია გაფართოვებების სახით. მასში ძალიან მარტივად მუშავდება ტექსტური დოკუმენტი, მოხერხებულად ხდება აბზაცებზე, წინადადებებსა და სიტყვებზე გადასვლა. მას აქვს ძიების მძლავრი მექანიზმი, იყენებს რა რეგულარულ გამოსახულებებს, კლავიატურის მაკროსებსა და სხვა. Emacs ყველანაირი ტექსტის დამუშავებისათვის შეგვიძლია გამოვიყენოთ, როგორც დიდ ტექსტებზე სამუშაოდ, ასევე ყოველდღიური მარტივი ჩანაწერების გასაკეთებლად, პროგრამების დასაწერად და სხვა. ის ყველანაირი გემოვნების ადამიანს დააკმაყოფილებს. მასში შეგიძლიათ ტექტრისი ითამაშოთ და ბევრი სხვა სახალისო რამ აღმოაჩინოთ. GNU Emacs, როგორც დასახელებიდან ჩანს, GNU პროექტის ფარგლებში შექმნილი პროდუქტია და დაწერილია C დაპროგრამების ენაზე. თავდაპირველი ვერსია თავად რიჩარდ სტალმანმა დაწერა. ამ პროგრამით სარგებლობა შესაძლებელია ბევრ სხვადასხვა სისტემაზე, როგორცაა: GNU/Linux, FreeBSD, NetBSD, OpenBSD, MacOS, MS Windows და Solaris.

სანამ GNU Emacs პროგრამას გავუშვებთ და სადემონსტრაციოდ პრაქტიკულ ღონიძიებებს მივმართავთ, ვთქვათ, თუ რას წარმოადგენს Emacs-ში ბუფერი. ეს ტერმინი ხშირად შეგხვდება Emacs-ში მუშაობისას და ძალიან მნიშვნელოვანია მისი ცოდნა. ბუფერი emacs-ში არის ის ადგილი, სადაც ჩვენ სასურველ ტექსტს ვკრეფთ. შეგიძლია ვთქვათ, რომ ყოველთვის ბუფერში ვმუშაობთ, უბრალოდ შესაძლებელია, ბუფერში არსებული ტექსტი შემდეგ ფაილში შევინახოთ. ტექსტის შემცველი ფაილის გახსნისასაც კი მისი შიგთავსი ბუფერში განთავსდება. თუმცა, არ არის აუცილებელი, რომ ბუფერი რომელიმე ფაილთან იყოს კავშირში. ეს კი ძალიან სასარგებლოა, როდესაც ტექსტის დროებითი დამუშავება გვსურს და არ გვინდა ზედმეტი ფაილის შექმნა. Emacs-ში შესაძლებელია, ბევრი ბუფერი გვქონდეს ერთდროულად გახსნილი, მაგრამ კონკრეტულ მომენტში მხოლოდ ერთ ბუფერში შეგიძლია მუშაობა. მას მიმდინარე ბუფერს უწოდებენ. ბუფერში, როგორც ვთქვით, ტექსტი ანუ ასო-ნიშნათა ერთობლიობა იწერება. დავსვათ კითხვა: რამდენი სიმბოლო შეიძლება ჩაეტიოს ბუფერში? აქვს მას საზღვარი? პასუხია: დიახ, ბუფერის ზომა შეზღუდულია. შესაბამისად, ბუფერში გარკვეულ ინფორმაციაზე მეტის ჩაწერა არ შეგიძლია. 64 ბიტისანი მანქანებისთვის ბუფერის მაქსიმალური ზომა $2^{61}-2$ ბაიტია, დაახლოვებით 2EiB. 32 ბიტისანი მანქანებისთვის კი $2^{29}-2$ ბაიტია, დაახლოვებით 512MiB. ბუფერის ზომა სისტემაში აგრეთვე შეზღუდულია ოპერატიული მესხიერების ზომით.

მოდიტ დავიწყეთ. გავუშვათ GNU Emacs პროგრამა.

```
achiko@debian:~$ emacs &
```

გრაფიკულ გარემოში ასეთი ფანჯარა გაიხსნება. იხილეთ სურათი 7.1.



სურ 7.1: GNU Emacs ტექსტური რედაქტორი გრაფიკულ გარემოში

რა თქმა უნდა, ასეთ გარემოში შეგიძლიათ მაუსის გამოყენება, თუმცა Emacs-ში არსებული ფუნქციონალის დიდი ნაწილი შეგიძლიათ კლავიატურის კლავიშების კომბინაციებით გამოიყენოთ. მაგალითად, ახალი ფაილის გახსნა **Ctrl^x** + **Ctrl^f** კლავიშებზე დაჭერით ხდება.

GNU Emacs-ით სარგებლობა, ცხადია, არაგრაფიკულ გარემოშიც შეგიძლიათ. მისი პირველი ვერსია იმ დროს შეიქმნა, როდესაც გრაფიკული გარემო არც კი არსებობდა. ტექსტურ რეჟიმში ამ პროგრამის გაშვება იგივენაირად, **emacs** ბრძანებით, შეგეძლებათ. ანუ, **emacs**-ის გაშვება ტერმინალის ემულატორში ახალი ფანჯრის შექმნას გამოიწვევს, ვირტუალური ტერმინალიდან კი სრულ ეკრანზე გაეშვება, ასე:

```
achiko@debian:~$ emacs
Welcome to GNU Emacs, one component of the GNU/Linux operating system.

Get help          C-h (Hold down CTRL and press h)
Emacs manual      C-h r          Browse manuals      C-h i
Emacs tutorial    C-h t          Undo changes      C-x u
```

```

Buy manuals      C-h RET      Exit Emacs      C-x C-c
Activate menubar M-`
(`C-' means use the CTRL key.  `M-' means use the Meta (or Alt) key.
If you have no Meta key, you may instead type ESC followed by the character.)
Useful tasks:
Visit New File      Open Home Directory
Customize Startup   Open *scratch* buffer

```

GNU Emacs 24.5.1 (x86_64-pc-linux-gnu, GTK+ Version 3.22.11)
of 2017-09-12 on hullmann, modified by Debian
Copyright (C) 2015 Free Software Foundation, Inc.

GNU Emacs comes with ABSOLUTELY NO WARRANTY; type C-h C-w for full details.
Emacs is Free Software--Free as in Freedom--so you can redistribute copies
of Emacs and modify it; type C-h C-c to see the conditions.
Type C-h C-o for information on getting the latest version.

If an Emacs session crashed recently, type Meta-x recover-session RET
to recover the files you were editing.

```

-UUU:%%--F1 *GNU Emacs*  All L1      (Fundamental) -----
For information about GNU Emacs and the GNU system, type C-h C-a.

```

მოდით, ჩამოვთვალოთ Emacs-ის ძირითადი კლავიშების კომბინაციები:

ფაილის/ბუფერის გახსნა, შენახვა, დახურვა

Ctrl[^]x + **Ctrl[^]f**

ფაილის გახსნა. ამ კომბინაციზე დაჭერით ფაილის სახელი უნდა შევიყვანოთ (ფანჯრის ქვედა ნაწილში). თუ ის უკვე არსებობს, მისი შიგთავსი გაიხსნება, თუ არა, მაშინ ახალი დასახელების ფაილი გაიხსნება, რომელიც მიმდინარე ბუფერზე იქნება მიბმული.

Ctrl[^]x + **Ctrl[^]s**

მიმდინარე ბუფერის შენახვა (Save).

Ctrl[^]x + **Ctrl[^]w**

მიმდინარე ბუფერის შენახვა მითითებული დასახელების ფაილში (Save as).

Ctrl^x + s	ყველას შენახვა (Save all).
Ctrl^x + k	მიმდინარე ბუფერის დახურვა.
Ctrl^x + b	გადართვა შემდეგ ბუფერზე. შეყვანის ღილაკზე (␣) დაჭერის შემდეგ გადავალთ ნაგულისხმევი მნიშვნელობით შემოთავაზებულ ბუფერზე.
Ctrl^x + b + buffer_name	გადართვა ბუფერზე დასახელებით buffer_name თუ ის უკვე არსებობს. წინააღმდეგ შემთხვევაში, ახალი ბუფერის შექმნა ამ დასახელებით.
Ctrl^x + Ctrl^b	ფანჯარა ჰორიზონტალურად ორად გაიყოფა და ქვედა დანაყოფში გამოჩნდება არსებული ბუფერების სია.
Ctrl^x + 1	ფანჯრის დაყოფის მოხსნა.
Ctrl^x + →	გადართვა შემდეგ ბუფერზე.
Ctrl^x + ←	გადართვა წინა ბუფერზე.
Ctrl^x + Ctrl^c	Emacs-დან გამოსვლა.

კოპირება, ჩასმა, გაუქმება, მონიშვნა ...


Ctrl^_	ბოლო ოპერაციის გაუქმება (Undo). შესაძლებელია ამ ოპერაციის მჯრავალჯერ გამოყენება. Ctrl^_ -ის ნაცვლად შეგიძლიათ Ctrl^/ -ც გამოიყენოთ.
Ctrl^g + Ctrl^/	ბოლო გაუქმბის გაუქმება (Redo). Ctrl^g + Ctrl^/ + Ctrl^/ + ... – მრავალი Redo.
Ctrl + გამოგოვება	მონიშვნის დაწყება.
Ctrl^x + h	ყველაფრის მონიშვნა (Select all).
Meta^w	მონიშნულის კოპირება (Copy). Meta იგივეა რაც, Alt კლავიში.
Ctrl^w	მონიშნულის ამოჭრა (Cut).
Ctrl^y	ჩასმა (Paste).





კურსორის მოძრაობა

Ctrl^f	კურსორის გადაადგილება მარჯვნივ ერთი ასო-ნიშნით. →
Ctrl^b	კურსორის გადაადგილება მარცხნივ ერთი ასო-ნიშნით. ←
Ctrl^p	კურსორის გადაადგილება ზედა ხაზზე. ↑
Ctrl^n	კურსორის გადაადგილება ქვედა ხაზზე. ↓
Ctrl^v	შემდეგი გვერდი. Page Down
Meta^v	წინა გვერდი. Page up

Ctrl^Home	ბუფერში მოცემული დოკუმენტის დასაწყისზე გადასვლა. იგივეა, რაც Meta^<
Ctrl^End	ბუფერში მოცემული დოკუმენტის ბოლოზე გადასვლა. იგივეა, რაც Meta^>
Ctrl^←	კურსორის გადაადგილება ერთი სიტყვით მარცხნივ. იგივეა, რაც Meta^b
Ctrl^→	კურსორის გადაადგილება ერთი სიტყვით მარცხნივ. იგივეა, რაც Meta^f
Meta^{	კურსორის გადაადგილება შემდეგ აბზაცზე.
Meta^}	კურსორის გადაადგილება წინა აბზაცზე.

წაშლა, მოძებნა, სხვა

Meta^d	სიტყვის წაშლა მარჯვნივ.
Meta^← (backspace)	სიტყვის წაშლა მარცხნივ.
Ctrl^k	კურსორის პოზიციიდან ხაზის ბოლომდე ყველა ასო-ნიშნის წაშლა.
Ctrl^x + 2	ფანჯრის გაყოფა ჰორიზინტალურად, ორ პანელად.
Ctrl^x + o	ფანჯრის ერთი პანელიდან მეორეზე გადასვლა (აქ გამოყენებულია ლათინური ანბანის პატარა „o“ ასო! არ აგერიოთ ის ნულში – „0“).
Ctrl^s	ტექსტში სიტყვის მოძებნა. Ctrl^s -ის დაჭერის შემდეგ უნდა შევიყვანოთ საძიებო სიტყვა, რომელიც ფანჯრის ბოლო ხაზზე გამოჩნდება. ასეთის არსებობის შემთხვევაში, Emacs სხვა ფერით მონიშნავს მოძებნილს. კვლავ Ctrl^s -ზე დაჭერით შემდეგი შემთხვევა გაფერადდება და ა.შ. წინა მოძებნილზე დაბრუნება Ctrl^r -ით ხდება. სასურველ, მოძებნილ შემთხვევაზე კურსორის გასაჩერებლად  ღილაკი უნდა გამოვიყენოთ.

emacs-ში მიმართულების სტანდარტული ღილაკებიც წარმატებით მუშაობს: , , , , **Home**, **End**, **Page up**, **Page Down**. თუ რომელიმე კლავიშების კომბინაციის აკრეფისას შეცდომა მოგივიდათ, **Esc** ღილაკზე რამდენჯერმე დაჭერით, ხშირ შემთხვევაში, დაბრუნდებით საწყის მდგომარეობაში. ყოველი შემთხვევისთვის, ყურადღებით დააკვირდით სტატუსის ხაზს, სადაც გარკვეული მინიშნება შეიძლება გამოჩნდეს.

emacs-ში როდესაც მონიშნულ ფრაგმენტს ვაკოპირებთ, მისი ჩასმა ნებისმიერ ადგილას, ნებისმიერი რაოდენობით შეგვიძლია. თუ ახალ ფრაგმენტს ვაკოპირებთ, ამ შემთხვევაში, ძველი ფრაგმენტი იგარეგება და მხოლოდ ბოლო ფრაგმენტის გამოყენება შეგვეძლება. იმ შემთხვევისათვის, თუ რამდენიმე ფრაგმენტის დამახსოვრება გსურთ ერთდროულად, მაშინ რეგისტრები უნდა გამოვიყენოთ. ერთი მონიშნულის დამახსოვრება ერთ რეგისტრში მოხდება, მეორე მონიშნულის – მეორეში და ა.შ. ჩასმისას კი ჩვენთვის სასურველ რეგისტრს ავირჩევთ. რეგისტრის შექმნა და მისი შიგთავსის გამოყენება ტექნიკურად ასე ხდება:

რეგისტრები

(Ctrl^x) + (r) + (s) + # მონიშნულის დამახსოვრება რეგისტრში სახელად # (რეგისტრის სახელწოდებად ხშირად რიცხვებს იყენებენ ხოლმე, მაგ: 1, 2 და ა.შ.).

(Ctrl^x) + (r) + (i) + 1 რეგისტრი „1“-ის შიგთავსის ჩასმა კურსორის ადგილზე.

რეგისტრების არსებობა ძალიან კომფორტულ გარემოს ქმნის მომხმარებლისთვის ტექსტებზე მუშაობის დროს. **emacs**-ში აგრეთვე შესაძლებელია გარკვეული ოპერაციების თანმიმდევრული ერთობლიობის დამახსოვრება, ჩაწერა და შემდეგ, სასურველი პოზიციიდან მისი გაშვება, ჩართვა. ოპერაციების ამ ერთობლიობას მაკროსს უწოდებენ. მაკროსის ჩაწერა და გაშვება ტექნიკურად ასე ხდება:

მაკროსები

(Ctrl^x) + ((მაკროსში ოპერაციების ჩაწერის დასაწყისი. ამ კლავიშების კომბინაციაზე დაჭერის შემდეგ ნებისმიერი ქმედება, რასაც გავაკეთებთ დამახსოვრდება მანამდე, სანამ მაკროსის ჩაწერას არ დავასრულებთ.

(Ctrl^x) +) მაკროსის ჩაწერის დასრულება.

(Ctrl^x) + e მაკროსის ჩართვა, გაშვება კურსორის პოზიციაზე.

მაგალითისთვის, ჩავეწეროთ მაკროსი შემდეგნაირად: ჩამოიყვანოთ კურსორი 1 ხაზით ქვევით და არსებული პოზიციიდან წავშალოთ 5 სიტყვა მარჯვნივ. ანუ: **(Ctrl^x) + ((** კლავიშების კომბინაციაზე დაჭერის შემდეგ, ერთხელ უნდა დაგაწვეთ **(Ctrl^n)**-ს (ან **(↓)**-ს) და შემდეგ 5-ჯერ გავიმეოროთ **(Meta^d)** . **(Ctrl^x) +)** -ზე დაჭერით ვასრულებთ მაკროსის ჩაწერას. შემდეგ, კურსორი გადავიყვანოთ სასურველ პოზიციაზე, საიდანაც გვსურს ამ ოპერაციების გამეორება და **(Ctrl^x) + e** კლავიშების კომბინაციით გავუშვათ ეს მაკროსი. დავინახავთ, რომ დამახსოვრებული ინსტრუქციები კვლავ შესრულდება კურსორის ახალი პოზიციიდან.

რეგისტრებისა და მაკროსების არსებობა იშვიათი ფუფუნებაა ტექსტურ რედაქტორებში. მოდით, აღვნიშნოთ **emacs**-ის მორიგი ღირსშესანიშნაობა. ის განსაკუთრებით კარგად არის მორგებული დაპროგრამების სხვადასხვა ენებზე მუშაობისას. კოდის წერისას შესაძლებელია რომელიმე ენის შესაბამის რეჟიმზე გადასვლა, რაც გამოიწვევს დაპროგრამების ამ კონკრეტული ენის ხელსაწყოების მყისიერ დამატებას მენიუ-ში, რომელიც შექმნის პროგრამისთვის მორგებულ გარემოს. ასევე, სხვადასხვა რეჟიმი შემოგთავაზებთ დამატებით ფუნქციას, რომელიც არამხოლოდ დაპროგრამების ენებზე არის ორიენტირებული.

ბუფერში მუშაობის სხვადასხვა რეჟიმები

c-mode დაპროგრამების C ენაზე კოდის წერისთვის, ამ რეჟიმზე გადართვა მენიუში დამატებს ახალ გრაფას „C“, სადაც ამ ენისთვის საჭირო დამატებითი ხელსაწყოები იქნება განთავსებული.

c++-mode დაპროგრამების C++ ენის რეჟიმი.

asm-mode დაპროგრამების Assembly ენის რეჟიმი.

tex-mode **L^AT_EX**-ის ფაილებზე მუშაობისთვის განკუთვნილი რეჟიმი.

text-mode ტექსტზე სამუშაო რეჟიმი.

დაპროგრამების ამ ენების გარდა, Emacs-ში მხარდაჭერილია შემდეგი ენები: Lisp, Objective C, Java, Javascript, Makefiles, AWK, python, perl, Pascal, Ruby, Tcl, SQL და ა.შ. რა თქმა უნდა, შესაძლებელია emacs-ის მორგება დაპროგრამების ახალ ენაზეც. Emacs-ში მუშაობისას ბუფერის რეჟიმის შეცვლა შემდეგი კლავიშების კომბინაციით ხდება: `[Meta^x] + „რეჟიმის სახელი (mode-name)“`. Emacs-ში დაინსტრალირებული რეჟიმების სრული სია კი შემდეგნაირად შეგიძლიათ ნახოთ: `[Meta^x] + [*-mode] + [tab]`.

მოდით, ვახსენოთ რამდენიმე სასარგებლო რეჟიმი: `[Meta^x] + [shell]` -ით შეგიძლიათ, Emacs-ის ბუფერში გაუშვათ shell, სადაც ყველა ბრძანებების გაშვება შეგეძლებათ, რასაც შელში გააკეთებთ. გამოწვევლიანია ისეთი პროგრამები, რომელიც ორიენტირებულია სრულ ეკრანზე, როგორიცაა vi, ის მთლიან ეკრანს იკავებს ტერმინალში, როგორც უკვე იცით. სახელმძღვანელო გვერდებისა გამოსატანად ცალკე რეჟიმი არსებობს: `[Meta^x] + [man]`. `[Meta^x] + [dired]` -ით ფაილების მენეჯერის გამოძახება შეგიძლიათ ფაილების მარტივად ასარჩევად. `[Meta^x] + [ftp]` -ით შეგიძლიათ FTP სერვერთან დაკავშირება და ა.შ. Emacs-ის სრული შესაძლებლობების სია იხილეთ მის ოფიციალურ ვებ-გვერდზე

https://www.gnu.org/software/emacs/manual/html_node/emacs/index.html

თავი 8

წვდომის უფლებები

მ პერაციულ სისტემებში რამდენიმე მომხმარებლის ანგარიში შეიძლება იყოს შექმნილი და სისტემით ხან ერთი მომხმარებელი სარგებლობდეს და ხან მეორე. Unix-ის ოჯახის ყველა ოპერაციული სისტემა მრავალმომხმარებლიანი სისტემაა. მრავალმომხმარებლიანობა იმას ნიშნავს, რომ კომპიუტერთან ერთდროულად შეუძლია მუშაობა რამდენიმე მომხმარებელს. მიუხედავად იმისა, რომ ჩვეულებრივ კომპიუტერს, როგორც წესი, ერთი კლავიატურა და ერთი მონიტორი ახლავს თან, თუ კომპიუტერი ქსელშია ჩართული, რამდენიმე მომხმარებელს ქსელური პროგრამის საშუალებით, ამ კომპიუტერთან დაკავშირება და მასზე დისტანციურად მუშაობა მაინც შეუძლია.

Linux-ში მრავალი მომხმარებლის მხარდაჭერა სიახლეს არ წარმოადგენს. ეს მახასიათებელი სისტემის ჩამოყალიბების დროსვე კარგად გათვალეს. აღვნიშნოთ თუ როგორი გარემო იყო მაშინ, როდესაც Unix იქმნებოდა. წლების წინ, სანამ კომპიუტერი „პერსონალური“ გახდებოდა (ახლა, ძირითადად, პერსონალურ კომპიუტერებს ვხვდებით – PC, Personal Computer), ის წარმოადგენდა ძალიან დიდ და ძვირადღირებულ მოწყობილობას. უნივერსიტეტებში ასეთი მოწყობილობა ერთ შენობაში იყო განთავსებული, ხოლო მომხმარებლები, სხვადასხვა ადგილზე განთავსებული ტერმინალების საშუალებით, უკავშირდებოდნენ ცენტრალურ კომპიუტერს მასზე საშუალოდ. ცალსახად, ასეთ კომპიუტერს მრავალი მომხმარებლის მხარდაჭერა უნდა ჰქონოდა.

ასეთ ვითარებაში გასათვალისწინებელი იყო ის, რომ ერთ მომხმარებელს არ უნდა შეეძლებოდა მეორე მომხმარებლის ფაილებთან წვდომა, თუ თავად ამ ფაილების მფლობელი არ მისცემდა უფლებას.

8.1 მფლობელი და ჯგუფი

Unix-ში მომხმარებელს შეუძლია შექმნას ფაილები და დირექტორიები. მას ამ ფაილებსა და დირექტორიებზე სრული კონტროლი აქვს, ის არის მათი მფლობელი. თავის მხრივ, ეს მომხმარებელი შეიძლება განეკუთვნებოდეს ჯგუფს, რომელიც, ამავდროულად, სხვა მომხმარებლებსაც აწევრიანებს. მფლობელს აქვს იმის საშუალება, რომ ამ ჯგუფის წევრებს გაუზიაროს საკუთარი ფაილები და დირექტორიები და მისცეს მათზე გარკვეული სახის წვდომა. მფლობელს დამატებით იმის საშუალებაც აქვს, რომ თავისი ჯგუფის წევრების გარდა, დანარჩენ მომხმარებლებსაც მისცეს მსგავსი უფლება.

მოდით, დეტალურად გავიაროთ ეს საკითხები. უპირველეს ყოვლისა, ვნახოთ რომელი მომხმარებლის სახელით ვართ შესული სისტემაში და რომელ ჯგუფს განეკუთვნება ეს მომხმარებელი. shell-ში `whoami` ბრძანება ჩვენი მომხმარებლის სახელს გვცუბნება.

```
achiko@debian:~$ whoami  
achiko
```

id ბრძანება მომხმარებლის იდენტიფიკაციის შესახებ უფრო მეტ ინფორმაციას გვანახებს.

```
achiko@debian:~$ id  
uid=1000(achiko) gid=1000(achiko) groups=1000(achiko),  
24(cdrom), 25(floppy), 29(audio), 44(video), 115(scanner)
```

Linux სისტემებში როდესაც მომხმარებელი იქმნება, მას სახელის გარდა, ნომერი ენიჭება – მომხმარებლის იდენტიფიკატორი ე.წ. UID (User ID). ამიერიდან სისტემაში ეს მომხმარებელი ცნობილია ამ ნომრით, UID-ით. ასევე, შექმნისთანავე მომხმარებელი წევრიანდება ჯგუფში – პირველად ჯგუფში (იგივე მომხმარებელი სხვა ჯგუფებშიც შეიძლება გაწევრიანდეს). ამ ჯგუფსაც თავისი ნომერი აქვს – ჯგუფის იდენტიფიკატორი, GID (Group ID).

ახლა უფრო გასაგები გახდა, რა გამოიტანა ეკრანზე id ბრძანებამ. ზოგადად, სისტემაში შექმნილი მომხმარებლისა და ჯგუფის შესახებ ინფორმაცია /etc/passwd და /etc/group ფაილებში ინახება.

/etc/passwd-ში თითოეულ ხაზში თითოეული მომხმარებლის მონაცემებია მოცემული, ისინი დაყოფილია ველებად, სვეტებად და ერთმანეთისგან ორწერტილით (:) არის გამოყოფილი. პირველ ველში წერია მომხმარებლის სახელი, მეორეში მისი პაროლი, მესამეში UID, მეოთხეში GID, მეხუთეში დამატებითი ინფორმაცია მომხმარებლის შესახებ – ამ ველის შევსება ნებაყოფლობითია, ის შეიძლება ცარიელი იყოს, მეექვსეში მომხმარებლის პირადი დირექტორიაა ჩაწერილი. ის დირექტორია, რომელშიც, ნაგულისხმევი მნიშვნელობით, აღმოვჩნდებით ხოლმე სისტემაში შესვლისას, მეშვიდეში კი ინტერპრეტატორია მოცემული, shell, რომელიც უნდა გაეშვას მომხმარებლის მიერ ტერმინალის გახსნისას. Linux-ისთვის, ნაგულისხმევი მნიშვნელობით, ის bash-ია, როგორც უკვე ვახსენეთ.

```
achiko@debian:~$ cat /etc/passwd  
root:x:0:0:root:/root:/bin/bash  
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin  
bin:x:2:2:bin:/bin:/usr/sbin/nologin  
sys:x:3:3:sys:/dev:/usr/sbin/nologin  
sync:x:4:65534:sync:/bin:/bin/sync  
...
```

დიდი დაკვირვება არ დაგვჭირდება იმის დასადგენად, რომ ამ ფაილში იმაზე მეტი მომხმარებელია შექმნილი, ვიდრე ერთი შეხედვით ჩანს. ვნახოთ რა დანიშნულება აქვთ მათ. მომხმარებელი სახელად root სისტემის ადმინისტრატორია და მას ყველაფრის უფლება აქვს. ამის გამო, მკაცრად რეკომენდირებულია ადმინისტრატორის უფლებებით მხოლოდ მაშინ შევიდეთ სისტემაში, როდესაც ამის რეალური საჭიროება არსებობს. იმ მომხმარებლებს, რომლებიც თავიანთ პირადი დირექტორიაში საკუთარი პაროლის საშუალებით შედიან – ნამდვილ მომხმარებლებს უწოდებენ, ხოლო დანარჩენს – სისტემის მომხმარებლებს. ისინი სისტემის უსაფრთხოებისა და მისი გამართულებული მართვისთვისაა შექმნილი. სწორედ ასეთი მომხმარებლები ჭარბობს, როგორც წესი, /etc/passwd-ში.

/etc/group ფაილშიც ჯგუფების შესახებ ინფორმაცია სვეტებად არის ორგანიზებული. პირველ ველში ჯგუფის სახელია მოცემული, მეორეში – პაროლი. ჯგუფის

პაროლი მაშინაა საჭირო, როდესაც მომხმარებელს ამ ჯგუფში გაწევრიანება სურს, მესამეში კი იმ მომხმარებლების სახელებია, რომლებიც ამ ჯგუფშია გაწევრიანებული.

```
achiko@debian:~$ cat /etc/group
root:x:0:
...
cdrom:x:24:achiko
floppy:x:25:achiko
tape:x:26:
sudo:x:27:
...
```

თუ კარგად დაუკვირდებით `/etc/passwd` ფაილის შიგთავსს, მეორე სვეტში პაროლის ნაცვლად `x` სიმბოლოა ჩაწერილი (ისევე როგორც `/etc/group`-ში). ეს იმას არ ნიშნავს, რომ მომხმარებლის პაროლი „x“ ასო-ნიშანია, არამედ იმას, რომ პაროლები სხვაგან, `/etc/shadow` ფაილში ინახება. ჯგუფის პაროლების შემთხვევაში – `/etc/gshadow`-ში.

საკუთარი პაროლის შეცვლა ყველა მომხმარებელს შეუძლია, სხვა მომხმარებლების პაროლების შეცვლა კი მხოლოდ `root`-ის უფლებებითაა შესაძლებელი.

ვნახოთ, როგორ შეგიძლიათ თქვენივე პაროლის შეცვლა. ზოგადად, უსაფრთხოებისათვის რეკომენდებულია რეგულარულად ცვალოთ პაროლები.

```
achiko@debian:~$ passwd
Changing password for achiko.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
```

ეს ბრძანება ჯერ მოგთხოვთ არსებული პაროლის შეყვანას (უსაფრთხოების მიზნებიდან გამომდინარე. ხომ შეიძლება დროებით დატოვოთ თქვენი სამუშაო გარემო და ამ დროს სხვა ადამიანმა სცადოს თქვენი პაროლის შეცვლა?) შემდეგ კი ახალი პაროლის დაფიქსირებას.

კოტყვით და, ცნობისმოყვარეობამ წაგძლიათ და მოისურვებთ `/etc/shadow` ფაილში ჩაწერილ სხვა მომხმარებლების პაროლებს შეავლოთ თვალი. ვნახოთ რა გამოვა:

```
achiko@debian:~$ cat /etc/shadow
cat: /etc/shadow: Permission denied
```

აღმოჩნდა, რომ სხვა მომხმარებლების პაროლების ნახვის უფლება არ გვაქვს. ეს საქმე მხოლოდ ადმინისტრატორის პრეროგატივაა. მართალია, პაროლები ფაილში არა პირდაპირი, არამედ დაშიფრული სახით ინახება, თუმცა, უსაფრთხოების მიზნით, ამ ინფორმაციის საჯაროდ გამოჩენა არარეკომენდებულია. ამ ფაილის შიგთავსის დეტალურ განხილვას მოგვიანებით დავუბრუნდებით.

8.2 ძირითადი ატრიბუტები

მიმოვიხილოთ, წვდომის რა უფლებები არსებობს და როგორ ვნახოთ ამა თუ იმ ფაილსა თუ დირექტორიაზე ვის, რისი უფლება აქვს. ამაში კვლავ `ls` ბრძანება დაგვეხმარება.

-l ოფციით დეტალურ ინფორმაციას ვიღებთ ფაილის შესახებ, სადაც ჩვენთვის საინტერესო ინფორმაციაც არის მოცემულია.

```
achiko@debian:~$ ls -l LICENSE.txt
-rw-r--r-- 1 achiko achiko 12767 Oct 10 2017 LICENSE.txt
```

თუ დირექტორიის შესახებ გვსურს მსგავსი დეტალური ინფორმაციის მიღება, მაშინ -l ოფციასთან ერთად -d ოფციაც უნდა გამოვიყენოთ. მის გარეშე გაშვებული ბრძანება არა თავად დირექტორიის, არამედ ამ დირექტორიაში არსებული ფაილებისა და დირექტორიების შესახებ მოგვცემს ინფორმაციას.

```
achiko@debian:~$ ls -ld .
drwxr-xr-x 18 achiko achiko 4096 Jun 6 17:50 .
```

```
achiko@debian:~$ ls -l .
total 84
drwxr-xr-x 2 achiko achiko 4096 Oct 6 2017 Desktop
drwxr-xr-x 2 achiko achiko 4096 Oct 6 2017 Documents
drwxr-xr-x 2 achiko achiko 4096 Oct 10 2017 Downloads
-rw-r--r-- 1 achiko achiko 12767 Oct 10 2017 LICENSE.txt
...
```

ეგრანზე გამოსულ მონაცემებში თითოეული ხაზი ერთი კონკრეტული ფაილის მონაცემებს შეიცავს. პირველი ველის პირველი სიმბოლო აღნიშნავს თუ რა ტიპის ფაილთან გვაქვს საქმე. ტირეთი (-) ჩვეულებრივი ფაილი აღნიშნავს, **d** სიმბოლოთი კი დირექტორიის ტიპის ფაილი. შემდეგი 9 სიმბოლოდან, რომლებსაც ფაილის ატრიბუტები ჰქვიათ, 3 თავად ამ ფაილის მფლობელის წვდომის უფლებებს გამოხატავს თავისთავად ფაილზე, შემდეგი 3 გვეუბნება რა უფლებები აქვთ მფლობელის ჯგუფის სხვა წევრებს ამ ფაილზე, ხოლო ბოლო 3 კი ყველა სხვა დანარჩენი მომხმარებლების უფლებებს აღნიშნავს. ხაზის მესამე და მეოთხე ველში, შესაბამისად, ამ ფაილის მფლობელი და ჯგუფია მოცემული.

ავხსნათ, რას ნიშნავს ეს ატრიბუტები ფაილისთვის და დირექტორიისთვის ცალ-ცალკე.

ატრიბუტი	ფაილისთვის	დირექტორიისთვის
r	გვაქვს ფაილის შიგთავსის ნახვის უფლება, ანუ შეგვიძლია ვნახოთ თუ რა ინფორმაცია წერია მასში.	გვაქვს დირექტორიის შიგთავსის ნახვის უფლება, ანუ შეგვიძლია ვნახოთ თუ რა ფაილები და დირექტორიებია მასში შენახული.
	მაგალითად: cat file	მაგალითად: ls directory/

w	გვაქვს როგორც ფაილის შიგთავსის შეცვლის, ასევე დამატების, წაშლის და შემდეგ მისი შენახვის უფლება.	გვაქვს დირექტორიის შიგთავსში ცვლილების შეტანის უფლება: მასში ფაილის ან დირექტორიის შექმნის, წაშლის ან სახელის გადარქმევის.
	მაგალითად: cat file1 > file2 vi file	მაგალითად: touch directory/file rm -r dir/{f1,dir1}
x	ფაილი აღიქმება, როგორც პროგრამა და შესაძლებელი ხდება მისი გაშვება. სკრიპტების შემთხვევაში ფაილს დამატებით კითხვის უფლება აუცილებლად უნდა ჰქონდეს.	გვაქვს ამ დირექტორიაში გადასვლის, მასში შესვლის უფლება.
	მაგალითად: cat file1 > file2	მაგალითად: cd directory/

ყურადღება!

ფაილის წაშლის უფლება მოიცემა არა თავად ფაილის ატრიბუტებით, არამედ იმ დირექტორიის ატრიბუტებით, რომელშიც ეს ფაილი იმყოფება.

უფლებების განაწილების რამდენიმე მაგალითები:

-rwx-----	ამ ფაილზე მხოლოდ მფლობელს აქვს კითხვის, შეცვლის და შესრულების უფლება.
-rw-----	ამ ფაილზე მხოლოდ მფლობელს აქვს კითხვისა და შეცვლის უფლება.
-rw-r--r--	აქ კითხვის უფლება ყველა მომხმარებელს აქვს, ცვლილებების გაკეთების – მხოლოდ მფლობელს, შესრულების კი – არავის.
drwxr-x---	ამ დირექტორიაზე მფლობელს ყველაფრის უფლება აქვს, მისი ჯგუფის სხვა წევრებს კი მხოლოდ დათვალიერების და მასში გადასვლის.

როგორ შევუცვალოთ ფაილს ატრიბუტები?

ამის გაკეთება **chmod** ბრძანებით შეგვიძლია. უნდა აღინიშნოს, რომ ფაილებზე უფლებების შეცვლა მხოლოდ მფლობელს ან სისტემის ადმინისტრატორს (root მომხმარებელს) შეუძლია. **chmod** ბრძანება ორი სხვადასხვა სინტაქსური ფორმით იწერება – **ციფრული** და **სიმბოლური**.

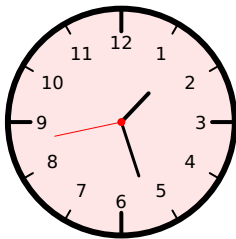
ციფრული გამოსახვისთვის დაგვჭირდება რიცხვების წარმოდგენის 8-ობითი სისტემის ცოდნა. ალბათ, გსმენიათ, რომ კომპიუტერში ხშირად გამოიყენება ინფორმაციის ჩაწერის ორობითი, რვაობითი და თექვსმეტობითი სისტემები. ყველა ეს სისტემა რიცხვების მოცემის სხვადასხვა ფორმაა.

მოდით, თემიდან ოდნავ გადავუხვიოთ და ავხსნათ თუ როგორ ხდება ინფორმაციის წარმოდგენა ციფრულ სამყაროში, ანუ კომპიუტერულ სისტემებში.

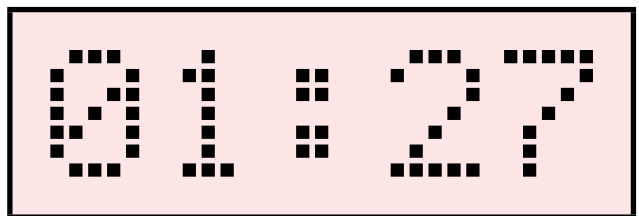
ჩვენ, ადამიანებს, გაგვიმართლა და ვცხოვრობთ არა ციფრულ, არამედ ანალოგურ სამყაროში. აქ, ჩვენ გვაქვს ფერების უსასრულო რაოდენობა, ხმის უსასრულო რაოდენობის ტონალობა, უსასრულო გრადაცია სურნელსა და გემოში და ა.შ. ასეთ სამყაროში ნებისმიერი ანალოგური ნიშნის უსასრულო რაოდენობა შესაძლებელი მნიშვნელობების მქონეა. ტერმინი ანალოგურიც იქიდან მოდის, რომ ჩვენ ნიშნის რაიმე მახასიათებლის გაზომვისას შესატყვისობას, ანალოგიას ვაკეთებთ.

ციფრულ სამყაროში კი ყველაფერს სასრული, დისკრეტული მნიშვნელობა აქვს. ეს იმას ნიშნავს, რომ აქ გვაქვს მინიმალური საზომი ერთეული და მასზე პატარის გაზომვა აღარ შეგვიძლია. ციფრულ სამყაროში ყველაფერი რიცხვებით გამოისახება. სწორედ ამიტომ ვიყენებთ ტერმინს „ციფრული“. მაგალითად, ხმა, გამოსახულება თუ ტექსტი კომპიუტერებში გარკვეული მექანიზმით რიცხვებში გარდაიქმნება. შემდეგ ხდება ამ რიცხვების შენახვა, დამუშავდება, გადაცემა, აღდგენა მის თავდაპირველ ვერსიამდე და სხვა.

ანალოგურსა და ციფრულს შორის განსხვავება ნათლად ჩანს ანალოგური (სურ. 8.1) და ციფრული საათების (სურ. 8.2) შედარებისას.



სურ 8.1: ანალოგური საათი



სურ 8.2: ციფრული საათი

ანალოგურ საათში წუთების ისარი ნელ-ნელა, უწყვეტად გადადის. ციფრულში კი წუთები რიცხვებით მოიცემა და ის ნახტომისებურად გადადის ერთიდან მეორეზე. მსგავსი შედარების გაკეთება სხვა საყოფაცხოვრებო ინსტრუმენტების (თერმომეტრი, წნევის საზომი აპარატი, სასწორი და სხვ) მაგალითებზეც შეგვიძლია.

დღესდღეობით არსებობს ტენდენცია, რომ ყველაფერი ციფრულ სამყაროში გადავიყვანოთ. ეს ტენდენცია იმითაა განპირობებული, რომ რიცხვებში გადაყვანილი ინფორმაციის შენახვა გაცილებით ადვილი და საიმედოა.

ჩვენი რეალური სამყაროდან ინფორმაცია რომ გავაციფრულოთ და შევინახოთ კომპიუტერში და პირიქით, კომპიუტერიდან რომ წავიკითხოთ ეს ინფორმაცია, დაგვჭირდება შესაბამისი გადაწყვენი მოწყობილობები და პროგრამები, ისეთი, როგორიცაა მიკროფონი – ხმის ჩასაწერად და ყურსასმენი – ხმის მოსასმენად, კამერა და სკანერი – გამოსახულების მისაღებად, პრინტერი და ეკრანი – გამოსახულების სანახავად და ა.შ.

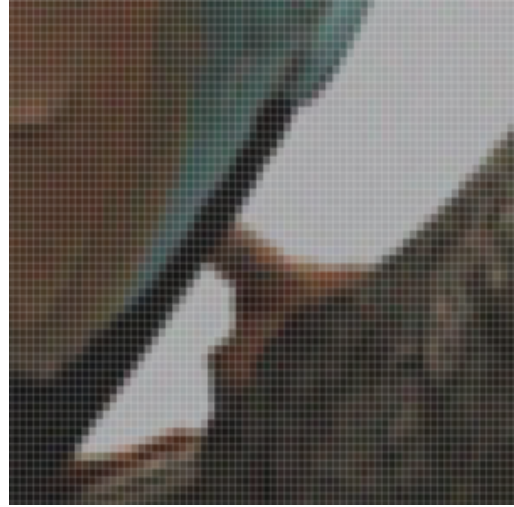
თუმცა, ერთი კი უნდა დავიმახსოვროთ: ანალოგური სამყაროდან ციფრულში გადაყვანილი „ასლი“ ვერასოდეს იქნება ისეთივე ხარისხიანი, როგორიც ის ორიგინალ ვერსიაში იყო. ასეთი გადაყვანისას გარკვეული ინფორმაცია სამუდამოდ იკარგება. ამის თვალსაჩინოდ სანახავად ავიღოთ ნებისმიერი კადრის ანალოგური (სურ. 8.3) და მისი ციფრული ვერსია.

ციფრულ ვერსიაში თუ სურათის რომელიმე ნაწილს გავადიდებთ, ნათლად დავინახავთ წერტილების (პიქსელების) ბადეს (სურ. 8.4). რაც უნდა კარგი ხარისხის ფოტოაპარატი ავიღოთ, რამდენჯერმე გადიდებისას პიქსელებს მაინც დავინახავთ. ანალოგურ ვერსიაში კი გადიდებისას (მაგალითად, დურბინით ან მსგავსი მოწყობილობით) გამოსახულებას ხარისხი არ შეეცვლება. რაც უნდა დიდი იყოს გადიდება, წერტილებამდე თეორიულადაც კი ვერ დავალთ.

იგივეა ხმის ჩაწერის დროსაც. კომპიუტერში მიკროფონით ხმის ჩაწერისას მიმდინარეობს სიგნალის ანალოგურიდან ციფრულში გადაყვანის პროცესი. ამ დროს გაზომვა



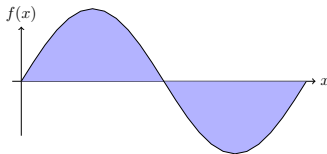
სურ 8.3: სრული სურათი



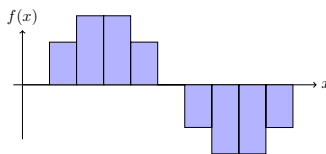
სურ 8.4: გადიდებული ფრაგმენტი

ხდება რეგულარულ ინტერვალში და არა უწყვეტად. შესაბამისად, გარკვეული ინფორმაცია აქაც იკარგება. რაც უფრო პატარა ინტერვალში (ანუ, უფრო დიდი სიხშირით) ვიწერთ ხმას, მით უფრო ზარისხიანი იქნება ჩანაწერი.

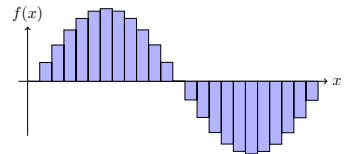
თვასაჩინოებისთვის წარმოვიდგინოთ შემდეგი გრაფიკები. ხმა ტალღებით ვრცელდება სამყაროში. მასი აღწერა კი მათემატიკური ფუნქციებით შეგვიძლია. წარმოვიდგინოთ მისი გრაფიკი. ანალოგურ ვერსიაში ხმას უწყვეტი ფორმა აქვს (სურ. 8.5). ციფრულში კი ამ ფორმის მიახლოებული ვერსია გვაქვს (სურ. 8.6). რაც მეტია ჩაწერის სიხშირე, მით მეტადაა მიახლოებული ჩანაწერი ორიგინალ ხმას. (სურ. 8.7)



სურ 8.5: ბუნებაში არსებული, ანალოგური, ხმა.



სურ 8.6: ცუდ ზარისხში ჩაწერილი ხმის ვერსია.



სურ 8.7: უკეთეს ზარისხში ჩაწერილი ხმა.

კლავიატურიდან შეტანილი ასო-ნიშნებიც საბოლოოდ რიცხვებში დაიყვანება. თუმცა, ამ რიცხვების უზარმაზარ ერთობლიობაში ისმის კითხვა, თუ სად მთავრდება ერთი ასო-ნიშნის შესაბამისი რიცხვი და სად იწყება მეორე, რამდენი რიცხვი შეესაბამება ერთ ასო-ნიშანს და სხვა. ამ კითხვებზე პასუხი მხოლოდ ჩვენზეა დამოკიდებული. ჩვენი გადასაწყვეტია, როგორ შესაბამისობას გავუკეთებთ, როგორ წარმოდგენაში გადავიყვანოთ. ყოველ ჯერზე სხვადასხვა წარმოდგენის სისტემის მოფიქრება და გამოყენება ინფორმაციას სხვებისთვის ძნელად გასაგებს გასდის. ის მხოლოდ ჩვენ გვეცოდინება თუ სხვას არ გავუზიარეთ. ამიტომ, ყველა თანხმდება, რომ თუ ერთად მუშაობა სურთ, წინასწარ დადგენილი წესით მოხდეს ინფორმაციის წარმოდგენა ერთი სისტემიდან, ერთი კოდირებიდან მეორეზე გადაყვანა. ერთ-ერთი ასეთი მარტივი კოდირების სქემაა სწორედ ASCII კოდირება.

ჩვენ, ადამიანები, დათვლას ათობითი სისტემაში ვართ მიჩვეულები.

რას უდრის 8+7 ? თუ თქვენი პასუხია **15**, მაშინ ათობით სისტემაში აზროვნებთ. ეს,

ალბათ, იქიდან გამომდინარეობს, რომ დაბადებიდან 10 თითი აქვს ადამიანს და თვლის სწავლასაც სწორედ თითების დახმარებით იწყებს. 10-ობით სისტემაში, როგორც უკვე ვიცით, სულ 10 სხვადასხვა ციფრი არსებობს. ესენია: **0, 1, 2, 3, 4, 5, 6, 7, 8** და **9** და მათი საშუალებით გამოვსახავთ ყველა დანარჩენ რიცხვებს. ათობითი სისტემა მსოფლიოში ყველაზე გავრცელებული სისტემაა, თუმცა ძველად ქართველები (და არამართო ქართველები) თვლისას ოცობით სისტემას იყენებდნენ. ამას მოწმობს რიცხვების აღმნიშვნელი დღემდე შემორჩენილი ტერმინები (ორმოცი, ანუ ორი ოცი), 60 (სამოცი, ანუ სამი ოცი) და 80 (ოთხმოცი, ანუ ოთხი ოცი).

ახლა მოდით, დროებით დავივიწყოთ ათობითი სისტემა, გადავერთოთ ორობით სისტემაზე, ამ სისტემაში დავიწყოთ აზროვნება და რიცხვების თვლა. რადგან აქ სულ ორი ციფრი გვაქვს: **0** და **1**, მათი საშუალებით უნდა გამოვსახოთ ყველა რიცხვი, ასე: **0, 1, 10, 11, 100, 101, 110, 111, 1000 ...**

თვლის რვაობით სისტემაში კი სულ 8 ციფრი გვაქვს: **0-დან 7-ის** ჩათვლით. ამ სისტემაში ჩაწერილი რიცხვებია: **0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20 ...**

თექვსმეტობით თვლის სისტემაში 16 აღმნიშვნელი ასო-ნიშანი გვაქვს **0-დან 9-ის** ჩათვლით და **A-დან F-ის** ჩათვლით. **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.**

ვნახოთ, როგორ გამოისახება ათობითი თვლის სისტემის ერთერთი რიცხვი, მაგალითად, 12, სხვადასხვა სისტემებში: $12_{10} \Leftarrow 1100_2 \Leftarrow C_{16} \Leftarrow 14_8$

ერთი თვლის სისტემიდან მეორეზე რიცხვების გადაყვანა ჩვენი წიგნის ფარგლებს ცდება. ცდებისთვის შეგიძლიათ ონლაინ გადამყვანები გამოიყენოთ. ისინი ინტერნეტში მრავლადაა წარმოდგენილი.

ორობითი სისტემა გასაგებია, რომ ძალიან მნიშვნელოვანია. კომპიუტერს ხომ თვლის მხოლოდ ეს სისტემა ესმის და მასში ნებისმიერი ინფორმაცია ორობით კოდში მოიცემა და გაიგება. მაშ, რაღაში გვჭირდება რვაობითი და თექვსმეტობითი სისტემები?

მოდით, მაგალითს მივმართოთ. კომპიუტერში ფერების წარმოდგენის ბევრი გზა არსებობს. ერთ-ერთია ცნობილი RGB (**Red**, **Green**, **Blue**) მოდელი. მასში წითელი, მწვანე და ლურჯი ფერების სხვადასხვა დოზით შერევით უამრავი კომბინაციის მიღება შეიძლება. კომპიუტერების დისპლეის თითოეული წერტილიც ამ სამი ფერით ხასიათდება.

წარმოვიდგინოთ, რომ სამივე ფერის – წითლის, მწვანისა და ლურჯის, სათითაოდ 256 გრადაციად დაყოფა შეიძლება. გამოდის, რომ ამ გრადაციების შერევის ყველა შესაძლო კომბინაციით $256^3 = 16\,777\,216$ რაოდენობის სხვადასხვა ფერის მიღება შეგვიძლია.

ავილოთ ამ ფერების ერთ-ერთი შეხამება. მაგალითად, ხაკისფერი ორობით სისტემაში 24 ცალი ციფრით (0 და 1-ების კომბინაციით) ასე მოიცემა: **110000111011000010010001**

დამეთანხმებით, ცოტა ძნელია ფერის ასეთი ფორმით დამახსოვრება, ამიტომ მოიგონეს კომპიუტერში უფრო კომპაქტურად ჩაწერის სხვა გზები. ხაკისფერი თექვსმეტობით სისტემაში ასე გამოიყრება: **C3B091** (აქედან წითელი ფერის კომპონენტი – **C3**, მწვანის – **B0**, ლურჯის კი **91**). ცხადია, ასეთი აღნიშვნა უფრო გვიმარტივებს საქმეს.

RBG მოდელის გარდა, ცნობილია CMYK (**Cyan**, **Magenta**, **Yellow**, **Black**) მოდელი. ის დაფუძნებულია ამ ოთხ ფერზე (ცისფერი, მუქი ვარდისფერი, ყვითელი და შავი) და მას ძირითადად პრინტერები გამოიყენებენ.

სადღეისოდ, თვლის თექვსმეტობითი სისტემა უფრო გავრცელებულია, ვიდრე რვაობითი თვლის სისტემა, თუმცა რვაობით სისტემასაც აქვს თავისი გამოყენება. უფლებების განსასაზღვრად **chmod** ბრძანებაში სწორეს ეს უკანასკნელი გამოიყენება.

მოდით, რვაობითი სისტემის ციფრები ორობითში გადავიყვანოთ. შემდეგ **rx** უფლებებზე შესაბამისობა მოვახდინოთ, ასე: ორობით ჩანაწერში სადაც 1-ია, შესაბამისი ატრიბუტი ჩავსვათ, ხოლო სადაც 0-ია ატრიბუტის ნაცვლად ტირე (-). მივიღებთ შემდეგ სქემას:

რვაობითი ორობითი ატრიბუტი

0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	rx-
6	110	rw-
7	111	rxw

ახლა უფრო მარტივად მისახვედრი იქნება თუ როგორ გამოისახება უფლებები ორობითი და რვაობითი სისტემების დახმარებით.

მოვიყვანოთ რამდენიმე მაგალითი. თუ გვსურს, რომ ფაილის უფლებები შემდეგნაირად გამოიყურებოდეს – `rwxr-xr--`, მაშინ `chmod` ბრძანებას საკმარისია გადავცეთ 764 (7 როგორც `rw`, 6 როგორც `rw-`, 4 როგორც `r--`). საბოლოოდ ბრძანება ასეთ სახეს მიიღებს:

```
achiko@debian:~$ chmod 764 file
```

`rw-r-----` უფლებების მისაცემად კი შემდეგი ბრძანებაა საჭირო:

```
achiko@debian:~$ chmod 640 file
```

`chmod` ბრძანება, როგორც ვთქვით, ციფრულის გარდა, სიმბოლურ ჩანაწერსაც იგებს. მის სინტაქსში სიმბოლოებით ნათლად იწერება თუ ვის რა უფლებას ვუმატებთ, ვართმევთ ან ვანიჭებთ.

ატრიბუტი	მნიშვნელობა
u	u, როგორც user, ნიშნავს მფლობელს.
g	g, როგორც group, ნიშნავს მფლობელის ჯგუფს.
o	o, როგორც other, ნიშნავს დანარჩენს მომხმარებლებს.
a	a, როგორც all, u, g და o-ს კომბინაციას აღნიშნავს, ანუ ერთბაშად ყველა მომხმარებელს.

თუ არ არის მითითებული ვის ეცვლება უფლება, მაშინ, ნაგულისხმევი მნიშვნელობით, a ანუ ყველა მომხმარებელი მოიაზრება.

ფაილებზე უფლების დამატება ხდება „+“ ოპერაციით, წართმევა „-“ ოპერაციით, „=“ ოპერაციით კი მინიჭებას ვახორციელებთ. მინიჭებისას მხოლოდ ის უფლება მიენიჭება რასაც მივუთითებთ, დანარჩენი კი ავტომატურად მოშორდება. თავად უფლებები კი ჩვენთვის უკვე ნაცნობი სიმბოლოებით აღინიშნება `r`, `w` და `x`.

`chmod` ბრძანების მაგალითებია:

```
achiko@debian:~$ chmod u+x filename
achiko@debian:~$ chmod u+r,g-x filename
achiko@debian:~$ chmod u+rw,go=rx filename
achiko@debian:~$ chmod +r filename
```

8.3 სპეციალური ატრიბუტები

კითხვის, ჩაწერისა და შესრულების უფლებების გარდა, shell-ში კიდევ სამი სპეციალური უფლება არსებობს. ესენია:

- SUID (Set User IDentification)
- SGID (Set Group IDentification)
- Sticky bit

ჩვენ ზემოთ აღვნიშნეთ, რომ მომხმარებელს უფლება აქვს თავისი პაროლი შეცვალოს. ვთქვით ისიც, რომ ეს პაროლები /etc/shadow-ში ინახება (დამიფრული სახით). გამოდის, რომ მომხმარებელს ცვლილების შეტანის უფლება ჰქონია ამ ფაილში.

გადავამოწმოთ და დეტალურად ვნახოთ, ვის რა უფლება აქვს ამ ფაილზე:

```
achiko@debian:~$ ls -l /etc/shadow
-rw-r----- 1 root shadow 8598 oct. 27 2017 /etc/shadow
```

/etc/shadow, როგორც ჩანს, ეკუთვნის root-ს. ადმინისტრატორია ამ ფაილის მფლობელი და მხოლოდ მას აქვს ამ ფაილის შიგთავსის შეცვლის უფლება. ჰმმ... საინტერესოა, მაშ როგორ ვცვლით ჩვენს პაროლებს?

ამ „გაუგებრობის“ გასარკვევად, ვნახოთ, რა უფლებები აქვს passwd ბრძანებას. ეს ბრძანებაც ხომ ფაილია, რომელზეც x ატრიბუტია გააქტიურებული. passwd ბრძანება /usr/bin დირექტორიაში ინახება. ზოგადად, ბრძანების აბსოლუტური გზის გასაგებად which ბრძანება გამოიყენება.

```
achiko@debian:~$ which passwd
/usr/bin/passwd
```

```
achiko@debian:~$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 59680 may 17 2017 /usr/bin/passwd
```

კარგად დააკვირდით და აღმოაჩენთ, რომ მფლობელის უფლებებში x-ის ნაცვლად s ატრიბუტია მოცემული. სწორედ ესაა SUID. როდესაც SUID გააქტიურებულია ბრძანებაზე, მომხმარებელი მას არა საკუთარი, არამედ მომხმარებლის უფლებებით გაუშვებს. /etc/passwd ფაილის მფლობელი არის root. root-ს კი, როგორც ვნახეთ, /etc/shadow-ში ცვლილებების შეტანის უფლება აქვს. ახლა უკვე გასაგებია, თუ რატომ შეუძლია მომხმარებელს საკუთარი პაროლების შეცვლა.

Setuid bit (SUID) ატრიბუტის გააქტიურებას აზრი აქვს მხოლოდ გამშვებ ფაილებზე, ანუ ბრძანებებზე. ტექნიკურად ამ ატრიბუტის მინიჭება ასე ხდება:

```
achiko@debian:~$ chmod u+s program_file
```

ციფრული მეთოდით ასე შეიძლება:

```
achiko@debian:~$ chmod 4555 program_file
```

მინიჭების შემდეგ, `ls -l` ბრძანების შედეგში მფლობელის გაშვების უფლება ანუ `x` ატრიბუტი, პატარა `s`-ად გადაკეთდება. ხოლო თუკი `x` ატრიბუტი მანამდე მინიჭებული არ ჰქონდა `program_file`-ს, მაშინ დიდ `S`-ად.

SUID ბიტის მოხსნა შეიძლება ასე:

```
achiko@debian:~$ chmod u-s program_file
```

SGID უფლება SUID -ის მსგავსია. განსხვავება ისაა, რომ როდესაც SGID გააქტიურებული აქვს ბრძანებას, მის გაშვებას უკვე ამ ბრძანების ჯგუფის უფლებებით ვახდენთ. SGID ბიტის მინიჭება/მოხსნა იდენტურად ხდება.

```
achiko@debian:~$ chmod g+s program_file  
achiko@debian:~$ chmod g-s program_file
```

ციფრული მეთოდით მინიჭება კი ასე ხდება:

```
achiko@debian:~$ chmod 2555 program_file
```

ამ შემთხვევაში ბრძანების უფლებებში ჯგუფის `x` ატრიბუტი შეიცვლება `s`-ად.

SGID უფლებას თავისი დატვირთვა აქვს დირექტორიისთვისაც. როდესაც SGID ჩართულია, მაშინ ამ დირექტორიაში შექმნილი ფაილები მიეკუთვნებიან იმავე ჯგუფს, რასაც ეკუთვნის თავად ეს დირექტორია და არა იმ ჯგუფს, რომელსაც მომხმარებელი მიეკუთვნება. ასეთი ფუნქციონირება გამოიყენება დირექტორიის გაზიარების დროს მრავალი მომხმარებლის მიერ.

SGID ბიტის ჩართვა/გამორთვა დირექტორიაზე ასე ხდება:

```
achiko@debian:~$ chmod g+s directory/
```

```
achiko@debian:~$ chmod g-s directory/
```

Sticky bit (სტიკი ბიტი) ატრიბუტიც გამოიყენება გაზიარებული დირექტორიებისთვის, როგორიცაა `/tmp`. მასში ყველა მომხმარებელს შეუძლია შექმნას ფაილები, მაგრამ მათი წაშლა მხოლოდ მათ მფლობელს შეეძლება.

მაგალითად, მომხმარებელმა `mishka`-მ შექმნა ფაილი `/tmp/file_mishka`, მეორე მომხმარებელს `nini`-ს არ შეუძლია წაშალოს ეს ფაილი მიუხედავად იმისა, რომ `/tmp` დირექტორიაზე `777` უფლებებია მინიჭებული. `nini` იმ შემთხვევაში შეძლება `file_mishka`-ს წაშლას, `/tmp` დირექტორიას Sticky bit მინიჭებული რომ არ ჰქონოდა.

ტექნიკურად ამ ბიტის გააქტიურება/გამორთვა ასე ხდება:

```
achiko@debian:~$ chmod +t directory/
```

```
achiko@debian:~$ chmod -t directory/
```

ან ციფრული მეთოდით ასე:

```
achiko@debian:~$ chmod 1777 directory/
```

თავი 9

პროცესები

თანამედროვე ოპერაციული სისტემები მრავალმომხმარებელიანის გარდა, მრავალამოცანიანიცაა. ეს იმას ნიშნავს, რომ სისტემას ერთდროულად რამდენიმე პროგრამის შესრულება შეუძლია. პროგრამა არის კომპიუტერისთვის გასაგები ინსტრუქციების ერთობლიობა და ის ინახება მუდმივ დამამახსოვრებელ მეხსიერებაში, მყარ დისკზე. მისი შესრულებისას ეს ინსტრუქციები იკითხება მყარი დისკიდან და კოპირდება ოპერაციულ მეხსიერებაში. პროგრამის ასეთ გაშვებულ ფორმას პროცესი ჰქვია. მომხმარებლების მიერ გაშვებული პროცესების გარდა, მეხსიერებაში ბევრი პროცესი თავად სისტემის მიერაა გაშვებული და მათი საშუალებით იმართება სისტემა.

vi ბრძანების გაშვება, მაგალითად, გამოიწვევს ერთი პროცესის შექმნას. მაგრამ თუ vi ბრძანება სისტემაში შემოსულმა 8 მომხმარებელმა გაუშვა, მაშინ 8 სხვადასხვა პროცესი იარსებებს მეხსიერებაში.

პროცესი იდენტიფიცირდება უნიკალური ნომრით, პროცესის იდენტიფიკატორით (Process ID), მოკლედ PID. არცერთ სხვა პროცესს არ შეუძლია ამ ნომრის - PID-ის მიღება, სანამ ეს პროცესი გაშვებულია. თუმცა მისი დასრულების შემდეგ ნომერი გათავისუფლდება და შესაძლებელია ის სხვა პროცესმა გამოიყენოს.

PID-ის გარდა, პროცესი სხვა ატრიბუტებითაც ხასიათდება. თითოეულ პროცესთან გარკვეული მომხმარებელი და ჯგუფი ასოცირდება: ვინ გაუშვა ეს პროცესი, ვინაა მისი მფლობელი და ჯგუფი. სწორედ ამ ინფორმაციით ირკვევა პროცესს სისტემამ რა რესურსები უნდა გამოუყოს და რა დოზით. მაგალითად ჩვეულებრივი მომხმარებლის მიერ გაშვებულ პროცესებისგან განსხვავებით, root მომხმარებლის მიერ გაშვებულ პროცესებს კომპიუტერის უფრო მეტ რესურსებთან აქვთ წვდომა.

მარტივი პროგრამის გაშვებისას, როგორც წესი, მისი შესაბამისი ერთი პროცესი იქმნება. კომპლექსური პროგრამის - აპლიკაციის - გაშვებამ კი შეიძლება რამდენიმე პროცესი შექმნას მეხსიერებაში.

პროცესის გაშვება ორი გზით არის შესაძლებელი - წინა და უკანა პლანზე გაშვებით. წინა პლანზე გაშვებული პროცესები (foreground processes) კონტროლდება ტერმინალიდან. უფრო მარტივად რომ ვთქვათ, ეს პროცესები მომხმარებლის მიერ - ხელითაა გაშვებული ტერმინალში, ისინი ავტომატურად არ ეშვება. უკანა პლანზე გაშვებული პროცესები (background processes), რომლებსაც ფონურ პროცესებსაც უწოდებენ, არ არის ტერმინალზე მიბმულნი. ისინი არა-ინტერაქციული, სისტემის მიერ ავტომატურად გაშვებული პროცესებია, და როგორც წესი, მომხმარებლის ჩარევას არ მოითხოვს. ფონურ პროცესად ძირითადად დემონები¹, სისტემის სერვისები, ეშვება. ისინი სისტემას ფუნქციონირებაში ეხმარებიან.

¹სიტყვა დემონი (Daemon) ბერძნული მითოლოგიიდან მოდის და აღნიშნავს ზებუნებრივ არსებას, რომელიც უკანა პლანზე დგას უხილავად და დამხმარე ძალას წარმოადგენს. Unix-ის მსგავს სისტემებში პროცესებს, რომლებიც ასე ოპერირებენ დემონებს უწოდებენ. ეს ტერმინი ამ სისტემებში ინსპირირებულია მაქსველის დემონისგან (Maxwell's daemon), რომელიც წარმოსახვითი არსება იყო და მაქსველი მას თავის ექსპერიმენტში თერმოდინამიკის მეორე

პროცესები ერთმანეთისგან სრულიად დამოუკიდებელი არ არის. როდესაც პროცესი იქმნება, ის ინარჩუნებს კავშირს თავის მშობელ პროცესთან, რომლის იდენტიფიკატორი აღინიშნება PPID-ით (Parent PID). თავის მხრივ, ეს უკანასკნელიც თავის მშობელთანაა მიბმული და საბოლოოდ დავდივართ პირველ პროცესამდე, რომელისგანაც ყველა დანარჩენი პროცესი დაიწყო. პირველი პროცესი არის systemd, ძველ სისტემებში - init და მისი PID არის ნომერი 1. ასე ქმნიან პროცესები ზისებრ სტრუქტურას, რომლის ძირიც არის systemd/init.

პროცესების სანახავად ყველაზე ხშირად **ps** ბრძანებას ვიყენებთ. მას უამრავი ოფცია ეხმოს, თუმცა მარტივად მათ გარეშეც ეშვება, ასე.

```
achiko@debian:~$ ps
PID TTY          TIME CMD
 916 pts/0        00:00:00 bash
 934 pts/0        00:00:00 ps
```

შედეგად, თქვენც ორ პროცესს დაინახავთ, **bash** და **ps** შესაბამისი 916 და 934 PID-ებით. TTY სვეტში მოცემულია საკონტროლო ტერმინალის სახელი, ანუ რომელი ტერმინალიდან იქნა გაშვებული ეს ბრძანებები (გრაფიკულ გარემოში ტერმინალის ემულატორებით გაშვებული ტექსტის შეტან/გამოტანის გარემოს პსევდოტერმინალს უწოდებენ და აღნიშნავენ pts-ით (pseudo-ttys), ხოლო გრაფიკული გარემოს უკან გაშვებული ტერმინალი კი tty-თი აღინიშნება), TIME ველში კი მოცემულია ის დრო, რა დროც დაჭირდა პროცესორს (CPU) ამ პროცესის შესასრულებლად ჩვენი მაგალითის შემთხვევაში. როგორც ჩანს, დიდი დატვირთვა არ გამოუწვევია ამ პროცესების გაშვებას. ბრძანებების დიდი რაოდენობა პროცესორის დროის² პატარა ნაწილს მოიხმარს. როგორც მოცემულ მაგალითშია ასახული - 0:00 (დამრგვალებულია წამის სიზუსტით), ამ პროცესებს გასაშვებად პროცესორის დროის ერთი წამიც კი არ დასჭირდა. ოფციების გარეშე **ps** ბრძანება გვანახებს მიმდინარე ტერმინალის სესიაში გაშვებულ პროცესებს. მათ შესახებ უფრო დეტალური ინფორმაციის მისაღებად გავუშვათ:

```
achiko@debian:~$ ps u
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
achiko    897  0.0  0.4  20968  4800 tty1    S+   14:11   0:00 -bash
achiko    916  0.0  0.5  20980  5148 pts/0    Ss   14:12   0:00 -bash
achiko    940  0.0  0.3  38312  3240 pts/0    R+   14:19   0:00 ps u
```

დამატებით გამოჩნდა ის, რომ მომხმარებელ **achiko**-ს გაუხსნია **bash** (achiko-ს მიერ ტერმინალის გაშვებით ის ავტომატურად ეშვება), შემდეგ კი ამ shell-ში გაუშვია ბრძანება „**ps u**“. აგრეთვე, ჩანს, თუ როდის გაუშვა achiko-მ ეს ბრძანებები (START), მათ გაშვებას რა რესურსები დაჭირდა: რამდენად დატვირთა პროცესორი (%CPU), ოპერატიულ მეხსიერების რა ზომა დაიკავა პროცენტულად (%MEM), რა ზომას იკავებს ფიზიკურად მეხსიერებაში კონკრეტულ მომენტში (კილობაიტებში) (RSS – resident set size). VSZ (virtual set size) კი

კანონის ასახსნელად იყენებდა. უნდა აღინიშნოს, რომ Daemon განსხვავდება სიტყვა demon-სგან. Demon - უზილავე ბოროტი სულიერი ქმნილებაა. სწორედ ამიტომ Daemon ფორმულირებული იქნა როგორც „Disk And Execution MONitor“ აკრონიმი, თუმცა ეს შემდგომი ჩანაფიქრი იყო.

²პროცესორის დრო* (CPU time) - CPU (Central processing unit) time ან process time დროის ის ოდენობაა, რასაც CPU ხარჯავს პროგრამის დასამუშავებლად. ის იზომება წამებში ან ტაქტებში/ციკლებში (clock ticks). ტაქტი/ციკლი/ძველა ყველაზე პატარა ერთეულია დროს გამოსახატად კომპიუტერებში. ის პირობითი სიდიდეა და დაკავშირებულია მისი საათის მუშაობასთან. მისი სიჩქარე დამოკიდებულია სისტემაზე და იზომება პერცენტში. მაგალითად, თუ მისი სიჩქარე 2 GHz-ია, ეს ნიშნავს, რომ 1 წამში 2 მილიარდი ტაქტი, ციკლი, ძველა ხდება. CPU კი ინსტრუქციებს ტაქტებში ასრულებს.

გვანახებს მთლიანობაში რამხელა მეხსიერებაა გამოყოფილი ამ პროცესისთვის. STAT ველში პროცესის მდგომარეობა მოიცემა და ის სხვადასხვა ასო-ნიშნით აღინიშნება:

პროცესის მდგომარეობები

R	გაშვებული, მომუშავე პროცესი.
S	დაძინებული პროცესი, რომელიც ინსტრუქციას ელოდება შესასრულებლად.
Z	ზომბი პროცესი. ის აღარაფერზე რეაგირებს (პროცესი დამთავრდა, თუმცა მამა პროცესის მიერ ვერ გაიწმინდა მეხსიერებიდან. როგორც წესი, ასეთი შემთხვევები ცუდად დაწერილ პროგრამების გაშვებისას გვხვდება).
<	მაღალ პრიორიტეტულ პროცესს აღნიშნავს, ანუ უფრო მეტი CPU დრო იხარჯება მის შესრულებაზე.
N	დაბალ პრიორიტეტული.
+	წინა პლანზე არსებული პროცესი.
s	სესიის ლიდერი, კონკრეტულ სესიაში წამყვანი პროცესი.

არამხოლოდ მიმდინარე ტერმინალში, არამედ ჩვენს მიერ გაშვებული ყველა პროცესი რომ ვნახოთ, მაშინ უნდა გავუშვათ: **ps x** ან **ps xu**.

```
achiko@debian:~$ ps x
  PID TTY          STAT TIME COMMAND
   891 ?            Ss    0:00 /lib/systemd/systemd --user
   893 ?            S      0:00 (sd-pam)
   897 tty1        S+    0:00 -bash
   915 ?            R      0:00 sshd: achiko@pts/0
   916 pts/0       Ss    0:00 -bash
   967 pts/0       R+    0:00 ps x
```

ბევრი პროცესი არა ტერმინალზე მიბმული, არამედ უკანა პლანზე, ფონურადაა გაშვებული სისტემის სხვადასხვა მომხმარებლის მიერ. მათ სანახავად **a** ოფცია უნდა გამოვიყენოთ. ყველა მომხმარებლის ყველა პროცესის სანახავად ვუშვებთ:

```
achiko@debian:~$ ps aux
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.6 138892 6752 ?        Ss   14:11   0:00 /sbin/init
root           2  0.0  0.0      0     0 ?        S    14:11   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        S    14:11   0:00 [ksoftirqd/0]
...
```

თუ ბევრი პროცესია გაშვებული და, სავარაუდოდ, თქვენს შემთხვევაშიც ასე იქნება, მათი გვერდებად დასათვალიერებლად **less** ბრძანების გამოყენება შეგიძლიათ:

```
achiko@debian:~$ ps aux | less
```

ალბათ, შეამჩნევდით, რომ `ps` ბრძანებას ოფციები ტირე-ს გარეშე მივუთითეთ. ეს BSD სისტემებში გავრცელებული ოფციების ტიპია და `ps` ბრძანებას ისინი ჩვენთვის უკვე ნაცნობ Unix³ და GNU⁴ ოფციების ტიპთან ერთად კარგად ესმის.

ყველა პროცესის ნახვა ტრადიციული - Unix-ის სტილით ასე მოიცემა:

```
achiko@debian:~$ ps -e
  PID TTY          TIME CMD
    1 ?            00:00:00 systemd
    2 ?            00:00:00 kthreadd
    3 ?            00:00:00 ksoftirqd/0
...
```

ან

```
achiko@debian:~$ ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root          1        0  0  14:11 ?            00:00:00 /sbin/init
root          2        0  0  14:11 ?            00:00:00 [kthreadd]
root          3        2  0  14:11 ?            00:00:00 [ksoftirqd]
...
```

შეგიძლიათ პროცესების ხისებრი სტრუქტურის ნახვაც, ასე:

```
achiko@debian:~$ ps axjf
PPID    PID    PGID    ...  STAT   UID    TIME COMMAND
...
    1    629    629    ...  Ss      0      0:00 /usr/sbin/sshd -D
  629    908    908    ...  Ss      0      0:00  \_ sshd: achiko [priv]
  908    915    908    ...  S      1000    0:00    \_ sshd: achiko@pts/0
  915    916    916    ...  Ss      1000    0:00        \_ -bash
  916   1011   1011    ...  R+      1000    0:00          \_ ps axjf
...
```

ან ასე:

```
achiko@debian:~$ ps -ejH
PID    PGID    SID TTY          TIME CMD
...
  629    629    629 ?            00:00:00 sshd
  908    908    908 ?            00:00:00 sshd
  915    908    908 ?            00:00:00 sshd
  916    916    916 pts/0        00:00:00 bash
 1029   1029    916 pts/0        00:00:00 ps
...
```

³Unix ოფციების ტიპი მოიცემა ერთი ტირეთი. მაგალითად, `-a`

⁴GNU ოფციების ტიპი მოიცემა სიტყვებით და ორი ტირეთი. მაგალითად, `--all`

ასეთი ფორმით უკეთ დავინახავთ ვინაა ამა თუ იმ პროცესის მამა თუ შვილი პროცესი. ხისებრი სტრუქტურის ნახვა შეგვიძლია აგრეთვე **pstree** ბრძანებით.

```
achiko@debian:~$ pstree
systemd+-ModemManager+-{gdbus}
|                                     `--{gmain}
|--NetworkManager+-{gdbus}
|                                     `--{gmain}
|--VGAuthService
|--anacron
|--atd
|--avahi-daemon---avahi-daemon
|--cron
|--dbus-daemon
|--dhclient
|--login---bash---watch
|--minissdpd
|--polkitd+-{gdbus}
|                                     `--{gmain}
|--rsyslogd+-{in:imklog}
|                                     |--{in:imuxsock}
|                                     `--{rs:main Q:Reg}
|--sshd+-sshd---sshd---bash
|                                     `--sshd---sshd---bash---pstree
...

```

თუ გვსურს, რომ რომელიმე კონკრეტული მომხმარებლის მიერ (მაგალითად root) გაშვებული პროცესები ვნახოთ, მაშინ ასეთი ბრძანება უნდა გავუშვათ:

```
achiko@debian:~$ ps -U root -u root
PID TTY          TIME CMD
  1 ?            00:00:00 systemd
  2 ?            00:00:00 kthreadd
  3 ?            00:00:00 ksoftirqd/0
...

```

ოფციები	მნიშვნელობა
-U, --User	ნამდვილი მომხმარებლის პროცესები გამოგვაქვს.
-u, --user	ევექტური მომხმარებლის პროცესები გამოგვაქვს, ანუ ის პროცესები, რომლებიც ამ მომხმარებლის უფლებებით სხვა მომხმარებელმა გაუშვა (იხილეთ SUID ატრიბუტი).

ნაგულისხმევი მნიშვნელობით, **ps** პროცესებს PID-ის მიხედვით ახარისხებს. **--sort** ოფციით ჩვენ შეგვიძლია, სურვილის მიხედვით, დახარისხების წესი შევცვალოთ პროცესორის გამოყენებადობის მიხედვით ასე:

```
achiko@debian:~$ ps aux --sort -pcpu
```

მეხსიერების გამოყენებადობის მიხედვით ასე:

```
achiko@debian:~$ ps aux --sort -pmem
```

ასევე შეგვიძლია ჩვენთვის სასურველი ფორმატით გამოვიტანოთ ეკრანზე პროცესების შესახებ ინფორმაცია -o ოფციის საშუალებით, მაგალითად ასე:

```
achiko@debian:~$ ps -eo pid,user,vsz,rss,comm
PID USER      VSZ   RSS COMMAND
   1 root    138892 6752 systemd
   2 root         0     0 kthreadd
   3 root         0     0 ksoftirqd/0
...
```

შეგვიძლია თან დავაზარისხოთ:

```
achiko@debian:~$ ps -eo pid,user,vsz,rss,comm --sort=-rss
PID USER      VSZ   RSS COMMAND
 423 root    153488 18160 VGAuthService
 433 root    480860 16452 NetworkManager
 273 root     49784  7296 systemd-udev
 908 root    101436 6908  sshd
...
```

ამ მაგალითში, `--sort=-rss` ოფციით, პროცესები დავაზარისხეთ ოპერატიულ მეხსიერებაში მათ მიერ დაკავებული მეხსიერების ზომის კლებადობით მიხედვით. `--sort=rss` ოფციით კი ზრდადობით მოხდებოდა დალაგება.

`ps` ბრძანება ძალიან დიდ და სასარგებლო ინფორმაციას გვაძლევს პროცესების შესახებ, თუმცა ეს მხოლოდ მყისიერი, `ps`-ის გაშვების მომენტში არსებული სურათია. დინამიკურ რეჟიმში პროცესების ნახვა `top` ბრძანებით შეგვიძლია. ის უწყვეტ რეჟიმში (ნაგულისხმევი მნიშვნელობით, 3 წამის ინტერვალით) ანახლევს ინფორმაციას პროცესების შესახებ.

```
achiko@debian:~$ top
top - 14:34:44 up 1 day, 3:31, 3 users, load average: 0.14, 0.06, 0.01
Tasks: 166 total, 1 running, 165 sleeping, 0 stopped, 0 zombie
%Cpu(s): 5.5 us, 1.7 sy, 0.0 ni, 92.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0
KiB Mem : 2033708 total, 808168 free, 563852 used, 661688 buff/cache
KiB Swap: 1046524 total, 1046524 free, 0 used. 1284724 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1	root	20	0	204616	6808	5148	S	0.0	0.3	0:05.16	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.04	kthrea+

3	root	20	0	0	0	0	S	0.0	0.0	0:00.03	ksofti+
5	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworke+
7	root	20	0	0	0	0	S	0.0	0.0	0:03.13	rcu_sc+
8	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_bh
9	root	rt	0	0	0	0	S	0.0	0.0	0:00.13	migrat+
10	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	lru-ad+
11	root	rt	0	0	0	0	S	0.0	0.0	0:00.25	watchd+
12	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cpuhp/0
13	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cpuhp/1
14	root	rt	0	0	0	0	S	0.0	0.0	0:00.23	watchd+
...											

მოლით, გავარჩიოთ თუ რა ინფორმაცია გამოიტანა ამ ბრძანებამ:

- ა) მიმდინარე დრო.
- ბ) uptime - დრო, რაც სისტემის ჩართვის/გაშვების შემდეგაა გასული.
- გ) სისტემაში შემოსული მომხმარებლების რაოდენობა.
- დ) პროცესორის საშუალო დატვირთვა ბოლო 1, 5 და 15 წუთის განმავლობაში.
- ე) პროცესების სხვადასხვა ჯამური რაოდენობები.
- ვ) პროცესორის დროის დეტალური განაწილება: მომხმარებლების მიერ დახარჯული დრო, სისტემის მიერ დახარჯული დრო, შეცვლილი პრიორიტეტით გაშვებული ამოცანებისთვის დახარჯული დრო და ა.შ.
- ზ) ოპერატიული მეხსიერებისა და ვირტუალური მეხსიერების დეტალური განაწილება: სრული, დაკავებული, თავისუფალი, გაზიარებული, ბუფერში/კეშიში არსებული, ხელმისაწვდომი მეხსიერება კილობაიტებში.
- თ) გაშვებული პროცესების ჩამონათვალი.

top ბრძანების საწყისი ნაწილი, თავსართი, სამი სხვა დამოუკიდებელი ბრძანების საშუალებითაც შეგვიძლია ვნახოთ:

- **uptime**
- **free** - ოპერატიული მეხსიერების შესახებ ინფორმაცია. უმჯობესია -h, --human ოფციის გამოყენება, რადგან ის მომხმარებლისთვის უფრო გასაგებ ფორმატს გვთავაზობს.
- **vmstat** - სისტემის რესურსების სტატისტიკური ინფორმაცია.

top პროგრამას რამდენიმე შიდა ბრძანება აქვს და მათი გაშვება კლავიატურიდან სხვადასხვა ასო-ნიშანის დაჭერაზეა მიბმული: M - ახარისხებს პროცესებს დაკავებული მეხსიერების მიხედვით.

M ახარისხებს პროცესებს მეხსიერების გამოყენების მიხედვით.

P უკან ვაბრუნებთ პროცესორით დაღაცების სტილს.

V გვანახებს ზისებრ სტრუქტურას.

h ან ? გვანახებს ყველა შესაძლო ბრძანებას.

ბრძანებები, რომლებიც პროცესებზე მოქმედებენ მათ შესახებ დიდწილ ინფორმაციას /proc ფაილური სისტემიდან იღებენ. თითოეული პროცესი ამ დირექტორიაში ქმნის შესაბამისი PID-ის დასახელებით დირექტორიას და სწორედ მასში ინახება პროცესის შესახებ დეტალური ინფორმაცია.

კეში, ბუფერი, რეგისტრი

კეში მეხსიერების უფრო მაღალ სიჩქარიანი წვდომის ნაწილია. განთავსებულია სტატიკურ მეხსიერებაში - SRAM (SRAM მოთავსებულია CPU-სა და RAM-ს შორის. თუმცა მძლავრი, თანამედროვე პროცესორები ხშირად L1 cache-ის მსგავსად, პროცესორის შიგნითვე ათავსებენ ასეთ CPU L2 და L3 cache-ს), ბუფერი კი უფრო ნელი - დინამიკური მეხსიერების (DRAM) ნაწილია. DRAM - ოპერატიული მეხსიერებაა (მაგალითად, DDR3-ც DRAM-ია).

ბუფერი ძირითადად გამოიყენება პროცესებს შორის მონაცემების შეტან/გამოტანისთვის. ამ დროს პროცესორი შედარებით ნაკლებად იტვირთება. კეში, სწრაფი წვდომის მიზნით, ხშირად გამოყენებული მონაცემების ასლი კეთდება.

კეში შესაძლებელია იყოს მყარი დისკის ნაწილიც. ბუფერი კი მხოლოდ ოპერატიული მეხსიერების ნაწილია.

კლავიატურიდან წვდომა გვაქვს მხოლოდ **ბუფერში** არსებულ მონაცემებზე (მაგალითად, ტექსტის რედაქტირებისას მონიშნული ნაწილის კოპირება და ჩასმა ხდება ბუფერში და ბუფერიდან), კეშიში შენახულ მონაცემებზე - არა.

კეშის გარდა, პროცესორში არის უფრო სწრაფი წვდომის დროებით შესანახი ადგილი - **რეგისტრი**. რაც უფრო მეტია რეგისტრთა რაოდენობა, მით უფრო მძლავრია პროცესორი.

9.1 ფონური რეჟიმი

თუ გრაფიკულ გარემოში მუშაობთ და ტერმინალი გვაქვს გახსნილი, აქედანვე შეგიძლიათ გაუშვათ გრაფიკული პროგრამები - მაგალითად, **xlogo** (ის X window სისტემის ლოგოს გვანახებს უბრალოდ), **xclock**, **gedit** ან **kwrite**.

```
achiko@debian:~$ xclock
```

გაშვების შემდეგ საათი გამოვა პატარა ფანჯარაში, თუმცა ტერმინალში სხვა ბრძანების აკრეფის საშუალება აღარ გვაქვს. ბრძანების ასაკრეფი ველი, ე.წ. shell-ის მოსაწვევი აღარ ჩანს და ეს მანამ გაგრძელდება, სანამ **xclock** პროგრამას არ დავხურავთ. სხვა სიტყვებით რომ ვთქვათ, **xclock** წინა პლანზეა გაშვებული.

დავუბნოვნდეთ ტერმინალს, დავაჭიროთ **Ctrl^C** კლავიშების კომბინაციას. ამით შევწყვეტთ პროგრამის მუშაობას, ფანჯარა გაქრება და shell-ის მოწვევაც გამოჩნდება. ახლა შეგვიძლია გაუშვათ სხვა ბრძანებები. ამ ხერხით shell-ში ბევრი ბრძანების გათიშვა შეგვიძლია.

თუ გვსურს, რომ პროგრამა გაუშვათ და თან ამასობაში სხვა ბრძანებების აკრეფის საშუალება გვქონდეს, მაშინ ეს ბრძანება უკანა პლანზე უნდა გადავწიოთ. ამისთვის ბრძანებას ბოლოში **&** ოპერატორი უნდა მივუწეროთ და ისე გაუშვათ.

```
achiko@debian:~$ xclock &
[1] 5821
```

`xclock` გაეშვა უკან პლანზე, ეკრანზე დამატებით მისი პროცესის PID დაიწერა. გავუშვათ, `ps` ბრძანება და დავრწმუნდეთ, რომ ჩვენი პროცესი გაშვებულია.

```
achiko@debian:~$ xclock &
[1] 5821
achiko@debian:~$ ps
  PID TTY          TIME CMD
 5812 pts/0    00:00:00 bash
 5821 pts/0    00:00:02 xclock
 6181 pts/0    00:00:00 ps
```

Shell-ში ბრძანება `jobs` საშუალებით შეგვიძლია მოცემული ტერმინალიდან უკანა პლანზე გაშვებული ბრძანებების სია ვნახოთ.

```
achiko@debian:~$ jobs
[1]+  Running                  xclock &
```

კიდევ გავუშვათ ბრძანებები უკანა პლანზე.

```
achiko@debian:~$ xlogo &
achiko@debian:~$ gedit &
```

ორი ახალი ფანჯარა გამოვიდა: ერთში ლოგო, მეორეში კი ტექსტის აკრეფა შეგვიძლია. სცადეთ და დარწმუნდით, რომ მუშაობს. შევამოწმოთ:

```
achiko@debian:~$ jobs
[1]  Running                  xclock &
[2]- Running                  xlogo &
[3]+ Running                  gedit &
```

`jobs` ნომრავს უკანა პლანზე გაშვებულ ბრძანებებს და ეს ნომრები, `fg` ბრძანებასთან კომბინაციაში, სწორედ მაშინ დავგჭირდება, როდესაც მოგვინდება რომელიმე ბრძანების წინა პლანზე გადმოტანა.

```
achiko@debian:~$ fg %3
```

ასე წინა პლანზე გადმოვიდა უკანა პლანზე გაშვებული მესამე ბრძანება, ჩვენს შემთხვევაში - `gedit`. მოდით, ახლა, `Ctrl^c` კლავიშების კომბინაციის ნაცვლად, `Ctrl^z` კლავიშების კომბინაციას დავაჭიროთ. ამ დროს `gedit`-ის პროცესი დაპაუზდება და ასეთ მდგომარეობაში გადავა უკანა პლანზე. გავაგრძელოთ `gedit`-ის ფანჯარაში ტექსტის აკრეფა... აღარაფერი იკრიფება. ჩანს, რომ პროცესი მართლაც შეჩერებულია.

bg ბრძანებით შეგვიძლია უკანა პლანზე გადასმული დაპაუზებული ბრძანება გავაგრძელოთ ასე:

```
achiko@debian:~$ bg %3
```

ახლა უკვე შეგედლებათ ტექსტის აკრეფა გააგრძელოთ.

გრაფიკული პროგრამების ბრძანებათა ხაზიდან გაშვება იმ დროსაა სასარგებლო, როდესაც პროგრამა გრაფიკულად არ ეშვება, ან ეშვება ხარვეზებით. ტერმინალში მისი გაშვება შეცდომების შეტყობინებების ნახვის საშუალებას მოგცემთ. ამასთან, ბევრ გრაფიკულ პროგრამას ბრძანებათა ხაზიდან გაშვების ბევრი საინტერესო და სასარგებლო ოფცია აქვს.

როგორც ზემოთ ვნახეთ, პროცესები ქმნიან ხესებრ სტრუქტურას. ეს იმას ნიშნავს, რომ თუ რომელიმე პროცესი დასრულდება, მისი მემკვიდრე პროცესებიც ავტომატურად მოკვდება. მაგალითად ტერმინალიდან გაშვებული პროცესები ტერმინალის დახურვის ან ზოგადად სამუშაო სესიიდან გამოსვლის შემთხვევაში, ავტომატურად გაითიშება. ხშირია შემთხვევები, როდესაც მომხმარებელს ისეთი ამოცანის გაშვება სურს, რომლის შესრულებაც წინასწარ არ იცის რა დროს გასტანს. ასეთ შემთხვევაში, სასურველია, ეს ბრძანება ისე გავუშვათ, რომ ტერმინალის ან სამუშაო გარემოს დახურვის შემთხვევაში, მაინც გაგრძელდეს და არ გაითიშოს. სწორედ ამ დროს გამოიყენება ბრძანება **disown**. ის საშუალებას იძლევა, რომ ტერმინალიდან გაშვებული პროცესი მისგან განაცალკევოს და მოაცილოს.

მოდით, პატარა ცდა ჩავატაროთ. გავუშვათ ტერმინალიდან ერთ-ერთი ბრძანება უკანა პლანზე და შემდეგ დავხუროთ სესია **exit** ბრძანებით:

```
achiko@debian:~$ xclock &
achiko@debian:~$ exit
```

დავინახავთ, რომ საათის ფანჯარაც დაიხურება. ახლა, ასე ვცადოთ:

```
achiko@debian:~$ xclock & disown
achiko@debian:~$ exit
```

ამ შემთხვევაში საათის პროცესი აღარ გაითიშება.

მსგავსი ეფექტის მიღება შეიძლება **nohup** ბრძანებითაც. სინტაქსურად სწორი ჩანაწერი ასეთია:

```
achiko@debian:~$ nohup xclock &
nohup: ignoring input and appending output to 'nohup.out'
```

ამ ორ ბრძანებას შორის განსხვავება ისაა, რომ **disown** გამოაცალკევებს პროცესს შელის **job** კონტროლისგან (ანუ **job** ბრძანების სიაში ამ ბრძანებას ვერ ვნახავთ), თუმცა ბრძანების სტანდარტული შესასვლელ/გამოსასვლელი და შეცდომების გამოსასვლელი კვლავ ტერმინალზე მიბმული რჩება, მაშინ, როდესაც **nohup**-ის შემთხვევაში, სტანდარტული და შეცდომების გამოსასვლელი **nohup.out** ფაილისკენ იქნება გადამისამართებული.

9.2 სიგნალები

წინა მაგალითში წინა პლანზე გაშვებული ბრძანება `Ctrl^c` კლავიშების კომბინაციით გავთიშეთ. პროცესს გათიშვის სიგნალი გადავეცით. მოდით, ვნახოთ, როგორ შეიძლება პროცესზე სიგნალების გადაცემა და საერთოდ, რა სახის სიგნალები არსებობს.

ბრძანება `kill` სწორედ სიგნალს უგზავნის პროცესებს. მისი სინტაქსი ასეთია:

```
achiko@debian:~$ kill [-signal] PID
```

სიგნალი შეიძლება მოიცეს მისი სახელით ან ნომრით.

№	სახელი	მნიშვნელობა
1	HUP ან SIGHUP	ტრადიციულად, ეს სიგნალი პროცესს ასრულებს. ის აგრეთვე გამოიყენება ბევრი დემონის მიერ პროცესის რეინიციალიზაციისათვის, ანუ ამ სიგნალის მიღების შემდეგ დემონი გადაიტვირთება (გამოირთვება და ხელახლა ჩაირთვება). შესაბამისად, ხელახლა წაიკითხავს კონფიგურაციის ფაილს.
2	INT ან SIGINT	შეწყვეტა. იგივეა, რაც კლავიატურიდან გადაცემული <code>Ctrl^c</code> კლავიშების კომბინაციით გადაცემული სიგნალი.
15	TERM ან SIGTERM	დამთავრება. ესაა <code>kill</code> ბრძანებაში ნაგულისხმევი მნიშვნელობა, თუ სიგნალი არ არის მითითებული.
9	KILL ან SIGKILL	მოკვლა. ავარიული გამორთვა. გამოიყენება მაშინ, როდესაც პროცესი დასრულების სხვა სიგნალებზე არ რეაგირებს. ამ სიგნალის გადაცემით პროცესის სწორი გაწმენდა არ ხდება, შესაბამისად, მდგომარეობა არ ინახება.
19	STOP ან SIGSTOP	შეჩერება, დაპაუზება. პროგრამას არ შეუძლია ამ სიგნალის იგნორირება. ის მას გვერდს ვერ აუვლის.
18	CONT ან SIGCONT	STOP-ით შეჩერებული გაგრძელება.
20	TSTP ან SIGTSTP	შეჩერება, დაპაუზება. იგივეა რაც <code>Ctrl^z</code> . პროგრამას აქვს შესაძლებლობა უგულებელყოს ეს სიგნალი.

მოდით, `kill` ბრძანების რამდენიმე მაგალითი მოვიყვანოთ. გავუშვათ `xclock` უკანა პლანზე და შემდეგ დავხუროთ იგი სიგნალის გადაცემით.

```
achiko@debian:~$ xclock &
[4] 8526
achiko@debian:~$ kill -15 8526
```

ამ ბრძანების დახურვა ასეც შეგვეძლო:

```
achiko@debian:~$ kill -TERM 8526
```

ან ასე:

```
achiko@debian:~$ kill -SIGTERM 8526
```

მარტივად ასეც შესაძლებელია, რადგან, ნაგულისხმევი მნიშვნელობით, მე-15 სიგნალს იყენებს `kill` ბრძანება:

```
achiko@debian:~$ kill 8526
```

`kill` ბრძანებას შეუძლია არგუმენტად რამდენიმე PID მიიღოს. შესაბამისად, შესაძლებელია რამდენიმე პროცესის ერთდროულად დასრულება.

```
achiko@debian:~$ ps aux | less      #ჯერ ვნახულობთ PID-ებს
achiko@debian:~$ kill PID1 PID2 PID3 ...
```

შეგვიძლია პროცესი არა მისი PID ის მითითებით, არამედ სახელით დავასრულოთ. ამისთვის `killall` ბრძანება დაგვჭირდება. თუმცა ამ შემთხვევაში, ამ დასახელების ყველა პროცესი დასრულდება. მაგალითად, გავუშვათ `xclock` რამდენჯერმდე. რამდენიმე პროცესი შეიქმნება. `killall`-ით ერთბაშად შეიძლება მათზე სიგნალის გადაცემა და ამ შემთხვევაში `xclock`-ის ყველა პროცესი დასრულდება.

```
achiko@debian:~$ xclock &
achiko@debian:~$ xclock &
achiko@debian:~$ xclock &
achiko@debian:~$ killall xclock
[5] Terminated          xclock
[6]- Terminated         xclock
[7]+ Terminated         xclock
```

`pkill` ბრძანებაც უგზავნის პროცესებს სიგნალს სახელის მიხედვით, თუმცა, თუ `killall`-ზე პროცესის სრულ დასახელების მითითებაა საჭირო, აქ საკმარისია პროცესის დასახელების შემადგენელი ნაწილი მივუთითოთ.

```
achiko@debian:~$ xclock &
achiko@debian:~$ xclock &
achiko@debian:~$ pkill cloc
```

სანამ `pkill` ბრძანებას გამოვიყენებთ, სასურველია, ჯერ გადავამოწმოთ თუ რომელი პროცესების დასახელებაში შედის ეს ფრაგმენტი, ასე:

```
achiko@debian:~$ pgrep -l cloc
```

გრაფიკული პროგრამის გათიშვა უფრო მარტივადაც არის შესაძლებელი, `xkill` ბრძანებით.


```
achiko@debian:~$ xkill
achiko@debian:~$ killall xclock
Select the window whose client you wish to kill with button 1.
```

ამ ბრძანების გაშვების შემდეგ რომელ ფანჯარაზეც დავაკლიკებთ მაუსს (დავაწკაპუნებთ), ის ფანჯარა დაიხურება.

9.3 პროცესის პრიორიტეტები

გაშვებულ პროცესებს, როგორც ვიცით, პროცესორი ასრულებს. პროცესორის რესურსები ყველა გაშვებულ პროცესზე თანაბრად გადანაწილდება. თუმცა, შესაძლებელია პროცესის სხვადასხვა პრიორიტეტით გაშვება, ისე რომ სისტემამ მის შესრულებას უფრო მეტი ან ნაკლები რესურსი დაუთმოს. პრიორიტეტი, ე.წ. **nice** მნიშვნელობა, [-20,19] ინტერვალში მოიცემა. ყველაზე მეტი რესურსი -20 მნიშვნელობით აღინიშნება, 19 კი ყველაზე დაბალი პრიორიტეტია. პრიორიტეტის მითითების გარეშე, ნაგულისხმევი მნიშვნელობით, ეს მნიშვნელობა 0-ის ტოლია. პროცესებზე პრიორიტეტების მინიჭებისას უნდა ვიცოდეთ შემდეგი:

- ჩვეულებრივ მომხმარებელს შეუძლია **nice**-ის მხოლოდ დადებითი მნიშვნელობების გამოყენება 0-დან 19-ის ჩათვლით.
- ჩვეულებრივ მომხმარებელს არსებული პრიორიტეტის მხოლოდ შემცირება შეუძლია, გაზრდა - არა. მაგალითად, თუ პროცესის პრიორიტეტის მიმდინარე მნიშვნელობა არის 5, მას 2-მდე ვერ დაიყვანთ, არ გექნებათ ამის უფლება.
- მომხმარებელს პრიორიტეტის შეცვლა მხოლოდ თავის პროცესებზე აქვს ნებადართული.
- ადმინისტრატორს, **root**-ს, ნებისმიერი მომხმარებლის პროცესზე პრიორიტეტის ნებისმიერი მნიშვნელობის მინიჭება შეუძლია.

პრიორიტეტის განსხვავებული მნიშვნელობით ბრძანების გაშვება **nice** ბრძანებითაა შესაძლებელი. **renice**-ით არსებული პროცესის პრიორიტეტი მიმდინარე მნიშვნელობის შეცვლა შეგიძლიათ.

```
achiko@debian:~$ nice -n 5 gedit &
[1] 9988
```

შევამოწმოთ. ამისთვის **top** ბრძანება გამოვიყენოთ.

```
achiko@debian:~$ top
...
მართლაც, NI ველის მნიშვნელობა 5 გახდა, ხოლო PR ველის
მნიშვნელობა კი 25 -მდე გაიზარდა
```

მოდით, შევეცადოთ, შევცვალოთ ეს მნიშვნელობა. **renice** ბრძანებას ამისთვის დაჭირდება ბრძანების PID. უკანა პლანზე გაშვებისას მისი PID იქვე დაიწერებოდა და მოსაძებნი არ გაგვიზღებოდა. სხვანაირად შეგიძლიათ **top**-ით ან **ps** ბრძანებით ნახოთ PID-ის მნიშვნელობა.

```
achiko@debian:~$ renice -n 11 -p 9988
9988 (process ID) old priority 5, new priority 11
```

შევცვადოთ, გავზარდოთ პრიორიტეტი და დავრწმუნდეთ, რომ ამის უფლება არ გვაქვს:

```
achiko@debian:~$ renice -n 5 -p 9988
renice: failed to set priority for 9988 (process ID):
Permission denied
```

`nice` და `renice` ბრძანებებს შესაძლებელია გადავცეთ პრიორიტეტის აღმნიშვნელი რიცხვი მოკლედ, `-n` ოფციის გარეშეც, ასე:

```
achiko@debian:~$ nice -5 gedit &          ### ეს ბრანება იგივეა, რაც
achiko@debian:~$ nice -n 5 gedit &
```

```
achiko@debian:~$ nice --5 gedit &          ### ეს ბრანება იგივეა, რაც
achiko@debian:~$ nice -n -5 gedit &
```

```
achiko@debian:~$ renice -5 -p 9988          ### ეს ბრანება იგივეა, რაც
achiko@debian:~$ renice -n -5 -p 9988
```

```
achiko@debian:~$ renice 5 -p 9988           ### ეს ბრანება იგივეა, რაც
achiko@debian:~$ renice -n 5 -p 9988
```

თავი 10

გარემო, ცვლადები

ყოველ ჯერზე, როდესაც shell-ის სესია იხსნება, იქმნება shell-ის გარემო (environnement). ამ გარემოში თავმოყრილია მნიშვნელოვანი მონაცემები shell-დან გასაშვები ბრძანებებისთვის. ეს მონაცემები, თავის მხრივ, გადანაწილებულია ცვლადებში მათ მნიშვნელობებად და სწორედ ეს მნიშვნელობები განსაზღვრავენ shell-დან გაშვებული პროგრამების ქცევას. ამ ცვლადებს გარემოს ცვლადები ჰქვიათ და ისინი ხელმისაწვდომია როგორც shell-ის პროცესებისთვის, ასევე მათი შვილი პროცესებისთვის. სხვაგვარად რომ ვთქვათ, გარემოს ცვლადებში ამ გარემოშივე გაშვებული პროცესების პარამეტრები (ე.წ. settings) ინახება. შეთანხმებისამებრ, გარემოს ცვლადებს დიდი ასოებით აღნიშნავენ.

გარემოს ცვლადების გარდა, shell-ში გვაქვს ლოკალური ცვლადები. ლოკალური ცვლადები მხოლოდ მიმდინარე სესიაში არსებობს და მათი მნიშვნელობები არ არის ხელმისაწვდომი ამ სესიიდან გაშვებული ბრძანებებისთვის ანუ შვილი პროცესებისთვის. ყოველ ჯერზე, როდესაც shell ეშვება, ცვლადების ინიციალიზაცია უნდა მოხდეს. bash-ში ახლად შექმნილი ყველა ცვლადი ლოკალურია. ის გარემოს ცვლადად რომ გაქციოთ, ჯერ უნდა დავაექსპორტოთ, გარემოში შევიტანოთ. დავექსპორტების შემდეგ სრულფასოვან გარემოს ცვლადებს მივიღებთ და ისინი უკვე ხელმისაწვდომი იქნება shell-დან გაშვებული პროგრამებისთვის. ლოკალურ ცვლადებს shell-ის ცვლადებსაც უწოდებენ. ისინი საჭიროა shell-ის გამართული ფუნქციონირებისთვის. shell-ის ცვლადების მნიშვნელობები გავლენას ახდენს არა shell-დან გაშვებულ პროგრამებზე, არამედ თავად shell-ის ქცევაზე.

სინამდვილეში, ცვლადი ოპერატიულ მეხსიერებაში იქმნება და წარმოადგენს ადგილს, სადაც მისი მნიშვნელობაა შენახული. მეხსიერების კონკრეტულად ამ უბანზე წვდომისთვის ცვლადის სახელი უნდა ვიცოდეთ. მისი მნიშვნელობა შეიძლება იყოს ნებისმიერი სახის: რიცხვი, ტექსტი, ფაილის სახელი ან სხვა. ანუ ცვლადი სხვა არაფერია, თუ არა მიმთითებელი გარკვეულ მონაცემზე. shell მომხმარებელს საშუალებას აძლევს შექმნას, წაშალოს ან შეცვალოს ცვლადი. თუმცა ეს ცვლადები მხოლოდ დროებითია და ისინი ავტომატურად წაიშლება მას შემდეგ, რაც shell-ის სესია დაიხურება.

ცვლადის სახელი shell-ში შეიძლება შეიცავდეს მხოლოდ ლათინურ ასო-ნიშნებს (a-z ან A-Z), ციფრებს (0-9) ან ქვედა ტირეს (-).

ცვლადის შექმნა და განსაზღვრა ასე ხდება:

```
achiko@debian:~$ Var="value"
```

სადაც, Var ცვლადის სახელია, ხოლო value მისი მნიშვნელობა. სხვაგვარად რომ ვთქვათ, ამ ბრძანებით Var ცვლადს ჩვენ მივანიჭეთ value მნიშვნელობა. მინიჭების დროს შელში გამოტოვება არ უნდა გამოვიყენოთ.

```
achiko@debian:~$ Var = "value"
-bash: Var : command not found
```

შედეგი ლოგიკურია, რადგან ასე შელმა `Var` აღიქვა ბრძანებად, ხოლო მინიჭების ოპერატორი „`=`“ კი მის პირველ არგუმენტად. `Var` დასახელების ბრძანება არ არსებობს და შეცდომაც ამიტომ გამოვიდა ეკრანზე.

ლოკალური ცვლადი გარემოს ცვლადად რომ გადავაქციოთ, შექმნის შემდეგ ის გარემოში უნდა დავაქსპორტოთ `export` ბრძანებით, ასე:

```
achiko@debian:~$ export Var
```

ან პირდაპირ, შექმნისთანავე გავაკეთოთ, ასე:

```
achiko@debian:~$ export Var="value"
```

ცვლადის მნიშვნელობის სანახავად `echo` ბრძანება გამოიყენება და ცვლადის სახელს წინ `$`(დოლარის ნიშანი) წარემძღვარება.

```
achiko@debian:~$ echo $Var
value
```

ცვლადის მნიშვნელობას თუ ხელახლა განვსაზღვრავთ, ძველი მნიშვნელობა დაიკარგება და ის ახლით შეიცვლება.

```
achiko@debian:~$ Var="next value"
achiko@debian:~$ echo $Var
next value
```

ცვლადის წაშლა კი `unset` ბრძანებით ხორციელდება:

```
achiko@debian:~$ unset $Var
```

გარემოს ცვლადების სრული ჩამონათვალის სანახავად, თავისი მნიშვნელობებით, უნდა გავუშვათ ბრძანება `printenv` ან `env` არგუმენტების გარეშე.

```
achiko@debian:~$ printenv
...
SHELL=/bin/bash
LANGUAGE=en_US:en
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
...
```

```
achiko@debian:~$ env
...
SHELL=/bin/bash
LANGUAGE=en_US:en
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
...
```

ორივე ბრძანება ერთსა და იმავე შედეგს მოგვცემს. ზოგადად, მათ შორის განსხვავება ისაა, რომ **printenv** ბრძანებით სრული ჩამონათვალის ნაცვლად კონკრეტული ცვლადი შეგვიძლია ვნახოთ:

```
achiko@debian:~$ printenv SHELL
/bin/bash
```

მეორეს მხრივ, **env** საშუალებას გვაძლევს გავუშვათ კონკრეტული პროგრამა განსხვავებულ გარემოში ანუ გარემოს ცვლადის განსხვავებული მნიშვნელობით. ამგვარად, **cmd** ბრძანება გაეშვება **Var**-ის შეცვლილი მნიშვნელობით, ხოლო გარემოში კი **Var** თავდაპირველი მნიშვნელობით დარჩება.

```
achiko@debian:~$ env Var="new value" cmd
```

set ბრძანებით შეგვიძლია ყოველგვარი ცვლადის სრული ჩამონათვალი ვნახოთ თავისი მნიშვნელობებით, მათ შორის **shell**-ის ფუნქციებიც (მათ მოგვიანებით განვიხილავთ). როგორც წესი, გრძელი სია დაგენერირდება ეკრანზე. მათ დასათვალიერებლად უპრიანია **less** გამოვიყენოთ:

```
achiko@debian:~$ set | less
```

ნაგულისხმევი მნიშვნელობით გაშვებული **bash** ზუსტად არ შეესაბამება POSIX სტანდარტს. **set -o posix** ბრძანებით შეგვიძლია ეს ნაგულისხმევი მნიშვნელობა გავუთიშოთ და POSIX სტანდარტთან შესაბამისობა გავააქტიუროთ. ასე უკვე, ბრძანება **set** ფუნქციებს აღარ გამოიტანს და ეკრანი ისე აღარ გადაიტვირთება. ნაგულისხმევი მნიშვნელობის დაბრუნება **set +o posix** ბრძანებით შეგვძლება.

მოდით, ვნახოთ რამდენიმე ცნობილი, ხშირად გამოყენებადი ცვლადი.

გარემოს ცვლადი	მნიშვნელობა
USER	მიმდინარე შემოსული მომხმარებლის სახელი.
HOME	მიმდინარე მომხმარებლის პირადი დირექტორია.
SHELL	ტერმინალში გაშვებული ინტერპრეტატორი, რომელმაც მომხმარებლის მიერ აკრეფილი ბრძანებები უნდა შეასრულოს. ნაგულისხმევი მნიშვნელობით, ჩვენ გვაქვს bash , თუმცა მისი შეცვლა შესაძლებელია.
PWD	მიმდინარე დირექტორია, სადაც ამჟამად ვმუშაობთ.

OLDPWD	წინა დირექტორია, სადაც მიმდინარე დირექტორიამდე ვიმყოფებოდით. <code>cd</code> - ბრძანება იყენებს სწორედ ამ ცვლადს.
PATH	დირექტორიების სია ორწერტილით (:) არის გამოყოფილი. ეს ის დირექტორიებია, რომლებშიც მოცემული რიგითობით სისტემა ჩვენს მიერ აკრეფილ ბრძანებებს შესასრულებლად ეძებს.
LANG	მიმდინარე ენის, ლოკალიზაციისა და კოდირების პარამეტრები.
-	ბოლოს გაშვებული ბრძანება.

Shell-ის ცვლადი	მნიშვნელობა
PS1	shell-ის მოსაწვევი (Prompt String). ტექსტი, რომელიც ბრძანების აკრეფამდე ჩანს ტერმინალზე. მისი გამოჩენა იმის მანიშნებელია, რომ shell მზადაა ახალი ბრძანება მიიღოს და შეასრულოს.
HISTFILE	ნაგულისხმევი მნიშვნელობით, <code>bash</code> ჩვენს მიერ გაშვებული ბრძანებების სახელებს ინახავს ამ ფაილში. სწორედ ამიტომ, რომ ზედა ისრით წინა ბრძანებების ნახვა და გაშვება შეგვიძლია.
HISTFILESIZE	ბრძანებების მაქსიმალური რაოდენობა, რომლის შენახვაც ზემოთ ნახსენებ ისტორიის ფაილშია შესაძლებელი.

ამ ცვლადების უკეთ გასაგებად, რამდენიმე მაგალითი მოვიყვანოთ.

```
achiko@debian:~$ echo $HOME
/home/achiko
achiko@debian:~$ echo $PWD
/home/achiko
```

მოდით, ახლა სხვა დირექტორიაში გადავიდეთ, მაგ. `Documents/`-ში და კვლავ ნახოთ ამ ცვლადების მნიშვნელობები:

```
achiko@debian:~$ cd Documents/
achiko@debian:~/Documents$ echo $HOME
/home/achiko
achiko@debian:~/Documents$ echo $PWD
/home/achiko/Documents
```

მომხმარებლის პირადი დირექტორია არ არის დამოკიდებული იმაზე, თუ რომელ დირექტორიაში ვიმყოფებით და ბუნებრივია, მისი მნიშვნელობა იგივე დარჩება. მიმდინარე დირექტორია კი შეიცვალა და შედეგიც შესაბამისია.

```
achiko@debian:~/Documents$ echo $OLDPWD  
/home/achiko/
```

ამ ცვლადში კი წინა დირექტორიის გზა დამახსოვრებული, სადაც ვიყავით.

ახლა PATH გარემოს ცვლადი განვიხილოთ დეტალურად. ის ერთ-ერთი მნიშვნელოვანი ცვლადია, რადგან ამ ცვლადის შიგთავსზეა დამოკიდებული სისტემა გაუშვებს თუ არა პირდაპირ ჩვენს მიერ აკრეფილ ბრძანებას.

```
achiko@debian:~/Documents$ ls  
Desktop/   file1  GPL2.txt  Pictures/  Videos/  
Documents/ file2  GPL3.txt  Public/  
Downloads/ file3  Music/    Templates/
```

შედეგი ეკრანზე დაიწერა. shell-მა ls ბრძანების ადგილმდებარეობა იპოვა (ბრძანებაც ზომ ფაილია და ისიც რომელიმე დირექტორიაშია შენახული) და გაუშვა. შევხედოთ PATH-ს.

```
achiko@debian:~/Documents$ echo $PATH  
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

ჩვენ მიერ აკრეფილი ბრძანება ერთ-ერთ ამ ჩამოთვლილ დირექტორიაში აღმოჩნდა და ამიტომაც გაეშვა. მოდით, დავრწმუნდეთ, რომ ეს ასეა:

```
achiko@debian:~/Documents$ PATH=""  
achiko@debian:~/Documents$ ls  
bash: ls: No such file or directory
```

როგორც კი PATH ცვლადის შიგთავსი დავაცარიელეთ, შელმა ls ბრძანება ვეღარ იპოვა. ამისდა მიუხედავად, ბრძანების გაშვება მაინც შეგვიძლია, მხოლოდ მისი სრული გზის მითითებით, ასე:

```
achiko@debian:~/Documents$ /bin/ls  
Desktop/   file1  GPL2.txt  Pictures/  Videos/  
Documents/ file2  GPL3.txt  Public/  
Downloads/ file3  Music/    Templates/
```

PATH კომფორტულ გარემოს გვიქმნის ბრძანებების გასაშვებად. საკმარისია, ის შეიცავდეს ბრძანებების შემცველ დირექტორიას, რომ პირდაპირ ბრძანების გამოძახებაც კმარა მის გასაშვებად. სწორედ ამიტომ, თუ საკუთარ ბრძანებებს შევქმნით, უმჯობესია, ისინი ერთ რომელიმე დირექტორიაში შევინახოთ და ამ დირექტორიის გზა დავუმატოთ PATH-ს, ასე:

```
achiko@debian:~/Documents$ PATH=$PATH:new_directory
```

ამგვარად, PATH-ს ბოლოში დავუმატებთ ახალი დირექტორიის გზას. შეგვიძლია წინაც დავუმატოთ ის, ასე:

```
achiko@debian:~/Documents$ PATH=new_directory:$PATH
```

ეს ორი ჩანაწერი ერთმანეთისგან განსხვავდება. ჩვენ მიერ აკრეფილი ბრძანება PATH-ის ჩამონათვალიდან ჯერ პირველი დირექტორიაში მოიძებნება, ვერ პოვნის შემთხვევაში, მეორე დირექტორიაში გადავა მოსაძებნად და ა.შ. შესაბამისად, თუ რომელიმე ბრძანების დასახელება გამეორებულია და სხვადასხვა დირექტორიაშია შენახული, ამ დასახელების აკრეფისას, ის ბრძანება გაეშვება, რომელიც პირველად შეხვდება shell-ს. PATH-ის ძველი მნიშვნელობის აღსადგენად მას ხელახლა უნდა მივანიჭოთ ძველი მნიშვნელობა ან შელის მიმდინარე სესია დავზუროთ და ახალი გავუშვათ, რათა თავიდან შეიქმნას ცვლადები.

მოდით ახლა, სანამ ჩვენ თავად შევქმნით ახალ ცვლადს, უკვე არსებული შელის ერთი ცვლადიც განვიხილოთ - PS1. როგორც უკვე ვთქვით, ის შელის მოსაწვევია და ბრძანების აკრეფამდე რასაც ვხედავთ, სწორედ PS1 ცვლადის მნიშვნელობას წარმოადგენს. ვნახოთ, რა წერია ამ ცვლადში:

```
achiko@debian:~/Documents$ echo $PS1
\[e0;\u@h: \w@a\]${debian_chroot:+($debian_chroot)}\u@h:\w$
```

ცოტა გაუგებარი შიგთავსია. ამდენი რამ ნამდვილად არ გამოდის ეკრანზე. ასე იმიტომ მოხდა, რომ PS1 დინამიკური ცვლადია და მას გარკვეული სპეციალური სიმბოლოების გაგება შეუძლია. ეს სიმბოლოები კოდირებულია და მათ ქვეშ სხვადასხვა მნიშვნელობა იგულისხმება. PS1 იგებს შემდეგ სიმბოლოებს:

`\h` კომპიუტერის სახელი.

`\u` მომხმარებლის სახელი.

`\w` მიმდინარე დირექტორიის სრული გზა.

`\W` მიმდინარე დირექტორიის სახელი.

`\d` მიმდინარე თარიღი.

`\t` მიმდინარე დრო 24 საათიანი HH:MM:SS ფორმატით.

`\T` მიმდინარე დრო 12 საათიანი HH:MM:SS ფორმატით.

`\@` მიმდინარე დრო 12 საათიანი AM/PM ფორმატით.

`\$` ეს სპეციალური სიმბოლო გადაიქცევა #-ად იმის ნიშნად, რომ, როგორც ადმინისტრატორი ისე შეხვედით სისტემაში. სხვა შემთხვევაში დარჩება \$-ად.

`\!` აკრეფილი ბრძანების რიგითი ნომერი.

PS1-ში სხვა კოდირებული სიმბოლოების გამოყენებაც შეიძლება. დეტალურად, მოგვიანებით, მე-2 ნაწილში დავუბრუნდებით PS1-ს.

```
achiko@debian:~$ PS1="Hello Georgia "
Hello Georgia
Hello Georgia echo Gamarjoba
Gamarjoba
```



```
Hello Georgia
Hello Georgia PS1="Hello Georgia2 "
Hello Georgia2
Hello Georgia2
```

```
achiko@debian:~$ PS1="\h "
debian
debian echo Gamarjoba
Gamarjoba
debian
```

```
achiko@debian:~$ PS1="User=\u\ndir=\w\ntime=\t \ $"
User=achiko
dir=~/.Downloads
time=14:29:31 $
User=achiko
dir=~/.Downloads
time=14:29:33 $ echo Gamarjoba
Gamarjoba
User=achiko
dir=~/.Downloads
time=14:29:33 $
```

მოდიტ, ახლა მიმდინარე სესიაში ჩვენ თავად შევქმნათ ახალი ცვლადი:

```
achiko@debian:~$ cvladi="Hello World!"
```

ეს ცვლადი shell-ის ცვლადია. ის მიმდინარე სესიაშია ხელმისაწვდომი.

```
achiko@debian:~$ set | grep cvladi
cvladi='HELLO World!'
```

ამით დავრწმუნდით, რომ ჩვენი შექმნილი ცვლადი ნამდვილად შეიქმნა მოცემული მნიშვნელობით. ახლა შევამოწმოთ, რომ იქ გარემოს ცვლადი არ არის.

```
achiko@debian:~$ printenv
...
```

ამ ჩამონათვალში ეს ცვლადი ნამდვილად არ არის. თუმცა ცვლადი არსებობს:

```
achiko@debian:~$ echo $cvladi
Hello World!
```

ახლა ვნახოთ, რომ მისი მნიშვნელობა შვილი პროცესისთვის მიუწვდომელია. გავუშვათ `bash`-ის ახალი პროცესი მიმდინარე `bash`-ში და შევამოწმოთ `cvladi`-ის მნიშვნელობა.

```
achiko@debian:~$ bash
achiko@debian:~$ echo $cvladi
```

არაფერი გამოვიდა ეკრანზე. ახლად გაშვებული `bash` რომ ნამდვილად ძველი `bash` პროცესის შვილი პროცესია, შეგვიძლია ასე დავრწმუნდეთ:

```
achiko@debian:~$ ps jf
PPID    PID    PGID    SID TTY          TPGID STAT   UID    TIME COMMAND
11435   11436   11436   11436 pts/1    16596 Ss      1000    0:00 -bash
11436   16588   16588   11436 pts/1    16596 S       1000    0:00 \_ bash
16588   16596   16596   11436 pts/1    16596 R+    1000    0:00 \_ ps jf
...
```

დავბრუნდეთ ორიგინალ `bash`-ში. ამისთვის მიმდინარე `bash`-დან `exit` ბრძანებით გამოვიდეთ. მოდით, ახლა ეს ლოკალური ცვლადი გარემოს ცვლადად გადავაქციოთ.

```
achiko@debian:~$ exit
achiko@debian:~$ export cvladi
```

შევამოწმოთ.

```
achiko@debian:~$ printenv | grep cvladi
cvladi=Hello World!
```

ყველაფერი რიგზეა. გარემოში ეს ცვლადი თავისი მნიშვნელობით შეტანილია. ახლა დავრწმუნდეთ, რომ ის შვილი პროცესისთვისაც გასაგები გახდა.

```
achiko@debian:~$ bash
achiko@debian:~$ echo $cvladi
Hello World!
```

მშვენიერი! ახლა ახალი ექსპერიმენტი ჩავატაროთ და შვილ პროცესში ახალი გარემოს ცვლადი შევქმნათ.

```
achiko@debian:~$ export cvladi2="Gamarjoba Georgia!"
```

ჯერ დავრწმუნდეთ, რომ გარემოში ახალი ცვლადი დაექსპორტდა, შემდეგ კი გამოვიდეთ მიმდინარე `bash`-დან და დავუბრუნდეთ საწყის `bash`-ს.

```
achiko@debian:~$ printenv | grep cvlad
cvladi2=Gamarjoba Georgia!
cvladi=Hello World!
achiko@debian:~$ exit
achiko@debian:~$ echo $cvladi2
```

ეკრანზე ვერაფერს დავინახავთ. მართალიც არის, რადგან გარემოს ცვლადები მემკვიდრეობით გადადიან მამიდან შვილ პროცესებზე. ჩვენ კი შვილ პროცესში შექმნილი ცვლადი მამა პროცესს გვინდა გავაგებინოთ. ეს კი შეუძლებელია, რადგან, როცა `exit` ბრძანებით შვილი პროცესიდან გამოვედით, ეს პროცესი დავხურეთ. შესაბამისად, ამით იქ შექმნილი ყველა ცვლადი განადგურდა.

თუმცა მამა პროცესში დაგვრჩა `cvladi` გარემოს ცვლადი. შეგვიძლია ის გარემოდან ამოვავდოთ და კვლავ ლოკალურ ცვლადად გადავაქციოთ, ასე:

```
achiko@debian:~$ export -n cvladi
```

შევამოწმოთ:

```
achiko@debian:~$ printenv | grep cvladi
```

მართლაც, გარემოში აღარაა ეს ცვლადი. ლოკალურ ცვლადად კი ნამდვილად დარჩა.

```
achiko@debian:~$ set | grep cvladi
cvladi=Hello World!
```

თუ გვსურს საბოლოოდ წავშალოთ ისიც, მაშინ `unset` ბრძანება უნდა გამოვიყენოთ.

```
achiko@debian:~$ unset cvladi
```

შევამოწმოთ.

```
achiko@debian:~$ echo $cvladi
```

მისი მნიშვნელობა აღარ გამოვიდა ეკრანზე, რადგან ცვლადი წაიშალა, აღარ არსებობს. ცვლადის შეცვლა თავისი მნიშვნელობით შესაძლებელია ცვლადის ფიგურულ ფრჩხილებში ჩასმით. ასე:

```
achiko@debian:~$ echo ${var}
```

ერთი შეხედვით ისმის კითხვა - თუ ფიგურული ფრჩხილების გარეშეც გამოდის მისი მნიშვნელობა ეკრანზე, რატომ არის საჭირო დამატებით ფრჩხილების გამოყენება? პასუხისთვის მაგალითს მივმართოთ:

```

achiko@debian:~$ a=123
achiko@debian:~$ echo $a
123
achiko@debian:~$ echo $a4      ### ასე შელი გამოიტანს a4 ცვლადის
მნიშვნელობას. მასში კი არაფერი წერია!

achiko@debian:~$ echo ${a}4    ### ასე კი შელი გამოიტანს a ცვლადის
მნიშვნელობას, რომელსაც ბოლოში მიუწერს 4-ს!
1234

```

ახლა უკვე ნათელი გახდა. თუ ცვლადის მნიშვნელობას გვსურს ასო-ნიშნების რაიმე რიგი მივუწეროთ, ფიგურული ფრჩხილების გამოყენება აუცილებელია.

10.1 ცვლადის დამუშავება

ცვლადებზე საუბრისას უნდა ვახსენოთ `expr` ბრძანება, რომელიც ცვლადის მნიშვნელობაში არსებულ სიმბოლოთა მწკრივს (ე.წ. string) ამუშავებს.

სადემონსტრაციოდ ავიღოთ `x` ცვლადი და `abcABC123ABCabc` მნიშვნელობა მივანიჭოთ.

```

achiko@debian:~$ x=abcABC123ABCabc
achiko@debian:~$ echo $x
abcABC123ABCabc

```

`expr` ბრძანებით ცვლადის სიმბოლოთა მწკრივზე სხვადასხვა მოქმედებების განხორციელება შეგვიძლია. მოდიოთ, ვნახოთ.

10.1.1 ცვლადის სიგრძე

```

achiko@debian:~$ expr length $x
15

```

ეს ბრძანება ცვლადის სიგრძეს გამოიტანს. ცვლადის სიგრძე ნიშნავს მის მნიშვნელობაში ჩაწერილი ასო-ნიშნების რაოდენობას. შესაბამისად, პასუხიც არის 15.

ცვლადის სიგრძის გამოტანა სხვაგვარი ჩანაწერებითაც ხდება. ასე:

```

achiko@debian:~$ expr $x : '.*'
15
achiko@debian:~$ expr ${#x}
15

```

ეს უკანასკნელი სინტაქსური ჩანაწერი `echo` ბრძანებას კარგად ესმის.

```

achiko@debian:~$ echo ${#x}
15

```

შესაბამისად, შეგვიძლია, `echo` ბრძანებასაც გადავცეთ ასეთი არგუმენტი. `expr` ბრძანებაში წერტილით (`.`) ნებისმიერი სიმბოლო აღინიშნება. ფიფქით (`*`) კი - მის წინ მიწერილი სიმბოლოს ბევრჯერ გამეორება.

10.1.2 ცვლადის მწკრივში ფრაგმენტის სიგრძე

```
achiko@debian:~$ expr match "$string" '$substring'
achiko@debian:~$ expr "$string": '$substring'
```

ამ ბრძანებებით, რომლებიც ერთსა და იმავე შედეგს იძლევა, ვხედავთ არა მთლიანი ცვლადის, არამედ მისი რომელიმე ფრაგმენტის სიგრძეს.

მაგალითად:

```
achiko@debian:~$ expr match $x '[a-z]*A'
4
```

აქ მითითებული ფრაგმენტი იწყება პატარა ასოთი (`[a-z]`), მეორდება რამდენჯერმე (`*`) და მთავრდება `A` ასო-ნიშნით. ასეთი ჩანაწერი ცვლადში არის „abcA“, ხოლო მისი სიგრძე 4.

მეორე ჩანაწერთაც იგივე შედეგს მივიღებთ:

```
achiko@debian:~$ expr $x : '[a-z]*A'
4
```

```
achiko@debian:~$ expr match $x '[a-z].*A'
10
```

ამ უკანასკნელი ბრძანებით მოვძებნით შემდეგ „abcABC123A“ ფრაგმენტს (რადგან „.*“ უმორეს `A`-მდე მისვლას ნიშნავს) და მისი სიგრძე არის 10.

10.1.3 ფრაგმენტის პოზიციის რიგითი ნომერი მწკრივში

```
achiko@debian:~$ expr index $string $substring
```

ამ ბრძანებით ცვლადში ფრაგმენტის პოზიციური ნომერი გამოგვაქვს. ანუ ვარკვევთ თუ რომელ პოზიციაზეა განთავსებული ცვლადში ეს ფრაგმენტი.

მაგალითად:

```
achiko@debian:~$ expr index $x '123'
7
```

ამ ბრძანებაში „123“ ფრაგმენტი `x` ცვლადში მე-7 პოზიციაზე იმყოფება.

10.1.4 მწკრივიდან ფრაგმენტის ამოღება

```
achiko@debian:~$ expr substr $string $position $length
```

ამ ბრძანებით კი ცვლადიდან თავად ფრაგმენტი გამოგვაქვს.

მაგალითად:

```
achiko@debian:~$ expr substr $x 2 2
bc
```

ასე `x` ცვლადიდან მე-2 პოზიციაზე მყოფი ასო-ნიშნიდან დაწყებული შემდეგი 2 სიმბოლო გამოვიდა - „bc“.

ფრაგმენტის გამოსატანად `echo` ბრძანებაც შეგვიძლია გამოვიყენოთ შემდეგი სინტაქსით:

```
achiko@debian:~$ echo ${string:position:length}
```

მაგალითად:

```
achiko@debian:~$ echo ${x:2:2}
cA
```

ეს ბრძანება `x` ცვლადიდან მე-2 პოზიციის შემდეგ 2 ასო-ნიშნის გამოიტანს - „cA“. შესაძლებელია, შემოკლებული სინტაქსიც გამოვიყენოთ, ასე:

```
achiko@debian:~$ echo ${string:position}      # ან
achiko@debian:~$ echo ${string::length}
```

```
achiko@debian:~$ echo ${x:2}
cABC123ABCabc
achiko@debian:~$ echo ${x:0:2}
ab
achiko@debian:~$ echo ${x::2}
ab
```

პირველი ბრძანება `x` ცვლადიდან მე-2 პოზიციის შემდეგ ყველა სიმბოლოს გამოიტანს - `cABC123ABC123`. მეორე ბრძანება კი, `x` ცვლადის დასაწყისიდან 2 სიმბოლოს გამოიტანს - `ab`. მესამე ბრძანება მეორის უფრო შემოკლებული ვარიანტია. ასეთი სინტაქსური ჩანაწერი `expr` ბრძანებასაც ესმის:

```
achiko@debian:~$ expr ${x:2}
cABC123ABCabc
```

```
achiko@debian:~$ expr ${x:2:2}
cA
```

ფრაგმენტის სიგრძის პოვნისას რა ჩანაწერიც გამოვიყენეთ, იგივე სინტაქსის გამოყენება თავად ფრაგმენტის გამოსატანადად გამოგვადგება. უბრალოდ, ეს ფრაგმენტი „(\“ სტრუქტურაში უნდა მოვაქციოთ, ასე:

```
achiko@debian:~$ expr match "$string" '\($substring\)'      # ან
achiko@debian:~$ expr "$string": '\($substring\)'
```

თუ ფრაგმენტის განსაზღვრა *x* ცვლადის ბოლოდან გვსურს დავიწყეთ, მაშინ ასეთი სინტაქსი უნდა გამოვიყენოთ:

```
achiko@debian:~$ expr match "$string" '.*\($substring\)'      # ან
achiko@debian:~$ expr "$string": '.*\($substring\)'
```

მაგალითად:

```
achiko@debian:~$ expr match $x '\([a-z]*A\)'
abcA
achiko@debian:~$ expr match $x '\([a-z].*A\)'
abcABC123A
achiko@debian:~$ expr match $x '\(..\)'      # გამოვა სანყისი 2 სიმბოლო
ab
achiko@debian:~$ expr match $x '.*\(..\)'      # გამოვა ბოლო 2 სიმბოლო
bc
```

10.1.5 მწკრივიდან ფრაგმენტის წაშლა

ცვლადიდან რომელიმე ფრაგმენტის ამოსაღებად 4 საშუალება არსებობს. ესენია:

```
achiko@debian:~$ echo ${string#substring}
achiko@debian:~$ echo ${string##substring}
achiko@debian:~$ echo ${string%substring}
achiko@debian:~$ echo ${string%%substring}
```

პირველი ბრძანებით ვახორციელებთ უმოკლეს წაშლას „string“ ცვლადის დასაწყისიდან. უმჯობესია, ზოგადი პრინციპით იყოს განსაზღვრული „substring“-ში, თუ რა ფრაგმენტი უნდა წაიშალოს.

მაგალითად:

```
achiko@debian:~$ echo $x
abcABC123ABCabc
```

```
achiko@debian:~$ echo ${x#a*A}  
BC123ABCabc
```

ამ ბრძანებით `x` ცვლადის დასაწყისიდან წავშაღეთ ასო-ნიშანთა ერთობლიობა, რომელიც იწყება „a“-ით და მთავრდება უახლოეს შემხვედრ „A“-სთან. მეორე ბრძანებით - უშორეს შემხვედრ „A“-სთან. ანუ, განვახორციელეთ უშორესი წაშლა დასაწყისიდან. მესამეთი - ცვლადის ბოლოდან უმოკლესი წაშლა, მეოთხეთი კი ბოლოდან უშორესი წაშლა.

```
achiko@debian:~$ echo ${x##a*A}  
BCabc  
achiko@debian:~$ echo ${x%C*c}  
abcABC123AB  
achiko@debian:~$ echo ${x%%C*c}  
abcAB
```

10.1.6 მწკრივიდან ფრაგმენტის ჩანაცვლება სხვა გამოსახულებით

```
achiko@debian:~$ echo ${string/substring/replacement}  
achiko@debian:~$ echo ${string//substring/replacement}  
achiko@debian:~$ echo ${string/#substring/replacement}  
achiko@debian:~$ echo ${string/%substring/replacement}
```

პირველი ბრძანებით „string“ ცვლადში პირველ შემხვედრ „substring“-ს შევცვლით „replacement“-ით. მეორეთი - „string“ ცვლადში ყველა შემხვედრ „substring“-ს შევცვლით „replacement“-ით. შემდეგით მხოლოდ იმ შემთხვევაში შეიცვლება „string“ ცვლადში „substring“ „replacement“-ით, თუ „string“ იწყება „substring“-ით, ხოლო ბოლო ბრძანებით მხოლოდ იმ შემთხვევაში შეიცვლება „string“ ცვლადში „substring“ „replacement“-ით, თუ „string“ მთავრდება „substring“-ით.

მაგალითად:

```
achiko@debian:~$ echo $x  
abcABC123ABCabc  
achiko@debian:~$ echo ${x/abc/TEST}  
TESTABC123ABCabc  
achiko@debian:~$ echo ${x//abc/TEST}  
TESTABC123ABCTEST  
achiko@debian:~$ echo ${x/#abc/TEST}  
TESTABC123ABCabc  
achiko@debian:~$ echo ${x/#abcd/TEST}  
abcABC123ABCabc  
achiko@debian:~$ echo ${x/%abc/TEST}  
abcABC123ABCTEST  
achiko@debian:~$ echo ${x/%abcd/TEST}  
abcABC123ABCabc
```


10.1.7 მწკრივში დიდი და პატარა ასოების შეცვლა/გადანაცვლება

```
achiko@debian:~$ echo ${string^}      # პირველი ასოს შეცვლა დიდი ასოთი
achiko@debian:~$ echo ${string^^}      # ყველა ასოს შეცვლა დიდი ასოთი
achiko@debian:~$ echo ${string,}      # პირველი ასოს შეცვლა პატარა ასოთი
achiko@debian:~$ echo ${string,,}      # ყველა ასოს შეცვლა პატარა ასოთი
achiko@debian:~$ echo ${string~}      # პირველი ასოს გადანაცვლება
achiko@debian:~$ echo ${string~~}      # ყველა ასოს გადანაცვლება
```

მაგალითად:

```
achiko@debian:~$ echo $x
abcABC123ABcabc
achiko@debian:~$ echo ${x^}
AbcABC123ABcabc
achiko@debian:~$ echo ${x^^}
ABCABC123ABCABC
achiko@debian:~$ echo ${x,}
abcabc123abcabc
achiko@debian:~$ echo ${x,,}
abcABC123ABcabc
achiko@debian:~$ echo ${x~}
AbcABC123ABcabc
achiko@debian:~$ echo ${x~~}
ABCabc123abcABC
```

ზოგადად, `${...}` კონსტრუქცია, `echo` ბრძანების გარდა, `expr` ბრძანებისთვისაც არის სამართლიანი. ზემოთმოყვანილ, ნებისმიერ მაგალითში `echo`-ს ნაცვლად `expr` რომ ვიხმაროთ, მსგავს შედეგს მივიღებთ. თუმცა, როდესაც ცვლადის შიგთავსის დასამუშავებლად ამ კონსტრუქციას ვიყენებთ, უმჯობესია, `echo` ბრძანება გამოვიყენოთ. საქმე ისაა, რომ თუ `${...}` კონსტრუქციაში გამოყენებული ცვლადი არ არსებობს ან არაფერი (NULL) წერია მასში, `expr` ბრძანება წარუმატებლად დასრულდება, მაშინ, როდესაც `echo` ბრძანებით წარმატებულ შედეგს ვიღებთ.

```
achiko@debian:~$ expr ${y:2}; echo $?      # y ცვლადი შექმნილი არ არის
expr: missing operand
Try 'expr --help' for more information.
2
achiko@debian:~$ echo ${y:2}; echo $?      # y ცვლადი შექმნილი არ არის
0
achiko@debian:~$
```

ასეთ დროს, შეიძლება, სასარგებლო აღმოჩნდეს ცვლადის მოცემის შემდეგი ფორმების ცოდნა:

<code>\${Variable:-word}</code>	თუ Variable არ არსებობს ან NULL-ია (ანუ, მასში მნიშვნელობა არ არის ჩაწერილი), მაშინ სტანდარტულ გამოსასვლელზე მივიღებთ word-ს. სხვა შემთხვევაში - Variable-ის მნიშვნელობას.
<code>\${Variable:=word}</code>	თუ Variable არ არსებობს ან NULL-ია, მაშინ სტანდარტულ გამოსასვლელზე მივიღებთ word-ს, და ამავედროულად, Variable-ს მიენიჭება word მნიშვნელობა. სხვა შემთხვევაში, Variable-ის მნიშვნელობა დაიბეჭდება.
<code>\${Variable:+word}</code>	თუ Variable არ არსებობს ან NULL-ია, მაშინ არაფერი შეიცვლება. სხვა შემთხვევაში, სტანდარტულ გამოსასვლელზე მივიღებთ word-ს.
<code>\${Variable:?word}</code>	თუ Variable არ არსებობს ან NULL-ია, მაშინ შეცდომების გამოსასვლელზე მივიღებთ word შეტყობინებაზე. სხვა შემთხვევაში, Variable-ის მნიშვნელობას.

სადემონსტრაციოდ ავიღოთ სამი ცვლადი a, b, c. პირველი შევქმნათ და მასში რაიმე მნიშვნელობა ჩავწეროთ, მეორე შევქმნათ და მასში არაფერი ჩავწეროთ, ხოლო მესამე არ შევქმნათ. ვნახოთ, რა გამოვა:

```
achiko@debian:~$ a=TEST
achiko@debian:~$ b=' '
achiko@debian:~$ echo ${a:-word}
TEST
achiko@debian:~$ echo ${b:-word}
word
achiko@debian:~$ echo ${c:-word}
word
achiko@debian:~$ echo ${a:+word}
word
achiko@debian:~$ echo ${b:+word}

achiko@debian:~$ echo ${c:+word}

achiko@debian:~$ echo ${a:?word}
TEST
achiko@debian:~$ echo ${b:?word}
bash: b: word
achiko@debian:~$ echo ${c:?word}
bash: c: word
achiko@debian:~$ echo ${a:=word}
TEST
achiko@debian:~$ echo ${b:=word}
word
achiko@debian:~$ echo ${c:=word}
word
achiko@debian:~$ echo $b $c
word word
```

ახლა პრაქტიკული გამოყენების მაგალითები მოვიყვანოთ. დავუშვათ, გვაქვს ფაილის გრძელი სრული გზა, რომელიც ცვლადშია მოცემული და გვსურს ამ ჩანაწერიდან მხოლოდ

ფაილის სახელი გამოვიტანოთ, ან მხოლოდ იმ დირექტორიის სრული სახელი, სადაც ეს ფაილი იმყოფება. ზემოთ ნახსენები ბრძანებებით მარტივად გამოდის ეს ყველაფერი. ვნახოთ:

```
achiko@debian:~$ x=/usr/share/common-licenses/GPL-3
```

ჩვენი მიზანია, ცალ-ცალკე გამოვიტანოთ „usr/share/common-licenses“ და „GPL-3“.

```
achiko@debian:~$ echo ${x%/*}
/usr/share/common-licenses
achiko@debian:~$ echo ${x##*/}
GPL-3
```

პირველ ბრძანებაში უმოკლესი წაშლა გავაკეთეთ ბოლოდან „/“ სიმბოლოს ჩათვლით. მეორეში კი - უმორესი წაშლა დასაწყისიდან, „/“-დან დაწყებული ბოლო შემხვედრ „/“-ის ჩათვლით.

სხვაგვარად ასე გავკეთდება:

```
achiko@debian:~$ expr match $x \"/(.*)\"/
/usr/share/common-licenses
achiko@debian:~$ expr match $x '.*/(.*)'
GPL-3
```

პირველ ბრძანებაში გამოგვაქვს ფრაგმენტი „/“-დან დაწყებული უმორეს „/“-მდე. მეორეში კი - ბოლო „/“-მდე გამოგვაქვს ყველაფერი.

ეს ფუნქცია მარტივად სრულდება უკვე არსებული სხვა ბრძანებებით. მაგალითად, **dirname**-ით პირდაპირ დირექტორიის სახელი გამოვა სრული გზიდან, ხოლო **basename**-ით მხოლოდ ფაილის სახელი (**basename**-ს სხვა დატვირთვაც აქვს). ასე:

```
achiko@debian:~$ dirname $x
/usr/share/common-licenses
achiko@debian:~$ basename $x
GPL-3
```

ეს ხერხი გაცილებით მარტივია, თუმცა უნდა აღინიშნოს ისიც, რომ **dirname** და **basename** მხოლოდ ფაილებზე მოქმედებს (ანუ გამყოფი „/“ უნდა იყოს და არა სხვა სიმბოლო) **expr** და **echo** ბრძანებები კი ზოგადია. მაგალითად, თუ გვსურს, რომ **PATH** ცვლადიდან პირველი და ბოლო დირექტორიის სრული გზა ამოვიღოთ, შემდეგი ბრძანებები უნდა გავუშვათ:

```
achiko@debian:~$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
achiko@debian:~$ echo ${PATH##*/:*}          # ბოლო დირექტორია
/usr/games
achiko@debian:~$ expr $PATH : '.*:(.*)'      # ბოლო დირექტორია
```

```

/usr/games
achiko@debian:~$ echo ${PATH%:*}      # პირველი დირექტორია
/usr/local/bin
achiko@debian:~$ expr $PATH : '\([^:]*\)'  # პირველი დირექტორია
/usr/local/bin

```

10.2 მარტივი არითმეტიკული გამოთვლები

`expr` ბრძანება აგრეთვე გამოიყენება მარტივი მათემატიკური გამოთვლებისთვის. მას შეუძლია შემდეგი არითმეტიკული ოპერაციების შესრულება (ოპერატორების წინ „\“ იწერება):

```

achiko@debian:~$ expr 7 \+ 5      # მიმატება
12
achiko@debian:~$ expr 7 \- 5      # გამოკლება
2
achiko@debian:~$ expr 7 \* 5      # გამრავლება
35
achiko@debian:~$ expr 7 \/ 5      # გაყოფა
1
achiko@debian:~$ expr 7 \% 5      # ნაშთით გაყოფა
2
achiko@debian:~$ expr \( 11 \+ 5\) \* 3
48

```

არითმეტიკული გამოთვლები `echo` ბრძანებითაც ხორციელდება. ამ დროს გამოსახულება ორმაგ ფრჩხილში (()) უნდა მოვათავსოთ. შესაძლებელია, კვადრატული ფრჩხილების გამოყენებაც [], ასე:

```

achiko@debian:~$ echo $((7+5))
12
achiko@debian:~$ echo ${7+5}
12
achiko@debian:~$ echo $((2**5))      # ხარისხში აყვანა
32
achiko@debian:~$ echo $[(7+5*3-1)/4]
5

```

როგორც ვხედავთ, ასე მხოლოდ მთელ რიცხვებთან მუშაობა შეგვიძლია. ათწილადებისთვის კი `bc` (Basic Calculator) ბრძანება უნდა გამოვიყენოთ. ის შელში კალკულატორის ფუნქციას ასრულებს. მოდით, გავუშვათ და ვნახოთ მარტივი მაგალითები:

```

achiko@debian:~$ bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation,
Inc.
This is free software with ABSOLUTELY NO WARRANTY.

```

```

For details type `warranty'.
4+3
7

3^4
81

(23-15)/4+98
100

15/4
3

scale=5 # ციფრების რაოდენობა მძიმის შემდეგ
15/4
3.75000

quit # ამ ბრძანებით გამოვდივართ bc ბრძანებიდან

```

bc ბრძანებით რთული გამოთვლების წარმოებაც არის შესაძლებელი. ტრიგონომეტრიული ($\sin(x)$, $\cos(x)$, $\arctg(x)$...), ლოგარითმული ($\ln(x)$, $\log_n(x)$...) და სხვა ფუნქციების გამოსაყენებლად ბრძანებას უნდა ჩავუერთოთ სტანდარტული მათემატიკური ბიბლიოთეკა, -l ოფციით. ეს ოფცია, ნაგულისხმევი მნიშვნელობით, მძიმის შემდეგ 20 ციფრს გვაჩვენებს.

```

achiko@debian:~$ bc -l
1/17
.05882352941176470588

```

ეს ყველაფერი, რაც აქამდე ვნახეთ, შეგვიძლია, აგრეთვე არაინტერაქტიულ რეჟიმში შევასრულოთ, ასე:

```

achiko@debian:~$ echo "4.5*3" | bc
25.71
achiko@debian:~$ echo "scale=5; (123.45/78.9)*2" | bc
3.12926
achiko@debian:~$ echo "2^1000" | bc          # 2 ხარისხად 1000
10715086071862673209484250490600018105614048117055336074437503883703\
51051124936122493198378815695858127594672917553146825187145285692314\
04359845775746985748039345677748242309854210746050623711418779541821\
53046474983581941267398767559165543946077062914571196477686542167660\
429831652624386837205668069376

```

```

achiko@debian:~$ x=7; echo "scale=10; sqrt($x)" | bc -l          #  $\sqrt{x}$ 
2.6457513110

```

```
achiko@debian:~$ echo "scale=500; 4*a(1)" | bc -l      # arctg(1)= $\pi/4$ 
3.141592653589793238462643383279502884197169399375105820974944592307\
81640628620899862803482534211706798214808651328230664709384460955058\
22317253594081284811174502841027019385211055596446229489549303819644\
28810975665933446128475648233786783165271201909145648566923460348610\
45432664821339360726024914127372458700660631558817488152092096282925\
40917153643678925903600113305305488204665213841469519415116094330572\
70365759591953092186117381932611793105118548074462379962749567351885\
75272489122793818301194912
```

როგორც ვნახეთ, bc ბრძანებას დიდი შესაძლებლობები აქვს. თუმცა, shell ნამდვილად არ შექმნილა რთული არითმეტიკული გამოთვლების საწარმოებლად.

თავი 11

ტექსტის დამუშავება

ტექსტური ფაილები ფართოდაა წარმოდგენილი UNIX-ის მსგავს ყველა სისტემაში. ბევრი მომხმარებელი, როდესაც საქმე პატარა ტექსტს ეხება, დოკუმენტებს ტექსტურ ფორმატში ინახავს. ასეთ ფორმატში დიდი დოკუმენტებიც შენახვაცაა შესაძლებელი. ერთ-ერთი გავრცელებული მიდგომა დიდი დოკუმენტების შექმნისას, სწორედ ისაა, რომ მათ ჯერ ტექსტურ ფორმატში ქმნიან და შემდეგ კი სვამენ ერთ-ერთ ისეთ სტრუქტურაში, სადაც გარკვეული აღნიშვნებით, მონიშვნებით, ტექსტის გამოტანის გარეგნულ სტილს განსაზღვრავენ. მათ მონიშვნების ენას (markup language) უწოდებენ. მსოფლიოში ყველაზე გავრცელებული ელექტრონული დოკუმენტი, ალბათ, ვებ-გვერდებია. ისინი ტექსტური ფორმატის ფაილებია, რომლებიც დოკუმენტის ვიზუალური ფორმატის აღსაწერად ჰიპერტექსტის (HTML) ან გაფართოებულ მონიშვნის ენას (XML) იყენებენ. ელექტრონული ფოსტაც ტექსტზე დაფუძნებული საშუალებაა. ელ. ფოსტაში დაწერილი ტექსტი, მასზე მიბმული ნებისმიერი ფაილი, საბოლოოდ, ტექსტად წარმოდგინდება და ისე იგზავნება ადრესატთან. UNIX-ის მსგავს სისტემებში პრინტერს დასაბეჭდად ფაილის შიგთავსი ტექსტად გადაეცემა. თუ დასაბეჭდი გვერდი გრაფიკულ გამოსახულებას შეიცავს, ის ჯერ კონვერტირდება ტექსტურ ფორმატში პოსტსკრიფტ (PostScript) - გვერდის აღწერის ენით, შემდეგ კი გადაეცემა პროგრამას, რომელიც ქმნის გრაფიკულ წერტილებს. საბოლოოდ, სწორედ ეს წერტილები იბეჭდება.

არც ჩვენთვის უკვე კარგად ნაცნობი ბრძანებებია გამონაკლისი. ამ პროგრამების საწყისი კოდი, თავდაპირველად, სწორედ ტექსტურ ფორმატში დაიწერა და შემდეგ დაკონვერტირდა ორობით ფორმატში.

განვიხილოთ ბრძანებები, რომლებსაც ტექსტის დამუშავებისას ხშირად ვიყენებთ.

11.1 tac

tac ბრძანება cat ბრძანებისგან განსხვავებით, ფაილის შიგთავსს შებრუნებული რიგითობით, ამოტრიალებულად გვაჩვენებს. მაგალითისთვის შევქმნათ ერთი მარტივი ტექსტური ფაილი test.txt.

```
achiko@debian:~$ echo erti > test.txt
achiko@debian:~$ echo ori >> test.txt
achiko@debian:~$ echo sami >> test.txt
```

ზოგადად, სატესტოდ გამოსაყენებელი ტექსტებისთვის ხშირად გამოიყენებენ „Lorem

ipsum“¹ ტექსტს.

```
achiko@debian:~$ cat test.txt
erti
ori
sami
```

```
achiko@debian:~$ tac test.txt
sami
ori
erti
```

11.2 head, tail

ბრძანება **head**-ს ფაილის საწყისი ნაწილი გამოაქვს, **tail** ბრძანებას - ბოლო ნაწილი.

```
achiko@debian:~$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

head, ნაგულისხმევი მნიშვნელობით, **/etc/passwd** ფაილის პირველ 10 ხაზს გამოიტანს.

head ბრძანება საქმეში როგორც ფილტრი, ისე რომ ვნახოთ, ჯერ **/etc/passwd** გავხსნათ, შემდეგ, ეკრანზე გამოტანის ნაცვლად, **pipe**-ის დახმარებით, **head** ბრძანებას გადავცეთ გასაფილტრად. შედეგი ერთი და იგივე იქნება:

```
achiko@debian:~$ cat /etc/passwd | head
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

¹ Lorem ipsum არის ნიმუშისთვის აღებული უმინიმუმ ტექსტი, რომელიც საბეჭდო და ტიპოგრაფიულ ინდუსტრიაში გამოიყენება. იგი სტანდარტად 1500-იანი წლებიდან იქცა, როდესაც უცნობმა მბეჭდავმა ამაყობა დაზგაზე წიგნის საცდელი ეგზემპლარი დაბეჭდა. Lorem Ipsum-ის გამოყენებით ვლენულობთ იმაზე მეტ-ნაკლებად სწორი გადართილების ტექსტს, ვიდრე ერთი და იგივე გამეორებადი სიტყვებია ზოლმე. შედეგად, ტექსტი ჩვეულებრივ ინგლისურს ჰგავს, მისი წაითხვა და გაგება კი შეუძლებელია. გავრცელებული მოსაზრებით, Lorem Ipsum შემთხვევითი ტექსტი სულაც არაა. მისი ფესვები ჯერ კიდევ ჩვ. წ. აღ-მდე 45 წლის დროინდელი კლასიკური ლათინური ლიტერატურიდან მოდის.


```
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

ეკრანზე ხაზების სასურველი რაოდენობის გამოსატანად -n ოფცია უნდა გამოვიყენოთ. პირველი 3 ხაზის გამოტანა თუ გვსურს, ბრძანება ასე უნდა ჩავწეროთ:

```
achiko@debian:~$ cat /etc/passwd | head -n 3
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

n ოფციის მითითება აუცილებელი არ არის. ასეთი ჩანაწერიც იმავე შედეგს მოგვცემს.

```
achiko@debian:~$ cat /etc/passwd | head -3
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

ხაზების გარდა, head-ს შეუძლია საწყისი N რაოდენობის ბაიტი გამოიტანოს ეკრანზე -c ოფციით.

```
achiko@debian:~$ cat /etc/passwd | head -c 8
root:x:0achiko@debian:~$
```

ყურადღება

8 ბიტიან ASCII კოდირებაში ერთი ასო-ნიშანი ერთ ბაიტი ზომისაა.

შესაბამისად, ეს ბრძანება /etc/passwd-დან პირველ 8 ასო-ნიშანს გვანახებს.

head ბრძანებაზე ჩვენ მიერ ნახსენებ ყველა ოფციას - tail ბრძანებაც იგებს, მხოლოდ, საწყისი ნაწილის ნაცვლად, მას ბოლო ნაწილი გამოაქვს (ბოლო ხაზები, ბოლო ასო-ნიშნები).

```
achiko@debian:~$ cat /etc/passwd | tail -3
sshd:x:112:65534:./run/sshd:/usr/sbin/nologin
saned:x:114:118:./var/lib/saned:/bin/false
achiko:x:1000:1000:Archil:/home/achiko:/bin/bash
```

ამ ჩანაწერით tail გამოიტანს /etc/passwd-ის ბოლო 3 ხაზს.

```
achiko@debian:~$ cat /etc/passwd | tail -c 8
in/bash
achiko@debian:~$
```

ეს ბრძანება `/etc/passwd` ფაილის ბოლო 8 ასო-ნიშანს გამოიტანს. აქ, მე-8 ასო-ნიშანი უხილავია, ის ახალ ხაზზე გადასვლის სიმბოლოა. ამიტომაც აღმოჩნდა `shell`-ის მოსაწვევი შემდეგ ხაზზე.

`tail`-ს, `head`-ისგან განსხვავებით, დამატებით ერთი ძალიან სასარგებლო ოფცია აქვს, `-f`. როგორც ვნახეთ, `head` და `tail` ბრძანებები ეკრანზე გვაჩვენებენ შედეგს, ამით პროცესი მთავრდება და სხვა ბრძანების აკრეფის საშუალება გვეძლევა `shell`-ში. `-f` ოფციის მითითებით კი ეს პროცესი არ წყდება, არამედ გააგრძელებს ფაილში დამატებული ტექსტის ჩვენებას ეკრანზე დინამიკურად და მანამდე გაგრძელდება, სანამ ჩვენ თვითონ ხელით, `Ctrl+C` კლავიშების კომბინაციით, არ შევწყვეტთ. ანუ ასე ფაილის შიგთავსს რეალურ დროში ვუყურებთ. დემონსტრირებისთვის გავხსნათ 2 ტერმინალი და ავკრიფოთ შესაბამისი ბრძანებები:

ტერმინალი 1

```
$ tail -f test.txt
erti
ori
sami
...
```

ტერმინალი 2

```
$ ls >> test.txt
$ ls /bin >> test.txt
$
```

პირველ ტერმინალზე დავინახავთ იმას, რაც `test.txt`-ს ბოლოში დავუმატეთ მეორე ტერმინალიდან. რა ინფორმაციაც უნდა დავუმატოთ `test.txt`-ს ბოლოში, ყველაფერი გამოჩნდება პირველ ტერმინალში, სანამ `Ctrl+C`-ით არ დავხურავთ `tail -f` ბრძანებას.

11.3 sort

ბრძანება `sort`-ით ფაილების შიგთავსს ვახარისხებთ, ნაგულისხმევი მნიშვნელობით ლათინური ანბანის მიხედვით. მაგალითისთვის ავიღოთ `/etc/passwd` ფაილი.

```
achiko@debian:~$ cat /etc/passwd | sort
avahi:x:111:116::/var/run/avahi-daemon:/bin/false
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
colord:x:113:117::/var/lib/colord:/bin/false
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
...
```

დავინახავთ, რომ ფაილის შიგთავსი ხაზების პირველი ასო-ნიშნების გათვალისწინებითაა დალაგებული.

```
achiko@debian:~$ cat /etc/passwd | sort -r
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
```

```
usbmux:x:106:46:usbmux daemon,,,:/var/lib/usbmux:/bin/false
sys:x:3:3:sys:/dev:/usr/sbin/nologin
...
```

-r ოფციით შედეგი უკუღმა, შებრუნებული რიგითობით გამოვა ეკრანზე.

თუ ფაილი შეიცავს რიცხვებს, შესაძლებელია, იგი დავალაგოთ არა ანბანის, არამედ რიცხვების მიხედვით.

/etc/passwd ფაილში, როგორც ვნახეთ, ინფორმაცია სვეტებადაა ორგანიზებული და მათ შორის გამყოფი ასო-ნიშანია ორწერტილი :. ამ ფაილს დეტალურად მოგვიანებით განვიხილავთ. მესამე სვეტში მხოლოდ რიცხვებია მოცემული. მოდით, დავახარისხოთ ფაილის შიგთავსი ხაზების მესამე სვეტში ჩაწერილი რიცხვების მიხედვით. -n სწორედ რიცხვითი მნიშვნელობით ახარისხებს ხაზებს. -k ოფციით კი შესაძლებელია სასურველი სვეტის მითითება. რადგან საქმე სვეტს ეხება, უნდა ვიცოდეთ სად მთავრდება ერთი სვეტი და სად იწყება მეორე, ანუ სვეტებს შორის გამყოფი სიმბოლო მითითებაც საჭიროა. ეს -t ოფციითაა შესაძლებელი. მის გარეშე გამოტოვება ან ტაბულაცია იგულისხმება. ჩვენს შემთხვევაში, ეს სიმბოლო ორწერტილია.

```
achiko@debian:~$ cat /etc/passwd | sort -n -k3 -t:
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
...
```

მოდით, -k ოფციის უფრო ფართო შესაძლებლობები ვნახოთ. ნიმუშისთვის ავიღოთ ერთი ტექსტური ფაილი, სადაც ჩამოწერილი გვაქვს ლინუქსის დისტრიბუტივების სახელები, მათი ვერსიების ნომრები და გამოშვების თარიღები (დღე/თვე/წელი).

```
achiko@debian:~$ cat distros.txt
Debian 9.4 10/03/2018
Debian 8.10 09/12/2017
Debian 7.11 04/06/2016
Ubuntu 18.04 26/04/2018
Ubuntu 17.10 19/10/2017
Ubuntu 16.10 13/10/2016
RHEL 7.5 10/04/2018
RHEL 7.4 01/08/2017
RHEL 7.3 03/11/2016
Fedora 28 01/05/2018
Fedora 27 14/11/2017
Fedora 26 11/07/2017
```

ახლა დავახარისხოთ ეს სია გამოშვების თარიღის მიხედვით.

```
$ cat distro.txt | sort -k3.7nbr -k3.4nbr -k3.1nbr
Fedora 28 01/05/2018
Ubuntu 18.04 26/04/2018
RHEL 7.5 10/04/2018
Debian 9.4 10/03/2018
Debian 8.10 09/12/2017
Fedora 27 14/11/2017
Ubuntu 17.10 19/10/2017
RHEL 7.4 01/08/2017
Fedora 26 11/07/2017
RHEL 7.3 03/11/2016
Ubuntu 16.10 13/10/2016
Debian 7.11 04/06/2016
```

-k3.7 ჩანაწერით ვუთითებთ, რომ დალაგება მოხდეს მე-3 სვეტის მე-7 ელემენტიდან (ანუ წელიწადით), -k3.4 და -k3.1 კი დალაგებას მე-3 სვეტის მე-4 (თვე) და პირველი (დღე) ელემენტიდან ახორციელებს. n და r კლებადობით დალაგებას უზრუნველყოფს რიცხვებით, b ოფცია კი უზრუნველყოფს დასაწყისში არსებული გამოტოვებების (leading blanks) უგულებელყოფას.

მოდით, უფრო დეტალურად შევხედოთ: sort ბრძანება ასე აღიქვამს ხაზს - „Debian 9.4 10/03/2018“. პირველი სვეტია „Debian“, მეორე - „9.4“, მესამე კი - „10/03/2018“. ანუ მეორე და მესამე სვეტებს წინ გამოტოვებები შერჩათ. სწორედ ამიტომაც აუცილებელია b ოფციის მითითება. სვეტებს შორის გამყოფი რომ ორწერტილი (:) ყოფილიყო, b ოფცია საჭირო აღარ იქნებოდა.

ლინუქსში ოფციების მითითებისას ყურადღება უნდა მივაქციოთ ასო-ნიშნის რეგისტრს. ასე, მაგალითად, -r (--reverse) თუ ანბანის მიხედვით უკუღმა ახარისხებს ხაზებს, -R (--random-sort) ოფციით დახარისხება შემთხვევითი წესით ხდება. -R ოფციით ყოველ ჯერზე სხვადასხვა რიგითობით გამოვა ხაზები.

```
achiko@debian:~$ cat distro.txt | sort -R
Debian 8.10 09/12/2017
RHEL 7.4 01/08/2017
...
achiko@debian:~$ cat distro.txt | sort -R
Debian 7.11 04/06/2016
Fedora 27 14/11/2017
...
```

11.4 uniq

ბრძანება **uniq** ფილტრავს ხაზებს და შეუძლია გამოიტანოს უნიკალური (ყველასგან განსხვავებული) ან გამეორებული ხაზები, გვანახოს მათი რეოდენობა და ა.შ.

ავიღოთ კვლავ ჩვენი სატესტო ფაილი test.txt და ისე დავარედაქტიროთ, რომ მასში ზოგი ხაზი მეორედ მოხდეს, ზოგიც - არა.

```
achiko@debian:~$ cat test.txt
erti
ori
ori
sami
erti
ori
```

თუ ბრძანება `uniq`-ს გადავცემთ ამ გამოსასვლელს, ერთმანეთზე მიწყობილად გამეორებული ხაზების ნაცვლად, მხოლოდ ერთი ეგზემპლიარი გამოვა ეკრანზე. ჩვენს შემთხვევაში, ასეთი მე-2 და მე-3 ხაზებია. ერთმანეთზე მიჯრილი სხვა ერთნაირი ხაზები ჩვენს ფაილში არ გვაქვს.

```
achiko@debian:~$ cat test.txt | uniq
erti
ori
sami
erti
ori
```

რადგან `uniq` მიწყობილ ხაზებს უყურებს, ამიტომ ის ხშირად გამოიყენება `sort`-თან კომბინაციაში. `sort` ჯერ ყველა ერთნაირ ხაზებს მიაწყობს ერთმანეთს, შემდეგ კი `uniq` მხოლოდ ერთ ეგზემპლიარს გამოიტანს.

```
achiko@debian:~$ cat test.txt | sort | uniq
erti
ori
sami
```

მოდით, მხოლოდ უნიკალური ხაზები გამოვიტანოთ, `-u` (`--unique`) ოფციით:

```
achiko@debian:~$ cat test.txt | sort | uniq -u
sami
```

ახლა კი მხოლოდ გამეორებული ხაზები, დუბლიკატები გამოვიტანოთ. ამისთვის `-d` (`--repeated`) ოფცია უნდა გამოვიყენოთ:

```
achiko@debian:~$ cat test.txt | sort | uniq -d
erti
ori
```

შეგვიძლია, დავთვალოთ კიდევ ეს ხაზები `-c` (`--count`) ოფციის დამატებით:

```
achiko@debian:~$ cat test.txt | sort | uniq -dc
      2 erti
      3 ori
```

ოფციის გრძელი ფორმატით ეს ბრძანება ასე ჩაიწერება. შედეგი ერთი იქნება:

```
achiko@debian:~$ cat test.txt | sort | uniq --repeat --count
      2 erti
      3 ori
```

uniq-ით ხაზების შედარებისას შესაძლებელია პირველი N ასო-ნიშნის გამოტოვება **-s** (**--stable**) ოფციით. ასეთი შემთხვევა ძირითადად მაშინ გვხვდება, როდესაც შესადარებელ ხაზებს წინ თავსართი უწერიათ, მაგალითად, როდესაც ისინი დანომრილია და საჭიროა ამ ნომრების უგულებელყოფა. მოდით, ავიღოთ ჩვენი სატესტო ფაილი დანომრილი ფორმით და გამოვიტანოთ მხოლოდ უნიკალური ხაზი.

```
achiko@debian:~$ cat test.txt
1 erti
2 ori
3 ori
4 sami
5 erti
6 ori
achiko@debian:~$ cat test.txt | sort -k2b | uniq -s2 -u
4 sami
```

შედეგი სწორია. ხაზი „sami“, იმის გათვალისწინებით, რომ პირველი 2 სიმბოლო (ნომერი და გამოტოვება) უგულებელვყავით, ნამდვილად ერთადეთია ამ ფაილში.

ბრძანება **uniq**-ს სხვა შესაძლებლობებიც აქვს და მათი ნახვა მის სახელმძღვანელო გვერდზე შესაძლებელია: **man uniq**.

11.5 cut

ბრძანება **cut**-ით ფაილებში ხაზის გარკვეული ნაწილის გამოტანა შეიძლება. მისი შემდეგი ორი ოფცია ყველაზე ხშირად გამოიყენება. ესენია:

-c N გამოაქვს ფაილის ყოველი ხაზის მე-N ბაიტი.

-f N გამოიტანს მე-N სვეტს, თუ ფაილის შიგთავსი სვეტებად არის ორგანიზებული. აქ მნიშვნელოვანია ვიცოდეთ, რა ასო-ნიშნითაა გამოყოფილი სვეტები ერთმანეთისგან. ნაგულისხმევი მნიშვნელობით, როგორც სხვა ბრძანებების მსგავსად, **cut** ბრძანებისთვისაც სვეტებს შორის გამყოფი ტაბულაციაა. თუ სხვა ასო-ნიშანი გვსურს ასეთ სიმბოლოდ გამოვიყენოთ, მაშინ **-f** ოფციასთან კომბინაციაში **-d** ოფცია უნდა მივუწეროთ და მას ეს სიმბოლო ასე მივუთითოთ **-d:**.

N შეიძლება იყოს რიცხვი, მძიმით გამოყოფილი ჩამონათვალი ან ინტერვალი.

N-ის შესაძლო მნიშვნელობები	
7	მე-7
2,5,11	მე-2, მე-5 და მე-11
3-8	მე-3-დან 8-ის ჩათვლით
3-	მე-3-დან ბოლომდე
-5	დასაწყისიდან მე-5-ს ჩათვლით

მოდით, მაგალითები ვნახოთ. სატესტოდ Lorem ipsum ტექსტი ავიღოთ.

```
achiko@debian:~$ cat lorem.txt
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Nam ultricies elit at cursus iaculis.
Nunc suscipit lacus egestas ullamcorper efficitur.
Maecenas hendrerit sapien suscipit tincidunt egestas.
Cras ultrices diam quis faucibus rhoncus.
```

ეს ბრძანება lorem.txt-ის თითოეული ხაზიდან მე-7 ბაიტს (ასო-ნიშანს) გამოიტანს.

```
achiko@debian:~$ cat lorem.txt | cut -c7
i
t
u
a
l
```

ეს კი - მე-7 ბაიტიდან ყველა ასო-ნიშანს ბოლომდე:

```
achiko@debian:~$ cat lorem.txt | cut -c7-
orempit amet, consectetur adipiscing elit.
am urat cursus iaculis.
unc s egestas ullamcorper efficitur.
aecessapien suscipit tincidunt egestas.
ras tquis faucibus rhoncus.
```

შემდეგი ბრძანება lorem.txt-დან თითოეული ხაზის მე-2-დან მე-5-ს ჩათვლით, მე-8-სა და მე-20-დან დაწყებული ყველა ასო-ნიშანს გამოიტანს.

```
achiko@debian:~$ cat lorem.txt | cut -c 2-5,8,20-
ipsum dolor sit amet, consectetur adipiscing elit.
tricies elit at cursus iaculis.
uscipit lacus egestas ullamcorper efficitur.
as hendrerit sapien suscipit tincidunt egestas.
ltrices diam quis faucibus rhoncus.
```

მოდით, ფაილიდან ახლა სვეტები გამოვიტანოთ. ამისთვის ჯერ საჭიროა, გვქონდეს ისეთი ფაილი, რომლის შიგთავსიც სვეტებადაა ორგანიზებული. ჩვენთვის ნაცნობი ერთ-ერთი ასეთი ფაილია /etc/passwd. გამოვიტანოთ მე-3 სვეტი:

```
achiko@debian:~$ cat lorem.txt | cut -f3
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
...
```

ეკრანზე მთლიანი შიგთავსი გამოვიდა და არა მესამე სვეტი. საქმე ისაა, რომ ჩვენ არ მიგვითითებია სვეტებს შორის გამყოფი სიმბოლო. შესაბამისად, cut ბრძანებამ ტაბულაციის სიმბოლო იგულისხმა. თუმცა, რადგან ამ ფაილში ტაბულაციაც არ არის ჩაწერილი, ამიტომ გამოვიტანა ასეთი გამოსასვლელი.

მოდით, მივუთითოთ სვეტებს შორის გამყოფი სიმბოლო და ვნახოთ რა გამოვა:

```
achiko@debian:~$ cat lorem.txt | cut -f3 -d:
0
1
2
3
...
```

ეს შედეგი ნამდვილად მესამე სვეტია.

```
achiko@debian:~$ cat lorem.txt | cut -f1,7 -d:
root:/bin/bash
daemon:/usr/sbin/nologin
bin:/usr/sbin/nologin
sys:/usr/sbin/nologin
...
```

ამგვარად, პირველი და მეშვიდე სვეტი გამოვა.

11.6 paste

paste ბრძანება, cut-სგან განსხვავებით, სხვადასხვა ფაილების შიგთავსს სვეტებად აერთიანებს.

ავიღოთ 3 ფაილი (names.txt, version.txt, date.txt), სადაც სათითაოდ ჩაწერთ ჩვენს მიერ უკვე გამოყენებული distro.txt-დან თითო სვეტს. ხელით რომ არ ვაკეთოთ ეს ყველაფერი, მოდით უკვე ნასწავლი ბრძანებები გამოვიყენოთ.

```
achiko@debian:~$ cat distros.txt | cut -f1 -d " " > names.txt
achiko@debian:~$ cat distros.txt | cut -f2 -d " " > versions.txt
achiko@debian:~$ cat distros.txt | cut -f3 -d " " > dates.txt
```


შევამოწმოთ.

```
achiko@debian:~$ cat names.txt version.txt date.txt
```

```
Debian
Debian
Debian
Ubuntu
Ubuntu
Ubuntu
RHEL
RHEL
RHEL
Fedora
Fedora
Fedora
9.4
8.10
7.11
18.04
17.10
16.10
7.5
7.4
7.3
28
27
26
10/03/2018
09/12/2017
04/06/2016
26/04/2018
19/10/2017
13/10/2016
10/04/2018
01/08/2017
03/11/2016
01/05/2018
14/11/2017
11/07/2017
```

ნამდვილად ისაა, რაც გვინდოდა. ახლა კი **paste** ბრძანებით შეგვიძლია გავაერთიანოთ ამ სამი დამოუკიდებელი ფაილის შიგთავსები იმ თანმიმდევრობით, როგორითაც გვსურს.

```
achiko@debian:~$ paste dates.txt names.txt versions.txt
```

```
10/03/2018      Debian  9.4
09/12/2017      Debian  8.10
04/06/2016      Debian  7.11
26/04/2018      Ubuntu  18.04
```

19/10/2017	Ubuntu	17.10
13/10/2016	Ubuntu	16.10
10/04/2018	RHEL	7.5
01/08/2017	RHEL	7.4
03/11/2016	RHEL	7.3
01/05/2018	Fedora	28
14/11/2017	Fedora	27
11/07/2017	Fedora	26

შეგვიძლია, აგრეთვე, სვეტებს შორის გამოყოფ სიმბოლოდ ტაბულაციის ნაცვლად, სხვა, მაგალითად, ორწერტილი ავირჩიოთ, ასე:

```
achiko@debian:~$ paste -d: dates.txt names.txt versions.txt
10/03/2018:Debian:9.4
09/12/2017:Debian:8.10
04/06/2016:Debian:7.11
26/04/2018:Ubuntu:18.04
19/10/2017:Ubuntu:17.10
13/10/2016:Ubuntu:16.10
10/04/2018:RHEL:7.5
01/08/2017:RHEL:7.4
03/11/2016:RHEL:7.3
01/05/2018:Fedora:28
14/11/2017:Fedora:27
11/07/2017:Fedora:26
```

-s ოფციით შესაძლებელია ფაილის თითოეული ხაზის შიგთავსი ერთ ხაზზე განვათავსოთ (ნაგულისხმევი მნიშვნელობით აქაც ტაბულაციით გამოყოფილი) და ასეთი ფორმატი გამოვიტანოთ შედეგი:

```
achiko@debian:~$ paste -s dates.txt names.txt versions.txt
Debian Debian Debian Ubuntu Ubuntu Ubuntu RHEL ...
9.4      8.10    7.11    18.04   17.10   16.10   7.5    ...
10/03/2018      09/12/2017      04/06/2016      ...
```

ამგვარად, მოვახდინეთ ე.წ. ტრანსპონირება.²

11.7 join

join ბრძანება paste ბრძანებისგან იმით განსხვავდება, რომ ის მხოლოდ 2 ფაილის შიგთავსს აერთებს სვეტებად და ისიც, მხოლოდ იმ შემთხვევაში, თუ ამ 2 ფაილს საერთო სვეტი აქვს.

დემონსტრაციისთვის ავიღოთ შემდეგი 2 ფაილი - countries.txt და cities.txt:

²მატრიცის ტრანსპონირებისას, ვიღებთ ახალ მატრიცას, რომლის სვეტები საწყისი მატრიცის სტრიქონებს წარმოადგენს.

```
achiko@debian:~$ cat countries.txt
1 Georgia
2 France
3 USA
4 Germany
5 Great Britain
achiko@debian:~$ cat cities.txt
1 Tbilisi
2 Paris
3 Washington
5 London
4 Berlin
```

```
achiko@debian:~$ join countries.txt cities.txt
1 Georgia Tbilisi
2 France Paris
3 USA Washington
join: cities.txt:5: is not sorted: 4 Berlin
5 Great Britain London
```

ამიტომ, უმჯობესია, `join` ბრძანების გამოყენებამდე ფაილების შიგთავსი დახარისხებული იყოს, რაც მსგავს შემთხვევას გამორიცხავს. `join` ბრძანება, ნაგულისხმევი მნიშვნელობით, შედარებისას ორივე ფაილის პირველ სვეტს ითვალისწინებს. თუმცა, შეგვიძლია ეს ქცევა შევცვალოთ `-1` და `-2` ოფციების საშუალებით. მოდით, შევცვალოთ მოცემულობა:

```
achiko@debian:~$ cat countries.txt
Georgia 1
France 2
USA 3
Germany 4
Great Britain 5
achiko@debian:~$ cat cities.txt
1 Tbilisi
2 Paris
3 Washington
5 London
4 Berlin
```

მოცემულ შემთხვევაში საერთო სვეტი ერთი ფაილისთვის პირველია, მეორისთვის კი მეორე. ვიმოქმედოთ `join` ბრძანებით, ასე:

```
achiko@debian:~$ join -1 2 -2 1 countries.txt cities.txt
1 Georgia Tbilisi
2 France Paris
3 USA Washington
```

```
join: cities.txt:5: is not sorted: 4 Berlin
```

join-ის უფრო მეტი შესაძლებლობების სანახავად მის სახელმძღვანელო გვერდს შეგიძლიათ მიმართოთ `man join` ბრძანებით.

11.8 tr

ფაილის შიგთავსის შესაცვლელად რომელიმე ტექსტურ რედაქტორს მივმართავთ ხოლმე - კურსორის მიმართულების ისრები მიგვყავს შესაბამის ადგილზე და ვანახლებთ, ვამატებთ ან ვშლით ასო-ნიშნებს. თუმცა, არსებობს ტექსტის მოდიფიცირების არაინტერაქციული გზაც. ამ დროს შესაძლებელია ცვლილებების გაკეთება ბრძანებებით, თანაც რამდენიმე ფაილში ერთდროულად, ხელის ერთი მოხშით (ასეთი ხერხით ცვლილების გაკეთებას ინგლისურად „editing on the fly“-ს ეძახიან. ზემოთ ნახსენები ზოგიერთი ბრძანებით სწორედ ასე ვცვლიდით ფაილის შიგთავსს. მაგალითად, `cut`, `head`, `tail` ბრძანებებით). შეცვლილი შიგთავსი ფაილშივე არ ინახება, არამედ მხოლოდ ეკრანზე გამოდის. თუმცა, გადამისამართებით თავისუფლად შეგვიძლია მისი შენახვაც.

`tr` ბრძანებაც სწორედ ასეთი ბრძანებაა. ამ ბრძანებით შესაძლებელია ასო-ნიშნების წაშლა, ჩანაცვლება, შეკუმშვა. მოდით, ვნახოთ `tr`-ს პრაქტიკული გამოყენების რამდენიმე მაგალითი. შევცვალოთ პატარა ასოები დიდი ასოებით:

```
achiko@debian:~$ echo "lowercase letters" | tr abcdefghijklmnopqrstuvwxyz  
ABCDEFGHIJKLMNOPQRSTUVWXYZ  
LOWERCASE LETTERS
```

ასო-ნიშნების ერთობლიობა შეგვიძლია ინტერვალით მივუთითოთ ან POSIX-ის ასო-ნიშანთა კლასი გამოვიყენოთ:

```
achiko@debian:~$ echo "lowercase letters" | tr a-z A-Z  
LOWERCASE LETTERS
```

```
achiko@debian:~$ echo "lowercase letters" | tr [:lower:] [:upper:]  
LOWERCASE LETTERS
```

მოდით, მიეღოებით გამეორებული სიმბოლოები შეკვეცით, `-s` ოფციით:

```
achiko@debian:~$ echo "This is for testing" | tr -s "[:space:]"  
This is for testing
```

`-d` ოფციით შეგვიძლია კონკრეტული ასო-ნიშნების წაშლა:

```
achiko@debian:~$ echo "my UID is 432234" | tr -d "[:digit:]"  
my UID is
```

მოცემულ ასო-ნიშანთა ერთობლიობის ნაცვლად `-c` ოფციით შესაძლებელია

ვიმოქმედოთ მის დამატებაზე. წავშალოთ ციფრების დამატება, ანუ ციფრების გარდა სხვა ასო-ნიშნები:

```
achiko@debian:~$ echo "my UID is 432234" | tr -cd "[:digit:]"
432234
```

მოდით, ახლა ყველა ხაზი ერთ ხაზში მოვაქციოთ, ანუ ახალ ხაზზე გადასვლა შევცვალოთ გამოტოვებით:

```
achiko@debian:~$ tr '\n' ' ' < test.txt
erti ori ori sami erti ori  achiko@debian:~$
```

ასე კი ყველა უხილავი სიმბოლოს წაშლა შეგვიძლია (ახალ ხაზზე გადასვლაც უხილავი სიმბოლოა):

```
achiko@debian:~$ tr -cd '[:print:]' < test.txt
ertioriorisamiertioriachiko@debian:~$
```

tr ბრძანებას კიდევ ერთი საინტერესო გამოყენება აქვს ტექსტის ROT13 კოდირებით ჩაწერაში. ROT13-ით ლათინურ ანბანის ყოველი ასო 13 პოზიციით გადანაცვლდება. სახელიც სწორედ აქედან მოდის - rotate by 13 places. ამ მეთოდის ლათინურ ტექსტზე ორჯერ განხორციელებით კვლავ თავდაპირველ ტექსტს მივიღებთ, რადგან ლათინური ანბანი 26 ასო-ნიშნისაგან შედგება.

```
achiko@debian:~$ echo "secret text" | tr a-zA-Z n-za-mN-ZA-M
frperg grkg
achiko@debian:~$ echo "frperg grkg" | tr a-zA-Z n-za-mN-ZA-M
secret text
```

მსგავს კონტექსტში, კოდირების გარდა, ხშირად ვიყენებთ სხვა ტერმინებსაც. ასეთებია - დაშიფვრა და ჰეშირება. მოდით, დეტალურად განვმარტოთ ისინი და ვთქვათ, რა განსხვავებაა მათ შორის:

კოდირება - ამ დროს ხდება მონაცემების გარდაქმნა, გადაყვანა ერთი სისტემიდან მეორეზე და გადაყვანის ეს წესი საჯაროდაა ცნობილი (მაგალითად: სპეციალური სიმბოლოების ნახვა ვებ გვერდზე ან ბინარული მონაცემის გაზავნა ელ.ფოსტით). ამ დროს მიზანი არ არის ინფორმაციის დაფარვა, არამედ ის, რომ ადრესატმა სწორად დაინახოს გზავნილი. კოდირების მაგალითებია: ASCII, Unicode, URL encoding, Base64 და სხვა. პირველი კოდირება, რომელიც მე-19 საუკუნიდან გამოიყენებოდა, იყო მორზეს ანბანი.

დაშიფვრა - მისი მიზანია მონაცემი ისე გარდაქმნას სხვადასხვა ალგორითმებით, რომ ადრესატის გარდა, სხებისთვის გაუგებარი იყოს გზავნილი. ადრესატს დამატებითი შიფრით, ალგორითმით ან გასაღებით შეეძლება ამ მონაცემის გაშიფვრა. დაშიფვრის მაგალითებია: AES, Blowfish, RSA და სხვა.

ჰეშირება - მისი მიზანია მონაცემის მთლიანობის უზრუნველყოფა, ანუ მიმღებმა გაიგოს მისთვის გამოგზავნილი შეტყობინება გზაში შეიცვალა თუ არა. ტექნიკურად ასე ხდება - გასაგზავნი შეტყობინების დაწერის შემდეგ, სპეციალური ფუნქციის მეშვეობით, მოხდება მისი გარდაქმნა ანუ ჰეშირება. შედეგად, გენერირდება ფიქსირებული სიგრძის

გამოსასვლელი (მოკლედ ჰეში) ისე, რომ:

- იგივე შესასვლელით ყოველთვის იგივე გამოსასვლელი უნდა გენერირდებოდეს.
- სხვა არცერთი შესასვლელით არ უნდა გენერირდებოდეს ეს გამოსასვლელი.
- გამოსასვლელიდან ვერ უნდა ვღებულობდეთ შესასვლელს.
- ყოველგვარი ცვლილება მოცემულ შესასვლელში უნდა იწვევდეს ჰეშის მკვეთრ ცვლილებას.

იმისთვის, რომ ზუსტად დადგინდეს, რომ შეტყობინება არ შეიცვალა ჰეშირებასთან ერთად, გამოიყენება ავთენტიკაცია. ამისთვის აიღება მიღებული ჰეში და ხდება მისი დაშიფვრა გამგზავნის დახურული გასაღებით. მიღებული შედეგი იგზავნება შეტყობინებასთან ერთად და წარმოადგენს ციფრულ ხელმოწერას.

როდესაც ადრესატი მიიღებს შეტყობინებას, მან უნდა გაშიფროს მიღებული ინფორმაცია გამგზავნის ღია გასაღებით (რომელიც მას წინასწარ აქვს). თუ არსებულმა ღია გასაღებმა გაშიფრა დოკუმენტი, ე.ი. გამოგზავნი სწორია, რადგან მხოლოდ ადრესატისა და გამგზავნის გასაღებები წარმოადგენს წყვილს. გაშიფვრის შედეგად მივიღებთ ჰეშს. შემდეგ ადრესატი თავად შექმნის მიღებული შეტყობინების ჰეშს და შეადარებს გაშიფრულს. თუ ისინი დაემთხვევა ერთმანეთს, ე.ი. შეტყობინება არ შეცვლილა ხელმოწერის შემდეგ. სადღეისოდ ციფრული ხელმოწერის პროცესი მთლიანად ავტომატიზებულია თანამედროვე კომპიუტერული პროგრამებით. ჰეშირების მაგალითებია: SHA-3, MD5 (უკვე მოძველებული) და ა.შ.

ყურადღება

კოდირება ასოცირდება მონაცემების გამოყენებადობის შენარჩუნებასთან (ერთი და იგივე მონაცემი სხვადასხვა კოდირებით სხვადასხვა ფორტაში გადაიყვანება, რათა მოხდეს განსვავებულ სისტემებში მისი გამოყენება, ეფექტიანი ტრანსმისია ან შენახვა), დაშიფვრა - კონფიდენციალობის შენარჩუნებასთან, ხოლო ჰეშირება კი მონაცემთა მთლიანობის დაცვასთან.

tr ბრძანებით შეგვიძლია, შევქმნათ მარტივი შემთხვევითი პაროლების გენერატორი.

```
achiko@debian:~$ cat /dev/urandom | tr -dc _?A-Za-z-0-9 | head -c8; echo
dgAqC1X-
achiko@debian:~$ cat /dev/urandom | tr -dc _?A-Za-z-0-9 | head -c8; echo
JKA?5k8x
```

ამ შემთხვევაში, 8 სიმბოლოიანი პაროლების შემადგენელ ასოებად აღებული გვაქვს დიდი და პატარა ასოები, ციფრები, ტირე, ქვედა ტირე და კითხვის ნიშანი. შეგვიძლია პაროლის სიგრძე გაგზარდოთ და სირთულისთვის მასში სხვა სიმბოლოებიც დავამატოთ.

```
$ cat /dev/urandom | tr -dc _?!\@#%$\\(\_)A-Za-z-0-9 | head -c12; echo
u#24ks%cktHQ
$ cat /dev/urandom | tr -dc _?!\@#%$\\(\_)A-Za-z-0-9 | head -c12; echo
ARkrCGWZ#)v!
```

ლინუქსისმაგვარ სისტემებში /dev/urandom ფაილი არის სპეციალური ფაილი,

რომლითაც ყოველ მიმართვაზე გენერირდება შემთხვევითი სიმბოლოების თანმიმდევრობა. მის გარდა, არსებობს აგრეთვე `/dev/random` ფაილიც. მეტი სირთულისთვის აპლიკაციებში გამოსაყენებლად სწორედ `/dev/random` ფაილის გამოყენება არის რეკომენდებული, რადგან ის მხოლოდ მაშინ დააგენერირებს კოდს, როდესაც სისტემაში საკმარისი ენტროპია³ შეიქმნება. თუ ენტროპია არ არის საკმარისი `/dev/random` მომხმარებლის მიმართვას დაბლოკავს. `/dev/urandom` ფაილი კი ყოველ მიმართვაზე დაუბრუნებს მონაცემს. სინამდვილეში, ეს იმას ნიშნავს, რომ `/dev/urandom` ფაილით დაგენერირებული კოდი თეორიულად მოწყვლადი იქნება კრიპტოგრაფიული შეტევისას. `/dev/random` კი უზრუნველყოფს ძალიან მაღალი ხარისხის შემთხვევითობას მის მიერ დაგენერირებულ კოდში.

³ენტროპია ფიზიკაში თერმოდინამიკული სისტემის მახასიათებელი ფუნქციაა და სისტემის მოუწესრიგებლობის ხარისხს აღნიშნავს. ინფორმაციულ სისტემებში კი ენტროპია აღნიშნავს შემთხვევითობის ხარისხს და მის წყაროს სისტემაში არსებული ფიზიკური მოწყობილობები წარმოადგენს. მაგ: მაუსის ზმირი მოძრაობითაც შესაძლებელია სისტემაში ენტროპია გავზარდოთ. ზოგადად, ენტროპია მით უფრო გაიზრდება, რაც უფრო დაიტვირთება სისტემა სხვადასხვა ამოცანების შესრულებით.

თავი 12

სხვა სასარგებლო ბრძანებები

მ თავში განვიხილავთ ფაილებთან მიმართებაში რამდენიმე სასარგებლო ბრძანებას. ვისაუბრებთ დიდი ზომის ფაილების დაყოფის ხერხებზე, ფაილის მოძებნაზე სხვადასხვა კრიტერიუმით, ტექსტური ფაილების ფორმატირებაზე და ა.შ.

12.1 ტექსტის შედარება

ხშირად, მნიშვნელოვანი ფაილის დაკარგვის თავიდან აცილების მიზნით, მიზანშეწონილია, მისი რამდენიმე ასლის გაკეთება. თუმცა, თუ რომელიმე მათგანში მცირე ცვლილებას შევიტანთ, შეიძლება მოგვიანებით გაგვიჭირდეს განსხვავებების პოვნა ან გადამოწმება იმისა, არის თუ არა სხვაობა კონკრეტულ ფაილებს შორის.

cmp ბრძანება ბაიტ-ბაიტ ადარებს ორ ფაილს და გვეუბნება რომელ ხაზზე და რომელი ბაიტიდან იწყება განსხვავება.

მოდით, ავიღოთ ჩვენი სატესტო ფაილი გამარტივებული შიგთავსით, ასევე, მისი ასლი პატარა ნაირსხვაობით და შევადაროთ ისინი.

```
achiko@debian:~$ cat test.txt
erti
ori
sami
achiko@debian:~$ cat test2.txt
erti
otxi
sami
```

```
achiko@debian:~$ cmp test.txt test2.txt
test.txt test2.txt differ: byte 7, line 2
```

თუ განსხვავება არ არის ამ ორი ფაილის შიგთავსს შორის, ეკრანზე არაფერი გამოვა. ეს ბრძანება საუკეთესო საშუალებაა იმის გასაგებად, ფაილების შიგთავსი იდენტურია თუ არა.

comm ბრძანება cmp-ს მსგავსად, ადარებს ორ ფაილს და შედეგი ეკრანზე 3 სვეტად გამოაქვს. პირველში მოცემულია ის ხაზი, რომელიც მხოლოდ პირველ ფაილშია, მეორე სვეტში მხოლოდ მეორე ფაილის უნიკალური სტრიქონები, ხოლო მესამეში კი ერთნაირი, საერთო ხაზებია მოცემული.

```
achiko@debian:~$ comm test.txt test2.txt
      erti
ori
      otxi
      sami
```

შესაძლებელია, რომელიმე სვეტი ეკრანზე არ გამოვიტანოთ. -1, -2 ან/და -3 ოფციის გამოყენებით დავფარავთ, შესაბამისად, პირველ, მეორე ან/და მესამე სვეტს.

diff ბრძანებაც ფაილების შესადარებლად გამოიყენება, თუმცა ის უფრო კომპლექსური პროგრამაა და მეტი შესაძლებლობები აქვს. **diff**-ით ერთდროულად დიდი რაოდენობით ფაილების შედარებაც შეიძლება. მას აგრეთვე შეუძლია ფაილებს შორის განსხვავება სხვადასხვა ფორმატში გამოიტანოს. **diff** ხშირად გამოიყენება პროგრამისტების მიერ საკუთარი პროგრამების საწყისი კოდების სხვადასხვა ვერსიების შესადარებლად. ამ ბრძანების მიერ ეკრანზე გამოტანილ ფაილებს შორის განსხვავება, როგორც წესი, მომხმარებლისთვის ბოლომდე გასაგები არაა, თუმცა ეს ინფორმაცია ძალიან მნიშვნელოვანია. თუ მას ცალკე ფაილში დავიმახსოვრებთ, ანუ შევქმნით ე.წ. განსხვავების ფაილს (patch_file ან diff_file), **patch** ბრძანებით ამ ფაილითა და ერთ-ერთი ორიგინალი ფაილით მეორე ორიგინალი ფაილის შიგთავსის აღდგენას შევძლებთ.

მოდით, მოვახდინოთ ამის დემონსტრირება:

```
achiko@debian:~$ diff test.txt test2.txt
2c2
< ori
---
> otxi
achiko@debian:~$ diff test.txt test2.txt > t1_t2.diff
```

ახლა **patch** ბრძანებითა და **t1_t2.diff** ფაილით აღვადგინოთ ერთ-ერთი ორიგინალი ფაილი:

```
achiko@debian:~$ patch test.txt t1_t2.diff
patching file test.txt
achiko@debian:~$ diff test.txt test2.txt > t1_t2.diff
```

ამ ბრძანებით **test.txt**-ში აღვადგენთ **test2.txt**-ს შიგთავსს. ანუ, **test.txt**-ს თავდაპირველი შიგთავსი დაიკარგება. თუ გვსურს, რომ **test2.txt** შევინარჩუნოთ, მაშინ **-b**, **--backup** ოფცია უნდა გამოვიყენოთ. ამით **test.txt**-ის შიგთავსი **test.txt.orig**-ში შეინახება და შემდეგ აღდგება **test2.txt**-ის ინფორმაცია.

```
achiko@debian:~$ patch -b test.txt t1_t2.diff
achiko@debian:~$ ls test*
test2.txt  test.txt  test.txt.orig
```

ფაილის შიგთავსის აღდგენას განსხვავების ფაილითა და ერთ-ერთი ორიგინალი ფაილით მაშინ აქვს აზრი, როდესაც მეორე ორიგინალ ფაილს დავკარგავთ.

12.2 find

`find` ბრძანება ფაილებს მოცემული დირექტორიების მთლიან იერარქიულ სტრუქტურაში, ანუ მათ ყველა ქვე-დირექტორაშიც ეძებს.

პარამეტრების გარეშე `find` მიმდინარე დირექტორიიდან, `.`-დან დაიწყებს ძებნას და მოგვიძებნის მასში არსებულ ყველა ფაილს, შემდეგ კი ეკრანზე გამოგვითავს ამ სიას.

```
achiko@debian:~$ find
...
./versions.txt
./lessshst
./Music
./wget-hsts
```

შეგვიძლია კონკრეტული დირექტორია მივუთითოთ `find`-ს:

```
achiko@debian:~$ find ~
...
/home/achiko/versions.txt
/home/achiko/.lessshst
/home/achiko/Music
/home/achiko/.wget-hsts
```

ეს ბრძანება ჩვენს პირად დირექტორიაში არსებულ ყველა ფაილს გვანახებს ეკრანზე. სია, სავარაუდოდ, გრძელი იქნება. მოდით, დავთვალოთ ისინი:

```
achiko@debian:~$ find ~ | wc -l
839
```

`find` ბრძანების ერთ-ერთი ძლიერი მხარე ისაა, რომ მას დირექტორიებში ბევრი სხვადასხვა კრიტერიუმით შეუძლია ფაილების ძებნა - სახელით, ზომით, ტიპით და სხვა.

მოდით, მაგალითები მოვიყვანოთ. ვთქვათ, გვსურს დირექტორიების მოძებნა. მაშინ ხელით უნდა მივუთითოთ, რომ მხოლოდ დირექტორიის ტიპის ფაილების ძებნა გვსურს, ასე:

```
achiko@debian:~$ find ~ -type d
...
/home/achiko/Templates
/home/achiko/.ssh
/home/achiko/Music
```

`d`-ს გარდა `type` კრიტერიუმს შეიძლება გადავცეთ: `f` - ჩვეულებრივი ფაილების აღსანიშნავად, `l` მაღსახმობისთვის (ე.წ. shortcut)

```
achiko@debian:~$ find ~ -type f | wc -l
708
```

ასე დავითვლით მხოლოდ ჩვეულებრივი ტიპის ფაილებს. მოდით, დასახელებით მოვძებნოთ ფაილები.

```
achiko@debian:~$ find ~ -name test.txt
/home/achiko/test.txt
```

ფაილის სახელის მითითებისას ზოგადობისთვის და უფრო მეტი მოქნილობისთვის შეგვიძლია * სიმბოლოს გამოყენება.

```
achiko@debian:~$ find ~ -name "*.txt"
...
/home/achiko/test.txt
/home/achiko/lorem5.txt
/home/achiko/distros.txt
/home/achiko/versions.txt
```

ამ ბრძანებით მოვძებნით ყველა იმ ფაილს, რომლის სახელიც .txt-ზე მთავრდება.

```
achiko@debian:~$ find ~ -name "*.txt" -type f
```

ასე კი ყველა იმ ფაილს ვეძებთ, რომელიც ორივე კრიტერიუმს აკმაყოფილებს: სახელი .txt-ზე მთავრდება, და ამავდროულად, ჩვეულებრივი ტიპის ფაილია.

მოდით, ახლა, ზომის მიხედვით მოვძებნოთ ფაილები. ეს -size კრიტერიუმის საშუალებით ხორციელდება. მას ზომა და ერთეული უნდა მივუთითოთ. ზომის ერთეულებია: c - ბაიტის¹ აღსანიშნავად, w - ორ-ბაიტიანი სიტყვის აღსანიშნავად, k - კილობაიტი (=1024 ბაიტი), M - მეგაბაიტი (=1048576 ბაიტი), G გიგაბაიტი (=1073741824 ბაიტი) ან b - 512-ბაიტიანი ბლოკი. თუ ერთეულს არ მივუთითებთ, ნაგულისხმევი მნიშვნელობა სწორედ b იქნება.

ზოგადად, ბაიტი აღინიშნება „B“ სიმბოლოთი. არ აგვრიოთ ის „b“-ში. IEEE²-ს სტანდარტით „b“-თი ბიტი აღინიშნება. IEC³ სტანდარტით კი ბიტი არის „bit“.

რაოდენობრივად მეტის აღსანიშნავად შემდეგი თავსართები გამოიყენება: კილო (kilo), მეგა (mega), გიგა (giga), ტერა (tera), პეტა (peta), ეგზა (exa), ძეტა (zetta). მაგალითად, ბაიტის შემთხვევაში, KB ნიშნავს $10^3=1000$ ბაიტს. ანუ ეს თავსართი 10-ის ხარისხებით მოიცემა. რადგან ორობითი სისტემა ინფორმაციულ სისტემებში ძალიან მნიშვნელოვანია, IEC-მ 2-ის ხარისხებიანი შემდეგი თავსართები შემოიტანა: კიბიბაიტი (kibibyte), მებიბაიტი (mebibyte)... 1 KiB არის $2^{10}=1024$ ბაიტი. IEC-მ ეს ერთეულები

¹ბაიტი არის ზომის აღმნიშვნელი ერთეული ინფორმაციულ სისტემებში. ის შედგება 8 ბიტისგან. ბიტი არის მონაცემის ყველაზე პატარა ერთეული, რომელსაც კომპიუტერი იყენებს. ბიტით აღინიშნება ინფორმაციის მდგომარეობა, როგორიც „კი“ და „არა“, „1“ და „0“, „ჭეშმარიტი“ და „მცდარი“ ან სხვა. კომპიუტერის შემადგენელ ნაწილებში სწორედ ასეთი მდგომარეობით აღიწერება ინფორმაცია: მაღალი/დაბალი ძაბვა, დენი გადის/არ გადის (ჩიპებში), ელექტრული მუხტი არის/არ არის (მეხსიერებაში, ფლეშ-დისკებზე), ბილიკი დამაგნეტებული არის/არ არის (დისკებზე)... 4 ბიტს ერთად ეწოდება ნიბლი. 1 ბაიტით შეიძლება გადმოიცეს ინფორმაციის 256 მდგომარეობა. 1 ბაიტი საბაზო ლათინური ანბანის 1 სიმბოლოს ზომაა. მახვილიანი და მსგავსი ასო-ნიშნები (როგორიცაა: é, è, ã, ñ...) 2 ბაიტის ტოლია. ქართული ასო-ნიშნები კი 3 ბაიტი ზომისაა.

²IEEE (Institute of Electrical and Electronics Engineers) არის საერთაშორისო არამომგებიანი, პროფესიული ორგანიზაცია გადრმავებულ ტექნოლოგიებში. სტანდარტების შემოთავაზებასა და დამკვიდრებაში ის მსოფლიოს ერთ-ერთი ლიდერია.

³IEC (International Electrotechnical Commission) არის სტანდარტების მსოფლიოს ერთ-ერთი წამყვანი ორგანიზაცია ელექტრულ, ელექტრონულ და მასთან დაკავშირებულ ტექნოლოგიებში.

მოგვიანებით, 1998 წელს შემოიტანა, თუმცა ორგანიზაციების ნაწილმა დაამკვირდა ისინი თავისთან, ნაწილმა - ჯერ არა. შესაბამისად, 90-იანი წლების ბოლომდე KB-ში 1024 ბაიტი იგულისხმებოდა და ასეთი აღნიშვნა დღესაც შენარჩუნებულია სხვადასხვა ოპერაციულ სისტემასა თუ ბრძანებაში. როგორც უკვე ვნახეთ, find ბრძანების გამოყენებისას 1 KB-ში 1024 ბაიტი იგულისხმება.

უფრო დიდი ზომის აღსანიშნავად შემოღებულია შემდეგი დასახელების ერთეულები:

1 კიბიბაიტი (KiB)	1024 ბ, $1024=2^{10}$ ბაიტი
1 მებიბაიტი (MiB)	1024 კიბ, $1048576=2^{20}$ ბაიტი
1 გიბიბაიტი (GiB)	1024 მიბ, $1073741824=2^{30}$ ბაიტი
1 ტებიბაიტი (TiB)	1024 გიბ, $1099511627776=2^{40}$ ბაიტი
1 პეტიბაიტი (PiB)	1024 ტიბ, $1125899906842624=2^{50}$ ბაიტი
1 ეგზაბიბაიტი (EiB)	1024 პიბ, $1152921504606846976=2^{60}$ ბაიტი
1 ძეტიბაიტი (ZiB)	1024 ეიბ, $1180591620717411303424=2^{70}$ ბაიტი
1 იოტიბაიტი (YiB)	1024 ძიბ, $1208925819614629174706176=2^{80}$ ბაიტი

1 კილობაიტი (KB)	1000 ბ, $1000=10^3$ ბაიტი
1 მეგაბაიტი (MB)	1000 კბ, $1000000=10^6$ ბაიტი
1 გიგაბაიტი (GB)	1000 მბ, $1000000000=10^9$ ბაიტი
1 ტერაბაიტი (TB)	1000 გბ, $1000000000000=10^{12}$ ბაიტი
1 პეტაბაიტი (PB)	1000 ტბ, $1000000000000000=10^{15}$ ბაიტი
1 ეგზაბაიტი (EB)	1000 პბ, $1000000000000000000=10^{18}$ ბაიტი
1 ძეტაბაიტი (ZB)	1000 ებ, $1000000000000000000000=10^{21}$ ბაიტი
1 იოტაბაიტი (YB)	1000 ძბ, $1000000000000000000000000=10^{24}$ ბაიტი

არსებობს კიდევ უფრო დიდი ზომის აღმნიშვნელი ერთეულებიც: ბრონტობაიტი და ჯეოპაიტი. 1 ბრონტობაიტი (BB, Brontobyte) = 10^{27} ბაიტი და 1 ჯეოპაიტი (GPB, Geopbyte) = 10^{30} ბაიტი.

ინფორმაციის მიახლოებული ზომის უკეთ აღსაქმელად ბევრი საინტერესო კვლევაა ჩატარებული. იხილეთ ზოგიერთი მათგანი:

ინფორმაციის ობიექტი	რამდენი ბაიტია
ბინარული მდგომარეობა	1 ბიტი
1 გვერდი ლათინური ტექსტი (A4 ზომა)	≈ 3 კბ
ვეფხისტყაოსნის სრული ტექსტი	< 1 მბ
1 წუთიანი მაღალ ხარისხში ⁴ ჩაწერილი ხმა	≈ 15 მბ
1 წუთიანი მაღალ ხარისხში ⁵ ჩაწერილი ვიდეო	≈ 10 გბ

ადამიანის 1 დნმ ⁶ -ის მოლეკულა	750 მბ
ადამიანის ტვინის ტევადობა	1-100 ტბ

სანამ ზომის მიხედვით დავიწყებთ ფაილების ძებნას, მანამდე ვნახოთ, როგორ შეიძლება ფაილების ზომის ნახვა shell-ში. თუ კარგად დავაკვირდებით `ls -l` ბრძანების შედეგს, აღმოვაჩენთ, რომ მე-5 სვეტში სწორედ ფაილის ზომაა მოცემული ბაიტებში.

```
achiko@debian:~$ ls -l
...
-rw-r--r-- 1 achiko achiko 5447 Oct 5 2013 Tux.jpg
-rw-r--r-- 1 achiko achiko 53 Jun 13 17:57 versions.txt
drwxr-xr-x 2 achiko achiko 4096 Oct 6 2017 Videos
```

ფაილების ზომის გასაგებად მართებულია `du` (Disk Usage) ბრძანება გამოვიყენოთ. როგორც ამ ბრძანების სახელიდან ჩანს, ის გვანახებს თუ რა ზომა დაიკავა ამ ფაილმა დისკზე. ყურადღება მივაქციოთ იმას, რომ ფაილის ზუსტი ზომა და ის, თუ რა ზომა დაიკავა ამ ფაილმა დისკზე, შეიძლება განსხვავდებოდეს (ამ საკითხებს უფრო დეტალურად ფაილური სისტემის განხილვისას დავუბრუნდებით). მოვიყვანოთ მაგალითები.

„`du *`“ ბრძანება გვანახებს რამდენი ბლოკი დაიკავა დისკზე მიმდინარე დირექტორიის ფაილებმა. `du *` ბრძანებისთვის ბლოკის ზომად, ნაგულისხმევი მნიშვნელობით, 1024 ბაიტი მოიაზრება თუ ის ხელით არ არის განსაზღვრული (მაგალითად shell-ში `DU_BLOCK_SIZE` გარემოს ცვლადით ან თავად ბრძანებაში `--block-size` ოფციით).

```
achiko@debian:~$ du *
...
8      Tux.jpg
4      versions.txt
4      Videos
```

`versions.txt` ფაილი რომ განვიხილოთ, დავინახავთ, რომ დისკზე მას 4 ბლოკი, ანუ $4 \cdot 1024 = 4096$ ბაიტი აქვს დაკავებული, მაშინ, როდესაც თავად ფაილის ზომა გაცილებით პატარა, 53 ბაიტია.

-b ოფციით `du` ბრძანება ბაიტებში გვანახებს ფაილის რეალურ ზომებს.

```
achiko@debian:~$ du -b *
...
5447   Tux.jpg
53     versions.txt
4096   Videos
```

როგორც უკვე ვთქვით, ეს შედეგი დაემთხვა `ls -l` ის შედეგს.

`du` ბრძანებას კიდევ რამდენიმე სასარგებლო ოფცია აქვს. -b-ს ნაცვლად შეგვიძლია -k ან -m ოფციით კილობაიტებში ან მეგაბაიტებში გამოვიტანოთ ზომები, ან სულაც -h-ით

⁴Sample rate: 44,1 kHz(CD), Bit rate: 8 bit, channels: 2 (Stereo), შეუკუმშავი (მაგალითად wav, aiff ან სხვა).

⁵Resolution:1920x1080, Frame rate: 23.98, Uncompressed 1080 10-bit. FHD (Full High Definition)

⁶დნმ - დეოქსირიბონუკლეინის მუკავის ცოცხალ ორგანიზმები ინახავს გენეტიკურ კოდს

ადამიანისთვის უფრო გასაგებ ფორმატში (human readable). -s-ით თითოეული არგუმენტის ჯამური ზომა გამოგვაქვს.

ეს ბრძანება გამოიტანს პირადი დირექტორიის ჯამურ ზომას:

```
achiko@debian:~$ du -sh ~
36M    /home/achiko
```

ეს კი გამოიტანს პირადი დირექტორიაში შემავალი ყველა ფაილისა და დირექტორიის ჯამურ ზომას ბაიტებში, ასევე ერთბაშად სრულ ზომასაც (-c)

```
achiko@debian:~$ du -sbc ~/*
...
5447   /home/achiko/Tux.jpg
53     /home/achiko/versions.txt
4096   /home/achiko/Videos
60609  total
```

დავუბრუნდეთ find ბრძანებას.

როდესაც ფაილების ზუსტი ზომა არ ვიცით და გვსურს მისი მოძებნა ზომის მიხედვით, შეგვიძლია + ან - ნიშანი ვიხმაროთ, რომელიც მოცემულ ზომაზე მეტს ან ნაკლებს ნიშნავს მოძებნის პროცესში.

```
achiko@debian:~$ find ~ -type f -size +1M
```

ეს მოგვიძებნის 1 მბ-ზე მეტი ზომის ყველა ფაილს ჩვენს პირად დირექტორიაში.

```
achiko@debian:~$ find ~ -type f -size 504c
```

ეს მოგვიძებნის ზუსტად 504 ბაიტის ზომის ფაილებს, თუ, რა თქმა უნდა, ასეთი მოგვეპოვება.

```
achiko@debian:~$ find ~ -type f -size 5
```

რადგან აქ ზომის ერთეული არ არის მიწერილი, იგულისხმება b ერთეული (512 ბაიტისანი ბლოკი) და ასეთი ჩანაწერი მოგვიძებნის იმ ფაილებს, რომლებიც ზომით [4-5] ბლოკს იკავებენ. ანუ რომელთა ზომები $4 \cdot 512 = 2048$ ბაიტსა და $5 \cdot 512 = 2560$ ბაიტს შორისაა წარმოდგენილი. ამ ინტერვალში 2048 არ ჩაითვლება, ხოლო 2560 კი ჩაითვლება. n ბლოკის მითითებისას მოგვიძებნის $(n-1) \cdot 512$ -სა და $n \cdot 512$ ბაიტებს შორის არსებული ზომის ფაილებს.

```
achiko@debian:~$ find ~ -size +2G -size -4G
```

ეს კი მოგვიძებნის ფაილებს, რომელთა ზომა 2 გბ-სა და 4 გბ-ს შორისაა. ზშირად გამოიყენება, აგრეთვე, შემდეგი კრიტერიუმები:

-iname file	-name-ის მსგავსია, მხოლოდ ფაილის დასახელებაში დიდსა და პატარა ასოებს აღარ გაარჩევს.
-empty	ცარიელი ფაილი ან დირექტორია.
-user username	მიესადაგება ყველა იმ ფაილს, რომლის მფლობელი არის username.
-group groupname	მიესადაგება ყველა იმ ფაილს, რომლის მფლობელის ჯგუფი არის groupname.
-perm mode	აღნიშნავს ყველა ფაილს, რომელზე წვდომის უფლებებიც ემთხვევა მითითებული mode-ს. mode-ის ჩაწერა, როგორც chmod ბრძანების შემთხვევაში, 2 სახითაა შესაძლებელი: ციფრული ან სიმბოლური.

მაგალითები მოვიყვანოთ:

```
achiko@debian:~$ find . -perm 664
```

შეესაბამება ყველა იმ ფაილს, რომელსაც მხოლოდ 664 უფლება აქვს, ანუ მფლობელსა და მის ჯგუფს აქვთ კითხვისა და ჩაწერის უფლება და სხვა მომხმარებელს მხოლოდ წაკითხვის უფლება.

```
achiko@debian:~$ find . -perm u=rw,g=rw,o=r
```

ეს იდენტური ბრძანებაა, მხოლოდ სიმბოლური ჩანაწერით.

```
achiko@debian:~$ find . -perm -664
```

შეესაბამება ყველა იმ ფაილს, რომელსაც მინიმუმ 664 უფლება აქვს. ანუ მფლობელსა და მის ჯგუფს აქვთ მინიმუმ კითხვისა და ჩაწერის უფლება და სხვა მომხმარებელს მინიმუმ კითხვის უფლება. აქ შესაძლებელია სხვა უფლებებიც ჰქონდეს ფაილს. მაგალითად, 777 კრიტერიუმით ფაილი მოიძებნება, რადგან 777 უფლება მოიცავს 664 უფლებას.

```
achiko@debian:~$ find . -perm /664
```

ეს კი ყველა იმ ფაილს შეესაბამება, რომლის კითხვისა და ჩაწერის (6) უფლება ან მფლობელს აქვს ან მისი ჯგუფის წევრებს. სხვა მომხმარებლებს კი მხოლოდ კითხვის (4) უფლება.

```
achiko@debian:~$ find . -perm -444 -perm /222 ! -perm /111
```

მოძებნის ყველა ფაილს რომლის კითხვაც ყველას შეუძლია (-perm -444), ცვლილების გაკეთება - ერთ-ერთს (მფლობელს, ჯგუფს ან სხვა მომხმარებელს) მაინც (-perm /222) და შესრულება - არავის (! -perm /111).


```
achiko@debian:~$ find . -perm -4000
```

მოძებნის ფაილებს, რომლებსაც SUID ატრიბუტი აქვს გააქტიურებული.

ახლა ისეთი კრიტერიუმები განვიხილოთ, რომლებიც ფაილის დროის მახასიათებლებს (timestamp) იყენებს. ლინუქსში ფაილს სულ 4 დროითი მახასიათებელი აქვს:

mtime	ფაილის შიგთავსში განხორციელებული ბოლო ცვლილების თარიღი. ეს მახასიათებელი ავტომატურად განახლდება იმ დროზე, როდესაც ფაილს დავარდაქტირებთ. (მაგ. <code>vi file</code>). <code>ls -l</code> , ნაგულისხმევი მნიშვნელობით, სწორედ ფაილების mtime-ს გვანახებს მეექვსე და მეშვიდე სვეტში.
atime	ფაილის შიგთავსის წაკითხვის ბოლო თარიღი. ის ავტომატურად განახლდება მას შემდეგ, რაც ფაილს წავიკითხავთ (მაგ. <code>cat file</code>). მის სანახავად შეგვიძლია <code>ls -lu</code> გავუშვათ.
ctime	ფაილის მეტამონაცემის ბოლო ცვლილების თარიღი. ის ავტომატურად განახლდება მას შემდეგ, რაც ფაილს შეეცვლება უფლებები ან მფლობელი ან სახელი და ა.შ. (მაგ. <code>chmod 644 file</code>). მას <code>ls -lc</code> ბრძანებით ვნახავთ.
crtime	ფაილის შექმნის თარიღი. ეს მახასიათებელი მოგვიანებით დაემატა. შესაბამისად, <code>find</code> ბრძანება ძებნას ვერ უზრუნველყოფს ამ მახასიათებლის მიხედვით. ვერც <code>ls</code> ბრძანებას გამოაქვს ეს მონაცემი. ამ საკითხს მოგვიანებით დავუბრუნდებით.

როგორც გახსოვთ, წინა თავებში `touch` ბრძანებას ცარიელი ფაილების შესაქმნელად ვიყენებდით, თუმცა არ აღგვინიშნავს მისი მთავარი დანიშნულება. `touch` გამოიყენება სწორედ ფაილის დროითი მახასიათებლების შესაცვლელად.

```
achiko@debian:~$ ls -l test.txt
-rw-r--r-- 1 achiko achiko 14 Jun 20 15:26 test.txt
achiko@debian:~$ touch -m -t 201801020304 test.txt
```

ახე `test.txt`-ის mtime-ს დავაყენებთ 2018 წლის 02 იანვრის 03:04-ზე. შევამოწმოთ:

```
achiko@debian:~$ ls -l test.txt
-rw-r--r-- 1 achiko achiko 14 Jan  2 03:04 test.txt
```

„თურმე“ ეს ფაილი ბოლოს ამა წლის 02 იანვრის 03:04-ზე შეგვიცვლია.

```
achiko@debian:~$ touch -a -t 201801020305 test.txt
```

ახე კი `test.txt`-ის atime-ს დავაყენებთ 2018 წლის 02 იანვრის 03:05-ზე. აქაც გადავამოწმოთ:

```
achiko@debian:~$ ls -lu test.txt
-rw-r--r-- 1 achiko achiko 14 Jan  2 03:05 test.txt
```

„თურმე“ ამ ფაილის შიგთავსი ბოლოს ამა წლის 02 იანვარს 03:05-ზე ვნახეთ.

დავუბრუნდეთ კვლავ `find` ბრძანების კრიტერიუმებს.

<code>-mtime n</code>	მიესადაგება იმ ფაილებს, რომლებიც $n*24$ საათის წინ შეიცვალა. შესაძლებელია, აგრეთვე, <code>-n</code> ან <code>+n</code> გამოვიყენოთ <code>n</code> -ზე ნაკლების ან მეტის აღსანიშნავად.
<code>-atime n</code>	მიესადაგება იმ ფაილებს, რომლების $n*24$ საათის წინ წაიკითხეს.
<code>-ctime n</code>	მიესადაგება იმ ფაილებს, რომლებსაც სტატუსი $n*24$ საათის წინ შეუცვალეს.
<code>-mmin n</code> <code>-amin n</code> <code>-cmin n</code>	მიესადაგება ყველა ფაილს, რომელთა შესაბამისი მახასიათებელიც <code>n</code> წუთის წინ შეიცვალა. აქაც შესაძლებელია აგრეთვე <code>-n</code> ან <code>+n</code> გამოვიყენოთ <code>n</code> -ზე ნაკლების ან მეტის აღსანიშნავად.
<code>-newer file</code>	მიესადაგება ყველა ფაილს, რომლის <code>mtime</code> -იც უფრო ახალია ვიდრე <code>file</code> -ის <code>mtime</code> .
<code>-anewer file</code> <code>-cnewer file</code>	მიესადაგება ყველა ფაილს, რომლის <code>atime/ctime</code> უფრო ახალია ვიდრე <code>file</code> -ის <code>mtime</code> .

```
achiko@debian:~$ find . -type f -mmin -60
```

მოძებნის ყველა ფაილს, რომელიც შეიცვალა ბოლო 1 საათის (60 წუთის) განმავლობაში.

```
achiko@debian:~$ find . -type f -mtime +5 -mtime -8
```

მოძებნის ყველა ფაილს, რომლებიც შეიცვალა ბოლო 5-8 დღის განმავლობაში.

`find` ბრძანებას კიდევ ბევრი სხვა კრიტერიუმი აქვს. მათ სანახავად `find`-ის სახელმძღვანელო გვერდი დაგვეხმარება. `man find`

როგორც ზემოთ აღვნიშნეთ, `find` ბრძანებაში რამდენიმე კრიტერიუმის გამოყენება ნიშნავს იმას, რომ მოძებნილი ფაილები ყველა ამ კრიტერიუმს ერთდროულად უნდა აკმაყოფილებდნენ. თუ გვსურს, რომ მოვძებნოთ ფაილები, რომლებისთვისაც არა ყველა, არამედ ერთ-ერთი კრიტერიუმი დაკმაყოფილდება, მაშინ ლოგიკური ოპერატორები დაგვჭირდება.

-`or` ოპერატორი `find` ბრძანებაში **ლოგიკურ ან-ს** აღნიშნავს და ორ კრიტერიუმს შორის იწერება. ამ დროს მოიძებნება ყველა ის ფაილი, რომელისთვისაც ან ერთი კრიტერიუმი ან სხვა კრიტერიუმი ან მეორე. მოკლედ აღინიშნება ასეც `-o`.

-`and` **ლოგიკური დას** ნიშნავს, და როგორც უკვე შევნიშნეთ, ნაგულისხმევი მნიშვნელობითაც, სწორედ ისაა გააქტიურებული. ანუ, როდესაც კრიტერიუმებს შორის ოპერატორი საერთოდ არაა მითითებული **ლოგიკური და** იგულისხმება. **ლოგიკური და** მოკლედ ასეც აღინიშნება `-a`.

-`not` ლოგიკურ უარყოფას აღნიშნავს და კრიტერიუმის წინ იწერება. მოკლედ ასე აღინიშნება `!`.

() - ფრჩხილებით რთული გამოსახულების დაჯგუფება შეგვიძლია. როგორც წესი, `find` კრიტერიუმებს მარცხნიდან მარჯვნივ კითხულობს. ასეთი რიგითობის შესაცვლელად

სწორედ გამოსახულების ფრჩხილებში ჩასმა გამოგვადგება. მათ, გამოყენებისას წინ \ უნდა დავუწეროთ, ასე: \ (\)

```
$ find ~ -type f \( -name "a*" -or -name "b*" \)
```

ეს ბრძანება მოძებნის ჩვეულებრივი ტიპის ყველა ფაილს, რომლის სახელიც ან a-თი იწყება ან b-თი.

ფრჩხილები რომ არ გამოვიყენოთ:

```
$ find ~ -type f -name "a*" -or -name "b"
```

ეს ბრძანება მოგვიძებნის a-თი დაწყებულ ჩვეულებრივი ტიპის ყველა ფაილს ან b-თი დაწყებულ ყველა ფაილს (აქ b-ზე დაწყებული დირექტორიაც შეიძლება მოიძებნოს)

```
$ find ~ \( -user 1000 \( -name "*box" -or -name "pu*" \) \) ! \( -path  
".mozilla/*" -o -path ".cache/*" \)
```

ეს ბრძანება მოძებნის ყველა იმ ფაილს, რომელიც ეკუთვნის მომხმარებელს 1000 UID ნომრით და რომლის სახელიც მთავრდება box-ზე ან იწყება pu-ზე, და ამაღვდროულად, მის სრულ დასახელებაში (იგულისხმება მისი PATH) არ შევა სიტყვა .mozilla/ ან .cache/ (ამით გვსურს, გამოვრიცხოთ .mozilla/ და .cache/ დირექტორიებში ან მათ ქვედირექტორიებში არსებული ფაილები).

find ბრძანებას, იმის გარდა, რომ სხვადასხვა კრიტერიუმებით ეძებს ფაილებს, შეუძლია მოძებნილ ფაილებზე სხვადასხვა ბრძანებით იმოქმედოს.

არსებობს უკვე განსაზღვრული ქმედებები, თუმცა ზოგადი, ტრადიციული გზა ამისთვის ასეთია, find ბრძანებას ბოლოში შემდეგი ჩანაწერი უნდა გადავცეთ და ისე გავუშვათ:

```
achiko@debian:~$ find ... -exec cmd '{}' \;
```

სადაც, cmd-ის ნაცვლად ის ბრძანება უნდა მივუთითოთ, რითაც გვსურს ვიმოქმედოთ მოძებნილ ფაილზე. {} ფიგურული ფრჩხილების ქვეშ მოძებნილი ფაილი იგულისხმება. ; წერტილ-მძიმე ბრძანების დასასრულს ნიშნავს და რადგან ამ სიმბოლოს shell-ში სხვა დატვირთვა აქვს, შესაბამისად, უნდა დავუკარგოთ მნიშვნელობა ასე ';' ან ასე \;

```
achiko@debian:~$ find ... -exec cmd '{}' +
```

+ ჩანაწერით მოძებნილი ფაილები ერთმანეთის მიყოლებით ჩაიწერება ერთ ხაზზე და ასე გადავცემა შესასრულებლად cmd-ს.

მათ შორის განსხვავება ნათლად ჩანს შემდეგი მაგალითიდან:

```
$ find . -name "t*.txt" -exec echo "Result is:" '{}' \;  
Result is: ./test2.txt  
Result is: ./test.txt
```

```
$ find . -name "t*.txt" -exec echo "Result is:" '{}' +  
Result is: ./test2.txt ./test.txt
```

```
$ find . -type d -perm 777 -exec chmod 755 '{}' \;
```

ამ ბრძანებას პრაქტიკული დანიშნულება აქვს. ის მოძებნის ისეთ დირექტორიებს, რომლებსაც 777 უფლება აქვს (ყველას ყველა უფლება აქვს) და შეცვლის მას 755-ით (ამით, ჩვენს გარდა, ყველას წავართმევთ ამ დირექტორიებში ცვლილებების გაკეთების უფლება).

```
$ find /tmp -type f -empty -exec rm -f {} \;
```

წაშლის ყველა ცარიელ ფაილს /tmp-ში.

```
$ find ~ -type f -name "*.txt" -exec mv {} txt_dir/ \;
```

ეს ბრძანება გადაიტანს ყველა txt-ზე დამთავრებულ ფაილს პირადი დირექტორიიდან txt_dir/ დირექტორიაში (რა თქმა უნდა, ის უკვე უნდა არსებობდეს)

```
$ find ~ \( -type f -not -perm 0600 -exec chmod 0600 {} \; \) -or \( -type  
d -not -perm 0700 -exec chmod 0700 {} \;
```

ეს ბრძანება პირად დირექტორიაში არსებული ყველა ფაილისა და დირექტორიის უფლებებს შეცვლის და მხოლოდ მფლობელს მისცემს მათზე წვდომის უფლებებს, სხვა დანარჩენს წაართმევს რაც გააჩნიათ.

find ბრძანების უფრო მეტი შესაძლებლობების სანახავად გაუშვიტ man find.

12.3 split

მოდით, ახლა ერთი საინტერესო ახალი ბრძანება - split შევისწავოთ. split ბრძანებით შეგვიძლია დიდი ფაილი ნაწილებად დავყოთ. ნაწილებად დავყოფა შეიძლება ზომით ან საზების რაოდენობით განვახორციელოთ.

```
achiko@debian:~$ split -b200 lorem.txt  
achiko@debian:~$ ls -l  
-rw-r--r-- 1 achiko achiko 504 Jun 11 16:33 lorem.txt  
...  
-rw-r--r-- 1 achiko achiko 200 Jun 22 18:14 xaa  
-rw-r--r-- 1 achiko achiko 200 Jun 22 18:14 xab  
-rw-r--r-- 1 achiko achiko 104 Jun 22 18:14 xac
```

ამ ბრძანებით lorem.txt დაიყოფა 200 ბაიტის ნაწილებად და დანაყოფებს ავტომატურად დაერქმევა xaa, xab, xac ... მართლაც, თითოეული დანაყოფი 200 ბაიტის ზომისაა, ბოლო ფაილი კი იმ ზომის იქნება, რაც ნაშთად დარჩა.

თუ გვსურს `x`-ის ნაცვლად ჩვენით მივუთითოთ სუფიქსი, მაშინ ის ბრძანებას ბოლოში უნდა მივუწეროთ, ასე:

```
achiko@debian:~$ split -b200 lorem.txt Danakopi_  
achiko@debian:~$ ls -l Danakopi*  
-rw-r--r-- 1 achiko achiko 200 Jun 22 18:31 Danakopi_aa  
-rw-r--r-- 1 achiko achiko 200 Jun 22 18:31 Danakopi_ab  
-rw-r--r-- 1 achiko achiko 104 Jun 22 18:31 Danakopi_ac
```

შესაძლებელია ანბანური დასახელება ციფრულით შევცვალოთ `-d` ოფციის გამოყენებით.

```
achiko@debian:~$ split -b200 -d lorem.txt Danakopi_  
achiko@debian:~$ ls -l Danakopi*  
-rw-r--r-- 1 achiko achiko 200 Jun 22 18:33 Danakopi_00  
-rw-r--r-- 1 achiko achiko 200 Jun 22 18:33 Danakopi_01  
-rw-r--r-- 1 achiko achiko 104 Jun 22 18:33 Danakopi_02
```

`split`-ში დიდი ზომის მისათითებლად შესაძლებელია შემდეგი ზომის ერთეულების გამოყენება (ამ ბრძანებაში გათვალისწინებულია 2-ისა და 10-ის ხარისხებიანი ზომის ერთეულები): K, M, G, T, P, E, Z და Y (2-ის ხარისხებისთვის), ხოლო KB, MB, GB, TB, PB, EB, ZB და YB (10-ის ხარისხებისთვის).

```
achiko@debian:~$ split -b1M file  
achiko@debian:~$ ls -l Danakopi*
```

ასე, 1 მეგიბაიტიანი ზომის დანაყოფებს მივიღებთ.

`-l` ოფცია ხაზების რაოდენობებით დაყოფს ფაილს და არა მითითებული ზომით. ასე:

```
achiko@debian:~$ split -l100 file
```

თითოეულ დანაყოფში შევა ორიგინალი ფაილის ყოველი 100 ხაზი.

`-n` ოფციით კი თანაბარი ზომის დანაყოფებს მივიღებთ. ცხადია, ბოლო დანაყოფში შეიძლება ნაშთი დავტყუდოთ.

```
achiko@debian:~$ split -n5 file
```

ამგვარად, `file`-ს 5 ტოლ ნაწილად დავყოფთ.

12.4 echo

`echo` ბრძანება ერთ-ერთი ხშირად გამოყენებადი ბრძანებაა `bash`-ში. როგორც უკვე ვნახეთ, მას გამოაქვს არგუმენტად გადაცემული ტექსტი (პირდაპირი გაგებით ექოსავით იქცევა. რასაც გადასცემ, იმას ბეჭდავს ეკრანზე) და ცვლადის მნიშვნელობა. ისიც ვთქვით,

რომ ტექსტები შეიძლება შეიცავდეს სპეციალურ სიმბოლოებს (ე.წ. backslash-escaped characters. ასე იმიტომ ჰქვია, რომ მის წინ „\“ სიმბოლოა წარმოდგენილი). ამ სიმბოლოების დატვირთვა არა რომელიმე ასო-ნიშნის ეკრანზე გამოსახვა, არამედ იმ ფუნქციის შესრულებაა, რაც მისი კოდის ქვეშ იგულისხმება. ამისთვის echo ბრძანებას -e ოფცია უნდა გადავცეთ.

backslash-escaped characters	
\a	სისტემური ხმოვანი სიგნალის გამოტანა (ე.წ. BEL ⁷).
\b	ერთი სიმბოლოს წაშლა მარჯვნიდან მარცხნივ (იგივეა, რაც „backspace“ კლავიშზე დაჭერა).
\c	ამ სიმბოლოს შემდეგ არსებული ტექსტის წაშლა.
\f	ახალ გვერდზე გადასვლა და ამ სიმბოლოს შემდეგ არსებული ტექსტით ხაზის დაწყება (ე.წ. form feed ან page break).
\n	ამ სიმბოლოს შემდეგ არსებული ტექსტის ახალ ხაზზე გადატანა (new line).
\t	ჰორიზონტალური ტაბულაცია (horizontal tab).
\v	ვერტიკალური ტაბულაცია (vertical tab).
\r	ამ სიმბოლოს შემდეგ არსებული ტექსტის გადაწერა იმავე ხაზის დასაწყისიდან (carriage return).
\\	თავად „\“ სიმბოლო.
\Onnn	8 ბიტიანი ასო-ნიშანი, რომლის მნიშვნელობა რვაობით წარმოდგენაში არის nnn (3 რვაობითი სიმბოლო).
\xHH	8 ბიტიანი ასო-ნიშანი, რომლის მნიშვნელობა თექვსმეტობით წარმოდგენაში არის HH (2 თექვსმეტობითი სიმბოლო).
\uHHHH	უნიკოდის (ISO/IEC 10646) ასო-ნიშანი, რომლის მნიშვნელობა თექვსმეტობით წამოდგენაში არის HHHH (4 თექვსმეტობითი სიმბოლო).

მოდით, მაგალითებით ვნახოთ ამ სპეციალური სიმბოლოების მნიშვნელობა. სადემონსტრაციოდ ავიღოთ ერთ-ერთი აფორიზმი „ვეფხისტყაოსნიდან“.

\$ echo "ბოროტსა სძლია კეთილმან, არსება მისი გრძელია!"
 ბოროტსა სძლია კეთილმან, არსება მისი გრძელია!

ყურადღება მიაქციეთ იმას, რომ backslash-escaped სიმბოლოების გამოყენებისას, ისინი ბრჭყალებში (ან აპოსტროფებს შორის) აუცილებლად უნდა მოვაქციოთ და, ამავედროულად, echo ბრძანებას -e უნდა გადავცეთ. მათ გარეშე მხოლოდ ამ კოდების აღმნიშვნელი ასო-ნიშნები გამოისახება ეკრანზე.

⁷bell code საკონტროლო კოდი თავდაპირველად გამოიყენებოდა პატარა ელექტრომექანიკურ ზარზე სიგნალის გადასაცემად, ბეჭდვის დროს ხაზის ბოლოს მისანიშნებლად და ახალი შეტყობინების მოსვლისას.

[illegible]

მოდით, ახლა გნახოთ თუ როგორ სრულდება echo ბრძანებით თვლის სხვადასხვა წარმოდგენაში მოცემული ASCII ასო-ნიშნების კოდები. ავიღოთ ascii-ს სახელმძღვანელო გვერდიდან ასო-ნიშნების რვაობითი და თექვსმეტიობითი კოდები და ვცადოთ:

```

achiko@debian:~$ man ascii
ASCII(7)                Linux Programmer's Manual                ASCII(7)

NAME
    ascii - ASCII character set encoded in octal, decimal,
    and hexadecimal

DESCRIPTION
    ASCII is the American Standard Code for Information
    Interchange. It is a 7-bit code. Many 8-bit codes
    (e.g., ISO 8859-1) contain ASCII as their lower half.
    The international counterpart of ASCII is known as ISO
    646-IRV.

    The following table contains the 128 ASCII characters.

    C program '\X' escapes are noted.

Oct  Dec  Hex  Char
-----
000  0    00   NUL '\0' (null character)  100  64   40   @
001  1    01   SOH (start of heading)  101  65   41   A
002  2    02   STX (start of text)  102  66   42   B
003  3    03   ETX (end of text)  103  67   43   C
004  4    04   EOT (end of transmission)  104  68   44   D

```

005	5	05	ENQ (enquiry)	105	69	45	E
006	6	06	ACK (acknowledge)	106	70	46	F
007	7	07	BEL '\a' (bell)	107	71	47	G
010	8	08	BS '\b' (backspace)	110	72	48	H
011	9	09	HT '\t' (horizontal tab)	111	73	49	I
012	10	0A	LF '\n' (new line)	112	74	4A	J
...							

ამ ცხრილიდან ჩანს, რომ „A“ ასო-ნიშნის რვაობით კოდია „101“, ხოლო თექვსმეტობითი „41“. ვაჩვენოთ, თუ როგორ გამოიტანს ამ კოდს `echo` ბრძანება.

```
achiko@debian:~$ echo -e "\0101"
A
achiko@debian:~$ echo -e "\x41"
A
achiko@debian:~$ echo -e "\0115\0111\0123\0110\0113\0101"
MISHKA
```

ყველაფერი რიგზეა. ახლა, მოდით, უნიკოდის ცხრილიდან გამოვიტანოთ ქართული ანბანური დამწერლობის ასო-ნიშნები. მასში მოცემულია როგორც მხედრული (თანამედროვე ანბანი), ასევე ნუსხური (ხუცური, ნუსხა-ხუცური, კუთხოვანი) და ასომთავრული (ხუცური ასომთავრული, მრგვლოვანი) დამწერლობის სახეები. უნიკოდის ცხრილი შეგიძლიათ ნახოთ <https://unicode-table.com> ვებ-გვერდზე ან <https://unicode.org/charts/>-ზე ან სხვაგან.

```
achiko@debian:~$ echo -e "\u10A0 \u10A1 \u10A2 \u10A3 \u10A4"
Ⴀ Ⴁ Ⴂ Ⴃ Ⴄ
achiko@debian:~$ echo -e "\u10D0 \u10D1 \u10D2 \u10D3 \u10D4"
Ⴅ Ⴆ Ⴇ Ⴈ Ⴉ
achiko@debian:~$ echo -e "\u2D00 \u2D01 \u2D02 \u2D03 \u2D04"
ᲀ ᲁ ᲂ ᲃ ᲄ
```

`echo` ბრძანებაში, `-e` ოფციისგან განსხვავებით, `-E` ოფცია, პირიქით, ბლოკავს სპეციალური სიმბოლოების ფუნქციების შესრულებას. `-n` ოფცია ბოლო ახალ ხაზზე გადასვლის სიმბოლოს შლის.

```
achiko@debian:~$ echo -E "\u10AB \u10A8 \u10B8 \u10A9 \u10A0"
ႧႨႩႪႫ
achiko@debian:~$ echo -e "\u10AB \u10A8 \u10B8 \u10A9 \u10A0"
ႧႨႩႪႫ
achiko@debian:~$ echo -ne "\u10AB \u10A8 \u10B8 \u10A9 \u10A0"
ႧႨႩႪႫachiko@debian::~$
```

`echo`-ს გარდა, გვაქვს იმავე ფუნქციის სხვა ბრძანებაც - `printf`. ის მოგვიანებით დაემატა ლინუქს სისტემებს და, `echo`-სგან განსხვავებით, მას შეუძლია ფორმატირებული გამოსახულება დაბეჭდოს სტანდარტულ გამოსასვლელზე. ის დაპროგრამების C ენიდან გამომდინარეობს და, შესაბამისად, მისი ეს შესაძლებლობა უფრო მეტად დაპროგრამების

დროსაა გამოსადეგი. ჩვენ მას სკრიფტების დაწერისას გამოვიყენებთ და დეტალურად მაშინ დავუბრუნდებით.

ზოგადი სინტაქსის თვალსაზრისით, `printf` ბრძანებას ეკრანზე გამოაქვს არგუმენტად გადაცემული ტექსტი და ისიც იგებს სპეციალურ სიმბოლოებს.

```
$ printf "ბოროტსა სძლია კეთილმან, არსება მისი გრძელია!"  
ბოროტსა სძლია კეთილმან, არსება მისი გრძელია!$  
$ printf "ბოროტსა \013სძლია \011კეთილმან, \012არსება მისი გრძელია!\n"  
ბოროტსა  
    სძლია კეთილმან,  
არსება მისი გრძელია!
```

12.5 xargs

როდესაც ერთი ბრძანების მეორესთან გადაბმა გვსურს, `xargs` ბრძანება ძალიან სასარგებლო შეიძლება აღმოჩნდეს. ზოგადად, `xargs` კითხულობს მონაცემებს სტანდარტული შესასვლელიდან, ამუშავებს და მათზე ასრულებს მოცემულ ბრძანებას, ნაგულისხმევი მნიშვნელობით `/bin/echo`-ს. მოკლედ, შეგვიძლია ვთქვათ, რომ `xargs` თავად ქმნის ბრძანებათა ზაზს.

```
achiko@debian:~$ xargs  
Gamarjoba,  
Georgia  
Gamarjoba, Georgia
```

ასე, `xargs`-ს კლავიატურიდან გადავეცით მონაცემები (მონაცემების შეტანა `Ctrl^d` კლავიშების კომბინაციით დავასრულეთ), რომლებიც დამუშავდა (ერთ ზაზზე განთავსდა) და რადგანაც სხვა ბრძანება მითითებული არ იყო, `/bin/echo`-თი ეკრანზე გამოვიდა ერთხაზიანი ჩანაწერი.

მსგავს შედეგს მივიღებთ შემდეგ მაგალითშიც:

```
achiko@debian:~$ ls | xargs  
cities.txt cmd.sed cmd.txt countries.txt dates.txt Desktop distros.txt  
Documents Downloads GPL-3 hidden.txt LICENSE.txt lorem5.txt lorem.txt  
Music names.txt newfile.txt new.txt Pictures Public sizes.txt Templates  
test2.txt test.txt tmp Tux.jpg TXT versions.txt Videos
```

ამ შემთხვევაში `xargs`-ს სტანდარტულ შესასვლელზე გადავეცით `ls` ბრძანების სტანდარტულ გამოსასვლელზე მიღებული მონაცემები, რომლებიც ერთ ზაზში გამოისახა ეკრანზე.

`xargs`-ით მარტივად შეგვიძლია სისტემაში არსებული მომხმარებლის სია კომპაქტურად, ერთ ზაზში გამოვიტანოთ:

```
achiko@debian:~$ cat /etc/passwd | cut -d: -f1 | sort | xargs  
achiko _apt avahi backup bin colord daemon Debian-exim Debian-gdm dnsmasq
```

```
games geoclue gnats irc list lp mail man messagebus news nobody proxy pulse
root rtkit saned speech-dispatcher sshd sync sys systemd-bus-proxy systemd-
network systemd-resolve systemd-timesync usbmux uucp www-data
```

xargs ბრძანების ქცევა **-n** ოფციით ნათლად ჩანს ამ მაგალითიდან:

```
achiko@debian:~$ echo ერთი ორი სამი ოთხი ხუთი | xargs
ერთი ორი სამი ოთხი ხუთი
achiko@debian:~$ echo ერთი ორი სამი ოთხი ხუთი | xargs -n 2
ერთი ორი
სამი ოთხი
ხუთი
```

-n ოფციით ბრძანებათა ხაზს მაქსიმუმ ორი პარამეტრი გადავცემთ არგუმენტად. შესაბამისად, ჯამში 3 ბრძანებათა ხაზი შეიქმნა.

თუ **xargs**-ს სხვა ბრძანებას მივუწერთ, მის მიერ დამუშავებულ მონაცემებს ამ ბრძანებას გადავცემთ არგუმენტად. მოდით, კვლავ მაგალითებს მივმართოთ.

```
achiko@debian:~$ ls *.txt | xargs wc
  5   10   49 cities.txt
  5   11   51 countries.txt
...
 12   12   53 versions.txt
372 2149 14874 total
```

ამ ბრძანებით მიმდინარე დირექტორიაში ვეძებთ **txt** გაფართოების მქონე ყველა ფაილს და მათი სახელებით შემდგარ ხაზს არგუმენტად გადავცემთ **wc** ბრძანებას. შესაბამისად, ის სათითაოდ გამოიტანს ყველა ფაილში შემავალი ხაზების, სიტყვებისა და ასო-ნიშნების რაოდენობას.

xargs ბრძანების გარეშე სულ სხვა შედეგს მივიღებთ:

```
achiko@debian:~$ ls *.txt | wc
 527   4128  135489
```

ამ ჩანაწერით **ls *.txt**-ის შედეგი, როგორც ასო-ნიშნების ნაკადი, გადავაცემთ **wc** ბრძანებას სტანდარტულ შესასვლელში და მანაც, დათვალა თუ რამდენი ხაზი, სიტყვა და ბაიტი შედის ამ ნაკადში. ანუ, თუ **cmd | cmd2** ჩანაწერით **cmd1** ბრძანების სტანდარტულ გამოსასვლელზე მიღებული ნაკადი გადაეცემა **cmd2** ბრძანებას სტანდარტულ შესასვლელზე, **cmd | xargs cmd2** ჩანაწერით **cmd1** ბრძანების სტანდარტულ გამოსასვლელზე მიღებული ნაკადი **cmd2** ბრძანებას მიეწოდება როგორც ერთ ხაზზე განთავსებული არგუმენტ(ებ)ი.

ახლა, კიდევ ერთი მაგალითი მოვიყვანოთ. ვთქვათ, ერთი ფაილის შიგთავსში (**a.txt**) ჩაწერილია სხვა ფაილის სრული გზა (**/home/achiko/tmp/b.txt**) და გვსურს ამ უკანასკნელის შიგთავსის გამოტანა ეკრანზე.

```
achiko@debian:~$ cat a.txt | cat
/home/achiko/tmp/b.txt
achiko@debian:~$ cat a.txt | xargs cat
/home/achiko/tmp/b.txt FILE CONTENT
```

პირველ შემთხვევაში, `cat` ბრძანებას სტანდარტულ შესასვლელზე გადავცით ნაკადი `/home/achiko/tmp/b.txt` და სტანდარტულ გამოსასვლელზე მისი ასლი დაგვიბრუნა. მეორე შემთხვევაში, კი `cat`-ს არგუმენტად გადავცით `/home/achiko/tmp/b.txt` და რადგან ეს არგუმენტი ფაილის დასახელებაა, შედეგად მისი შიგთავსი მივიღეთ ეკრანზე.

ასეთი ამოცანის დროს შესაძლებელია, აგრეთვე, სულ სხვა მიდგომა - ბრძანების შეცვლა შედეგით (command substitution) - გამოვიყენოთ. ამ დროს თავად ბრძანება ჩანაცვლდება თავისივე სტანდარტული გამოსასვლელით. ამისთვის ბრძანება უნდა მოვათავსოთ `$(...)` ან ``...`` კონსტრუქციაში.

```
achiko@debian:~$ cat $(cat a.txt)
/home/achiko/tmp/b.txt FILE CONTENT
achiko@debian:~$ cat `cat a.txt`
/home/achiko/tmp/b.txt FILE CONTENT
```

შესაძლებელია, აგრეთვე, ბრძანების შეცვლა ბრძანების შეცვლაში რამდენიმეჯერმა გამოვიყენოთ (nested command substitution). ამ დროს, უმჯობესია პირველი ფორმა გამოვიყენოთ, რადგან მეორე ფორმაში შიდა მკვეთრ მახვილებს წინ ბექსლექშები უნდა მივუწეროთ.

```
achiko@debian:~$ cat $(cat $(cat a.txt))
achiko@debian:~$ cat `cat \`cat a.txt \``
```

მოდით, დავუბრუნდეთ `xargs` ბრძანებას და `find`-თან ერთად კომბინაციაში განვიხილოთ. მაგალითად, მოვძებნოთ `test` დირექტორიაში მყოფი ის დირექტორიები, რომელთა სახელიც „a“ ასო-ნიშნით იწყება და წავშალოთ ისინი. ასეთი ამოცანა მარტივად კეთდება მხოლოდ `find` ბრძანებითაც (`-exec`-ის გამოყენებით), თუმცა ამ ჯერზე `xargs` ბრძანება დავიხმაროთ.

ჯერ შევქმნათ ასეთი დირექტორიები:

```
achiko@debian:~$ mkdir -p test/{a1,a2,"a3 a4"}
achiko@debian:~$ ls -l test
total 12
drwxr-xr-x 2 achiko achiko 4096 Aug 13 12:37 a1
drwxr-xr-x 2 achiko achiko 4096 Aug 13 12:37 a2
drwxr-xr-x 2 achiko achiko 4096 Aug 13 12:37 a3 a4
```

რადგან `test` დირექტორია არ არსებობდა და გვსურდა, ისიც შექმნილიყო და, ამავედროულად, მასში სხვა ქვე-დირექტორიებიც, ამიტომ გამოვიყენეთ `-p`, `--parents` ოფცია.

ახლა კი წავშალოთ:

```
achiko@debian:~$ find test/ -type d -name "a*" | xargs rm -rf
```

ერთი შეხედვით, ეს ბრძანება საკმარისია, თუმცა თუ შევამოწმებთ, ვნახავთ, რომ test დირექტორია ცარიელი არ არის. არ წაიშალა მხოლოდ ის დირექტორია, რომლის დასახელებაშიც გამოტოვების სიმბოლოა გამოყენებული.

```
achiko@debian:~$ ls test
a3 a4
```

ამ ხარვეზის გამოსასწორებლად find ბრძანებაში -print0 ოფცია უნდა გამოვიყენოთ. ის უზრუნველყოფს, რომ სტანდარტულ გამოსასვლელზე გამოსული ფაილების სრული დასახელებები ნულოვანი ასო-ნიშნით იყვნენ ერთმანეთისგან გამოყოფილნი და არა ახალ ხაზზე გადატანის სიმბოლოთი, როგორც ეს, ნაგულისხმევი მნიშვნელობით, -print ოფციის დროს ხდება. ეს საშუალებას აძლევს სხვა ბრძანებებს სწორად აღიქვან ის ფაილები, რომელთა დასახელებებშიც უხილავი ასო-ნიშნები (მაგალითად, ახალ ხაზზე გადასხვლა ან სხვა) შედის. xargs ბრძანებამაც ისინი კორექტულად რომ აღიქვას, -0 ოფცია უნდა მივუთითოთ. ასე:

```
achiko@debian:~$ find test/ -type d -name "a*" -print0 | xargs -0 rm -rf
achiko@debian:~$ ls test
achiko@debian:~$
```

ახლა, შევეცადოთ ფაილ(ებ)ი ვაკოპიროთ ერთდროულად რამდენიმე დირექტორიაში.

```
achiko@debian:~$ cd test/ && mkdir d{1,2}; touch f{1,2}
achiko@debian:~$ ls -F
d1/ d2/ f1 f2
achiko@debian:~$ echo d1 d2 | xargs -n 1 cp -v f1 f2
'f1' -> 'd1/f1'
'f2' -> 'd1/f2'
'f1' -> 'd2/f1'
'f2' -> 'd2/f2'
achiko@debian:~$ ls d1
f1 f2
achiko@debian:~$ ls d2
f1 f2
```

ls ბრძანების -F, --classify უზრუნველყოფს უკეთეს ხილვადობას, სხვადასხვა ტიპის ფაილების ერთმანეთისგან გამორჩევას. მაგალითად დირექტორიის ტიპის ფაილს ბოლოში მიუწერს „/“-ს, მალსახმობს „@“-ს და ა.შ. xargs ბრძანების -n, --max-args= ოფცია კი თითო ბრძანებათა ხაზზე არგუმენტის რაოდენობას აღნიშნავს.

როდესაც ბრძანებას სხვადასხვა მონაცემს ვაწვდით, სინამდვილეში არ ჩანს, ეს ბრძანება მას სტანდარტულ შესასვლელზე მიიღებს თუ არგუმენტად. xargs ბრძანების გამოყენებით, ცალსახად ვსაზღვრავთ, რომ მონაცემებს არგუმენტად ვაწვდით ბრძანებას. მოვიყვანოთ მაგალითი:

```

achiko@debian:~$ ls *.txt | grep LICENSE
LICENSE.txt
achiko@debian:~$ ls *.txt | xargs grep LICENSE
file1.txt:PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2
file1.txt:1. This LICENSE AGREEMENT is between the Python Software
file2.txt:PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2
file2.txt:1. This LICENSE AGREEMENT is between the Python Software
LICENSE.txt:PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2
LICENSE.txt:1. This LICENSE AGREEMENT is between the Python Software
...

```

პირველი ბრძანებით ვეძებთ ისეთ ტექსტურ ფაილებს, რომელთა დასახელებაშიც შედის „LICENSE“ გამოსახულება, ხოლო მეორე ბრძანებით თვითონ ტექსტური ფაილების შიგთავსში ვეძებთ იმ ხაზებს, სადაც „LICENSE“ გამოსახულება შედის.

12.6 expand, unexpand

ლინუქსში ფაილების შიგთავსზე მუშაობისას, მათი დამუშავებისას შეიძლება ხშირად შეგვხვდეს სიტუაცია, როდესაც ტაბულაციის ნაცვლად ცარიელ სიმბოლოს ვისურვებდით გვექონოდა და პირიქით. ტექსტების დამუშავების ბევრი ბრძანებისთვის მნიშვნელოვანია სვეტებს შორის გამყოფი ასო-ნიშნის ცოდნა. სწორედ ამიტომ, ლინუქსში არსებობს ბრძანება **expand**, რომელსაც ტექსტში გამოყენებული ტაბულაციების ცარიელ სიმბოლოდ გარდაქმნა შეუძლია.

მოდით, მაგალითები მოვიშველიოთ. შევქმნათ ფაილი, სადაც გამოვიყენებთ ტაბულაციებს. ხაზში სვეტებს შორის 1, 2 და 3 ცალი ტაბულაცია გვაქვს ჩასმული:

```

achiko@debian:~$ cat test1.txt
one      two           three           four

```

გადავაქციოთ ტაბულაციები გამოტოვების სიმბოლოდ:

```

achiko@debian:~$ expand test1.txt
one      two           three           four

```

ერთი შეხედვით, თითქოს შედეგი იგივეა, თუმცა ეს ასე არ არის. **expand** ბრძანება, ნაგულისხმევი მნიშვნელობით, ტაბულაციაში გულისხმობს 8 ცარიელ სიმბოლოს. ამიტომაც ვიღებთ ვიზუალურად მსგავს შედეგს ტაბულაციების ცარიელ სიმბოლოებად გარდაქმნისას. უკეთ რომ დავინახოთ ეს ყველაფერი, მოდით, 1 ტაბულაცია 1 (და არა 8) დაშორების სიმბოლოდ გადავაქციოთ. ამისთვის -t ოფცია უნდა გამოვიყენოთ, ასე:

```

achiko@debian:~$ expand -t1 test1.txt
one two  three  four

```

თვალსაჩინოებისთვის **cat** ბრძანებაც გამოგვადგება შესაბამისი ოფციებით (რომ გვანახოს ტაბულაციები, სხვა უხილავი სიმბოლოები და ხაზის დაბოლოება).

```
achiko@debian:~$ cat -vet test1.txt
one^Itwo^I^Ithree^I^I^Ifour$
```

ასე, ტაბულაციები „^I“ სიმბოლოებად გადაკეთდა. -vet ოფციის ნაცვლად, მოკლედ შეგვიძლია -A ოფციაც გამოვიყენოთ.

```
achiko@debian:~$ expand test1.txt > test2.txt
achiko@debian:~$ cat -A test2.txt
one      two              three              four$
```

მივიღეთ ლოგიკური შედეგი.

unexpand ბრძანება expand-ის შებრუნებულად იქცევა. ის ცვლის ცარიელ სიმბოლოებს ტაბულაციებად. -a, --all ოფციით ყველა ცარიელ სიმბოლოს გადააქცევს ტაბულაციად.

```
achiko@debian:~$ unexpand -a test2.txt > test3.txt
achiko@debian:~$ cat -A test{1,2,3}.txt
one^Itwo^I^Ithree^I^I^Ifour$
one      two              three              four$
one^Itwo^I^Ithree^I^I^Ifour$
```

ასე, ყველაფერი ნათლად გამოჩნდა.

სხვადასხვა ბრძანების დემონსტრირებისას სხვადასხვა სატესტო ფაილს ვქმნიდით ხოლმე. ყოველ ჯერზე ახალი ფაილი რომ არ შევქმნათ და ასე არ „დავაბინძუროთ“ ჩვენი დირექტორია, შეგვიძლია ძველი სატესტო ფაილები გამოვიყენოთ. უბრალოდ მასში ძველი შიგთავსი უნდა წავშალოთ და ახალი სასურველი შიგთავსი ჩავწეროთ. ფაილის შიგთავსის წაშლა მარტივია. შეგვიძლია სასურველ ტექსტურ რედაქტორში (მაგალითად vi-ში) გავხსნათ ფაილი და ხელით წავშალოთ ის, თუმცა ძალიან დიდი ზომის ფაილის შემთხვევაში, ეს საქმე გაცილებით მეტ დროს წაიღებს, თუ მოკლე გზები არ იცით. მოდით, განვიხილოთ ისინი.

ლინუქსში, როგორც ვახსენეთ, /dev დირექტორიაში გვაქვს null დასახელების მქონე „მოწყობილობა“ სადაც, ძირითადად, არასასურველ შეტყობინებებს გადავამისამართებთ ხოლმე. ეს „მოწყობილობა“ სპეციალური ტიპის ფაილს წარმოადგენს და მასში რაც მოხვდება, ყველაფერი იშლება. მოკლედ რომ ვთქვათ, მასში არაფერი წერია. შესაბამისად, შეგვიძლია, მისი შიგთავსი გადავაწეროთ ფაილს, ასე:


```
achiko@debian:~$ cat /dev/null > test3.txt
achiko@debian:~$ ls -l test3.txt
-rw-r--r-- 1 achiko achiko 0 Aug 15 17:02 test3.txt
```

აგრეთვე, შეგვიძლია პირდაპირ cp ბრძანება გამოვიყენოთ.

```
achiko@debian:~$ cp /dev/null test3.txt
```

შეგვიძლია echo ბრძანებითაც ვიმოქმედოთ.

```
achiko@debian:~$ echo "" > test3.txt      # ან
achiko@debian:~$ echo > test3.txt
achiko@debian:~$ ls -l test3.txt
-rw-r--r-- 1 achiko achiko 1 Aug 15 17:03 test3.txt
```

თუმცა, ასე ფაილის შიგთავსი სრულიად არ განუღდება, რადგან `echo` ბრძანების შესრულება  ღილაკზე დაჭერის ექვივალენტია, რომელიც ავტომატურად ახალ ხაზზე გადასვლას იწვევს. ამის თავიდან ასაცილებლად შეგვიძლია, `-n` ოფცია გამოვიყენოთ:

```
achiko@debian:~$ echo -n "" > test3.txt    # ან
achiko@debian:~$ printf "" > test3.txt
achiko@debian:~$ ls -l test3.txt          # ორივე შემთხვევაში ზომა განუღდება
-rw-r--r-- 1 achiko achiko 0 Aug 15 17:04 test3.txt
```

`vi` რედაქტორში თუ გვაქვს გახსნილი ფაილი, მარტივად შეგვიძლია, ასე მოვიქცეთ:

```
achiko@debian:~$ vi test3.txt
~
~
~
:1,$d
```

ერთ-ერთი სხვა ხერხი ასეთია: შეგვიძლია, ნებისმიერი ისეთი ბრძანება გამოვიყენოთ, რომელიც წარმატებით სრულდება და ეკრანზე არაფერს ბეჭდავს. ჩვენ მხოლოდ ამ ბრძანების სტანდარტული გამოსასვლელის ფაილისკენ გადამისამართება დაგვჭირდება. ერთ-ერთი ასეთი ბრძანებაა „:“ ან ბრძანება `true`.

```
achiko@debian:~$ : > test3.txt
achiko@debian:~$ true > test3.txt
```

თუმცა, ფაილის შიგთავსის განუღების ყველაზე მარტივი მეთოდი შემდეგია:

```
achiko@debian:~$ > test3.txt
```

ამ ხერხით „არაფერს“, `null`-ს, არარსებულ ობიექტს ვამისამართებთ ფაილისკენ.

გვაქვს ერთი საინტერესო ბრძანებაც, რომლითაც იმ ზომის ფაილს ვქმნით, რა ზომისაც გვსურს. ეს ბრძანებაა `truncate`. ის უფრო ზოგადი დანიშნულების არის, ვიდრე ჩვენ მიერ დასახული ამოცანა.

```
achiko@debian:~$ truncate -s 0 test3.txt
-rw-r--r-- 1 achiko achiko 0 Aug 15 17:05 test3.txt
achiko@debian:~$ truncate -s 5G test3.txt
-rw-r--r-- 1 achiko achiko 5368709120 Aug 15 17:05 test3.txt
```

თუ არ არსებობს არგუმენტად გადაცემული ფაილი, ის შეიქმნება. `-s` ოფციით კი ზომის მითითება ხდება. შესაბამისად, ამ ბრძანებით ნებისმიერ ზომის ფაილის შექმნა წამიერად შეგვიძლია. `truncate` ბრძანება ქმნის ე.წ. `Sparse`⁸ ფაილებს. სწორედ ამიტომ არის, რომ `truncate`-ით შექმნილი დიდი ზომის ფაილები დისკის დაკავებულობას არ ცვლის (შეგიძლიათ `df` ბრძანებით გადაამოწმოთ).

12.7 `fmt, nl`

`fmt` ბრძანება ტექსტური ფაილების ფორმატირებასა და ოპტიმიზაციას უზრუნველყოფს. ეს ყველაფერი ხელითაც შეიძლება გაკეთდეს, თუმცა როდესაც დიდ ტექსტს ეხება საქმე, იქ `fmt` ბრძანება დიდ მნიშვნელობას იძენს. იხილეთ მისი ზოგიერთი სასარგებლო ოფცია:

<code>-w, --width=</code>	მიუთითებს ხაზზე რამდენი ასო-ნიშანი შეიძლება დაეტიოს. ნაგულისხმევი მნიშვნელობით, ეს 75-ია (ქართული ტექსტისთვის 3-ჯერ ნაკლებია)
<code>-t, --tagged-paragraph</code>	უკეთესი კითხვადობისთვის გამოყოფს აბზაცს.
<code>-s, --split-only</code>	უკეთესი კითხვადობისთვის გამოყოფს გრძელ ხაზებს.
<code>-u, --uniform-spacing</code>	აერთგვაროვნებს ტექსტს. სიტყვებს შორის სვამს მაქსიმუმ ერთ გამოტოვებას, ხოლო წინადადებებს შორის კი 2-ს.

```
achiko@debian:~$ cat lorem2.txt
```

```

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla hendrerit
leo at dolor eleifend viverra. Proin arcu arcu, dictum sit amet interdum
quis, elementum lobortis sem. Aenean at dignissim lectus, tristique
egestas felis. Nulla eleifend massa nec ipsum varius facilisis. Interdum
et malesuada fames ac ante ipsum primis in faucibus. Sed tincidunt eros
ut egestas dignissim. Aliquam erat volutpat. Nam in ornare dui. Integer
id nulla vehicula, pretium est sed, vulputate libero. Donec ultricies
turpis id facilisis tincidunt. Proin nulla sem, faucibus id tempor eget,
ultrices at lorem. Fusce sed eros in ligula lacinia commodo ut vitae
quam. Mauris rutrum lorem dignissim condimentum tempus.
```

```
achiko@debian:~$ cat lorem2.txt | fmt -tu -w 50
```

```

Lorem ipsum dolor sit amet, consectetur
    adipiscing elit. Nulla hendrerit leo at dolor
    eleifend viverra. Proin arcu arcu, dictum
    sit amet interdum quis, elementum lobortis
    sem. Aenean at dignissim lectus, tristique
    egestas felis. Nulla eleifend massa nec
    ipsum varius facilisis. Interdum et malesuada
    fames ac ante ipsum primis in faucibus. Sed
    tincidunt eros ut egestas dignissim. Aliquam
    erat volutpat. Nam in ornare dui. Integer id
```

⁸`Sparse` file არის ისეთი ფაილი, რომლის შიგთავსის დიდი ნაწილიც ცარიელია. ასეთი ფაილების დისკზე ეფექტიანად განსათავსებლად, მათი შიგთავსის ცარიელი ბაიტების შენახვის სანაცვლოდ, წარმოგდენილია მეტამონაცემი, რომელიც დისკზე დამატებით ბლოკებს არ იკავებს.


```
nulla vehicula, pretium est sed, vulputate  
libero. Donec ultricies turpis id facilisis  
tincidunt. Proin nulla sem, faucibus id tempor  
eget, ultrices at lorem. Fusce sed eros in  
ligula lacinia commodo ut vitae quam. Mauris  
rutrum lorem dignissim condimentum tempus.
```

ბრძანება nl-ს უზრუნველყოფს სტანდარტული გამოსასვლელის ხაზების დანომრვას.

```
achiko@debian:~$ cat lorem.txt | nl  
1 Lorem ipsum dolor sit amet, consectetur adipiscing elit.  
2 Nam ultricies elit at cursus iaculis.  
3 Nunc suscipit lacus egestas ullamcorper efficitur.  
...  
achiko@debian:~$ cat lorem.txt | nl -w1      # ნუმერაციის სვეტის პოზიციაა 1  
1 Lorem ipsum dolor sit amet, consectetur adipiscing elit.  
2 Nam ultricies elit at cursus iaculis.  
3 Nunc suscipit lacus egestas ullamcorper efficitur.  
...  
achiko@debian:~$ cat lorem.txt | nl -w4      # ნუმერაციის სვეტის პოზიციაა 4  
1 Lorem ipsum dolor sit amet, consectetur adipiscing elit.  
2 Nam ultricies elit at cursus iaculis.  
3 Nunc suscipit lacus egestas ullamcorper efficitur.  
...  
achiko@debian:~$ cat lorem.txt | nl -i5      # ნუმერაციის ინტერვალია 5  
1 Lorem ipsum dolor sit amet, consectetur adipiscing elit.  
6 Nam ultricies elit at cursus iaculis.  
11 Nunc suscipit lacus egestas ullamcorper efficitur.  
16 Maecenas hendrerit sapien suscipit tincidunt egestas.  
...
```

nl ბრძანების დამატებითი შესაძლებლობები შეგიძლიათ მის სახელმძღვანელო გვერდზე ნახოთ.

12.8 od

ბრძანება od ლინუქსში გამოიყენება ფაილის შიგთავსის სხვადასხვა ფორმატში გამოსატანად. ეს ბრძანება სასარგებლოა ლინუქსის სკრიფტების გამოსაკვლევადა, რათა მასში არასასურველი სიმბოლო ან ცვლილება დავაფიქსიროთ. მისი სინტაქსი მარტივია. ნაგულისხმევი მნიშვნელობით, მას შიგთავსი რვაობით წარმოდგენაში (-o ოფციით) გამოაქვს. შესაძლებელია, აგრეთვე, გამოვიყენოთ ათობითი (-i ოფციით) სისტემა, თექვსმეტობითი (-x ოფციით) სისტემა, სიმბოლური წარმოდგენა (-c ოფციით) ან სხვა.

```
achiko@debian:~$ echo "lorem ipsum" | od  
0000000 067554 062562 020155 070151 072563 005155  
0000014
```

```

achiko@debian:~$ echo "lorem ipsum" | od -d
00000000 28524 25970 8301 28777 30067 2669
0000014
achiko@debian:~$ echo "lorem ipsum" | od -x
0000000000 6f6c 6572 206d 7069 7573 0a6d
0000014
achiko@debian:~$ echo "lorem ipsum" | od -c
00000000  l   o   r   e   m           i   p   s   u   m   \n
0000014

```

პირველი სვეტი წარმოადგენს ბაიტების წანაცვლებას. მისი წარმოდგენაც შეგვიძლია სხვადასხვა ფორმატში (-Ax თექვსმეტობითი, -Ao რვაობითი, -Ad ათობითი, -An პირველი სვეტის გარეშე).

მოდით, ახლა შევიტანოთ ერთ-ერთი უხილავი სიმბოლო ფაილში და ვნახოთ, თუ როგორ გამოისახება ის ეკრანზე ლინუქსის ბრძანებებით. მაგალითისთვის ავიღოთ იმავე ხაზის დასაწყისზე გადასვლის სიმბოლო (carriage return, \r). ის უხილავი სიმბოლოა. თუ მისი ჩასმა vi რედაქტორით გვსურს უნდა დავაჭიროთ Ctrl^v Ctrl^m კლავიშების კომბინაციას თანმიმდევრობით. შედეგად ^M სიმბოლოებს დავინახავთ ტექსტში.

```

achiko@debian:~$ vi invisible.txt
Gamarjoba Geor^Mgia
achiko@debian:~$ cat invisible.txt
giaarjoba Geor

```

ფაილის შიგთავსის გამოტანისას შედეგიც შესაბამისი მივიღეთ. carriage return კოდი შესრულდა და რეალური შიგთავსის ნაწილი დაიფარა. სწორედ ასეთ დროს უნდა გამოვიყენოთ od ბრძანება, რომ დავინახოთ ფაილის რეალური შიგთავსი.

```

achiko@debian:~$ cat invisible.txt | od -Ad -c
00000000  G   a   m   a   r   j   o   b   a           G   e   o   r   \r   g
0000016  i   a   \n
0000019

```

აქ უკვე ჩანს, რომ მე-15 ბაიტში ნამდვილად carriage return კოდია ჩასმული. cat -vet invisible.txt ბრძანებითაც შეგვეძლო იმის ნახვა, რომ ფაილში უხილავი სიმბოლოა ჩასმული, თუმცა კონკრეტულად რომელი - ვერ გავიგებდით.

უხილავი სიმბოლოები შეგვიძლია tr ბრძანებით მარტივად წავშალოთ, ასე:

```

achiko@debian:~$ cat invisible.txt | tr -dc '[:print:]' > visible.txt
achiko@debian:~$ cat visible.txt
Gamarjoba Georgia

```

ამ ბრძანების ზერელედ გამოყენება საზიფათოა. ამით სასარგებლო უხილავი სიმბოლოებიც წაიშლება (მაგალითად ახალ ხაზზე გადასვლა).

სხვა მაგალითიც ვნახოთ:

```
achiko@debian:~$ printf "\014Ga\013mar\013jo\013ba \012Geor\013gia\n "
```

```
Ga
  mar
    jo
      ba
Geor
  gia
achiko@debian:~$
```

```
$ printf "\014Ga\013mar\013jo\013ba \012Geor\013gia\n " | od -c
00000000  \f  G  a  \v  m  a  r  \v  j  o  \v  b  a      \n  G
00000020  e  o  r  \v  g  i  a  \n
00000030
```

როგორც წესი, ტექსტურ რედაქტორებში ასო-ნიშნების ტექსტში შეყვანას შესაბამისი ლილაკებზე დაჭერით ვახორციელებთ. თუმცა, როგორც ვნახეთ, შესაძლებელია სიმბოლოების ჩაწერა მათი კოდური წარმოდგენის მითითებითაც. სწორედ ეს ხერხი გვაძლევს იმის საშუალებას, რომ უხილავი სიმბოლოები შევიტანოთ ტექსტში. **vi** რედაქტორში ეს მარტივად არის შესაძლებელი. ამისთვის საკმარისია, ვიცოდეთ სიმბოლოს კოდი რვაობით, თექვსმეტობით ან უნიკოდის წარმოდგენაში. მაგალითისთვის ავიღოთ ვერტიკალური ტაბულაცია. მისი რვაობითი კოდია **013**, თექვსმეტობითი **0B**, ხოლო უნიკოდში კი წარმოდგენილია **000B** კოდით. **vi** რედაქტორში **[Ctrl^v]** კლავიშების კომბინაციის შემდეგ უნდა ავკრიფოთ **[o]** + **[013Oct]** (რვაობითი წარმოდგენის კოდი) ან **[x]** + **[0BHex]** (თექვსმეტობითის კოდი) ან **[u]** + **[000BU]** (უნიკოდის კოდი) კომბინაცია.

მოდით, ვცადოთ. ავიღოთ კვლავ **invisible.txt** ფაილი და „Gamarjoba Georgia“ ხაზში, „Ga“-ს შემდეგ ავკრიფოთ **[Ctrl^v]** + **[o]** + **[013Oct]**, „mar“-ს შემდეგ **[Ctrl^v]** + **[x]** + **[0BHex]** და „jo“-ს შემდეგ **[Ctrl^v]** + **[u]** + **[000BU]**. ეკრანზე ასეთი სიმბოლოები გამოისახება. **cat** ბრძანებით მისი შიგთავსის ნახვისას კი ეს კოდები შესრულდება:

```
achiko@debian:~$ vi invisible.txt
Ga^Kmar^Kjo^Kba Georgia
achiko@debian:~$ cat invisible.txt
Ga
  mar
    jo
      ba Georgia
```

ეს მეთოდი უნივერსალურია და მისი გამოყენება ნებისმიერი (არა მარტო უხილავი) სიმბოლოების მოსაცემად შეიძლება. ეს მაშინ ხდება აქტუალური, როდესაც უცხო ასო-ნიშნის ასაკრეფად შესაბამისი ენის კლავიატურის დრაივერი არ გვაქვს დაყენებული სისტემაში.

12.9 date, cal

ბრძანება **date**, როგორც თვითონ სიტყვა მიგვითითებს, თარიღს გვიჩვენებს.

```
achiko@debian:~$ date
```

ამ ბრძანებამ გამოიტანა კვირის დღე, თარიღი, დრო, დროის სარტყელი და წელი. მართალია, ამ ბრძანებას ტექსტების დამუშავებასთან დიდი არაფერი საერთო არ აქვს, მაგრამ ის ხშირად გამოიყენება ფაილის დასახელების განსაზღვრისას. ძალიან მოსახერხებელია, როდესაც ფაილის დასახელებაში მისი შექმნის დრო შედის. **date**-ის სხვადასხვა ფორმატით არგუმენტის გადაწოდებით დროის ცალკეული ელემენტების გამოტანა შეგვიძლია. მხოლოდ მიმდინარე წელი, თვე, საათი და ა.შ.

მოდით ვნახოთ რამდენიმე ელემენტი, რისი გამოტანაც შეუძლია ამ ბრძანებას:

%H	მიმდინარე საათი.
%M	მიმდინარე წუთი.
%S	მიმდინარე წამი.
%a	კვირის დღე (შემოკლებული ფორმით).
%B	თვის სახელი (შემოკლებული ფორმით).
%d	თვის დღე (1..31 ფორმატით).
%m	თვე (1..12 ფორმატით).
%j	წლის დღე (1..365 ფორმატით).
%Y	მიმდინარე წელი.
%s	ეპოქიდან ⁹ გასული წამების რაოდენობა.

ამ ელემენტების გამოტანისას წინ აუცილებლად „+“ ნიშანი უნდა წარუმიღვაროთ. ასე:

```
achiko@debian:~$ date +%Y          # მხოლოდ მიმდინარე წელი გამოგვაქვს
2018
achiko@debian:~$ ps aux > process_$(date +%Y%m%d%H%M%S).txt
achiko@debian:~$ ls
...
process_20180727162943.txt
```

ამ ბრძანებით კონკრეტული დროის პროცესების მონაცემებს ვინახავთ ფაილში, რომლის დასახელებაში შესაბამისი დრო არის მოცემული: წელი, თვე, რიცხვი, საათი, წუთი და წამი.

date ბრძანებას **-d** (**--date**) ოფციით შეუძლია გვანახოს თარიღი გარკვეულ დრომდე ან შემდეგ. მაგალითად:

⁹Unix-ის მსგავს სისტემებში ოპოქა არის Unix ოპერაციული სისტემის შექმნის დროს დროის ათვლის საწყისი წერტილი. ათვლა იწყება 1970 წლის 1 იანვრიდან წამებით. ეპოქა ცნობილია სხვა ტერმინებითაც, როგორიცაა: POSIX დრო ან Unix-ის დრო.

```

achiko@debian:~$ date -d '5 days'          # თარიღი 5 დღეში
Wed Aug  1 16:45:38 +04 2018
achiko@debian:~$ date -d '5 days ago'      # თარიღი 5 დღის წინ
Sun Jul 22 16:57:01 +04 2018
achiko@debian:~$ date +%A -d '18 oct 2010' # 18.10.2010-ის კვირის დღე
Monday

```

`date` ბრძანებას `-u` (`--utc`, `--universal`) ოფციით გამოაქვს UTC (Coordinated Universal Time) დრო. სადღეისოდ UTC დროის სტანდარტად არის აღიარებული. მისი წინანდელი სტანდარტი იყო გრინვიჩის მერიდიანის დრო (GMT (Greenwich Mean Time), ახლა GMT მხოლოდ დროის სარტყელს წარმოადგენს. UTC დრო სინქრონიზირდება საერთაშორისო ატომურ დროზე, რომელიც დაფუძნებულია 400 მაღალი სიზუსტის ატომურ საათზე და ასტრონომიულ დროზე, რომელიც, თავის მხრივ, დაფუძნებულია მზის სისტემაში დედამიწის ტრიალზე.

```

achiko@debian:~$ date -u
Fri Jul 27 13:13:04 UTC 2018
achiko@debian:~$ date
Fri Jul 27 17:13:05 +04 2018

```

`date` ბრძანებით შესაძლებელია დროის შეცვლა კომპიუტერზე. თუმცა, ამისთვის ადმინისტრატორის უფლებებია საჭირო. ახალი დასაყენებელი დრო არგუმენტად ასეთი ფორმატით უნდა გადავცეთ: `MMDDhhmmYY[.ss]` (თვე,დღე,საათი,წუთი,წელი,.წამი), წამების სიზუსტე ნებაყოფლობითია და შეგვიძლია, არ მივუთითოთ. ამას გარდა, ბევრი სხვაგვარი ფორმატის მითითებაც შესაძლებელია.

```

achiko@debian:~# date 010214152015.34
Fri Jan  2 14:15:34 +04 2015

```

`date` ბრძანების გარდა, დროის დასაყენებლად სხვა ხერხებიც არსებობს. ზოგადად, კომპიუტერში დროის შესახებ დეტალურ განხილვას მოგვიანებით დავუბრუნდებით.

`cal` ბრძანებით მიმდინარე თვის კალენდარი გამოდის ეკრანზე. მთლიანი წლის კალენდრის ნახვა `-y` ოფციით არის შესაძლებელი. თუ კონკრეტული წლის კალენდრის ნახვა გვსურს, მაშინ არგუმენტად შესაბამისი წელი უნდა მივუთითოთ.

```

achiko@debian:~# cal
      July 2018
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

achiko@debian:~# cal 1121
1121

```

January							February							March							
Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	
						1			1	2	3	4	5			1	2	3	4	5	
2	3	4	5	6	7	8	6	7	8	9	10	11	12	6	7	8	9	10	11	12	
9	10	11	12	13	14	15	13	14	15	16	17	18	19	13	14	15	16	17	18	19	
16	17	18	19	20	21	22	20	21	22	23	24	25	26	20	21	22	23	24	25	26	
23	24	25	26	27	28	29	27	28						27	28	29	30	31			
30	31																				
April							May							June							
Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	
					1	2	1	2	3	4	5	6	7				1	2	3	4	
3	4	5	6	7	8	9	8	9	10	11	12	13	14	5	6	7	8	9	10	11	
10	11	12	13	14	15	16	15	16	17	18	19	20	21	12	13	14	15	16	17	18	
17	18	19	20	21	22	23	22	23	24	25	26	27	28	19	20	21	22	23	24	25	
24	25	26	27	28	29	30	29	30	31					26	27	28	29	30			
July							August							September							
Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	
					1	2			1	2	3	4	5	6					1	2	3
3	4	5	6	7	8	9	7	8	9	10	11	12	13	4	5	6	7	8	9	10	
10	11	12	13	14	15	16	14	15	16	17	18	19	20	11	12	13	14	15	16	17	
17	18	19	20	21	22	23	21	22	23	24	25	26	27	18	19	20	21	22	23	24	
24	25	26	27	28	29	30	28	29	30	31				25	26	27	28	29	30		
31																					
October							November							December							
Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	
						1			1	2	3	4	5					1	2	3	
2	3	4	5	6	7	8	6	7	8	9	10	11	12	4	5	6	7	8	9	10	
9	10	11	12	13	14	15	13	14	15	16	17	18	19	11	12	13	14	15	16	17	
16	17	18	19	20	21	22	20	21	22	23	24	25	26	18	19	20	21	22	23	24	
23	24	25	26	27	28	29	27	28	29	30				25	26	27	28	29	30	31	

როგორც ჩანს, დიდგორის ბრძოლა (1121 წლის 12 აგვისტო) პარასკევ დღეს მოუგია დავით აღმაშენებელს :)

თავი 13

რეგულარული გამოსახულება

მ თავში ჩვენ განვიხილავთ ტექსტებთან მანიპულირების ინსტრუმენტებს. როგორც უკვე აღვნიშნეთ, Unix-ის მსგავს სისტემებში ტექსტურ მონაცემებს დიდი დატვირთვა აქვს. სწორედ ასეთი ინსტრუმენტია რეგულარული გამოსახულება (regular expression, მოკლედ regexp ან regex). ის წარმოადგენს სტრუქტურულ ჩანაწერს, შაბლონს, რომელიც ტექსტებში გარკვეული ბლოკის გამოსაცნობად გამოიყენება. მისი შემადგენელი თითოეული ასო-ნიშანი ან მეტასიმბოლოა, რომელსაც სპეციალური მნიშვნელობა აქვს, ან ჩვეულებრივი სიმბოლო, რომელსაც პირდაპირი მნიშვნელობა აქვს.

რეგულარული გამოსახულებები ხშირად გამოიყენება საძიებო სისტემებში, ტექსტურ რედაქტორებში, ლექსიკულ ანალიზში, და ზოგადად, დაპროგრამების ბევრ ენაში ტექსტებთან სამუშაოდ. ლინუქსში ბრძანებათა ხაზის ბევრ ბრძანებას აქვს რეგულარული გამოსახულების მხარდაჭერა. ჩვენ სწორედ ამ მიმართულებით განვიხილავთ მას. ერთი შეხედვით, ეს ყველაფერი ჰგავს ბრძანებათა ხაზში ჩვენთვის უკვე კარგად ნაცნობ მაგენერირებელ სიმბოლოებს, რომლებიც ფაილების სრული დასახელების შესავსებად გამოიყენება, თუმცა რეგულარული გამოსახულება გაცილებით დიდ მასშტაბს ეხება. ის ყველა ინსტრუმენტსა თუ დაპროგრამების ენაში შეიძლება ერთნაირი არ იყოს და შესაძლებელია, პატარა ნაირსხვაობით ხასიათდებოდეს. ჩვენ ამ თავში მხოლოდ POSIX სტანდარტით განსაზღვრული რეგულარული გამოსახულებით შემოვიფარგლებით.

13.1 grep

ერთ-ერთი ყველაზე სასარგებლო და ხშირად გამოყენებადი ბრძანება, რომელიც რეგულარულ გამოსახულებებს იყენებს, არის ბრძანება **grep**. მისი დასახელება მოდის „global regular expression print“-დან. **grep**-ით შეგვიძლია მოვნახოთ მისი სტანდარტული შესასვლელიდან ის ხაზები, რომლების მოსაძიებნ შაბლონს დაემთხვევა.

მაგალითისთვის ავიღოთ ერთ-ერთი ტექსტური ფაილი `/usr/share/common-licences` დირექტორიდან.

```
achiko@debian:~$ cp /usr/share/common-licences/GPL-3 .
achiko@debian:~$ grep "GNU" GPL-3
GNU GENERAL PUBLIC LICENSE
The GNU General Public License is a free, copyleft license for
...
```

ეს ბრძანება არგუმენტად გადაცემულ ფაილის შიგთავსში მოძებნის და გამოიტანს ყველა იმ ხაზს, რომლებიც შეიცავს სიტყვა GNU-ს.

უკეთესი ხილვადობისთვის შეგვიძლია მოსაძებნ გამოსახულებას ეკრანზე გამოტანისას გამოკვეთილი ფერი მივცეთ `--color` ოფციით. ასე უკეთ დავინახავთ და დავრწმუნდებით, რომ ხაზი გამოსახულებას შეიცავს.

```
achiko@debian:~$ grep --color "GNU" GPL-3
GNU GENERAL PUBLIC LICENSE
The GNU General Public License is a free, copyleft license for
...
```

ხშირად, ის სიტყვა, გამოსახულება, რომლის მოძებნაც გვსურს, ტექსტში ზოგჯერ დიდი ასოებით არის მოცემული, ზოგჯერ პატარა ან შერეული ასოებით. თუ გვსურს, რომ გამოსახულება მოვძებნოთ ისე, რომ მის დიდ და პატარა ასოებს ყურადღება არ მიექცეს, მაშინ `-i` (`--ignore-case`) ოფცია უნდა გამოვიყენოთ.

```
achiko@debian:~$ grep --color -i "license" GPL-3
GNU GENERAL PUBLIC LICENSE
of this license document, but changing it is not allowed.
The GNU General Public License is a free, copyleft license for
...
```

როგორც ვხედავთ, ამ ბრძანებამ გამოგვიტანა ის ხაზები, სადაც `license` ჩაწერილია დიდი ან პატარა ასოებით. შერეული ვარიანტიც რომ ყოფილიყო, მასაც გამოიტანდა.

დიდ ტექსტში გამოსახულების მოძებნისას ხშირად სასარგებლოა, ვიცოდეთ იმ ხაზების ნომრები, რომლებიც გამოსახულებას შეიცავენ. `-n` (`--line-number`) ოფციით, მოძებნილ ხაზებთან ერთად, ეკრანზე მათი ნუმერაციაც გამოჩნდება.

```
achiko@debian:~$ grep -n GNU GPL-3
1: GNU GENERAL PUBLIC LICENSE
10: The GNU General Public License is a free, copyleft license for
15: the GNU General Public License is intended to guarantee your freedom to
...
```

ეკრანზე გამოსახულების შემცველი ხაზი გამოდის, თუმცა კონტექსტის უკეთ გასაგებად, სასურველია მოძებნილი ხაზის წინა და შემდეგი ხაზების დანახვაც. ამ შესაძლებლობას `grep` ბრძანება უზრუნველყოფს `-A`, `-B` და `-C` ოფციებით.

```
achiko@debian:~$ grep -A3 GNU GPL-3
```

ეს ბრძანება გამოიტანს ხაზებს, რომლებიც GNU-ს შეიცავენ და დამატებით, ამ ხაზების შემდეგ 3 ხაზს.

```
achiko@debian:~$ grep -B3 GNU GPL-3
```

ეს კი ხაზებს, რომლებიც GNU-ს შეიცავენ და დამატებით, ამ ხაზების წინა 3 ხაზს.


```
achiko@debian:~$ grep -C3 GNU GPL-3
```

ხოლო ეს ბრძანება დამატებით წინა და შემდეგ 3 ხაზსაც.

-A, -B და -C ოფციებს გრძელი ფორმატის ექვივალენტური ჩანაწერები აქვს. ესენია: --after-context=N, --before-context=N და --context=N.

ზოგ შემთხვევაში, შეიძლება გამოსახულების გამორიცხვა დაგვჭირდეს ხაზებიდან. ანუ, იმ ხაზების მოძებნა, რომლებიც, პირიქით, არ შეიცავენ მოსაძებნ გამოსახულებს. -v (--invert-match) ოფციით სწორედ შებრუნებული ხაზები გამოვა.

```
achiko@debian:~$ grep -v GNU GPL-3
```

როგორც ვთქვით, ეკრანზე გამოტანილი არცერთი ხაზი არ შეიცავს GNU გამოსახულებას.

რიგ შემთხვევებში, უფრო მეტად ხაზების რაოდენობა გვაინტერესებს და არა თვითონ ის ხაზები, რომლებიც ამ გამოსახულებას შეიცავენ. ასეთ სიტუაციაში, შეგვიძლია, ხაზები wc ბრძანებას დავათვლევირთ, თუმცა grep-ის -c (--count) ოფცია უზრუნველყოფს ხაზების რაოდენობას დათვლას.

```
achiko@debian:~$ grep -c GNU GPL-3
19
```

grep-ით გამოსახულების მოძებნა შესაძლებელია არა მხოლოდ ერთი ფაილის მიგთავსში, არამედ იმდენში, რამდენ ფაილსაც გადავცემთ ბრძანებას არგუმენტად.

```
achiko@debian:~$ grep GNU LICENSE.txt GPL-3
...
LICENSE.txt:previously distributed under the GNU General Public License
GPL-3:The GNU General Public License does not permit incorporating
GPL-3:the library. If this is what you want to do, use the GNU Lesser
...
```

შეგვიძლია ფაილების ერთობლიობა *-თაც მივუთითოთ.

```
achiko@debian:~$ grep GNU TXT/*
```

ასე, GNU გამოსახულება TXT/ დირექტორიაში არსებულ ყველა ფაილში მოიძებნება. თუ გვსურს GNU გამოსახულება ქვედირექტორიებში არსებულ ფაილებშიც რეკურსურად მოიძებნოს, მაშინ -r ოფცია უნდა გამოვიყენოთ.

```
achiko@debian:~$ grep -r GNU .
...
GPL-3:GNU General Public License, you may choose any version
./TXT/foobar.txt:passing these types as a comma-separated list (GNU ext
...
```

რეკურსიული ძებნისთვის -R ოფციის გამოყენებაც შეგვიძლია. განსხვავებით -r-სგან, -R-ით მაღსახმობშიც მოიძებნება.

ასეთი მიდგომით, **grep** ბრძანება ეკრანზე გამოიტანს ამ გამოსახულების შემცველ ხაზებს, შესაბამისი ფაილის დასახელებებთან ერთად. შედეგად, გამოსავალი გადატვირთული იქნება. ამიტომ შეგვიძლია, მხოლოდ ფაილების სიის გამოტანით შემოვიფარგლოთ. ამისთვის -l (--file-with-matches) ოფცია უნდა გამოვიყენოთ.

```
achiko@debian:~$ grep -Rl GNU .
...
./TXT/foobar.txt
./LICENSE.txt
./GPL-3
...
```

-w ოფციის გამოყენებით შეგვიძლია, მოვძებნოთ ის ხაზები, რომლებიც კონკრეტულ სიტყვას შეიცავს.

```
achiko@debian:~$ grep -w ipsum lorem.txt
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Praesent sollicitudin ipsum vel pharetra gravida.
```

ეს ბრძანება გამოიტანს იმ ხაზებს, რომლებიც შეიცავს სიტყვა **ipsum**-ს. ეს ხაზები ეკრანზე -w ოფციის გარეშეც გამოვიდოდა, თუმცა დამატებით ის ხაზებიც გამოვა, სადაც ასო-ნიშნების ასეთი წყობა **ipsum** სხვა სიტყვის შემადგენელი ნაწილიცაა.

უკვე ვახსენეთ, რომ **grep**-ის შედეგით ხანდახან ეკრანი საკმაოდ გადაიტვირთება. -o ოფციით შესაძლებელია ეკრანზე მხოლოდ მოძებნილი გამოსახულება გამოვიდეს და არა მისი შემცველი მთლიანი ხაზი.

```
achiko@debian:~$ grep -o GNU GPL-3
GNU
GNU
GNU
...
```

ერთი შეხედვით, ახალი არაფერი მივიღეთ, ეკრანზე გამოვიდა მხოლოდ მოსაძებნი გამოსახულება, რომელიც ჩვენ თვითონ მივუთითეთ ბრძანებას. ამ ოფციის მნიშვნელობა უკეთ გამოჩნდება რეგულარული გამოსახულების გამოყენების დროს. მაშინ, როდესაც მოსაძებნი გამოსახულებაში ჩვეულებრივი სიმბოლოების გარდა, მეტასიმბოლოებსაც ჩავწერთ. მოდით, დეტალურად შევისწავლოთ ის.

დავიწყეთ!

grep ბრძანებაში შესაძლებელია, გამოვიყენოთ როგორც ძირითადი რეგულარული გამოსახულების ჩანაწერი (BRE – basic regular expression), ასევე გაფართოებული რეგულარული გამოსახულების ჩანაწერი (ERE – extended regular expression).

ძირითად რეგულარულ გამოსახულებებში შემდეგი მეტასიმბოლოები არსებობს:

ძირითადი რეგულარული გამოსახულების (BRE) მეტასიმბოლოები

- ^** ეს მეტასიმბოლო მხოლოდ ხაზის დასაწყისში მოძებნის გამოსახულებას.
მაგ: `^Mishka` იმ ხაზებს აღნიშნავს, რომლებიც `Mishka`-თი იწყება.
-
- \$** ეს მეტასიმბოლო ხაზის ბოლოში ეძებს გამოსახულებას.
მაგ: `Mishka$` იმ ხაზებს აღნიშნავს, რომლებიც `Mishka`-ზე მთავრდება.
-
- .** წერტილი აღნიშნავს ერთ ნებისმიერ ასო-ნიშანს.
მაგ: `M.shka`-ს შეიძლება შეესაბამებოდეს `Mishka`, `Mashka` და ა.შ.
-
- *** ეს მეტასიმბოლო შეიძლება მიეწეროს ნებისმიერ ასო-ნიშანს და ის ამ სიმბოლოს ნულჯერ ან ბევრჯერ გამოერებას ნიშნავს.
მაგ: `ab*c` გამოსახულებით მოიძებნება: `abc`, `abbc`, `abbbc` ... და `ac`-ც, რადგან `b*`, როგორც აღვნიშნეთ, `b`-ს ნულჯერ გამოერებასაც ნიშნავს.
-
- ** ეს სიმბოლო, როგორც ზოგადად შელში, უკარგავს მნიშვნელობას მეტასიმბოლოებს.
მაგ: `tree\.` მოძებნის `tree.`-ს და არა `trees`, რადგან წერტილმა დაკარგა თავისი მნიშვნელობა და როგორც ჩვეულებრივი სიმბოლო ისე აღიქმება.
-
- \N** ეს ჩანაწერი, სადაც `N` რიცხვია, მე-N მონიშნულ ქვეგამოსახულებს ნიშნავს რეგულარულ გამოსახულებაში.
-
- []** კვადრატულ ფრჩხილებში წარმოდგენილ ასო-ნიშნებიდან ის აღნიშნავს ერთ-ერთს.
მაგ: `[abc]`–თი აღინიშნება ან `a`, ან `b` და ან `c`. კვადრატული ფრჩხილებით, როგორც უკვე შელიდან ვიცით, შეიძლება მოიცეს ასო-ნიშანთა ერთობლიობა ინტერვალით ან კლასით. მაგ: ინტერვალი `[b-g]` აღნიშნავს `b`-დან `g`-ს ჩათვლით ერთ-ერთ სიმბოლოს. კვადრეტული ფრჩხილების დასაწყისში ჩაწერილი სიმბოლო `^` კი დანარჩენი ასო-ნიშნების გარდა, სხვა ასო-ნიშნებს აღნიშნავს. მაგ: `^[abc]` `a`-ს, `b`-სა და `c`-ს გარდა, სხვა ასო-ნიშნებს გულისხმობს. ასე რომ, ყურადღება მიაქციეთ, რომ კვადრატულ ფრჩხილებსა და მის გარეთ ჩაწერილ `^` სიმბოლოს სხვადასხვა მნიშვნელობა აქვს. ინტერვალის გარდა `grep`-ს ესმის `POSIX` კლასებიც.

ასო-ნიშანთა POSIX კლასები

- [:lower:]** ლათინური ანბანის პატარა ასო-ნიშნები.
-
- [:upper:]** ლათინური ანბანის დიდი ასო-ნიშნები.
-
- [:digit:]** ციფრები.
-
- [:alpha:]** ლათინური ანბანის ასო-ნიშნები. იგივეა, რაც `[A-Za-z]`, `[:upper:]` `[:lower:]`.
-
- [:alnum:]** ლათინური ანბანის დიდი და პატარა ასოები და ციფრები. იგივეა, რაც `[0-9A-Za-z]` ან `[:alpha:]` `[:digit:]`.
-
- [:blank:]** გამოტოვება და ტაბულაცია.

<code>[:space:]</code>	გამოტოვება, ახალ ხაზზე გადასვლა, ჰორიზონტალური და ვერტიკალური ტაბულაცია, იმავე ხაზის დასაწყისზე გადასვლა, ახალ გვერდზე გადასვლა.
<code>[:punct:]</code>	პუნქტუაციის ნიშნები. მასში შედის შემდეგი სიმბოლოები: <code>!"#\$%&'()*+,-./:;<=>?@[\\]^_`{ }~</code>
<code>[:print:]</code>	ხილული სიმბოლოები. იგივეა, რაც <code>[:graph:]</code> <code>[:blank:]</code> .
<code>[:cntrl:]</code>	უხილავი, კონტროლის სიმბოლოები.
<code>[:xdigit:]</code>	თექვსმეტობითი სისტემის ასო-ნიშნები. იგივეა, რაც <code>[0-9A-Fa-f]</code> .

იმისთვის, რომ **grep** ბრძანებას კლასი მივუთითოთ მისი შემადგენელი ერთ-ერთი სიმბოლო მოსაძებნად, ეს კლასი, თავის მხრივ, კიდევ უნდა ჩავსვათ კვადრატულ ფრჩხილებში. ასე: `[:lower:]`, `[:digit:]` და ა.შ.

გაფართოებული რეგულარული გამოსახულების (ERE) მეტასიმბოლოები

+	ეს მეტასიმბოლო შეიძლება, მიეწეროს ნებისმიერ ასო-ნიშანს და ის ამ სიმბოლოს ერთხელ ან ბევრჯერ გამეორებას ნიშნავს. მაგ: <code>ab*c</code> -ით მოიძებნება: <code>abc</code> , <code>abbc</code> , <code>abbbc</code> . აქ გამოირიცხება <code>ac</code> .
?	ეს მეტასიმბოლო შეიძლება, მიეწეროს ნებისმიერ ასო-ნიშანს და ის ამ სიმბოლოს ერთხელ ან ნულჯერ გამეორებას ნიშნავს. მაგ: <code>ab?c</code> -ით მოიძებნება: <code>abc</code> და <code>ac</code> .
	ამ მეტასიმბოლოთი ალტერნატივის ჩაწერა შესაძლებელი. მაგ: <code>Mishka Nini Ioane</code> აღნიშნავს <code>Mishka</code> -ს ან <code>Nini</code> -ს ან <code>Ioane</code> -ს.
()	ფრჩხილებში მოთავსება დაჯგუფებას ნიშნავს. მაგ: <code>(a b)c</code> ნიშნავს ან <code>ac</code> -ს ან <code>bc</code> -ს. ფრჩხილების გარეშე კი <code>a bc</code> გამოსახულებაში იგულისხმება ან <code>a</code> ან <code>bc</code> .
{ }	ფიგურულ ფრჩხილებში რიცხვი იწერება და ასეთი ჩანაწერი მის წინ მდგომი ასო-ნიშნის ამ რიცხვჯერ გამეორებას ნიშნავს. <code>{n}</code> ასო-ნიშნის ზუსტად <code>n</code> -ჯერ გამეორებაა. მაგ: <code>a{5}</code> ნიშნავს <code>aaaaa</code> -ს. <code>{n,}</code> ასო-ნიშნის <code>n</code> -ჯერ ან მეტჯერ გამეორებაა. მაგ: <code>a{2,}</code> არის <code>aa</code> , <code>aaa</code> , <code>aaaa</code> ... <code>{,m}</code> ასო-ნიშნის მაქსიმუმ <code>m</code> -ჯერ გამეორებაა. მაგ: <code>ba{,3}b</code> არის <code>bb</code> , <code>bab</code> , <code>baab</code> ან <code>baaab</code> <code>{n,m}</code> ასო-ნიშნის <code>n</code> -დან <code>m</code> -ჯერ გამეორებაა. მაგ: <code>a{2,4}</code> არის <code>aa</code> , <code>aaa</code> ან <code>aaaa</code> ყურადღება მიაქციეთ, რომ ასეთი ფორმა ეძებს წინ მიწერილი ერთი ასო-ნიშნის გამეორებას. თუ გვსურს, რომ რამდენიმე ასო-ნიშნის, სიტყვის გამეორება ვეძებოთ, მაშინ ასო-ნიშნთა ასეთი გაერთიანება უნდა დავაჯგუფოთ. მაგ: <code>Io{2,4}</code> ნიშნავს <code>Io</code> -ს, <code>Io</code> ო-ს ან <code>Io</code> ოო-ს, ხოლო <code>(Io){2,4}</code> ნიშნავს <code>IoIo</code> , <code>IoIoIo</code> -ს ან <code>IoIoIoIo</code> -ს.

`\b`, `\B` `\b` მეტასიმბოლოთი აღინიშნება სიტყვის კიდე. რეგულარულ გამოსახულებაში სიტყვად მოიაზრება მხოლოდ ანბანის ასოები ან ციფრები და ან ქვედა ტირე (-). `\B` მეტასიმბოლოთი კი სიტყვის შუა ნაწილი აღინიშნება, და არა კიდე. მაგ:

`\bMishka\b`-თი მოიძებნება მხოლოდ სიტყვა `Mishka`.

`\bMishka``\B`-თი ის სიტყვა, რომელიც იწყება `Mishka`-თი და აუცილებლად გრძელდება სხვა ერთი ასო-ნიშნით.

`\BMishka\b`-თი ის სიტყვა, რომელიც იწყება რომელიმე ასო-ნიშნებით და აუცილებლად მთავრდება `Mishka`.

`\BMishka``\B`-თი ის სიტყვა, რომელიც აუცილებლად იწყება და მთავრდება რომელიმე ასო-ნიშნებით და შუალედში აუცილებლად არის წარმოდგენილი `Mishka`.

`<`, `>` GNU-ს დამკვიდრებული ბრძანებებში, სიტყვის დაწყება-დამთავრების აღსანიშნავად, აგრეთვე გამოიყენება საკუთარი სინტაქსური ჩანაწერი. სწორედ ეს მეტასიმბოლოები აღნიშნავენ GNU-ს სინტაქსში კიდეებს სიტყვის დასაწყისსა და ბოლოში.

მაგ: `<word>` ნიშნავს სიტყვა `word`-ს. ამ სიმბოლოების ცალ-ცალკე ხმარებაც შეიძლება. `<word` ნიშნავს სიტყვას, რომელიც იწყება `word`-ით და უნდა გაგრძელდეს სხვა სიმბოლოებით. `word>` კი ნიშნავს სიტყვას, რომელიც მთავრდება `word`-ით და უნდა იწყებოდეს სხვა სიმბოლოებით.

თუ გვსურს ისეთი სიტყვის მოძებნა, რომელიც შედგება სხვა სიმბოლოებისგანაც, გარდა ანბანის ასოების, ციფრებისა და ქვედატირესგან (-), მაშინ მხოლოდ `-w` ოფცია გამოგვადგება ამაში.

ძირითად რეგულარულ გამოსახულებაში გაფართოებული რეგულარული გამოსახულების ზოგიერთ მეტასიმბოლოს აღარ გააჩნია სპეციალური მნიშვნელობა. ესენია: `?`, `+`, `{`, `|`, `(` და `)`. ამ ფუნქციის გამოსაყენებლად, ძირითად რეგულარულ გამოსახულებაში მათ წინ ბეჭდულში უნდა დავუმატოთ: `\?`, `\+`, `\{`, `\|`, `\(` და `\)`.

გაფართოებული რეგულარული გამოსახულების გამოყენებისას `grep` ბრძანებაში აუცილებლად `-E` (`--extended-regexp`) ოფცია უნდა გამოვიყენოთ. მის გარეშე ის მხოლოდ ძირითად რეგულარულ გამოსახულებას აღიქვამს.

დაიმახსოვრეთ!

რეგულარული გამოსახულება, სადაც მეტასიმბოლოები შედის ყოველთვის უნდა ჩავსვათ ბრჭყალებში `"` ან მოვათავსოთ აპოსტროფებს შორის `'`.

მოდით, ახლა მაგალითები მოვიყვანოთ. ავიღოთ ერთ-ერთი სატესტო ფაილი და გამოვიტანოთ ის ხაზები, რომლებიც შეიცავს გამოსახულებებს - `linux` ან `Linux`:

```
achiko@debian:~$ cat test.txt | grep "[lL]inux"
```

მარტივია. მეორენაირად ასეც შეგვიძლია დავწეროთ ბრძანება:

```
achiko@debian:~$ cat test.txt | grep -E "(l|L)linux"
```

იმისთვის, რომ არ გამოვიყენოთ გაფართოებული რეგულარული გამოსახულება და შევინარჩუნოთ მიდგომა, ასეც შეგვიძლია ჩაწეროთ:

```
achiko@debian:~$ cat test.txt | grep "(l|L)linux"
```

ამ ამოცანის ამოსახსნელად გაუმართლებელი იქნებოდა `-i` ოფციის გამოყენება.

```
achiko@debian:~$ cat test.txt | grep -i "linux"
```

რადგან ეს ბრძანება გამოიტანდა ისეთ ხაზებსაც, რომლებიც შეიცავს მაგალითად `LiNux`-ს ან `LINUX`-ს და ა.შ.

თუ პირობას ცოტა შევცვლით და მიზნად ისეთი ხაზების გამოტანას დავისახავთ, სადაც შედის სიტყვა და არა გამოსახულება `linux` ან `Linux`, მაშინ ბრძანება მიიღებს შემდეგ სახეს:

```
achiko@debian:~$ cat test.txt | grep -w "[lL]linux"
```

ან

```
achiko@debian:~$ cat test.txt | grep -Ew "(l|L)linux"
```

ან

```
achiko@debian:~$ cat test.txt | grep -E "\b(l|L)linux\b"
```

ან

```
achiko@debian:~$ cat test.txt | grep -E "<(l|L)linux>"
```

ვთქვათ, გვსურს მოვძებნოთ სიტყვა „`linux`“. აქ უკვე `\b`, `<` და `>` მეტასიმბოლოები აღარ გამოგვადგება. ისინი, ხომ სიმბოლო „`\`“-ს სიტყვის შემადგენელ ასო-ნიშნად ვერ აღიქვამენ. ამიტომ, ამ შემთხვევაში მხოლოდ `-w` ოფცია დაგვჭირდება.

```
achiko@debian:~$ cat test.txt | grep -w "linux"
```

ამის დემონსტრირება სხვა ელემენტარული მაგალითითაც ნათლად შეგვიძლია:

```
$ echo "Here /usr/bin is the word" | grep -w '/usr/bin'
Here /usr/bin is the word
```

შედეგი ეკრანზე გამოვიდა.

```
$ echo "Here /usr/bin is not the word" | grep -E '\b/usr/bin\b'
$
```

```
$ echo "Here /usr/bin is not the word" | grep -E '</usr/bin>'
$
```

ეს ბრძანებები კი ვერ მოძებნიან /usr/bin-ს, რადგან სიტყვაში „/“ სიმბოლო შედის. ახლა, მოძებნოთ ხაზები, რომლებიც იწყება linux ან Linux-ით ან და unix ან Unix-ით.

```
achiko@debian:~$ cat test.txt | grep -E '^(l|L)inux|(u|U)nix'
```

დავითვალდოთ ცარიელი ხაზები ფაილში:

```
achiko@debian:~$ cat test.txt | grep -c '^$'
```

აქ ვითვლით იმ ხაზებს, რომლებიც დაწყებისთანავე მთავრდება. სხვა მიდგომითაც შეგვიძლია ეს ამოცანა ამოვხსნათ, ასე:

```
achiko@debian:~$ cat test.txt | grep -vc '.'
```

„.“ ზომ ნიშნავს ერთ ნებისმიერ ასო-ნისანს, -v ოფციით კი ის ხაზი გამოვა, სადაც ეს ასო-ნიშანი არ შედის, ანუ ცარიელი ხაზი. -c-ით კი დავითვლით ასეთი ხაზების რაოდენობას.

გამოვიტანოთ ის ხაზები, რომლებიც მხოლოდ ციფრებს შეიცავს:

```
achiko@debian:~$ cat test.txt | grep -v '[^[:digit:]]'
```

-v ოფციის გარეშე გამოვიდოდა ის ხაზები, რომლებიც შეიცავს ციფრის გარდა სხვა სიმბოლოებს (ასეთ ხაზში შეიძლება მოხვდეს ციფრიც. ხაზი ზომ, მის გარდა, სხვა სიმბოლოებს შეიცავს). -v კი ამ ხაზების ინვერსიას აკეთებს და, მათ ნაცვლად, დანარჩენ ხაზებს გამოიტანს ეკრანზე. ანუ მხოლოდ ციფრებს შემცველ ხაზებს.

ეს ბრძანება ცარიელ ხაზსაც გამოიტანს (თუ, რა თქმა უნდა, ასეთი ამ ფაილში არსებობს), ამიტომ ისინი დამატებით უნდა გავფილტროთ.

```
achiko@debian:~$ cat test.txt | grep -v '[^[:digit:]]' | grep '.'
```

გამოვიტანოთ ხაზები, რომლებიც მხოლოდ 5 ასო-ნიშანს შეიცავს.

```
achiko@debian:~$ cat test.txt | grep -E '^.{5}$'
```

გამოვიტანოთ ხაზები, რომლებიც მობილური ტელეფონების მხოლოდ საქართველოში რეგისტრირებულ ნომრებს შეიცავს, ასეთი ფორმატით - +9955xxxxxxxx:

```
achiko@debian:~$ cat test.txt | grep -w '\+9955[0-9]\{8\}'
```

მოდით, ცოტა გავართულოთ ამოცანები.

დავითვალოთ, სულ რამდენი ფაილი გვაქვს ჩვენს პირად დირექტორიაში (ვგულისხმობთ მათ ქვე-დირექტორიებსაც):

```
$ ls -ARl $HOME | grep '^-' | tr -s ' ' | cut -f5 -d' ' | grep -c '^\.'
```

განვმარტოთ!

ჯერ რეკურსიულად გამოგვაქვს პირადი დირექტორიიდან ყველა ფაილის სია გრძელი ფორმატით, შემდეგ ვარჩევთ მხოლოდ ჩვეულებრივი ტიპის ფაილს (გრძელ ფორმატში მათი შესაბამისი ხაზები „-“-ით იწყება), შემდეგ გამოგვაქვს ფაილის დასახელებები (მე-9 სვეტშია განთავსებული). მე-9 სვეტი სწორად რომ გამოვიტანოთ, შევკვეცოთ სვეტებს შორის არსებული მრავალი გამოტოვების სიმბოლო ერთი დაშორებით (სვეტებს შორის დაშორება არაერთგვაროვანია და შეიძლება მრავალ გამოტოვებას შეიცავდეს). შემდეგ კი, ფაილის დასახელებებიდან გამოვიტანოთ მხოლოდ ისინი, რომლებიც წერტილით იწყება. **grep**-ში რადგან „.“ მეტასიმბოლოს წარმოადგენს „-“-ით ვუკარგავთ მის მნიშვნელობას.

გამოვიტანოთ ის ხაზები, რომლების სწორად ჩაწერილ IP მისამართებს შეიცავენ:

```
achiko@debian:~$ grep -E '\b([0-9]|[0-9]{2}|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.([0-9]|[0-9]{2}|1[0-9]{2}|2[0-4][0-9]|25[0-5])\b' test.txt
```

გამოვიტანოთ ის ხაზები, რომლების მხოლოდ ნამდვილ რიცხვებს შეიცავენ. მისაღებია შემდეგი ფორმატი: +0.45, -4.2, .9 (0.9-ს ექვივალენტად), ხოლო „9.“ არა. წერტილის შემდეგ ერთი ციფრი მაინც უნდა მოდიოდეს.

```
achiko@debian:~$ cat test.txt | grep -E '^(\\+|\\-)?[0-9]*\\. [0-9]+$'
```

grep ბრძანებას აქვს კიდევ ერთი სასარგებლო ოფცია **-e**. ის საშუალებას გვაძლევს რამდენიმე მოსაძებნი გამოსახულება გამოვიყენოთ ერთდროულად და მოვძებნოთ ხაზი, რომელშიც ერთ-ერთი გამოსახულება მაინც შედის. მოკლედ რომ ვთქვათ, „ლოგიკური ან“ გამოვიყენოთ.

```
achiko@debian:~$ grep -e pattern1 -e pattern2 -e pattern3 ...
```

ასეთ ამოცანას ვიცით კიდევ, როგორც უნდა მივუდგეთ. მაგალითად ასე:

```
achiko@debian:~$ cat test.txt | grep 'linux\\|Unix\\|Solaris'
```

თუმცა, **-e** ოფციის პირდაპირი დანიშნულება სწორედ ისაა, რომ მარტივი მოსაძებნი კრიტერიუმები სათითაოდ გადავცეთ **grep** ბრძანებას. ასე, ხილვადობასაც შევინარჩუნებთ.


```
achiko@debian:~$ cat test.txt | grep -e 'linux' -e 'Unix' -e 'Solaris'
```

რადგან „ლოგიკურ ან“ ვახსენეთ, ვთქვათ, როგორ ზდება „ლოგიკური და“ და „ლოგიკური არა“-ს ჩაწერა `grep`-ში. „ლოგიკური არა“ უკვე ავხსენით და ის, როგორც უკვე მიხვდით, `-v` ოფციით ზორციელდება, ხოლო „ლოგიკური და“-თვის `grep`-ში სპეციალური ოფცია არ გვაქვს. ჩვენ მხოლოდ მისი სიმულირება შეგვიძლია. ვთქვათ, გვსურს იმ ხაზების გამოტანა, რომელიც შეიცავს ერთდროულად როგორც `linux`-ს, ასევე `Unix`-ს.

ამ დროს ასე უნდა მოვიქცეთ:

```
achiko@debian:~$ cat test.txt | grep 'Linux.*Unix'
```

ამ ბრძანებით ეკრანზე გამოვა ის ხაზები, რომლებშიც შედის `Linux`, შემდეგ ნებისმიერი რაოდენობით გამეორებული ნებისმიერი ასო-ნიშანი და შემდეგ `Unix`. რა თქმა უნდა, შესაძლებელია, ხაზში ჯერ `Unix` შედიოდეს და შემდეგ `Linux`. ზოგადი ჩანაწერი ასეთი იქნება:

```
achiko@debian:~$ cat test.txt | grep -E 'Linux.*Unix|Unix.*Linux'
```

„ლოგიკური და“ სხვა მიდგომითაც შეგვიძლია განვახორციელოთ, `grep`-ის რამდენჯერმე გამოყენებით. ჩვენს მაგალითზე ეს ასე იქნება:

```
achiko@debian:~$ cat test.txt | grep Linux | grep Unix
```

`grep` გადაცემულ გამოსახულებას ეძებს ფაილში/ფაილებში ან მის სტანდარტულ შესასვლელზე გადაცემულ ტექსტში. შესაძლებელია, აგრეთვე, ძებნა ერთდროულად ორივეგან მოხდეს, როგორც სტანდარტულ შესასვლელზე (ის „-“თი აღინიშნება), ასევე არგუმენტად გადაცემულ ფაილებში, ასე:

```
achiko@debian:~$ cat file1 | grep 'pattern' - file2
```

`grep`-ის გაფართოებული რეგულარული გამოსახულების ზშირად გამოყენებიდან გამომდინარე, არსებობს `grep -E` ჩანაწერის ექვივალენტი ბრძანება `egrep`. ასევე, რეკურსიული ძებნისთვის არსებობს `rgrep` ბრძანება, რომელიც `grep -r` ჩანაწერის ექვივალენტურია. აგრეთვე, `grep -F` იგივეა, რაც `fgrep`.

`fgrep` ძალიან სასარგებლოა მაშინ, როდესაც ისეთი გამოსახულების მოძებნა გვსურს, რომელიც შეიცავს რეგულარული გამოსახულების მეტასიმბოლოებს. ეს ბრძანება უზრუნველყოფს სწორედ იმას, რომ ეს მეტასიმბოლოები აღქმულ იქნას მხოლოდ ჩვეულებრივ სიმბოლოებად და სხვა მნიშვნელობა აღარ მიენიჭოს.

```
achiko@debian:~$ cat test.txt | fgrep '(1|L)inux'
```

ასე მოვძებნით იმ ხაზს, რომელიც შეიცავს ასეთ სიმბოლოთა ერთიანობას - `(1|L)inux` და არა `Linux`-ს ან `linux`-ს.

13.2 sed

sed ბრძანება, **grep** ბრძანების მსგავსად, სრულად იყენებს რეგულარული გამოსახულების შესაძლებლობებს. ეს პროგრამა ერთ-ერთი გამორჩეული და მძლავრი აპარატია ტექსტების დამუშავების პროცესში. **sed** არის Stream EEditor-ის აკრონიმი და, როგორც წესი, ის GNU/Linux თითქმის ყველა სისტემაშია წარმოდგენილი. მისი სიტნაჟი, ერთი შეხედვით, შეიძლება ცოტა ჩახლართულად გვეჩვენოს, თუმცა კარგად გააზრების შემთხვევაში, **sed**-ის ბრძანებების მარტივი გამოყენებით საკმაოდ რთული ამოცანების გადაჭრაა შესაძლებელი. მისი შესაძლებლობები იმიტაც იზრდება, რომ მას რეგულარული გამოსახულების გაგება შეუძლია.

sed ბრძანების სინტაჟი ასეთია:

```
$ sed 'commands' file1 file2 ...
```

ეს ბრძანება სტანდარტულ შესასვლელზე იღებს არგუმენტად გადაცემული ფაილ(ებ)ის შიგთავსს, დაამუშავებს მათ ხაზ-ხაზ მოცემული ბრძანებებით (ეს ბრძანებები საკუთარ ინსტრუქციებს წარმოადგენს და არა შელის ბრძანებებს) და შემდეგ მიღებულ შედეგს გამოიტანს სტანდარტულ გამოსასვლელზე, ეკრანზე.

უფრო დეტალურად რომ აღვწეროთ, ეს ყველაფერი ასე ხდება:

1. **sed** კითხულობს შესასვლელის პირველ ხაზს და განათავსებს მას გამოსახულების ბუფერში. გამოსახულების ბუფერი არის **sed**-სთვის მეხსიერებაში გამოყოფილი ადგილი.
2. **sed** დაამუშავებს მოცემული ბრძანებებით გამოსახულების ბუფერში შენახულ მონაცემს.
3. **sed** გამოიტანს დამუშავებულ მონაცემებს გამოსასვლელზე და გაასუფთავებს გამოსახულების ბუფერს.
4. შემდეგ **sed** მეორე ხაზზე გადავა და იგივე პროცედურებს გაიმეორებს. ეს ციკლი გაგრძელდება მანამდე, სანამ ბოლო ხაზიც არ დამუშავდება.

რადგან **sed** ბრძანება ოპერირებს ბუფერში კოპირებულ მონაცემებზე, ეკრანზე ბუფერში არსებული შიგთავსის შეცვლილი ვერსია გამოდის, ხოლო ფაილის შიგთავსი უცვლელი რჩება.

მოდით, ავიღოთ ჩვენი სატესტო ფაილი და გცადოთ:

```
achiko@debian:~$ vi test.txt
Young Georgian Rugby Team wins European Championship by defeating France.
```

```
achiko@debian:~$ sed ' ' test.txt
Young Georgian Rugby Team wins European Championship by defeating France.
```

მოდით, გავაანალიზოთ რატომ მივიღეთ ასეთი შედეგი: **sed**-მა აიღო ხაზი, დააკოპირა ის გამოსახულების ბუფერში, რადგან ინსტრუქცია არ არის მითითებული, არ დაამუშავა ეს მონაცემები და პირდაპირ გამოიტანა ეკრანზე.

ახლა კი, განვიხილოთ **sed**-ის ბრძანებები, რათა უკვეთ გავიგოთ, თუ რა შეუძლია მას. ავიღოთ ერთ-ერთი სატესტო ფაილი **countries.txt**

```
achiko@debian:~$ cat countries.txt
1 Georgia
2 France
3 USA
4 Germany
5 Great Britain
```

13.2.1 წაშლა

ბრძანება `d` (delete) შლის სტრიქონებს გამოსახულების ბუფერში.

```
achiko@debian:~$ cat countries.txt | sed 'd'
achiko@debian:~$
```

არაფერი გამოვიდა ეკრანზე. ეს შედეგი ნორმალურია. ჩვენ ხომ მივუთითეთ, რომ ყველა ხაზი წაეშალა.

`sed`, როგორც ვთქვით, ყველა ხაზს ჩამოუვლის და დაამუშავებს. თუმცა, შესაძლებელია კონკრეტული ხაზების მისამართის მითითებაც. ხაზის მისამართი შეიძლება იყოს რიცხვი ან გამოსახულება. რიცხვის მითითებით, `sed` ამ რიცხვის შესაბამის ნომრიან ხაზს გამოიტანს. გამოსახულების მითითებით, (ის აუცილებლად „/“-ებში უნდა მოვათავსოთ) `sed` გამოიტანს ხაზს, რომელიც ამ გამოსახულებებს შეიცავს.

მისამართის მოცემის სხვადასხვა ფორმები არსებობს და ის ბრძანებებთან კომბინაციაში გამოიყენება. მაგალითად, ყველა ხაზის წაშლის მაგივრად, შეგვიძლია, მივუთითოთ თუ რომელი ხაზების წაშლა გვსურს.

ეს ბრძანება შლის მე-3 ხაზს:

```
achiko@debian:~$ sed '3d' countries.txt
1 Georgia
2 France
4 Germany
5 Great Britain
```

ასე ვეუბნებით, რომ წაშალოს მე-2-დან მე-4-ს ჩათვლით ხაზები:

```
achiko@debian:~$ sed '2,4d' countries.txt
1 Georgia
5 Great Britain
```

ვეუბნებით, რომ წაშალოს მე-2-დან მომდევნო 3 ხაზი:

```
achiko@debian:~$ sed '2,+3d' countries.txt
1 Georgia
```

წაშალოს პირველიდან დაწყებული ყოველი მე-2 ხაზი:

```
achiko@debian:~$ sed '1~2d' countries.txt
2 France
4 Germany
```

ჩამაღოს ხაზი, რომელიც **erm** გამოსახულებას შეიცავს:

```
achiko@debian:~$ sed '/erm/d' countries.txt
1 Georgia
2 France
3 USA
5 Great Britain
```

ჩამაღოს ყველა ხაზი, იმ ხაზიდან დაწყებული, რომელიც **ran** გამოსახულებას შეიცავს, იმ ხაზით დამთავრებული, რომელიც **erm** გამოსახულებას შეიცავს:

```
achiko@debian:~$ sed '/ran/,/erm/d' countries.txt
1 Georgia
5 Great Britain
```

ჩამაღოს ხაზი, რომელიც **erm** გამოსახულებას შეიცავს და კიდევ შემდეგი 2 ხაზი:

```
achiko@debian:~$ sed '/erm/,+2d' countries.txt
1 Georgia
2 France
3 USA
```

გამოსახულებებში თავისუფლად შეგვიძლია, ჩვენთვის უკვე ნაცნობი რეგულარული გამოსახულება გამოვიყენოთ.

13.2.2 ჩანაცვლება

ბრძანება **s** (substitute) ერთ გამოსახულებას მეორეთი ჩანაცვლებს. შესაცვლელი გამოსახულება შესაძლებელია, რეგულარული გამოსახულებით აღვწეროთ.

ეს ბრძანება ყოველ ხაზში პირველ შემხვედრ **a**-ს ჩანაცვლებს **XYZ**-ით:

```
achiko@debian:~$ sed 's/a/XYZ/' countries.txt
1 GeorgiXYZ
2 FrXYZnce
3 USA
4 GermXYZny
5 GreXYZt Britain
```

აქ, ხაზში მხოლოდ მე-2 შემხვედრი **a** ჩანაცვლდება **XYZ**-ით:

```
achiko@debian:~$ sed 's/a/XYZ/2' countries.txt
1 Georgia
2 France
3 USA
4 Germany
5 Great BritXYZin
```

ასე, ხაზში ყველა შემხვედრი a შეიცვლება XYZ-ით:

```
achiko@debian:~$ sed 's/a/XYZ/g' countries.txt
1 GeorgiXYZ
2 FrXYZnce
3 USA
4 GermXYZny
5 GreXYZt BritXYZin
```

ასე კი, ხაზში მე-3 შემხვედრი a-დან დაწყებული ყველა a შეიცვლება XYZ-ით. უბრალოდ, ჩვენს ფაილში ასეთი სამჯერ ან მეტჯერ გამოყენებული a ასო-ნიშანი ერთ ხაზში არ გვაქვს.

```
achiko@debian:~$ sed 's/a/XYZ/3g' countries.txt
1 Georgia
2 France
3 USA
4 Germany
5 Great Britain
```

ამგვარად, ყველა a (დიდი თუ პატარა ასო) შეიცვლება XYZ-ით. I, ისევე როგორც i, ამ შემთხვევაში, უზრუნველყოფს სწორედ დიდი და პატარა ასოების უგულებელყოფას.

```
achiko@debian:~$ sed 's/a/XYZ/Ig' countries.txt
1 GeorgiXYZ
2 FrXYZnce
3 USXYZ
4 GermXYZny
5 GreXYZt BritXYZin
```

ასე, ყოველი ხაზის ბოლოში XYZ-ს ვამატებთ:

```
achiko@debian:~$ sed 's/$/XYZ/' countries.txt
1 GeorgiaXYZ
2 FranceXYZ
3 USAXYZ
4 GermanyXYZ
5 Great BritainXYZ
```

ასე, ყოველი ხაზის პირველ 2 სიმბოლოს ვშლით. (ვცვლით არაფრით)

```
achiko@debian:~$ sed 's/^..//' countries.txt
XYZGeorgia
XYZFrance
XYZUSA
XYZGermany
XYZGreat Britain
```

გამოსახულების ჩანაცვლებისას, ამოცანიდან გამომდინარე, ზშირად სასარგებლოა მთლიანი ხაზი რეგულარული გამოსახულებით აღვწეროთ. ჩვენს შემთხვევაში ასეთი სურათი გვაქვს. პირველ სვეტში გვაქვს რიცხვი მეორეში კი ქვეყნის დასახელება. რეგულარული გამოსახულებით ასეთი ხაზი ასე აღიწერება: „[0-9][0-9]* . . *“ - ციფრს მინიმუმ ერთხელ მაინც ვიმეორებთ (* სიმბოლო ზომ 0-ჯერ გამეორებასაც ნიშნავს. ამიტომ დავწერეთ ორჯერ ციფრი). შემდეგ გამოტოვების ერთი სიმბოლო მოდის (სვეტებს შორის გამოყოფი სიმბოლო). ბოლოს კი ნებისმიერი სიმბოლო გამეორება მინიმუმ ერთხელ.

შემდეგ კი, საკმარისია, ამ გამოსახულების ნებისმიერი მონაკვეთი „\ (\)“ სტრუქტურაში მოვაქციოთ, რომ ეს მონაკვეთი sed-ის ცვლადში, „\1“-ში დამახსოვრდება. თუ სხვა მონაკვეთსაც ჩავსვამთ იგივე სტრუქტურაში, მაშინ მისი შიგთავსი „\2“-ში ჩაიწერება და ა.შ.

ამ ბრძანებით გამოვიტანთ მხოლოდ ქვეყნის დასახელებებს:

```
achiko@debian:~$ sed 's/[0-9][0-9]* \(..*\)/\1/' countries.txt
Georgia
France
USA
Germany
Great Britain
```

ასე კი, ჯერ ქვეყნის დასახელებას (ის უკვე \2 ცვლადშია) გამოვიტანთ, ხოლო შემდეგ ნუმერაციას (\1). შუაში კი ტაბულაცია გვაქვს ჩასმული (\t):

```
achiko@debian:~$ sed 's/\([0-9][0-9]*\) \(..*\)/\2\t\1/' countries.txt
Georgia 1
France 2
USA 3
Germany 4
Great Britain 5
```

ხაზის აღწერისას „*“-ის ნაცვლად, გაფართოებულ რეგულარული გამოსახულების ერთ-ერთი მეტასიმბოლო „+“ (ერთხელ ან ბევრჯერ გამეორება) შეგვიძლია, გამოვიყენოთ. გაფართოებული რეგულარული გამოსახულების მეტასიმბოლოების დროს sed ბრძანებას აუცილებლად უნდა მივუთითოთ -r ოფცია. ამ დროს ქვესტრუქტურის განსაზღვრისას, „()“ უნდა გამოვიყენოთ „\ (\)“-ს ნაცვლად.

ჩინა ბრძანება უკვე ასეთ სახეს მიიღებს უკვე:

```
achiko@debian:~$ sed -r 's/([0-9]+) (.+)/\2\t\1/' countries.txt
```

გამოსახულების შეცვლისას, **sed**-ის ცვლადების გარდა, შეგვიძლია, გამოვიყენოთ შემდეგი სპეციალური სიმბოლოებიც \l, \L, \u, \U და &.

- \l გამოსახულების პირველ ასო-ნიშანს პატარა ასოდ გადააკეთებს.
- \L გამოსახულების ყველა ასო-ნიშანს პატარა ასოდ გადააკეთებს.
- \u გამოსახულების პირველ ასო-ნიშანს დიდ ასოდ გადააკეთებს.
- \U გამოსახულების ყველა ასო-ნიშანს დიდ ასოდ გადააკეთებს.
- & ეს სიმბოლო **sed**-ში თვითონ შესაცვლელი გამოსახულების მოკლე აღნიშვნაა.

ამ ბრძანებით ყველა ხაზის პატარა ასოებს გადავაკეთებთ დიდ ასოებად:

```
achiko@debian:~$ sed 's/.*/\U&/' countries.txt
1 GEORGIA
2 FRANCE
3 USA
4 GERMANY
5 GREAT BRITAIN
```

ასე, ქვეყნების სახელების პირველი ასო გამოვა პატარა ასოებით:

```
achiko@debian:~$ sed 's/ \(.\) / \l\1/' countries.txt
1 georgia
2 france
3 uSA
4 germany
5 great Britain
```

ასე, მე-2 და დამატებით შემდეგ 2 ხაზზე მთლიან გამოსახულებას ჩავსვამთ ფრჩხილებში. დანარჩენ ხაზებს არ ვეხებით.

```
achiko@debian:~$ sed '2,+2s/.*/(&)/' countries.txt
1 Georgia
(2 France)
(3 USA)
(4 Germany)
5 Great Britain
```

s ბრძანებით შეცვლისას s/x/y/ ჩანაწერში, შეთანხმებისამებრ, „/“ სიმბოლოს იყენებენ ზოლმე. თუმცა, მის ნაცვლად ნებისმიერი სიმბოლოს გამოყენებაც შესაძლებელია. ყველა ეს ჩანაწერი ერთმანეთის ექვივალენტურია:

```
achiko@debian:~$ echo "xyz" | sed 's/x/A/'
Ayz
achiko@debian:~$ echo "xyz" | sed 's|x|A|'
```

```
Ayz
achiko@debian:~$ echo "xyz" | sed 's!x!A!'
Ayz
achiko@debian:~$ echo "xyz" | sed 's:x:A:'
Ayz
achiko@debian:~$ echo "xyz" | sed 'sTxTAT'
Ayz
achiko@debian:~$ echo "xyz" | sed 'sbxbAb'
Ayz
```

ასეთი ხერხი საშუალებას გვაძლევს, გამოსახულებაში არსებული სიმბოლოების გარდა, ჩანაცვლების ბრძანებაში სხვა სიმბოლო გამოვიყენოთ. შესაცვლელი გამოსახულება თუ ბევრ „/“-ს შეიცავს, უმჯობესია, მოვერიდოთ „/“-ის გამოყენებას და მის ნაცვლად, მაგალითად, „@“ სიმბოლო გამოვიყენოთ. ასე, მაგალითად:

```
achiko@debian:~$ echo $PATH | sed 's@/usr/local2/bin@/opt/local/bin@'
```

თუ მაინცდამაინც „/“-ის გამოყენება გვსურს, მაშინ მას წინ „\“ უნდა მივუწეროთ. ასე:

```
achiko@debian:~$ echo $PATH | sed 's\/usr\/local2\/bin\/opt\/local\/bin\/'
```

13.2.3 დაბეჭდვა

p (print) ბრძანება დაბეჭდვას, ეკრანზე გამოტანას ნიშნავს. ვცადოთ:

```
achiko@debian:~$ echo "Georgia " | sed 'p'
Georgia
Georgia
```

ეკრანზე მეორეჯერაც გამოვიდა იგივე ხაზი.

ეს შედეგი ნორმალურია, რადგან sed-ს, ნაგულისხმევი მნიშვნელობით, ისედაც გამოაქვს გამოსახულების ბუფერში გადატანილი ხაზი. იმისათვის, რომ ხაზის ავტომატური გამოტანა გამოვრთოთ, -n (--quiet, --silent) ოფცია უნდა გამოვიყენოთ. ამ შემთხვევაში, p ბრძანება მხოლოდ იმას გამოიტანს, ჩვენ რასაც ვეტყვი.

ამ ბრძანებით 2-დან მე-4-ს ჩათვლით ხაზებს გამოიტანს მხოლოდ:

```
achiko@debian:~$ sed -n '2,4p' countries.txt
2 France
3 USA
4 Germany
```

ასე, მხოლოდ ის ხაზი გამოვა, რომელშიც შედის Geo გამოსახულება. ეს ბრძანება grep-ის ექვივალენტურია.


```
achiko@debian:~$ sed -n '/Geo/p' countries.txt
1 Georgia
```

„!“-ით შეგვიძლია „ლოგიკური არა“ გამოვიყენოთ. ამ ბრძანებით ის ხაზები გამოვა, რომელშიც არ შედის Geo გამოსახულება.

```
achiko@debian:~$ sed -n '/Geo/!p' countries.txt
2 France
3 USA
4 Germany
5 Great Britain
```

ასე კი, მე-2 და დამატებით შემდეგი 2 ხაზის გარდა, დანარჩენ ხაზებს ჩავსვამთ ფრჩხილებში.

```
achiko@debian:~$ sed '2,+2!s/.*/(&)/' countries.txt
(1 Georgia)
2 France
3 USA
4 Germany
(5 Great Britain)
```

13.2.4 გამოსვლა

q (quit) ბრძანება გამოსვლას ნიშნავს.

ამ ბრძანებით ეკრანზე პირველი 3 ხაზი გამოვა და დასრულდება პროცესი. ეს ბრძანება head ბრძანების ექვივალენტურია.

```
achiko@debian:~$ sed '3q' countries.txt
1 Georgia
2 France
3 USA
```

ასე კი, ეკრანზე გამოვა საწყისი სტრიქონები Fra გამოსახულების შემცველ პირველ ხაზიანად.

```
achiko@debian:~$ sed '/Fra/q' countries.txt
1 Georgia
2 France
```

13.2.5 გარდაქმნა

y (transform) ბრძანება, tr ბრძანების მსგავსად, თითოეული ასო-ნიშნის გარდაქმნას, გადაკეთებას ახორციელებს.

ამ ბრძანებით a შეიცვლება X-ით, e Y-ით, i კი Z-ით.

```
achiko@debian:~$ sed 'y/aei/XYZ/' countries.txt
GYorgZX
2 FrXncY
3 USA
4 GYrmXny
5 GrYXt BrZtXZn
```

13.2.6 საზის ნომერი

„=“ სიმბოლო ბრძანებას წარმოადგენს sed-ში და მას დასამუშავებელი ხაზების ნომრები გამოაქვს.

```
achiko@debian:~$ sed '=' countries.txt
1
1 Georgia
2
2 France
3
...
```

ეს ბრძანება გამოიტანს საზის ნომერს. შემდეგ კი, ნაგულისხმევები მნიშვნელობით, მთლიან საზს. იმისათვის, რომ ეკრანზე მთლიანი საზი არ გამოვიდეს, სასურველი იქნება -n ოფციის გამოყენება, ასე:

```
achiko@debian:~$ sed -n '=' countries.txt
1
2
3
4
5
```

ეს ბრძანება გამოიტანს იმ საზის ნომერს, რომელშიც შედის Geo გამოსახულება.

```
achiko@debian:~$ sed -n '/Geo/=' countries.txt
1
```

ამ ბრძანებით, აგრეთვე, შეგვიძლია ფაილში ხაზების რაოდენობა დავთვალოთ. ამისთვის, საკმარისია, ბოლო საზის ნომერი გამოვიტანოთ, ასე:

```
achiko@debian:~$ sed -n '$=' countries.txt
5
```

13.2.7 შეცვლა

c (change) ბრძანება ხაზებს ცვლის მოცემული ტექსტით:

```
achiko@debian:~$ sed '1,4c text' countries.txt
text
5 Great Britain
```

ამგვარად, პირველი, მეორე, მესამე და მეოთხე ხაზი, მთლიანად შეიცვალა მოცემული ტექსტით (text). თუ გვსურს, რომ ხაზები ისეთი ტექსტით შევცვალოთ, რომელიც, თავის მხრივ, შეიცავს ახალ ხაზზე გადასვლის სიმბოლო(ებ)ს (ანუ ტექსტი მრავალხაზიანია), მაშინ „\“ უნდა გამოვიყენოთ ხაზების ბოლოს (ბოლო ხაზის გარდა). ასე:

```
achiko@debian:~$ cat countries.txt | sed '1,4c text1\
> test2\
> text3
> '
text1
test2
text3
5 Great Britain
```

13.2.8 ჩამატება

ბრძანებები a (append) და i (insert) ტექსტის ჩამატებას უზრუნველყოფს მოცემული ხაზის შემდეგ და წინ.

ამ ბრძანებით პირველი ხაზის დასაწყისში ჩავსვამთ გამოსახულებას START-ს.

```
achiko@debian:~$ sed '1i START' countries.txt
START
1 Georgia
2 France
3 USA
4 Germany
5 Great Britain
```

ასე კი, ბოლო ხაზის შემდეგ ჩავსვამთ END-ს

```
achiko@debian:~$ sed '$a END' countries.txt
1 Georgia
2 France
3 USA
4 Germany
5 Great Britain
END
```

-e ოფცია რამდენიმე ინსტრუქციის გადაცემის საშუალებას იძლევა ერთბაშად. მოდით, START და END ერთი ბრძანებით ჩავამატოთ და თან „გავალამაზოთ“:

```
achiko@debian:~$ sed -e '1i\\nSTART\\n' -e '$a\\nEND\\n' countries.txt
```

```
START
```

```
1 Georgia
2 France
3 USA
4 Germany
5 Great Britain
```

```
END
```

13.2.9 ფაილის კითხვა და ფაილში ჩაწერა

r ბრძანებით შესაძლებელია მოცემულ მისამართზე (ხაზზე) სხვა ფაილის შიგთავსი ჩავსვათ.

```
achiko@debian:~$ sed '$r distros.txt' countries.txt
```

```
...
4 Germany
5 Great Britain
Debian 9.4 10/03/2018
Debian 8.10 09/12/2017
...
```

ეს ბრძანება `countries.txt` ფაილის ბოლოში `distros.txt` ფაილის შიგთავსს ჩასვამს. **r** ბრძანების წინ მისამართი (ჩვენს შემთხვევაში, ფაილი ბოლო \$) რომ არ მიგვეითებინა, მაშინ ყოველი ხაზის შემდეგ ჩაჯდება `distros.txt`-ს შიგთავსი.

როგორც ვიცით, **sed** ბრძანება სინამდვილეში ფაილის შიგთავსს არ ეხება. ის უცვლელი რჩება. **-i** ოფციით კი შესაძლებელია ვუთხარტ **sed**-ს, რომ ცვლილების დაფიქსირება ეკრანზე კი არა, პირდაპირ ფაილში გვსურს.

```
achiko@debian:~$ sed -i 's/USA/Italy/' countries.txt
```

ასე, ცვლილებას პირდაპირ `countries.txt` ფაილში შევიტანთ.

-i ოფციას შესაძლებელია, აგრეთვე, სუფიქსი მივუწეროთ. ეს საშუალებას მოგვცემს ორიგინალი ფაილის შიგთავსი შევინახოთ ფაილის ახალი დასახელებით, შემდეგნაირად:

```
achiko@debian:~$ sed -i.bak 's/USA/Italy/' countries.txt
```

ასე, ორიგინალი შიგთავსი შეინახება `countries.txt.bak` ფაილში, ხოლო ცვლილება კი დაფიქსირდება `countries.txt`-ში.

თუ გვსურს, რომ ეკრანზე გამოსული შეცვლილი შიგთავსი, ამავედროულად, სხვა დასახელების ფაილშიც შევინახოთ, ხოლო ორიგინალ ფაილს არ შევეხოთ, ამაში **w** ბრძანება დაგვეხმარება. ასე:

```
achiko@debian:~$ sed -e 's/Great\Britain/Canada/' -e 'w newfile.txt'
countries.txt
1 Georgia
2 France
3 USA
4 Germany
5 Canada
```

13.2.10 დაჯგუფება

რამდენიმე ინსტრუქციის გადაცემა -e ოფციის გარეშე, „;“ სიმბოლოთიც არის შესაძლებელი. ასე:

```
achiko@debian:~$ sed 's/Great\Britain/Canada/; w newfile.txt' countries.txt
1 Georgia
2 France
3 USA
4 Germany
5 Canada
```

წერტილ-მძიმით, ძირითადად, მარტივ ბრძანებებს აცალკევებენ ხოლმე. ასე, მაგალითად:

```
achiko@debian:~$ sed '1d;3d;5d' countries.txt
2 France
4 Germany
```

ასე, ერთდროულად წავშლით პირველ, მესამე და მეხუთე ხაზს.

ფიგურული ფრჩხილებით „{ }“ შეგვიძლია დაჯგუფება მოვახდინოთ. მაგალითად, თუ გვსურს, რომ წავშალოთ ბოლო ხაზი, თუ ის შეიცავს Bri გამოსახულებას, მაშინ დაჯგუფება დაგვჭირდება.

```
achiko@debian:~$ sed '${/Bri/d}' countries.txt
1 Georgia
2 France
3 USA
4 Germany
```

ფიგურული ფრჩხილების გარეშე კი სინტაქსური შეცდომა იქნება:

```
achiko@debian:~$ sed '$/Bri/d' countries.txt
sed: -e expression #1, char 2: unknown command: `/'
```

13.2.11 შელის ბრძანების შესრულება

ე ბრძანებით, მილის საშუალებით, შესაძლებელია, შელს გამოსახულების ბუფერში

ჩაწერილი ბრძანებები გადაეცეს. ამისათვის **e** ბრძანებას არგუმენტი არ უნდა გადავცეთ. ზოლო, თუ **e**-ს არგუმენტი უწერია, ისიც აუცილებლად შელის ბრძანება უნდა იყოს. მოდით, ავიღოთ ერთი ფაილი და მასში ხაზ-ხაზ შელის ბრძანებები ჩავწეროთ:

```
achiko@debian:~$ cat cmd.txt
echo "Hello Georgia"
date
whoami
```

```
achiko@debian:~$ sed 'e' cmd.txt
Hello Georgia
Tue Jul 10 15:56:32 +04 2018
achiko
```

ამ ბრძანებაში **e** ბრძანებას არგუმენტი არ აქვს და შესაბამისად **cmd.txt** ფაილში ჩაწერილი ბრძანებები შესრულდება **sed**-ის მიერ.

```
achiko@debian:~$ sed 'e echo TEST' cmd.txt
TEST
echo "Hello Georgia"
TEST
date
TEST
whoami
```

ასე კი, **e** ბრძანებას არგუმენტად შელის ბრძანებას გადავცემთ. შესაბამისად, **cmd.txt** ფაილში ხაზ-ხაზ მოცემული ბრძანებები, ამ შემთხვევაში, მხოლოდ და მხოლოდ, მონაცემებს წარმოადგენს **e** ბრძანებისთვის.

მხოლოდ პირველი ხაზის წინ რომ ჩავსვათ შელის რამე ბრძანებ(ებ)ის შედეგი, მაშინ მისამართი უნდა მივუთითოთ.

```
achiko@debian:~$ sed "1e echo USER $USER read the file; date" cmd.txt
USER achiko read the file
Thu Jul 12 15:09:02 +04 2018
echo "Hello Georgia"
date
whoami
```

13.2.12 სარეზერვო ბუფერი

აქამდე რა მაგალითებიც მოვიყვანეთ, მათში მხოლოდ გამოსახულების ბუფერში არსებულ მონაცემებს ვიყენებდით დასამუშავებლად. გამოსახულების ბუფერის გარდა, **sed**-ს გააჩნია სარეზერვო ბუფერიც. შესაძლებელია, გამოსახულების ბუფერიდან მოხდეს მონაცემების გადატანა სარეზერვო ბუფერში მომავალში მისი დამუშავების მიზნით. ყოველი ციკლის შემდეგ გამოსახულების ბუფერი იშლება, თუმცა სარეზერვო ბუფერში მონაცემი რჩება. სარეზერვო ბუფერზე ბრძანებებით მოქმედება **sed**-ს პირდაპირ არ შეუძლია, მაგრამ

შესაძლებელია მონაცემების მოძრაობა სარეზერვო ბუფერიდან გამოსახულების ბუფერში და პირიქით. თავდაპირველად ორივე ბუფერი ცარიელია.

მოდი, სარეზერვო ბუფერიც ჩავრთოთ საქმეში და ვნახოთ, რამდენად გაზრდის ის sed-ის შესაძლებლობებს.

x (exchange)	გამოსახულების ბუფერისა და სარეზერვო ბუფერის შიგთავსებს გადანაცვლებს.
h (hold)	გამოსახულების ბუფერის შიგთავსს აკოპირებს სარეზერვო ბუფერში. ამ უკანასკნელში არსებული მონაცემი იშლება.
H (Hold)	ეს ბრძანება კი გამოსახულების ბუფერის შიგთავსს ბოლოში დაამატებს სარეზერვო ბუფერს. დამატებული შიგთავსი ახალი ხაზიდან ჩაიწერება.
g (get)	აკოპირებს სარეზერვო ბუფერის შიგთავსს გამოსახულების ბუფერში. ასე ეს უკანასკნელი იშლება.
G (Get)	ეს ბრძანება კი სარეზერვო ბუფერის შიგთავსს ბოლოში დაუმატებს გამოსახულების ბუფერს, ახალი ხაზიდან.

მათი გამოყენების მაგალითები:

```
achiko@debian:~$ cat countries.txt | sed 'G'
1 Georgia

2 France

3 USA

4 Germany

5 Great Britain
```

G ბრძენებით გამოსახულების ბუფერს ბოლოში, ახალ ხაზზე, დაემატა სარეზერვო ბუფერის შიგთავსი. ახლა ხაზზე, როგორც გამოჩნდა, არაფერი ჩაიწერა, რადგან სარეზერვო ბუფერი ცარიელია. შესაბამისად, ეს ბრძანება თითოეული ხაზის შემდეგ ერთ ცარიელ ხაზს ჩასვამს.

თუ გვსურს დავუმატოთ ორი ახალი ხაზი, მაშინ ორჯერ მივუთითებთ ამ ბრძანებას:

```
achiko@debian:~$ cat countries.txt | sed 'G;G'
1 Georgia

2 France

...
```

თითოეულ ხაზის წინ რომ ჩავსვათ ერთი ახალი ხაზი, ასე უნდა მოვიქცეთ:

```
achiko@debian:~$ cat countries.txt | sed 'x;p;x'
```

```
1 Georgia
2 France
3 USA
4 Germany
5 Great Britain
```

ორი ხაზის ჩასახმელად კი ასე:

```
achiko@debian:~$ cat countries.txt | sed 'x;p;p;x'
```

```
1 Georgia

2 France

...
```

მოდით, ახალი ხაზი ჩავსვათ მხოლოდ იმ ხაზამდე და მის შემდეგაც, რომელიც შეიცავს რაიმე გამოსახულებას. ჩვენს მაგალითში ავიღოთ გამოსახულება USA:

```
achiko@debian:~$ cat countries.txt | sed '/USA/x;p;x;G'
```

```
1 Georgia
2 France

3 USA

4 Germany
5 Great Britain
```

ამოვატრიალოთ ხაზების თანმიმდევრობა. sed-ით შევქმნათ იგივე tac ბრძანება.

```
achiko@debian:~$ cat countries.txt | sed -n '1!G;h;$p'
```

```
5 Great Britain
4 Germany
3 USA
2 France
1 Georgia
```


ავსხნათ, რა გავაკეთეთ: პირველ ხაზზე მხოლოდ **h** ბრძანებით ვმოქმედებთ, ანუ ხაზს ვაკოპირებთ სარეზერვო ბუფერში. **G** ბრძანება გამოვტოვეთ, რადგან **1!G** მიუთითებს, რომ მხოლოდ პირველი ხაზზე არ შესრულდეს ეს ბრძანება. მეორე ხაზიდან კი **G;h** ბრძანებები სრულდება. ანუ, ჯერ ვუმატებთ მეორე ხაზს სარეზერვო ბუფერს ბოლოში ახალ ხაზიდან (იქ უკვე გვაქვს პირველი ხაზი დამახსოვრებული), შემდეგ კი ვაკოპირებთ მას უკან, გამოსახულების ბუფერში. ასე გრძელდება ბოლომდე. **\$p** უზრუნველყოფს მხოლოდ ბოლო ხაზის დამუშავების შედეგის გამოტანას ეკრანზე.

\$-ის გარეშე პროცესის სრულ ციკლს დავინახავთ. ასე:

```
achiko@debian:~$ cat countries.txt | sed -n '1!G;h;p'
1 Georgia
2 France
1 Georgia
3 USA
2 France
1 Georgia
4 Germany
3 USA
2 France
1 Georgia
5 Great Britain
4 Germany
3 USA
2 France
1 Georgia
```

მოლით, რამდენიმე ბრძანება კიდევ განვიხილოთ.

ბრძანება **n** კითხულობს მიმდინარე ხაზს, გამოაქვს და ჩაანაცვლებს გამოსახულების ბუფერში არსებულ მონაცემს შემდეგი ხაზით. **N** ბრძანება კი გამოსახულების ბუფერს დაუმატებს შემდეგ ხაზს ახალ ხაზიდან.

D ბრძანება, **d**-სგან განსხვავებით, გამოსახულების ბუფერს არა მთლიანად, არამედ პირველ შემხვედრ ახალ ხაზამდე შლის, ანუ მთლიანი ხაზის პირველ პორციას.

ასეთივე განსხვავებაა **p**-სა და **P** ბრძანებებს შორის. **P**-თი, გამოდის გამოსახულების ბუფერის პირველი პორცია (მხოლოდ პირველ შემხვედრ ახალ ხაზმდე).

sed ბრძანებაში ნაკადების მართვაც არის შესაძლებელი. ამისთვის ჭდე უნდა გამოვიყენოთ. ასეთი შესაძლებლობებით, ფაქტობრივად, **sed** მინი დაპროგრამების ენას წარმოადგენს. ჭდეს დასმა „:“ ბრძანებით ხდება, რომელსაც უნდა მოსდევდეს ჭდის სახელი. ამ ჭდეზე დაბრუნება კი **b** ბრძანებით ხორციელდება. მასაც ჭდის სახელი უნდა მივუთითოთ.

მოლით, კვლავ მაგალითებს მივმართოთ - მოვნახოთ ხაზი, რომელიც შეიცავს, მაგალითად, **USA**-ს და წავშალოთ მისი მომდევნო ხაზი:

```
achiko@debian:~$ sed '/USA/{N;s/\n.*/g}' countries.txt
1 Georgia
2 France
3 USA
5 Great Britain
```

მოდით, მისი შემდეგი ორი ხაზი წავშალოთ:

```
achiko@debian:~$ sed '/USA/{N;N;s/\n.*//g}' countries.txt
1 Georgia
2 France
3 USA
```

ახლა ფაილში ყოველი ახალ ხაზზე გადასვლის სიმბოლო შევცვალოთ უბრალო გამოტოვებით. ერთი შეხედვით, ამოხსნა ასეთი მარტივი ჩანაწერი უნდა იყოს.

```
achiko@debian:~$ cat countries.txt | sed 's/\n/ /'
1 Georgia
2 France
3 USA
4 Germany
5 Great Britain
```

თუმცა შედეგს, როგორც ვხედავთ, ვერ ვაღწევთ. მოდით, დავფიქრდეთ რატომ? გავიხსენოთ, თუ როგორ მუშაობს `sed`. ის იღებს ხაზს, განათავსებს მას გამოსახულების ბუფერში და დამუშავება სწორედ გამოსახულების ბუფერში არსებულ მონაცემზე ხდება. იქ კი, ცხადია, ახალ ხაზზე გადასვლის სიმბოლო არ გვექნება, თუ `N` ბრძანება (ან მსგავსი - მაგალითად `H`) არ გამოვიყენეთ. ამიტომ, შეგვიძლია, ასე მოვიქცეთ - გამოსახულების ბუფერში არსებულ მონაცემს ახალ ხაზზე დავუმატოთ შემდეგი ხაზი `N` ბრძანებით (შედეგად, ახალ ხაზზე გადასვლის სიმბოლო ბუფერში იარსებებს), ყოველი შემდეგი ხაზის დასამატებლად გამოვიყენოთ ჭდე და ბოლოს უკვე, შეგვეძლება `\n` შევცვალოთ გამოტოვებით.

```
achiko@debian:~$ cat countries.txt | sed ': mishka; N; $!b mishka; s/\n/ /g'
1 Georgia 2 France 3 USA 4 Germany 5 Great Britain
```

ავხსნათ დეტალურად:

- ა) ექმნით ჭდეს სახელად `mishka`.
- ბ) მიმდინარე ხაზს გამოსახულების ბუფერში ვამატებთ შემდეგ ხაზს `N`-ით.
- გ) ვუბრუნდებით ჭდეს და ვმოქმედებთ `N`-ით მანამ, სანამ ბოლო ხაზს არ მივაღწევთ (`$!b mishka` ნიშნავს, რომ ბოლო ხაზზე ჭდეზე აღარ გადავიდეთ და, შესაბამისად, `N` აღარ გამოვიყენოთ).
- დ) ბოლოს, `s` ბრძანებით გვცვლით ახალ ხაზზე გადასვლის სიმბოლოს - `\n`-ს - გამოტოვებით.

თუ `sed` მიჯრით შეიცავს ბევრ ბრძანებას, მაშინ, უკეთესი კითხვადობისთვის, `sed`-ის ეს ინსტრუქციები შეგვიძლია ხაზ-ხაზ ახლა ფაილში განვათავსოთ და შემდეგ ეს ინსტრუქციები შესასრულებლად `sed` ბრძანებას `-f` ოფციის საშუალებით ასე გადავცეთ:

```
achiko@debian:~$ sed -f cmd.sed file.txt
```

მაგალითად, ჩვენი ბოლო ბრძანება ამ ხერხით ასე შესრულდება.

ჯერ შევიტანოთ ახალ ფაილში `cmd.sed`-ში ინსტრუქციები. ასეთ ფაილში #-ით დაწყებული ხაზი მხედველობაში არ მიიღება და ის კომენტარების ჩაწერის საშუალებას გვაძლევს დოკუმენტაციისთვის.

```
achiko@debian:~$ vi cmd.sed
# ვქმნით ჭდეს სახელად mishka
: mishka
# გამოსახულების ბუფერს ვუმატებთ შემდეგ ხაზს
N
# ვუბრუნდებით ჭდეს სანამ ბოლო ხაზს არ მივაღწევთ
$!b mishka
# ვცვლით ახალ ხაზზე გადასვლის სიმბოლოს გამოტოვებით
s/\n/ /g
achiko@debian:~$ sed -f cmd.sed countries.txt
1 Georgia 2 France 3 USA 4 Germany 5 Great Britain
```

`sed`-ს კიდევ ბევრი შესაძლებლობა აქვს. მათი სრული ჩამონათვალის სანახავად GNU `sed`-ის დოკუმენტაციას მიმართეთ: <https://www.gnu.org/software/sed/manual/sed.html>

ამ თავის ბოლოს, `sed` ბრძანების საშუალებით ვნახოთ ამოცანის გადაწყვეტის რამდენიმე „ეგზოტიკური“ ხერხი.

შემოვატრიალოთ ფრაზა - `sed`-ით განვახორციელოთ იგივე `rev` ბრძანება:

```
achiko@debian:~$ echo Mishka | rev
akhsiM
achiko@debian:~$ echo Mishka | sed '/\n/!G;s/\(.\)\(.*\n\)/&2\1/;/D;s/.//\'
akhsiM
```

მოვახდინოთ ტექსტის სწორება მარჯვენა კიდეზე:

```
achiko@debian:~$ x=${$(tput cols)-1}
achiko@debian:~$ sed -e ':a' -e "s/^\.{1,$x\}$/ &/" -e 'ta' countries.txt
1 Georgia
2 France
3 USA
4 Germany
5 Great Britain
```

`t` ბრძანება `x` ბრძანების მსგავსია, ოღონდ, თუ `x`-თი უპირობოდ გადავდივართ ჭდეზე, `t`-თი მხოლოდ იმ შემთხვევაში გადავდივართ ჭდეზე, თუ მის წინ მყოფი ჩანაცვლების `s` ბრძანება წარმატებით დასრულდა (წარმატებით დასრულება ნიშნავს, რომ ერთი გამოსახულება შეიცვალა მეორეთი)

თავი 14

არქივი, შეკუმშვა

კომპიუტერზე მუშაობისას ერთ-ერთი მთავარი ამოცანაა სისტემაში არსებული მონაცემების მთლიანობის, ინტეგრულობის უზრუნველყოფა. ერთ-ერთი გზა, რომელსაც სისტემის ადმინისტრატორი მიმართავს ამ ამოცანის გადასაწყვეტად არის სისტემის ფაილების დროული სარეზერვო ასლების გაკეთება. მაშინაც კი, თუ თქვენ არ ხართ სისტემის ადმინისტრატორი, სასარგებლოა, თქვენი ფაილების დიდი კოლექციის ასლები გადაიღოთ ერთი ადგილიდან მეორეში ან ერთი მოწყობილობიდან მეორეზე გადაიტანოთ.

ასლების გაკეთება, მეორეს მხრივ, მოითხოვს გქონდეთ საკმარისი თავისუფალი ადგილი შესანახ მოწყობილობაზე.

კომპიუტერის გამოჩენიდან დღემდე მუდამ დგას საკითხი, თუ როგორ შეიძლება, მაქსიმალურად დიდმა მონაცემმა მინიმალურად მცირე ადგილი დაიკავოს. ამისთვის მოიგონეს მონაცემთა შეკუმშვის, კომპრესიის ტექნიკა, რომელიც მონაცემის ორიგინალ ზომას გარკვეული ალგორითმების საშუალებით აპატარავებს. სწორედ ასეთი ტექნიკები იძლევა იმის საშუალებას, რომ გამოსახულება მაღალი გარჩევადობის ხარისხით მივიღოთ, მაგალითად, ტელევიზიით ან ინტერნეტით.

მარტივად რომ წარმოვიდგინოთ - მონაცემთა კომპრესია წარმოადგენს პროცესს, როდესაც ჭარბი ინფორმაციის ამოღება, ამოცლა ხდება და ამის ხარჯზე ზომაში ვიგებთ. წარმოვიდგინოთ ერთი ასეთი მაგალითი. დაუშვათ, გვაქვს სურათი, სრულიად შავი ფერის, რომლის ტექნიკური პარამეტრები ასეთია: მისი განზომილება, სიგრძე და სიგანე, 100x100 წერტილი/პიქსელია და თითოეული პიქსელის ფერი 24 ბიტისანი ზომის ინფორმაციით არის წარმოდგენილი. ფერის წარმოდგენის 24 ბიტისანი სისტემა გულისხმობს, რომ ასარჩევ ფერთა გამაში სულ გვაქვს 2^{24} ცალი ფერი და კონკრეტული პიქსელი, ერთ-ერთი ამ ფერთაგანია. ჩვენს შემთხვევაში, ყველა შავია - შავი ფერი კი 24 ცალი 0-იანით მოიცემა (000000000000000000000000).

სურათის სრული ზომა $100 * 100 * 3 = 30\,000$ ბაიტი გამოვა, რადგან თითოეული პიქსელი 24 ბიტის ანუ 3 ბაიტის ტოლია.

სურათის ამ ფორმატში წარმოდგენისას ჩვენ გვაქვს პიქსელი₁ ფერით 000000000000 000000000000, პიქსელი₂ ფერით 000000000000000000000000, ... პიქსელი₁₀₀₀₀ ასევე 00000 00000000000000000000 ფერით.

თუ უფრო ჰკვიანურად წარმოვიდგენთ ამ მოცემულობას, სურათი შეგვიძლია სხვა კოდირებით ასე ჩავწეროთ: გვაქვს პიქსელი₁, პიქსელი₂ ... პიქსელი₁₀₀₀₀ და ბოლოში მივუწეროთ, რომ ყველა მათგანის ფერი არის 000000000000000000000000. შედეგად, მოვახდინეთ მონაცემთა კომპრესია და ამოვიღეთ ჭარბი ინფორმაცია. ასეთ კოდირებას run-length encoding ჰქვია. ის ერთ-ერთი მარტივი კომპრესიის ტექნიკაა. სადღეისოდ, უფრო განვითარებული და კომპლექსური ტექნიკები გამოიყენება, თუმცა მიზანი მაინც ერთი რჩება - მოვაშუროთ ჭარბი ინფორმაცია.

კომპრესიის ალგორითმები ორ ძირითად კატეგორიად იყოფა: დანაკარგის გარეშე და დანაკარგით. დანაკარგის გარეშე ტექნიკით შეკუმშული ფაილის აღდგენის შემდეგ, ის ორიგინალ ფაილს სრულიად დაემთხვევა. დანაკარგის ტექნიკის გამოყენებისას კი, აღდგენილი ვერსია მაქსიმალურად მიუახლოვდება ორიგინალს, თუმცა 100%-ით ვერა. დანაკარგის ტექნიკის მაგალითებია JPEG (სურათებისთვის), MP3 (მუსიკისთვის). ამ თავში ჩვენ მხოლოდ დანაკარგის გარეშე შეკუმშვის ტექნიკას მიმოვიხილავთ.

14.1 gzip

ლინუქს სისტემებში ფაილების შესაკუმშად ხშირად გამოიყენება **gzip** ბრძანება. შესრულების შემდეგ, ეს ბრძანება ორიგინალ ფაილს შეცვლის კომპრესირებული ვერსიით და ფაილის დასახელებაში მისანიშნებლად **.gz** გაფართოებას დაუმატებს. შეკუმშული ფაილების აღდგენა (დეკომპრესია, განკუმშვა) **gunzip** ბრძანებით ხდება. მოვიყვანოთ მისი გამოყენების მარტივი მაგალითი. ჯერ შევქმნათ სატესტო ფაილი **foobar.txt**¹:

```
achiko@debian:~$ man {ls,find,man} > foobar.txt
achiko@debian:~$ ls -lh foobar.txt
-rw-r--r-- 1 achiko achiko 123K Jun 25 15:57 foobar.txt
achiko@debian:~$
achiko@debian:~$ gzip foobar.txt
achiko@debian:~$ ls -lh foobar.txt.gz
-rw-r--r-- 1 achiko achiko 38K Jun 25 15:57 foobar.txt.gz
achiko@debian:~$ achiko@debian:~$ gunzip foobar.txt.gz
achiko@debian:~$ ls -lh foobar.txt
-rw-r--r-- 1 achiko achiko 123K Jun 25 15:57 foobar.txt
```

ახლა **gzip** ბრძანების რამდენიმე საინტერესო ოფცია განვიხილოთ:

ვნახოთ, შეკუმშვის შედეგად (**-v**, **--verbose** ოფციის გამოყენებით) რამდენი პროცენტი მოვიგეთ ორიგინალი ფაილის ზომაში:

```
achiko@debian:~$ gzip -v foobar.txt
foobar.txt:      69.6% -- replaced with foobar.txt.gz
```

მოლით, ახლა რამდენიმე ფაილი შევკუმშოთ ერთდროულად.

```
achiko@debian:~$ gzip test.txt test2.txt
achiko@debian:~$ ls test.*
test2.txt.gz  test.txt.gz
```

შევამოწმოთ შეკუმშული ფაილების ინტეგრალობა (**-t** ოფციით). ხომ არ მოხდა ტექნიკური ხასიათის რამე შეფერხება შეკუმშვის პროცესში:

¹ტერმინი **foobar** ან **foo** კომპიუტერში მუშაობისას გამოიყენება ობიექტის, ცვლადის, ფუნქციის ან ბრძანების დასახელებისას. **foo** ხშირად **bar**-თან ერთად გამოიყენება. ეს ტერმინი II მსოფლიო ომის დროს სამხედროების მიერ გამოყენებულ ჯარგონთან, **FUBAR**-თან, ასოცირდება.

```
achiko@debian:~$ gzip -tv foobar.txt.gz
foobar.txt.gz: OK
```

შეკუმშოთ ფაილი და, ამავედროულად, შევინახოთ მისი ორიგინალი ვერსია:

```
achiko@debian:~$ gzip -c foobar.txt > foobar.txt.gz
achiko@debian:~$ ls foobar.txt*
foobar.txt  foobar.txt.gz
```

ამ მაგალითში, სინამდვილეში, ფაილის შეკუმშული ვერსია სტანდარტულ გამოსასვლელზე გამოგვაქვს (-c ოფცია უზრუნველყოფს ამას), რომელსაც შემდეგ გადავამისამართებთ foobar.txt.gz ფაილში. გადამისამართების გარეშე ეკრანზე დავინახავდით შეკუმშული ვერსიის შიგთავსს.

-k (--keep) ოფციით კი პირდაპირ ვაიძულებთ gzip-ს, რომ ორიგინალი ფაილი შეინახოს.

```
achiko@debian:~$ gzip -k foobar.txt
```

როგორ შევკუმშოთ დირექტორიაში არსებული ყველა ფაილი? ამისთვის დაგვჭირდება -r ოფცია.

```
achiko@debian:~$ mkdir TXT && cp *.txt TXT/
achiko@debian:~$ gzip -r TXT/
achiko@debian:~$ ls TXT/
cities.txt.gz      distros.txt.gz    lorem5.txt.gz     sizes.txt.gz
countries.txt.gz   foobar.txt.gz     lorem.txt.gz       versions.txt.gz
dates.txt.gz       LICENSE.txt.gz    names.txt.gz
```

შეკუმშული ფაილების სტატისტიკა შეგვიძლია -l ოფციით ვნახოთ (ორიგინალი და შეკუმშული ფაილის ზომები, ზომაში მოგება პროცენტულად, ფაილის ორიგინალი დასახელება).

```
achiko@debian:~$ gzip -rl TXT/
```

	compressed	uncompressed	ratio	uncompressed_name
...				
	4072	12767	68.3%	TXT//LICENSE.txt
	95	132	49.2%	TXT//dates.txt
	38138	125455	69.6%	TXT//foobar.txt

gzip ბრძანება ფაილის შესაკუმშად იყენებს Deflate ალგორითმს (Lempel-Ziv (LZ77) ალგორითმსა და Huffman-ის კოდირების კომბინაციას). ეს ბრძანება გაშვებისას შესაძლებელია ისე დავაკონფიგურიროთ, რომ მოხდეს მაქსიმალურად მეტი ჭარბი ინფორმაციის მოძიება და ამოღება. თუმცა, ამ ოპერაციისთვის უფრო მეტი დროა საჭირო. შეკუმშვის ხარისხისა და ამ ოპერაციისთვის საჭირო დროის თანაფარდობა, gzip ბრძანების გაშვებისას, შეგვიძლია, შემდეგი ოფციებით დავარეგულიროთ: -1 (--fast)-დან -9

(--best)-მდე, ანუ უსწრაფესი შეკუმშიდან საუკეთესო შეკუმშვამდე. ოფციის მიუთითებლობის შემთხვევაში, ნაგულისხმევი მნიშვნელობა არის -6.

სწრაფი შეკუმშვა:

```
achiko@debian:~$ gzip -1 foobar.txt
achiko@debian:~$ ls -l foobar.txt.gz
-rw-r--r-- 1 achiko achiko 45404 Jun 25 15:57 foobar.txt.gz
```

საუკეთესო შეკუმშვა:

```
achiko@debian:~$ gzip -9 foobar.txt
achiko@debian:~$ ls -l foobar.txt.gz
-rw-r--r-- 1 achiko achiko 37919 Jun 25 15:57 foobar.txt.gz
```

როგორც წესი, ფაილების შეკუმშვას მიმართავენ მათი შენახვის, სხვასთან გაგზავნის (მაგალითად ელ. ფოსტით) ან გადატანის მიზნით. გადატანის შემდეგ კი აუცილებლად უნდა მოვახდინოთ მათი დეკომპრესია, რათა ამ ფაილებზე სხვადასხვა ოპერაციის (მაგალითად, რედაქტირების) გაკეთება შეგვეძლოს. თუმცა, shell-ში არსებობს ბრძანებები, რომლებიც პირდაპირ შეკუმშულ ფაილებზე ოპერირებს. ასეთებია: `zcat`, `zless`, `zmore`, `zgrep`, `zdiff` და სხვა.

```
achiko@debian:~$ zcat foobar.txt.gz
```

ამ ბრძანებით პირდაპირ ვნახულობთ `file.txt` ფაილის შიგთავსს. `zcat „gunzip -c“` ბრძანების იდენტურია. დანარჩენი ბრძანებების მნიშვნელობა, დარწმუნებული ვარ, ინტუიციურად გასაგები იქნება თქვენთვის.

აღსანიშნავია ისიც, რომ `gzip` ბრძანებას მხოლოდ ჩვეულებრივი ტიპის ფაილების შეკუმშვა შეუძლია. ის უგულებელყოფს მალსახმობს.

14.2 bzip2

ფაილების შესაკუმშად, `gzip` ბრძანების გარდა, linux სისტემებში არსებობს `bzip2` ბრძანებაც. ის Burrows-Wheeler-ის ალგორითმსა და Huffman-ის კოდირებას იყენებს. ამ დროს შეკუმშვა, როგორც წესი, უკეთესი ხარისხისაა, თუმცა შეკუმშვისა და მით უფრო დეკომპრესიის დრო გაცილებით დიდია, ვიდრე `gzip` ბრძანების შემთხვევაში. `bzip2` შეკუმშვის შემდეგ ფაილს დასახელებაში „.bz2“ გაფართოება დაემატება.

მოდით, ვნახოთ მისი გამოყენების რამდენიმე მაგალითი. `bzip2`-ს, უმეტეს შემთხვევაში, `gzip`-ის მსგავსი სინტაქსი აქვს.

შევკუმშოთ ფაილი:

```
achiko@debian:~$ bzip2 foobar.txt
```

შევკუმშოთ ფაილი და, ამავედროულად, შევინახოთ მისი ორიგინალი ვერსია:


```
achiko@debian:~$ bzip2 -c foobar.txt > foobar.txt.bz2
```

აწ

```
achiko@debian:~$ bzip2 -k foobar.txt
```

გამოვიტანოთ შეკუმშვის შესახებ ინფორმაცია:

```
achiko@debian:~$ bzip2 -v foobar.txt
```

შეგვუმშოთ სწრაფად:

```
achiko@debian:~$ bzip2 -1 foobar.txt
```

bzip2 ბრძანებისთვისაც სამართლიანია შეკუმშვის ხარისხის დონის მითითება ოფციებით -1 (უსწრაფესი)-დან -9 (საუკეთესო შეკუმშვა)-მდე.

time ბრძანებით შეგვიძლია, ვნახოთ რა დრო დაიხარჯა მოცემული ბრძანების შესასრულებლად. ასე უკეთ დავინახავთ განსხვავებას უსწრაფესსა და საუკეთესო შეკუმშვას შორის.

```
achiko@debian:~$ time bzip2 -1 foobar2.txt
```

```
real    0m20.627s
user    0m14.304s
sys     0m6.168s
```

```
achiko@debian:~$ bunzip2 foobar2.txt.bz2
```

```
achiko@debian:~$ time bzip2 -9 foobar2.txt
```

```
real    0m45.857s
user    0m45.796s
sys     0m0.048s
```

რაც უფრო დიდია ფაილის ზომა, მით უფრო თვასაჩინოა ეს განსხვავება.

ახლა, მოდით, მოვახდინოთ რამდენიმე შეკუმშული ფაილის კონკატენაცია:

```
achiko@debian:~$ bzip2 -c names.txt > file.bz2
```

```
achiko@debian:~$ bzip2 -c versions.txt >> file.bz2
```

file.bz2 ახლა შეიცავს ორივე ფაილის (**names.txt** და **versions.txt**) შეკუმშულ ვერსიას. მისი დეკომპრესიის შედეგად მივიღებთ ფაილს **file**, სადაც ორიგინალი ფაილების შიგთავსი მიჯრით იქნება შენახული.

cat file იმავე შედეგს მოგვცემს, რასაც **cat names.txt versions.txt**

მარტივად კი შეგვიძლია, ასე ვნახოთ მისი შიგთავსი:

```
achiko@debian:~$ bzip file.bz2
```

ზოგადად, შეკუმშვის პროცესით ფაილის შიგთავსის სტრუქტურა იცვლება. მას თავსართი და ბოლოსართი ემატება, სადაც განსაზღვრულია ყველა ის წესი, თუ როგორ შეიკუმშება აღნიშნული ფაილი. სწორედ ამ თავსართისა და ბოლოსართის მეშვეობით ხდება სწორი დეკომპრესია. მაშ, რა მოხდება, თუ ზომით პატარა ან სულაც ცარიელი ფაილის შეკუმშვას შევცდებით (თუმცა, ეს ოპერაცია აზრს მოკლებულია)? რა თქმა უნდა, ამ შემთხვევაში ფაილის ზომა იზრდება.

```
achiko@debian:~$ echo "hello" > new_file
achiko@debian:~$ ls -l new_file
-rw-r--r-- 1 achiko achiko 6 Jun 25 17:57 new_file
achiko@debian:~$ gzip new_file
achiko@debian:~$ ls -l new_file.gz
-rw-r--r-- 1 achiko achiko 35 Jun 25 17:57 new_file.gz
```

როგორც ვთქვით პატარა ზომის ფაილის შეკუმშვის დროს ზომა პირიქით გაიზარდა.

დაიმახსოვრეთ

აზრი არ აქვს უკვე შეკუმშული ფაილის ხელახლა შეკუმშვას. მაგალითად, jpeg გაფართოების ფაილი, თავის მხრივ, უკვე კომპრესირებულია და მისი შეკუმშვა დიდ ვერაფერ შედეგს მოგვცემს.

14.3 tar

ერთ-ერთ წინა მაგალითში ვნახეთ, თუ როგორ შეიძლება, მოგვეთავსებინა რამდენიმე ფაილის შიგთავსის შეკუმშული ვერსია ერთ ფაილში, თუმცა, როდესაც საქმე ფაილების დიდ რაოდენობას ეხება, რომლებიც დირექტორიებში და ქვედირექტორიებშია განაწილებული, ეს მიდგომა არ გამოგვადგება. მეორე მხრივ, ჩვენთვის ნაცნობი შეკუმშვის ბრძანებები ვერ უზრუნველყოფს თავად დირექტორიების შეკუმშვას. სწორედ ამ დროსაა სასარგებლო, გამოვიყენოთ არქივი. დაარქივებას, როგორც წესი, მნიშვნელოვანი, მხოლოდ უკვე იშვიათად გამოსაყენებელი მონაცემების დიდი ვადით შესანახად მიმართავენ.

მონაცემთა დაარქივებისთვის linux სისტემებში გამოიყენება **tar**² ბრძანება. ამ ბრძანებით ფაილებისა და დირექტორიების დიდი კოლექცია გადაგვაქვს ერთ ფაილში - არქივში, რომელსაც, ჩვეულებისამებრ, tar გაფართოვებას ვუთითებთ და მას tarball-ს უწოდებენ. არქივი ზომით გაცილებით დიდი ფაილი გამოვა, ვიდრე მაში განთავსებული ფაილების ზომების ჯამი. თუმცა შეკუმშვის ბრძანებასთან ერთად კომბინაციაში, საკმაოდ კარგ შედეგს იძლევა და, შესაბამისად, მიღებულ ფორმადაც ითვლება.

tar ბრძანების შესაძლებლობების სანახავად მოდით, კვლავ მაგალითებს მივმართოთ. შევქმნათ არქივი დასახელებით archive-2018.06.25.tar და მასში განვათავსოთ დირექტორიები და ფაილები.

```
achiko@debian:~$ tar -cvf archive-2018.06.25.tar test.txt test2.txt TXT/
```

²ტერმინი tar მოდის Tape ARchive-დან. Linux/Unix სისტემის ადმინისტრატორები ხშირად მიმართავენ მონაცემების მაგნიტურ ლენტებზე (tape) გადატანას.

ვნახოთ, რომელი ოფცია რას აკეთებს:

-c (--create)	ქმნის ახალ არქივს.
-v (--verbose)	გვანახებს არქივის შექმნის პროცესს.
-f (--file) archive	გვუბნება, რომ არქივი იყოს ფაილი სახელად archive ან მოწყობილობა archive, ამ დროს მონაცემები დაარქივდება მოწყობილობაზე (როგორც აღვნიშნეთ, დაარქივება შეიძლება მოხდეს მაგნიტურ ლენტებზე).

tar ბრძანება იმ იშვიათ ბრძანებათა რიგს მიეკუთვნება, რომელსაც ოფცია შეგვიძლია ტირეს (-) გარეშეც მივუთითოთ, ასე:

```
achiko@debian:~$ tar cvf archive-2018.06.25.tar test.txt test2.txt TXT/
```

ვახსენეთ, რომ არქივის ზომა ყოველთვის უფრო მეტია, ვიდრე მასში შემავალი ფაილების ზომების ჯამი. შესაბამისად, უპრიანია, ის შევკუმშოთ **gzip**-ით ან **bzip2**-ით:

```
achiko@debian:~$ gzip archive-2018.06.25.tar
```

ამ მაგალითში ჩვენ ჯერ არქივი შევქმენით და შემდეგ ის, ზომის მოგების მიზნით, შევკუმშეთ. თუმცა **tar** ბრძანებით, **z** ოფციის გამოყენებით, ერთბაშად შეგვიძლო დაარქივებისა და შეკუმშვის ოპერაციის განხორციელება, ასე: (შევკუმშვა, ამ შემთხვევაში, **gzip** ბრძანებით ხდება):

```
achiko@debian:~$ tar -cvzf archive-2018.06.25.tar.gz test.txt test2.txt  
TXT/
```

მიღებულია, რომ **gzip**-ით შეკუმშულ არქივისთვის ან **.tar.gz** გაფართოება გამოვიყენოთ ან მოკლედ **.tgz**.

```
achiko@debian:~$ tar -cvzf archive-2018.06.25.tgz test.txt test2.txt TXT/
```

შესაკუმშად, **bzip2**-ის გამოყენების სურვილის შემთხვევაში, **z** ოფციის ნაცვლად, **tar** ბრძანებაში **j** ოფცია უნდა გამოვიყენოთ. ამის მისანიშნებლად არქივის დასახელებაშიც ან **.tar.bz2** გაფართოებას იყენებენ ან მოკლედ **.tbz2**-ს. ასე:

```
achiko@debian:~$ tar -cvjf archive-2018.06.25.tbz2 test.txt test2.txt TXT/
```

არქივიდან ფაილების ამოღება ხდება **-x** (--extract, --get) ოფციით და ისინი განთავსდება მიმდინარე დირექტორიაში. თუ სხვა დირექტორიაში გვსურს არქივის ფაილების ამოღება, ეს **-C** ოფციითაა შესაძლებელი. მას არგუმენტად სასურველი დირექტორია უნდა მივუთითოთ.

```
achiko@debian:~$ tar -xvzf archive-2018.06.25.tbz2
achiko@debian:~$ tar -xvzf archive-2018.06.25.tbz2 -C new_dir/
```

შესაძლებელია, აგრეთვე, არქივიდან კონკრეტული ფაილი ამოვიღოთ. .tgz აქრივიდან ეს ასე ხდება:

```
achiko@debian:~$ tar -xvzf archive-2018.06.25.tgz test.txt
```

ან

```
achiko@debian:~$ tar --extract --file=archive-2018.06.25.tgz test.txt
```

ზოლო .bz2 აქრივიდან კი ასე:

```
achiko@debian:~$ tar -xvzf archive-2018.06.25.tbz2 test.txt
```

ან

```
achiko@debian:~$ tar --extract --file=archive-2018.06.25.tbz2 test.txt
```

შესაძლებელია, აგრეთვე, მაგენერირებული სიმბოლოს გამოყენება. ყურადღება მიაქციეთ იმას, რომ ეს ოფცია მხოლოდ tar-ის GNU-ს ვერსიას აქვს:

```
achiko@debian:~$ tar -xvf archive-2018.06.25.tar --wildcards "*.txt"
```

```
achiko@debian:~$ tar -zxvf archive-2018.06.25.tgz --wildcards "*.txt"
```

```
achiko@debian:~$ tar -jxvf archive-2018.06.25.tbz2 --wildcards "*.txt"
```

ასე, არქივიდან ამოვიღებთ მხოლოდ txt გაფართოების ფაილებს.

მოდით, დავუმატოთ უკვე არსებულ არქივს ფაილები ან/და დირექტორიები:

```
achiko@debian:~$ tar -rvf archive-2018.06.25.tar Tux.jpg
```

მოდით, ასეთი სცენარი წარმოვიდგინოთ. ვთქვათ, გვსურს არსებულ JPG_archives.tar არქივს დავუმატოთ Documents/ დირექტორიაში არსებული jpg გაფართოების სხვა ფაილებიც.

```
achiko@debian:~$ find Documents/ -name "*.jpg" -exec tar -rvf
JPG_archives.tar "{}" +
```

თუ გვსურს, ამ მოძებნილი ფაილების არქივი შევქმნათ და შემდეგ, თან შევკუმშოთ, ასე უნდა მოვიქცეთ:

```
achiko@debian:~$ find Documents/ -name "*.jpg" | tar -cvf JPG_archives2.tar  
-T - | gzip > JPG_archives2.tgz
```

-T file მიუთითებს ფაილების სიას არქივის შესაქმნელად. ჩვენს შემთხვევაში, -T ოფციის არგუმენტი file-ის ნაცვლად არის „-“. tar ბრძანებაში კი „-“ ნიშნავს სტანდარტულ შესასვლელზე მიღებულ ფაილებს.

როგორ ამოვშალოთ კონკრეტული ფაილი ან/და დირექტორია არსებული არქივიდან?

```
achiko@debian:~$ tar -vf archives.tar --delete file1 file2
```

აღსანიშნავია, რომ ამოშლისას file ისეთი path-ით უნდა მივუთითოთ, როგორითაც არის ის არქივში შენახული.

```
achiko@debian:~$ tar -tf archive-2018.06.25.tar  
...  
TXT/distros.txt  
TXT/versions.txt  
Tux.jpg  
achiko@debian:~$ tar -vf archive-2018.06.25.tar --delete Tux.jpg  
achiko@debian:~$ tar -tf archive-2018.06.25.tar  
...  
TXT/distros.txt  
TXT/versions.txt
```

ერთ-ერთ წინა ბრძანებაში JPG_archives.tar არქივს დავუმატეთ სხვა ფაილები. თუმცა, მოდით, ჯერ ვთქვათ, თუ როგორ შეიძლება შევქმნათ find ბრძანებით მოძებნილი ფაილების არქივი. მაგალითისთვის მოვძებნოთ png გაფართოების ფაილები (სისტემაში ისინი მრავლად შეგვხვდება jpeg-სგან განსხვავებით) და დავაარქივოთ ისინი.

```
$ find /usr/share/doc -name "*.png" -exec tar -cvf png.tar "{}" +  
tar: Removing leading '/' from member names  
/usr/share/doc/groff-base/html/img/pic30.png  
/usr/share/doc/groff-base/html/img/pic43.png  
...
```

ამ ბრძანებით png.tar ფაილი ნამდვილად შეიქმნება, თუმცა არქივის შექმნისას გაფართოებებს გამოვა ეკრანზე. ეს შეტყობინება გვეუბნება, რომ უსაფრთხოების მიზნით tar ბრძანებამ დასაარქივებელი ფაილების აბსოლუტურ გზას წინ ავტომატურად „/“ მოაშორა. ეს იმიტომ ხდება, რომ არქივიდან ამოღების დროს მსგავსი დასახელების ფაილზე არ მოხდეს ახალი შიგთავსის გადაწერა და ძველის დაკარგვა. ამიტომ, ჯობს, რომ არქივში ფაილების ფარდობითი გზა მოვათავსოთ. ხოლო არქივიდან ამოღების დროს, -C ოფციის გამოყენებით, თავად მივუთითოთ თუ რომელ დირექტორიაში გვსურს ამ ფაილების ამოღება.

მოძებნილი ფაილების tar ბრძანებისთვის ერთ ზაზად გადაცემა, მილის და xargs

ბრძანების დახმარებით, ასე შეგვიძლია:

```
$ find /usr/share/doc -name "*.png" -print0 | xargs -0 tar -cvf png.tar
```

14.4 cpio

cpio ბრძანებას ფაილების დაარქივებისთვის არის შექმნილი. მისი დასახელება მოდის „copy in, copy out“-დან. **cpio** ბრძანებით შესაძლებელია ფაილების არქივში კოპირება, მათი არქივიდან ამოღება და ფაილების ერთი დირექტორიიდან მეორეში გადაცემა.

გნაზოთ მაგალითები. შევქმნათ არქივი:

```
$ ls *.txt | cpio -ov > txt.cpio
```

ასე შეიქმნება არქივი `txt.cpio`, სადაც მიმდინარე დირექტორიის ყველა `.txt` გაფართოების ფაილი შევა. `-o`, `--create` ოფცია არქივის ფაილის შექმნას ნიშნავს, `-v`, `--verbose`-ით კი ეკრანზეც დავინახავთ იმ ფაილების დასახელებას, რომლებიც არქივში კოპირდება.

თუ გვსურს, რომ დირექტორიის მთლიანი ხე დავაარქივოთ (ქვე-დირექტორიების ჩათვლით), მაშინ ასეთი ბრძანება უნდა გავუშვათ:

```
$ find . -print -depth | cpio -ov > dir.cpio
```

`-depth` ოფციით **find** ბრძანება მოძებნილი დირექტორიის სახელების გამოტანამდე ჯერ მასში არსებულ სტრუქტურას გამოიტანს.

ამოარქივება ასეთი ბრძანებით ხდება:

```
$ cpio -idv < dir.cpio
```

`-i`, `--extract` ხსნის არქივს, `-d` ოფციის მითითება აუცილებელია, რათა ამოღებულ დირექტორიებში ქვესტრუქტურა შენარჩუნდეს. სხვაგვარად ეს დირექტორიები ცარიელი იქნება. ფაილების ამოღებისას `-d` ოფცია შეგვიძლია არ გამოვიყენოთ, თუ არქივში მხოლოდ ფაილებია შენახული.

არქივის შიგთავსის ნახვა ასეა შესაძლებელი:

```
$ cpio -it < dir.cpio
```

ხაზგასასმელია ის ფაქტი, რომ **cpio** თავსებადია **tar**-თან. მას შეუძლია შექმნას და გახსნას **tar** არქივი. ზოგად შემთხვევებში, **tar** უფრო პოპულარულია და უფრო ხშირად გამოიყენება, ვიდრე ვიდრე **cpio**, მეტი სიმარტივისა და იმის გამო, რომ **tar** ბრძანებას დამატებით შეკუმშვის შესაძლებლობა აქვს.

cpio ბრძანების ერთ-ერთი ძლიერი მხარე ისაა, რომ ძალიან მარტივად ახდენს დიდი ხისებრი სტრუქტურის სხვა დირექტორიაში გადატანას. ამისთვის ჯერ უნდა შევქმნათ გადასატანი ფაილების სია და შემდეგ მთლიან გადავცეთ **cpio** ბრძანებას. `-p`, `--pass-through` ოფცია უზრუნველყოფს ფაილების გადაცემას სხვა დირექტორიაში. `-0`, `--null` ოფცია კი, **find**-ის `-print0`-თან ერთად, როგორც უკვე ვიცით, უზრუნველყოფს

სპეციალური სიმბოლოების შემცველი ფაილების დასახელებების კორექტულად დამუშავება/გადატანას. ეს ყველაფერი სინამდვილეში სხვა არაფერია, თუ არა კოპირების ოპერაცია. მხოლოდ, სარეზერვო ასლში ფაილების შიგთავსის გარდა, შენარჩუნებული უნდა იყოს ფაილების მახასიათებლებიც, როგორცაა მისი დროითი მახასიათებლები, მფლობელობა და ა.შ. ეს ოპერაცია სხვა ხერხებითაც არიც შესაძლებელი, მაგალითად **cp** ან **rsync** ბრძანებით. **rsync**-ს, დამატებით, მონაცემების გადატანა ქსელშიც შეუძლია. მოდით, განვიხილოთ ყველა ეს ვარიანტი. ასეთი სცენარი წარმოვიდგინოთ: დავუშვათ, გვსურს ჩვენი პირადი დირექტორიის სრული ასლი გადავიტანოთ **/tmp/dir** დირექტორიაში.

ვნახოთ, როგორ ხდება ეს ოპერაცია სხვადასხვა ბრძანებით. თან შედარებისთვის შევამოწმოთ, რა დრო მიაქვს თითოეულ ოპერაციას:

```
$ cd $HOME
$ time find . -depth -print0 | cpio -pvd0 /tmp/dir/

real    0m3.589s
user    0m0.036s
sys     0m1.220s
$ rm -rf /tmp/dir/{.[^.] ,.*,*}      # ვშლით ყველაფერს /tmp/dir-ში
$ cd $HOME; time tar -c . | tar -x -C /tmp/dir/

real    0m4.692s
user    0m0.060s
sys     0m0.984s
$ rm -rf /tmp/dir/{.[^.] ,.*,*}
$ time rsync -a . /tmp/dir

real    0m3.731s
user    0m1.092s
sys     0m0.848s
$ rm -rf /tmp/dir/{.[^.] ,.*,*}
$ time cp -a . /tmp/dir

real    0m4.561s
user    0m0.000s
sys     0m0.440s
```

time ბრძანებამ ეკრანზე გამოიტანა დეტალური ინფორმაცია: **Real** - არის დრო, რომელიც გავიდა პროცესის დაწყებიდან მის დამთავრებამდე. რეალურად, ამ დროში გათვალისწინებულია ის დრო, რაც ჩვენი პროცესის გარდა სხვა პროცესების შესრულებისთვის დაიხარჯა სისტემაში. **User** - არის პროცესორის ის დრო (ე.წ. CPU time), რომელიც ამ პროცესის შესრულებისთვის დაიხარჯა მომხმარებლის გარემოში, ზოლო **Sys** - არის პროცესორის ის დრო, რომელიც ამ პროცესის შესასრულებლად ბირთვმა გამოიყენა.

User+Sys არის კონკრეტულად ჩვენი პროცესისთვის დახარჯული დრო. გარკვეული წარმოდგენა შეიძლება შეგვექმნა იმაზე, თუ რომელი ბრძანება რამდენად უფრო წარმადია ერთი მეორესთან მიმართებაში, თუმცა ამ ერთი სურათით რომელიმე ბრძანების გამორჩევა არ შეიძლება, რადგან ძალიან ბევრ ფაქტორზეა დამოკიდებული, თუ რა დრო იხარჯება ამა თუ იმ ფაილის კოპირებისთვის. მაგალითად, როგორ არის სისტემა დატვირთული გადატანის პროცესში; რა საერთო ზომის მონაცემებია გადასატანი; რამხელა და რამდენი ფაილისგან შედგება ეს დირექტორია; ბევრ **Sparse** ფაილებთან გვაქვს თუ არა საქმე (ამ შემთხვევისთვის

ჩვენ მიერ განხილულ ბრძანებებს სპეციალური ოფციები აქვთ) და ა.შ.

ფაილების შესაკუმშად ჩვენ განვიხილეთ ბრძანებები **gzip** და **bzip2**, თუმცა, მათ გარდა, არსებობს სხვა პროგრამებიც, რომლებიც მსგავსი დანიშნულებისთვის გამოიყენება. ესენია: **rar**, **zip**, **7z** და სხვა. ისინი, შეკუმშვის გარდა, დაარქივების ფუნქციასაც ასრულებენ ამადროულად. მათი დეტალურად განვიხილვას აქ არ შევუდგებით. უბრალოდ, შევადაროთ ამ ბრძანებების წარმადობა ერთმანეთს შეკუმშვის მხრივ. მივიღებთ შემდეგი სახის ცხრილს:

შეკუმშვის ბრძანებების შედარება	
დეკომპრესიის სიჩქარე (სწრაფიდან ნელისკენ)	gzip, zip → 7z → rar → bzip2
შეკუმშვის სიჩქარე (სწრაფიდან ნელისკენ)	gzip, zip → bzip2 → 7z → rar
შეკუმშვის ხარისხი (უკეთესიდან უარესისკენ)	7z → rar, bzip2 → gzip → zip
ხელმისაწვდომობა (Unix/Linux)	gzip, bzip2 → zip → 7z → rar
ხელმისაწვდომობა (Windows)	zip → rar → 7z → gzip, bzip2

თავი 15

ამოცანების გაშვების ავტომატიზაცია

3 ომპიუტერზე მუშაობისას ხშირია ისეთი სიტუაცია, როდესაც ამოცანის პერიოდულად გაშვებაა საჭირო. ასეთ დროს, ამ ამოცანების ხელით გაშვების ნაცვლად, Unix-ის მსგავს სისტემებში არსებობს ამოცანების ავტომატურად გაშვების მძლავრი სერვისი - **cron**.

15.1 cron

cron წარმოადგენს დემონს, რომელიც სისტემაში ფონურ რეჟიმად არის გაშვებული და ის ასრულებს გარკვეულ ბრძანებებს მოცემულ თარიღსა და დროში. **cron** ხშირად გამოიყენება სისტემის ადმინისტრირების დროს ამოცანების გაშვების ავტომატიზირებისთვის - როგორცაა სარეზერვო ასლების გაკეთება, სისტემის განახლება, გარკვეული ამოცანების წარმატებულად ან, თუნდაც, წარუმატებლად გაშვების/დამთავრების შემთხვევაში, შეტყობინებების გაგზავნა და ა.შ.

ლინუქსის თითქმის ყველა დისტრიბუტივში, ნაგულისხმევი მნიშვნელობით, **cron** დემონი არის დაყენებული და ის ავტომატურადაა გაშვებული სისტემის ჩართვის შემდეგ. მომხმარებელს შეუძლია **cron**-ს თავისი ამოცანა გადასცეს და ამისთვის ის იყენებს **crontab** ბრძანებას. **crontab** (**cron table**-ის შემოკლებული ფორმა), ამავდროულად, **cron**-ის კონფიგურაციის ფაილს წარმოადგენს. **crontab** ბრძანებით პირდაპირ შეგვიძლია ამ კონფიგურაციის ფაილის რედაქტირება. მასში თითოეულ ხაზზე იწერება თუ რა ამოცანა უნდა გაეშვას და როდის.

მოდით ვნახოთ, როგორ შეგვიძლია გავხსნათ და დავამატოთ ჩვენი ამოცანა ჩვენს **crontab** ფაილში. ამისთვის, როგორც უკვე ვთქვით, ვიყენებთ **crontab** ბრძანებას **-e** ოფციით. გაუშვათ.

```
achiko@debian:~$ crontab -e
```

```
Select an editor. To change later, run 'select-editor'.
```

1. /bin/nano <---- easiest
2. /usr/bin/vim.basic
3. /usr/bin/vim.tiny

```
Choose 1-3 [1]:
```

ამ ბრძანების გაშვების შემდეგ, ეკრანზე გამოვა შეკითხვა - თუ რომელი ტექსტური რედაქტორის გამოყენება გსურთ. არჩევანის დაფიქსირების შემდეგ, თუ მისი შეცვლა

დაგვჭირდა, მაშინ EDITOR ან VISUAL ცვლადი უნდა დავარედაქტიროთ, ასე:

```
achiko@debian:~$ export EDITOR=vi
achiko@debian:~$ crontab -e
```

აქ ყოველი ხაზი, რომელიც #-ით იწყება, კომენტარს აღნიშნავს და მხედველობაში არ მიიღება. ამ კომენტარებით მოცემულია პატარა აღწერა, თუ როგორ უნდა ჩაიწეროს ხაზი სწორი სინტაქსით. ანუ ხაზი უნდა შედგებოდეს გამოტოვებით დაშორებული 6 ველისგან, სადაც თითოეული ნიშნავს შემდეგს:

```
achiko@debian:~$ crontab -e
#           ,-----> წუთი
#           | ,-----> საათი
#           | | ,-----> თვის დღე
#           | | | ,-----> თვე (1=იანვარი,2=თებერვალი, ...)
#           | | | | ,-----> კვირის დღე (1=ორშაბათი,2=სამშაბათი, ...)
#           | | | | | ,---> გასაშვები ბრძანება
#           | | | | |
#           | | | | |

1 2 3 * * echo "Hello Georgia" > /tmp/crontest.txt
```

წუთი	ნებისმიერი ნატურალური რიცხვია 0-დან 59-ის ჩათვლით.
საათი	ნებისმიერი ნატურალური რიცხვია 0-დან 23-ის ჩათვლით.
თვის დღე	ნებისმიერი ნატურალური რიცხვია 1-დან 31-ის ჩათვლით (თვის დაკონკრეტების შემთხვევაში ნამდვილი რიცხვია უნდა იყოს. მაგ. თებერვლის შემთხვევაში 31 არ გამოგვადგება).
თვე	ნებისმიერი ნატურალური რიცხვია 1-დან 12-ის ჩათვლით.
კვირის დღე	ნებისმიერი ნატურალური რიცხვია 0-დან 7-ის ჩათვლით, სადაც 0 და 7 წარმოადგენს კვირა დღეს.
ბრძანება	აქ ნებისმიერი თქვენთვის სასურველი ბრძანება უნდა ჩაწეროთ.

ჩამოთვლილი მნიშვნელობების გარდა, შესაძლებელია გამოვიყენოთ შემდეგი ასო-ნიშნები: ფიფქი (*) ნებისმიერის აღსანიშნავად, ტირე (-) ინტერვალის აღსანიშნავად და მძიმე (,) ჩამონათვალის მისათითებლად. შესაძლებელია სლექსის (/) გამოყენება რიცხვითი მნიშვნელობის ბიჯის მოსაყვამად. მაგალითად:

01 02 03 * * cmd	ყოველი თვის 3 რიცხვში კვირის ნებისმიერ დღეს 02 საათსა და 01 წუთზე შესრულდეს cmd ბრძანება.
0 03,15 15 * * cmd	ყოველი თვის 15 რიცხვში კვირის ნებისმიერ დღეს დღისა და ღამის 3 საათზე შესრულდეს cmd ბრძანება.

0 03-15/5 15 * * cmd	ყოველი თვის 15 რიცხვში კვირის ნებისმიერ დღეს ღამის 3 საათიდან დაწყებული ყოველ 5 საათში დღის 3 საათის ჩათვლით შესრულდეს cmd ბრძანება.
*/15 3,4,5 * 2-12/2 1-5 cmd	თებერვლიდან დაწყებული ყოველი მეორე თვის (ანუ თებერვალი, აპრილი, ივნისი ...) ყოველ დღე შაბათ-კვირის გარდა ყოველ 3, 4 და 5 საათსა და ყოველ 15 წუთში შესრულდეს cmd ბრძანება.

Crontab-ში დროის მარტივად მოსაგემად უკვე არსებობს მოკლე ჩანაწერები:

@reboot	გაეშვას ერთხელ, სისტემის ჩართვის დროს
@yearly	გაეშვას წელიწადში ერთხელ: 0 0 1 1 *
@annually	იგივეა, რაც @yearly
@monthly	გაეშვას თვეში ერთხელ: 0 0 1 * *
@weekly	კვირაში ერთხელ: 0 0 * * 0
@daily	დღეში ერთხელ: 0 0 * * *
@midnight	იგივეა რაც @daily
@hourly	გაეშვას საათში ერთხელ: 0 * * * *

მოდით, დავაკვირდეთ ჩვენს პირად დირექტორიას და გავარკვიოთ, ყოველდღიურად, 6 საათში ერთხელ, თუ რომელი 5 ფაილი ან დირექტორია იკავებს ყველაზე დიდ ადგილს დისკზე. საბოლოოდ, მათი სია ფაილში ჩაწეროთ, რომელსაც სახელად მიმდინარე თარიღს დავარქმევთ.

```
achiko@debian:~$ crontab -e
00 */6 * * * du -sh $HOME/*|sort -rh|head -5 >>/tmp/$(date +%Y%m%d).txt
```

Crontab კონფიგურაციის ფაილში **date** ბრძანების გამოყენებისას % სიმბოლოს კორექტულად აღსაქმელად წინ \ უნდა მივუწეროთ. Crontab-დან გამოსვლის შემდეგ ეკრანზე ასეთი შეტყობინება დაიწერება:

```
crontab: installing new crontab
```

ეს იმას ნიშნავს, რომ კონფიგურაცია შეინახა ფაილში, რომელიც ძვეს შემდეგ დირექტორიაში /var/spool/cron/crontabs/ და მას ჩვენი მომხმარებლის სახელი ჰქვია. ყურადღება მიაქციეთ, რომ ამ დირექტორიაზე წვდომის უფლებები აქვს მხოლოდ root-ს.

```
achiko@debian:~$ ls /var/spool/cron/crontabs/
ls: cannot open directory '/var/spool/cron/crontabs/': Permission denied
```

მიუხედავად ამისა, მომხმარებელს მაინც აქვს იმის უფლება, რომ ნახოს ან წაშალოს თავის მიერ შექმნილი კონფიგურაციის ფაილი. ეს კი იმის წყალობით მიიღწევა, რომ

ჩვეულებრივი მომხმარებელი `crontab` ბრძანებას მისი მფლობელი ჯგუფის უფლებებით უშვებს (ანუ მასზე `SGID` ბიტია გააქტიურებული).

```
achiko@debian:~$ ls -l /usr/bin/crontab
-rwxr-sr-x 1 root crontab 40264 May  3 2015 /usr/bin/crontab
```

```
achiko@debian:~$ crontab -l
```

ამ ბრძანებით შეგვიძლია ვნახოთ ჩვენი `crontab`-ის ფაილი.

```
achiko@debian:~$ crontab -r
```

ასე კი წავშლით ჩვენსავე ფაილს. გადავამოწმოთ:

```
achiko@debian:~$ crontab -l
no crontab for achiko
```

ნაგულისხმევი მნიშვნელობით, `cron`-ის გამოყენება ყველა მომხმარებელს შეუძლია. ანუ, მათ შეუძლიათ საკუთარი `crontab` შექმნან, სადაც ბრძანებების გაშვებას დაგეგმავენ სურვილისებრ. მიუხედავად ამისა, `root` მომხმარებელი, რომელიც სისტემის ადმინისტრატორია, ინარჩუნებს უფლებას აუკრძალოს კონკრეტულ მომხმარებლებს `cron`-ით სარგებლობა. ადმინისტრატორი ამ უფლებებს არეგულირებს `/etc/cron.allow` და `/etc/cron.deny` ფაილების საშუალებით.

თუ `/etc/cron.allow` ფაილი არსებობს, `cron`-ით სარგებლობა მხოლოდ ამ ფაილში ხაზ-ხაზ ჩამოთვლილ მომხმარებლებს შეუძლიათ. დანარჩენ მომხმარებლებს `cron`-ით სარგებლობა ავტომატურად ეკრძალებათ. ამ ფაილის არ არსებობის შემთხვევაში, `/etc/cron.deny` ფაილი მოიძებნება. მასში გაწერილ მომხმარებლებს ეკრძალებათ `cron`-ით სარგებლობა. დანარჩენებს კი ავტომატურად ეძლევათ მისი გამოყენების უფლება.

იმ შემთხვევაში, თუ არცერთი ფაილი არ არსებობს, ან ყველა მომხმარებელს შეუძლია `crontab`-ის გამოყენება ან მხოლოდ `root`-ს. ეს დამოკიდებულია კონკრეტული დისტრიბუტივის შემქმნელებზე (Debian-ში, ამ შემთხვევაში, ყველა მომხმარებელს აქვს ამის უფლება).

`root` მომხმარებელსაც შეუძლია შექმნას საკუთარი `crontab` ფაილი ზემოთ ხსენებული მეთოდებით, თუმცა ამის გარდა, მას შეუძლია, გამოიყენოს `/etc/crontab` ფაილი და `/etc/cron.d` დირექტორია. ამ ორ უკანასკნელ მეთოდს ის უპირატესობა აქვს, რომ მათში შესაძლებელია იმ მომხმარებლის სახელის დაფიქსირება, ვისი უფლებებითაც `root`-ს მოცემული ბრძანებების გაშვება სურს. მომხმარებლის ეს სახელი გასაშვები ბრძანების წინ, დამატებით ველში მოიცემა.

```
achiko@debian:~$ cat /etc/crontab
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the `crontab' command to
# install the new version when you edit this file and files in /etc/cron.d.
# These files also have username fields, that none of the other crontabs do.
SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
```

```
# m h dom mon dow user  command
17 * * * * root    cd / && run-parts --report /etc/cron.hourly
...
```

ამას გარდა, /etc-ში, ნაგულისხმევი მნიშვნელობით, შექმნილია შემდეგი დირექტორიები: cron.hourly/, cron.daily/, cron.weekly/ და cron.monthly/ და მათში ჩაწერილი სკრიფტები თუ ბრძანებები გაეშვება, შესაბამისად, ყოველ საათში, ყოველდღიურად, ყოველკვირეულად და ყოველთვიურად.

როგორც წესი, cron-ში გაწერილი ამოცანების გაშვება ღამის საათებშია დაგეგმილი. ეს ის დროა, როდესაც, ხშირ შემთხვევებში, კომპიუტერი გამორთულია და, შესაბამისად, cron ვერ შეასრულებს მითითებულ ბრძანებებს. ასეთი შემთხვევების აღმოსაფხვრელად არსებობს დემონი anacron. ის, ფაქტობრივად, ავსებს cron-ს და როგორც კი კომპიუტერი ჩაირთვება, anacron ყველა დარჩენილ ამოცანას შეასრულებს.

ასეთი გამოტოვებული ბრძანებების სია შეიძლება დიდი იყოს. ეს კი, კომპიუტერს საგრძნობლად დატვირთავს ჩართვის შემდეგ, რომელიც მის მუშაობას შეანელებს. ამიტომ, /etc/anacrontab ფაილში გაწერილი ამოცანები nice ბრძანებით ეშვება, რათა კომპიუტერმა მათ შესრულებაზე ნაკლები რესურსი დახარჯოს.

/etc/anacrontab-ის სინტაქსი განსხვავდება /etc/crontab ფაილის ჩანაწერისგან. მისი ხაზი შედგება 4 ველისგან: period delay job-identifier cmd, სადაც:

Period	არის ამოცანის შესრულების სიხშირე მოცემული დღეებში. მაგ: 1 - ნიშნავს ყოველდღე, 5 - ხუთ დღეში ერთხელ.
Delay	აქ მოიცემა რიცხვი და ის აღნიშნავს წუთების რაოდენობას, რომლის განმავლობაშიც სისტემა ყოვნიდება ამოცანების გაშვებამდე.
Job-identifier	ამოცანის დასახელება, რომელი სახელწოდებითაც ის log ფაილებში ჩაიწერება.
Cmd	გასაშვები ბრძანება.

```
achiko@debian:~$ cat /etc/anacrontab
1      5      cron.daily      nice run-parts /etc/cron.daily
7      15     test.daily      /bin/sh /home/achiko/backup.sh
...
```

cron ძირითადად გამოიყენება სერვერული ტიპის კომპიუტერებზე, რომლებზეც სისტემა მუდამ მუშაობს, anacron კი - პერსონალურ კომპიუტერებზე ან ლეპტოპებზე, სადაც სისტემა ითიშება ყოველდღიურად ან კვირების განმავლობაში. cron ამოცანების გაშვებას გეგმავს წუთების დონეზე, anacron - დღეებზე დაბლა ვერ ჩადის. cron დემონს იყენებს ჩვეულებრივი მომხმარებელიც და root-იც, ხოლო anacron მხოლოდ root-ის მიერ გამოიყენება (თუ სპეციალური კონფიგურაცია არ არის შექმნილი ჩვეულებრივი მომხმარებლისთვის).

ფურადღება!

Debian დისტრიბუტივში **cron** ითვალისწინებს დროითი სარტყელის შეცვლის შედეგად გამოწვეულ დროის ცვლილებას. მაგალითად, თუ დროის შეცვლის შემდეგ საათმა წინ გადაიწია და გადაახტა ამოცანის გაშვების დაგეგმილ დროს, ის ერთხელ მაინც შესრულდება დროითი სარტყელის შეცვლის შემდეგ. მეორეს მხრივ, თუ საათმა, პირიქით, უკან გადაიწია, დაგეგმილი ბრძანება კი მანამდე უკვე შესრულებული იყო და საათის გადაწევის გამო კვლავ უწევს ამ ბრძანებას შესრულება, ის ხელმეორედ აღარ შესრულდება.

15.2 at

at ბრძანება უშვებს მომხმარებლის მიერ მითითებულ ბრძანებებს ერთხელ, მოცემულ დროს და არა პერიოდულად, როგორც ამას **cron** აკეთებს. ამგვარად, მომხმარებელს შეუძლია დაგეგმოს ამოცანების გაშვება კონკრეტულ დროს. ამისთვის, აუცილებელია სისტემაში გაშვებული იყოს **atd** დემონი. სინტაქსურად **at**-ს დრო უნდა გადავცეთ არგუმენტად, გავუშვათ და შემდეგ, ხაზ-ხაზ მივუთითოთ ბრძანებების სია. **Ctrl^d** -თი ვამთავრებთ **at** ბრძანებას.

```
achiko@debian:~$ at 22:00 19.07.2018
at> touch /tmp/$(date +%Y%M)
at> <EOT>
job 1 at Thu Jul 19 16:46:00 2018
```

atq ბრძანებით (იგივეა, რაც **at -l**) ვნახულობთ თუ რა ბრძანებებია დაგეგმილი და რა დროს. პირველ სვეტში კი ამოცანის ნომერია მოცემული.

```
achiko@debian:~$ atq
1 Thu Jul 19 22:00:00 2018 a achiko
```

უკვე დაგეგმილი ამოცანის გაუქმება შეგვიძლია **atrm** ბრძანებით (იგივეა **at -d**), თუ ის ჯერ არ არის გაშვებული. არგუმენტად მას ამოცანის ნომერი უნდა მივუთითოთ.

```
achiko@debian:~$ atrm 1
achiko@debian:~$ atq
```

ახლა, აღარაფერია დაგეგმილი. **at** ბრძანებას შესაძლებელია, დრო ბევრი სხვადასხვა ფორმატით გადავცეთ. მისაღები ფორმატებია (დავუშვათ მიმდინარე დრო არის 10:00 AM, October 18, 2018):

at ბრძანების დროის ფორმატი

noon	12:00 PM October 18 2018
midnight	12:00 AM October 19 2018
teatime	4:00 PM October 18 2018

tomorrow	10:00 AM October 19 2018
noon tomorrow	12:00 PM October 19 2018
next week	10:00 AM October 25 2018
next monday	10:00 AM October 22 2018
fri	10:00 AM October 19 2018
9:00 AM	9:00 AM October 19 2018
2:30 PM	2:30 PM October 18 2018
1430	2:30 PM October 18 2018
2:30 PM tomorrow	2:30 PM October 19 2018
2:30 PM next month	2:30 PM November 18 2018
2:30 PM Fri	2:30 PM October 19 2018
2:30 PM Oct 21	2:30 PM October 21 2018
2:30 PM 10/21/2018	2:30 PM October 21 2018
2:30 PM 21.10.18	2:30 PM October 21 2018
now + 30 minutes	10:30 AM October 18 2018
now + 1 hour	11:00 AM October 18 2018
now + 2 days	10:00 AM October 20 2018
4 PM + 2 days	4:00 PM October 20 2018
now + 3 weeks	10:00 AM November 8 2018
now + 4 months	12:00 PM October 18 2018
now + 5 years	10:00 AM October 18 2023

მისაღები დროის ფორმები მოცემულია /usr/share/doc/at/timespec ფაილში.

at-ის გამოყენების უფლებებიც, cron-ის მსგავსად, /etc/at.allow და /etc/at.deny ფაილებით რეგულირდება. თუ /etc/at.allow არსებობს, მხოლოდ მასში ჩაწერილ მომხმარებლებს აქვთ მისი გამოყენების უფლება, დანარჩენებს - არა. თუ ის არ არსებობს, მაშინ მოიძებნება /etc/at.deny ფაილი, და მისი არსებობის შემთხვევაში მასში ჩაწერილ მომხმარებლებს ეკრძალებათ at-ის გამოყენება, დანარჩენს კი - არა. თუ არცერთი ფაილი არ არსებობს, მხოლოდ root რჩება ეს პრივილეგია.

batch ბრძანება (იგივე at -b) შეასრულებს მითითებულ ბრძანებას მაშინ, როდესაც სისტემის დატვირთვა 1.5-ზე ნაკლებია. შესაბამისად, დროის მითითება არგუმენტად აღარ არის საჭირო.

```
achiko@debian:~$ batch
warning: commands will be executed using /bin/sh
at> touch /tmp/$(date +%Y%Y)
at> <EOT>
```

job 19 at Thu Jul 19 17:18:00 2018

გასაშვებად დაგეგმილი ბრძანებები განთავსდება /var/spool/cron/atjobs/
დირექტორიაში.

ნაწილი II

Shell-ის სკრიფტები

თავი 16

შელის სკრიფტები

ჩვენს პირველ ნაწილში ჩვენ არაერთი ბრძანება განვიხილეთ (წინ უფრო მეტი გველის). დავრწმუნდით იმაშიც, რომ ბევრი ამოცანის გადაწყვეტა გაცილებით იოლია ბრძანებათა საზიდან. რაც უფრო მეტს ვსწავლობთ ბრძანებათა საზში, მით მეტ პოტენციალს ვხედავთ მასში. წარმოვიდგინოთ, რომ შეგვიძლია რამდენიმე ბრძანება გავაერთიანოთ შელში და ისე გავუშვათ. სწორედ ასე იქმნება შელის სკრიფტი. ის უბრალო ტექსტური ფაილია, რომელშიც საზ-საზ არის ჩაწერილი გასაშვები ბრძანებები. ყველაფერი, რისი გაშვებაც ბრძანებათა საზიდან შეგვიძლია, შესაძლებელია სკრიპტიდანაც გავუშვათ. ასეთი მიდგომით მინიმუმ დროს დავზოგავთ, რაც ძალიან მნიშვნელოვანია. თუმცა, ქვემოთ ვნახავთ, რომ შელის სკრიფტები კიდევ უფრო მეტ შესაძლებლობასა და მოქნილობას გვთავაზობს.

16.1 ინტერპრეტატორები

როდესაც Shell-ს ვახსენებთ, როგორც წესი, ვგულისხმობთ ინტერპრეტატორს, პროგრამას, რომელიც ასრულებს ჩვენ მიერ აკრეფილ ბრძანებებს. თუმცა შელი, სინამდვილეში, უფრო მეტია, ვიდრე ბრძანებათა საზის ინტერპრეტატორი. ის, თავის მხრივ, დაპროგრამების ენაცაა (სკრიფტების ენა) საკუთარი სტრუქტურით, ციკლებით, ცვლადებით, ფუნქციებით და ა.შ. ლინუქსის უმრავლეს დისტრიბუტივში, ისევე როგორც Apple's macOS-ში, ნაგულისხმევი მნიშვნელობის შელი bash-ით არის წარმოდგენილი.

Unix-ის ოჯახის სისტემებში პირველი shell კენ ტომპსონმა (Ken Thompson) შექმნა Bell Labs-ში და ის, ძირითადად, მე-7 ვერსიამდელ Unix-ებში გამოიყენებოდა. ყველაზე გავრცელებული და პოპულარული shell კი საწყის პერიოდში Bourne shell და C shell იყო.

Bourne shell (sh) სტეფან ბურნმა (Stephen Bourne) შექმნა Bell Labs-ში (სინამდვილეში ის გადაწერილი ტომპსონის შელია, რომელიც დამატებითი ფუნქციებითაა აღიჭურვილი) და მე-7 ვერსიის Unix-ში გამოჩნდა. C shell (csh, მისი გაუმჯობესებული ვერსია tcsh) ბილ ჯოიმ კალიფორნიის უნივერსიტეტში 1970 წელს შექმნა და ის ფართოდ გამოიყენებოდა BSD სისტემებში. სახელი დაპროგრამების C ენის სინტაქსთან მსგავსების გამო დაერქვა. ცნობილი შელებია ასევე: Korn shell (ksh), Z shell (zsh), Almquist shell (ash) და Friendly interactive shell (fish).

Bash ბრაიან ფოქსმა (Brian Fox) 1989 წელს GNU პროექტისთვის დაწერა. Bash დასახელება გამომდინარეობს „Bourne Again SHell“. ამ აკრონიმში ორი იდეაა ჩადებული: პირველი - Bash Bourne shell-ის თავისუფალი პროგრამული ჩანაცვლება და მეორე - მასში შელის თავიდან დაბადების კონცეფციაა ნაგულისხმევი. Bash სრულიად თავსებადია POSIX სისტემასთან. მისი ფუნქციონალი გამდიდრებულია sh, csh და ksh ინტერპრეტატორებიდან.

16.2 პირველი shell სკრიფტი

Unix-ის მსგავს სისტემებში დღემდე აგრძელებენ `/bin/sh` გამშვების გამოყენებას და, რადგან სხვადასხვა დასახელებით შელის ბევრი ვარიანტი არსებობს, `/bin/sh` წარმოადგენს ბმულს სასურველი შელისკენ. სწორედ ამიტომ, ხშირად `bash`-ზე დაწერილ სკრიფტებს shell სკრიფტებსაც უწოდებენ.

`bash` სკრიფტებისთვის `.sh` გაფართოების გამოყენებაა მიღებული. ფაილების სიის გამოტანისას ასე უფრო ალქმადი იქნება, რომ საქმე შელის სკრიფტს ეხება. თუმცა, გაფართოება აუცილებელი ფაქტორი არ არის, მის გარეშეც შეგვიძლია სკრიფტი დავასათაუროთ. არ დავარღვიოთ „ტრადიცია“ და ჩვენი პირველი შელ სკრიფტით „Hello World!“ გამოვიტანოთ ეკრანზე. გავხსნათ `script1.sh` ფაილი ერთ-ერთ ტექსტური რედაქტორში და ავკრიფოთ:

```
#!/bin/bash
# ჩემი პირველი სკრიფტი

echo "Hello World!"
```

როგორც წესი, სკრიფტის პირველ ხაზზე იმ შელის გამშვები ფაილის სრულ გზას უთითებენ, რომლითაც გვსურს, რომ ქვემოთ აკრეფილი ბრძანებები შესრულდეს, ანუ - ინტერპრეტატორს. მის წინ კი „#!“ ასო-ნიშნები აუცილებლად უნდა დავწეროთ. მას Shebang¹-ს უწოდებენ და სწორედ მას მოსდევს იმ პროგრამის სახელი, რომელიც ინტერპრეტაციას უკეთებს ჩვენს სკრიფტს. ჩვენს შემთხვევაში, ეს არის „`/bin/bash`“. შესაბამისად, ჩვენი სკრიფტი არის `bash` სკრიფტი. ამ მექანიზმს იყენებენ სკრიფტების სხვა ენებიც, როგორიცაა `perl`, `awk`, `Tcl`, `Tk` და `python`. შემდეგი ხაზი არის კომენტარი. ყველაფერი, რაც „#“ სიმბოლოს შემდეგ არის ჩაწერილი (Shebang-ის გარდა), შესრულებისას უგულებელყოფილია შელის მიერ. როგორც წესი, პროგრამისტები კომენტარს კოდის აღსაწერად იყენებენ.

შემდეგი ეტაპი სკრიფტზე გაშვების უფლების მინიჭებაა, `chmod` ბრძანებით:

```
achiko@debian:~$ chmod u+x script1.sh
```

სკრიფტის გაშვება კი ასე ხდება:

```
achiko@debian:~$ ./script1.sh
Hello World!
```

ალბათ, დაგებადებათ კითხვა - რა საჭიროა „./“-ს მიწერა სკრიფტის წინ? ჩვენ ზომ მიჩვეულები ვართ, რომ, როდესაც რომელიმე ბრძანებას ვუშვებთ, მხოლოდ მის სახელს ვკრეფთ ბრძანებათა ხაზზე. გავიხსენოთ `PATH` გარემოს ცვლადის დანიშნულება და ყველაფერი თავის ადგილზე დადგება. როგორც ვიცით, `PATH` ცვლადში დირექტორიების სიაა მოცემული. ბრძანების გაშვებისას ჩვენი შელი სწორედ ამ ცვლადში მოცემულ დირექტორიებში ეძებს ამ ბრძანებას და მხოლოდ მისი პოვნის შემთხვევაში შეძლებს მის

¹სახელი Shebang გამომდინარეობს ორი სიტყვის, Sharp და bang, არასწორად წარმოთქმულის ჩაწერით. Sharp როგორც დიეზი მუსიკაში და bang როგორც ძახილი ნიშანი (1950-იან წლებში ასეც აღნიშნავდნენ ჩანაწერებში). მეორე ვერსიით sh მოდის სიტყვა shell-დან. Shebang-ს აგრეთვე უწოდებენ: sha-bang, hashbang, pound-bang ან hash-pling.

გაშვებას. სხვა შემთხვევაში კი ბრძანების ან სკრიფტის გასაშვებად აუცილებელია მისი აბსოლუტური თუ ფარდობითი გზა ჩავწეროთ. ისიც გავიხსენოთ, რომ „.“ მიმდინარე დირექტორიას აღნიშნავს და ნათელი მოეფინება ამ ჩანაწერს. „./script1.sh“ ნიშნავს მიმდინარე დირექტორიაში არესებულ „script1.sh“ ფაილს. ანუ, სკრიფტს ვუშვებთ მისი ფარდობითი გზის მოცემით. აბსოლუტური გზის გამოყენებით, ასეც შეგვიძლია გავუშვათ ჩვენი სკრიფტი:

```
achiko@debian:~$ pwd
/home/achiko
achiko@debian:~$ /home/achiko/script1.sh
Hello World!
```

თუ გვსურს, რომ პირდაპირ, სახელის მითითებით შევძლოთ სკრიფტის გაშვება, მაშინ ჯობს ჩვენი სკრიფტები ერთ-ერთ დირექტორიაში მოვათავსოთ და ამ დირექტორიის გზა PATH ცვლადს დავამატოთ. ამისთვის ხშირად პირად დირექტორიაში ქმნიან bin დასახელების დირექტორიას და მის სრულ გზას ამატებენ PATH ცვლადში.

რადგან სკრიფტის გაშვების სხვასახვა ხერხებზე ვსაუბრობთ, ისიც ვიკითხოთ - რამდენად არის აუცილებელი სკრიფტში Shebang-ით ინტერპრეტატორის მითითება?

თუ bash-ის სკრიფტი გვაქვს და ინტერპრეტატორად ტერმინალში bash არის გაშვებული, მაშინ არ არის აუცილებელი სკრიფტში „პირველი ხაზის“ განსაზღვრა. თუმცა, არ დაგვავიწყდეს, რომ სკრიფტი Linux/Unix სისტემებში არის პორტაბული. ის შეგიძლიათ ერთი სისტემიდან მეორეში გადაიტანოთ და იქ გაუშვათ. ამისთვის კი აუცილებელია, სხვა სისტემაშიც იგივე შელი დაგვხდეთ. წინააღმდეგ შემთხვევაში სკრიფტი არ გაეშვება. ამ ყველაფრის თავიდან ასაცილებლად, მიზანშეწონილია სკრიფტში გამშვები ინტერპრეტატორის მითითება!

16.3 გამოსახულების შეფასება

სკრიფტში ბრძანებების შესრულებამდე, ზოგჯერ საჭიროა გარკვეული გამოსახულების, პირობის შეფასება: არსებობს თუ არა ესა თუ ის ფაილი, A ცვლადის მნიშვნელობა მეტია თუ არა B ცვლადის მნიშვნელობაზე და ა.შ.

ბრძანება **test** სწორედ ამ მიზნებისთვის გამოიყენება. ის აფასებს პარამეტრად გადაცემულ გამოსახულებას და გამოაქვს დასკვნა ჭეშმარიტია თუ მცდარი ეს გამოსახულება. ტექნიკურად ამის გაგება დასრულების კოდის საშუალებით არის შესაძლებელი. ჭეშმარიტი პასუხის შემთხვევაში, დასრულების კოდს მიენიჭება რიცხვი 0, ხოლო სხვა შემთხვევაში 0-სგან განსხვავებული მნიშვნელობა. თუ **test** ბრძანებას პარამეტრს არ გადავცემთ, ამ შემთხვევაშიც, დასრულების კოდი ნულისგან განსხვავებული იქნება. ზოგადად, **test** ბრძანების მსგავსად, ნებისმიერ ბრძანებას აქვს საკუთარი დასრულების კოდი. სწორედ ამ კოდის მნიშვნელობით ხდება სისტემა რამდენად წარმატებით შესრულდა ბრძანება. მისი წარმატებით შესრულების შემდეგ კოდი ყოველთვის 0-ია, ხოლო წარუმატებლობის შემთხვევაში კი 0-სგან განსხვავებული. დასრულების კოდის მნიშვნელობა ნატურალურ რიცხვია და ის [0-255] ინტერვალში ვარირებს. სხვადასხვა არანულოვანი მნიშვნელობა შეცდომის განსხვავებულ ტიპზე მიუთითებს. ბრძანების დასრულების ეს სტატუსი ყოველთვის ინახება ცვლადში დასახელებით „?“.

```
achiko@debian:~$ ls Tux.jpg
Tux.jpg
```

```

achiko@debian:~$ echo $?
0
achiko@debian:~$ ls Tux.jpgggg
ls: cannot access 'Tux.jpgggg': No such file or directory
achiko@debian:~$ echo $?
2

```

ამ მაგალითის პირველ შემთხვევაში ფაილი Tux.jpg ნამდვილად არსებობს და დასრულების კოდი 0-ია, ხოლო მეორე შემთხვევაში Tux.jpgggg ფაილი არ არსებობს და მისი დასრულების კოდი 0-სგან განსხვავდება, კერძოდ, მისი მნიშვნელობაა 2. ჩვენთვის უკვე კარგად ნაცნობი პირობითი ოპერატორები სწორედ ბრძანებების დასრულების კოდებს იყენებენ იმის გასაგებად, ბრძანება წარმატებით დასრულდა თუ არა. თვალსაჩინოებისათვის შევადგინოთ შემდეგნაირი ბრძანება: თუ ეს ფაილი არსებობს, მაშინ ერთი ბრძანება შესრულდეს, ხოლო თუ არ არსებობს - მეორე ბრძანება.

```

achiko@debian:~$ ls Tux.jpg && echo "არსებობს" || echo "არ არსებობს"
Tux.jpg
არსებობს
achiko@debian:~$ ls Tux.jpgggg && echo "არსებობს" || echo "არ არსებობს"
ls: cannot access 'Tux.jpgggg': No such file or directory
არ არსებობს

```

შეგვიძლია დასრულების კოდის მნიშვნელობებიც გამოვიტანოთ და ისე დავრწმუნდეთ ზემოთ გაშვებული ბრძანებების მართებულობაში, ასე:

```

achiko@debian:~$ ls Tux.jpg && echo "დასრულების კოდი არის $?" || echo
"დასრულების კოდი არის $?"
Tux.jpg
დასრულების კოდი არის 0
achiko@debian:~$ ls Tux.jpgggg && echo "დასრულების კოდი არის $?" || echo
"დასრულების კოდი არის $?"
ls: cannot access 'Tux.jpgggg': No such file or directory
დასრულების კოდი არის 2

```

ბრძანების წარმატებით ან წარუმატებლად დასრულების სიმულაციისთვის შელში არსებობს წინასწარ შექმნილი ორი მარტივი ბრძანება: **true** და **false**. **true** ყოველთვის წარმატებით სრულდება, ხოლო **false** ყოველთვის წარუმატებლად.

```

achiko@debian:~$ true; echo $?
0
achiko@debian:~$ false; echo $?
1

```

ეს ბრძანებები ხშირად გამოსადევია იმის შესამოწმებლად, თუ რამდენად სწორად მუშაობენ დიდი, შედგენილი ბრძანების ცალკეული ფრაგმენტები.

ახლა, მოდით, დავუბრუნდეთ **test** ბრძანებას. მისი სინტაქსი შემდეგნაირია:

test Expression

ის აფასებს **Expression** გამოსახულებას - ჭეშმარიტია ის, თუ მცდარი. მაგალითად, გავუშვათ შემდეგი ბრძანება:

```
achiko@debian:~$ test -f Tux.jpg
achiko@debian:~$
```

ამ ბრძანებით გამოწმებთ არსებობს თუ არა `Tux.jpg` და, ამავდროულად, არის თუ არა ის ჩვეულებრივი ტიპის ფაილი (და არა, მაგალითად, დირექტორია). ეს ბრძანება შეამოწმებს ამ პირობას, თუმცა, როგორც უკვე ვნახეთ, შედეგი ეკრანზე არ იბეჭდება. ეს ყველაფერი იმას ნიშნავს, რომ შელმა პასუხი იცის, თუმცა ჩვენ არ გვითხვია ჩვენთვისაც ეცნობებინა ის. თუ გვსურს გავიგოთ შედეგი, შეგვიძლია ან გამოსასვლელი კოდის მნიშვნელობა ვნახოთ, ან პირობითი ოპერატორი გამოვიყენოთ, ასე:

```
achiko@debian:~$ test -f Tux.jpg && echo "არსებობს" || echo "არ არსებობს"
არსებობს
```

test ბრძანება ხელმისაწვდომია [დასახელებითაც და მის სრულიად იდენტურ ვერსიას წარმოადგენს, რომლის სინტაქსიც ასეთია:

[Expression]

```
achiko@debian:~$ [ -f Tux.jpg ]
achiko@debian:~$ [ -f Tux.jpg ] && echo "არსებობს" || echo "არ არსებობს"
არსებობს
```

[- მარჯვენა კვადრატული ფრჩხილი **[** ბრძანების ბოლო არგუმენტია და მისი გამოყენება აუცილებელია. აქვე, არ უნდა დაგვავიწყდეს, რომ, რადგან **[** ბრძანებაა, მის შემდეგ გამოტოვება აუცილებელია. გამოტოვება აუცილებელია, აგრეთვე, **]**-ის წინაც, რადგანაც **]**-იც არგუმენტია. სხვა შემთხვევაში შეცდომა დაფიქსირდება.

```
achiko@debian:~$ [ -f Tux.jpg]
-bash: [: missing `]'
achiko@debian:~$ [-f Tux.jpg ]
-bash: [-f: command not found
```

[ბრძანება წარმოადგენს `bash`-ის შიდა ბრძანებას. შიდა ბრძანება (ე.წ. builtin), სწრაფქმედების მიზნით, პირდაპირ ჩაშენებულია `bash`-ში. გარე ბრძანებებისგან განსხვავებით, რომლებიც სხვადასხვა დირექტორიებშია განთავსებული (აქამდე, ჩვენ სწორედ გარე ბრძანებებთან გვექონდა საქმე), შიდა ბრძანებები პირდაპირ ოპერატიულ მეხსიერებაშია წარმოდგენილი ინტერპრეტატორის გაშვებისთანავე. შესაბამისად, ისინი არც `PATH` ცვლადის შიგთავსზე არიან დამოკიდებულნი და არც შვილი პროცესი იქმნება მათი გაშვებისას. ისტორიულად, **[** და **test** ბრძანებები იყო და ახლაც წარმოადგენენ გარე

ბრძანებებს. თუმცა, სწრაფქმედების მიზნით, ეს ბრძანებები, ამავედროულად, ინტერპრეტატორების უმრავლესობაში ჩაშენებულია იმავე დასახელებით. ასეთი ბრძანების გაშვებისას პრიორიტეტი შიდა ბრძანებას ენიჭება. გარე ბრძანების გასაშვებად კი სრული გზა უნდა გამოვიყენოთ. იმის გაგებაში ბრძანება შიდაა თუ გარე **type** ბრძანება დაგვეხმარება.

```
achiko@debian:~$ type test
test is a shell builtin
achiko@debian:~$ type [
[ is a shell builtin
achiko@debian:~$ type -a test
test is a shell builtin
test is /usr/bin/test
achiko@debian:~$ type -a [
[ is a shell builtin
[ is /usr/bin/[
achiko@debian:~$ type cat
cat is /bin/cat
```

test ბრძანებას კიდევ ბევრი კრიტერიუმით შეუძლია გამოსახულების შემოწმება. მოდით, ვნახოთ ძირითადი მათგანი.

შემოწმებები ფაილებზე	
-e file	ჯეშმარიტია თუ file არსებობს (მნიშვნელობა არ აქვს რომელი ტიპის).
-f file	ჯეშმარიტია თუ file არსებობს და ის ჩვეულებრივი ტიპის ფაილია.
-d file	ჯეშმარიტია თუ file არსებობს და ის დირექტორიაა.
-h file, -L file	ჯეშმარიტია თუ file არსებობს და ის მალსაზმობია.
-r file	ჯეშმარიტია თუ file არსებობს და მიმდინარე პროცესს მისი კითხვის უფლება აქვს.
-w file	ჯეშმარიტია თუ file არსებობს და მიმდინარე პროცესს მასში ჩაწერის უფლება აქვს.
-x file	ჯეშმარიტია თუ file არსებობს და მიმდინარე პროცესისთვის ამ ფაილს x ატრიბუტი გააქტიურებული აქვს.
-u file	ჯეშმარიტია თუ file არსებობს და ამ ფაილზე გააქტიურებულია SUID ბიტი.
-g file	ჯეშმარიტია თუ file არსებობს და ამ ფაილზე გააქტიურებულია SGID ბიტი.
-k file	ჯეშმარიტია თუ file არსებობს და ამ ფაილზე გააქტიურებულია Sticky bit.
-O file	ჯეშმარიტია თუ file არსებობს და თქვენ (როგორც ეფექტური მომხმარებელი) ხართ ამ ფაილის მფლობელი.

<code>-G file</code>	ჰემმარიტია თუ <code>file</code> არსებობს და თქვენ ხართ ამ ფაილის ეფექტური ჯგუფის წევრი.
<code>-s file</code>	ჰემმარიტია თუ <code>file</code> არსებობს და მისი ზომა 0-ზე მეტია.
<code>file1 -nt file2</code>	ჰემმარიტია თუ <code>file1</code> ახალია <code>file2</code> -ზე (საუბარია ფაილების შიგთავსის ბოლო ცვლილების დროზე).
<code>file1 -ot file2</code>	ჰემმარიტია თუ <code>file1</code> ძველია <code>file2</code> -ზე (საუბარია ფაილების შიგთავსის ბოლო ცვლილების დროზე).
<code>file1 -ef file2</code>	ჰემმარიტია თუ <code>file1</code> და <code>file2</code> ერთსა და იმავე კვანძს (inode) იზიარებენ, ამაზე მოგვინებით ვისაუბრებთ. ახლა კი, უფრო გასაგები ენით რომ ვთქვათ, ჰემმარიტია თუ ეს ორი ფაილი ერთმანეთის ბმულია, ანუ ფაილს ორი სხვადასხვა დასახელება აქვს <code>file1</code> -სა და <code>file2</code> -ის სახით.

მაგალითები:

```
achiko@debian:~$ [ -d Documents ] && echo "არსებობს" || echo "არ არსებობს"
არსებობს
```

```
achiko@debian:~$ echo a > 1.txt
achiko@debian:~$ echo b > 2.txt      #ცხადია, 2.txt ფაილი 1.txt-ზე ახალია
achiko@debian:~$
achiko@debian:~$ [ 2.txt -nt 1.txt ] && echo "2.txt უფრო ახალია" || echo
"1.txt უფრო ახალია"
2.txt უფრო ახალია
```

```
achiko@debian:~$ [ -s 1.txt ]; echo $?
0
achiko@debian:~$ du -b 1.txt      #გადავამოწმოთ მისი ზომა
2      1.txt
```

შემოწმებები `string`-ებზე

<code>-n String</code>	ჰემმარიტია თუ <code>String</code> -ის ზომა არ არის ნული. <code>String</code> -ის შემცველი ცვლადის ზომა მაშინ არის ნული, როდესაც ის არცერთ ასო-ნიმანს არ შეიცავს.
<code>String</code>	იგივეა რაც, <code>-n String</code> .
<code>-z String</code>	ჰემმარიტია თუ <code>String</code> -ის ზომა ნულია.
<code>"String1" = "String2"</code>	ჰემმარიტია თუ <code>String1</code> და <code>String2</code> იდენტურია.
<code>"String1" != "String2"</code>	ჰემმარიტია თუ <code>String1</code> და <code>String2</code> იდენტური არ არის.

მაგალითები:

```
achiko@debian:~$ [ "abc" = "abc" ]; echo $?  
0
```

```
achiko@debian:~$ a="abc"  
achiko@debian:~$ [ -n $a ]; echo $?  
0  
achiko@debian:~$ [ $a ]; echo $?  
0  
achiko@debian:~$ [ -z $a ]; echo $?  
1
```

```
achiko@debian:~$ a="abc"  
achiko@debian:~$ b="xyz"  
achiko@debian:~$ [ $a = $b ]; echo $?  
1  
achiko@debian:~$ [ $a=$b ]; echo $?  
0  
achiko@debian:~$ [ $a != $b ]; echo $?  
0
```

დაიმახსოვრეთ!

String-ების შედარებისას ტოლობის ნიშნის წინ და შემდეგ გამოტოვება აუცილებელია! წინააღმდეგ შემთხვევაში შედარება სწორად არ მოხდება, რადგან გამოტოვების გარეშე მიღებული გამოსახულება bash-ის მიერ არქმული იქნება როგორც სხვა, ერთი მთლიანი String.

არითმეტიკული შემოწმებები

<code>\$a -gt \$b</code>	ჰემშარიტია თუ <code>a</code> ცვლადში ჩაწერილი რიცხვითი მნიშვნელობა მეტია <code>b</code> ცვლადში ჩაწერილ რიცხვით მნიშვნელობაზე. ანუ $a > b$
<code>\$a -lt \$b</code>	ჰემშარიტია თუ <code>a</code> ცვლადში ჩაწერილი რიცხვითი მნიშვნელობა ნაკლებია <code>b</code> ცვლადში ჩაწერილ რიცხვით მნიშვნელობაზე. $a < b$.
<code>\$a -ge \$b</code>	ჰემშარიტია თუ <code>a</code> ცვლადში ჩაწერილი რიცხვითი მნიშვნელობა მეტია ან ტოლი <code>b</code> ცვლადში ჩაწერილ რიცხვით მნიშვნელობაზე. $a \geq b$.
<code>\$a -le \$b</code>	ჰემშარიტია თუ <code>a</code> ცვლადში ჩაწერილი რიცხვითი მნიშვნელობა ნაკლებია ან ტოლი <code>b</code> ცვლადში ჩაწერილ რიცხვით მნიშვნელობაზე. $a \leq b$.
<code>\$a -eq \$b</code>	ჰემშარიტია თუ <code>a</code> ცვლადში ჩაწერილი რიცხვითი მნიშვნელობა ტოლია <code>b</code> ცვლადში ჩაწერილ რიცხვით მნიშვნელობაზე. $a = b$.

`$a -ne $b` ჭეშმარიტია თუ `a` ცვლადში ჩაწერილი რიცხვითი მნიშვნელობა არ არის ტოლი `b` ცვლადში ჩაწერილ რიცხვით მნიშვნელობაზე. $a \neq b$.

მაგალითები:

```
achiko@debian:~$ [ 8 -gt 5 ]; echo $?  
0
```

```
achiko@debian:~$ [ 8 -eq 5 ]; echo $?  
1
```

```
achiko@debian:~$ a=39;b=12  
achiko@debian:~$ [ $a -ge $b ]; echo $?  
0
```

`test` ბრძანებით შეგვიძლია, აგრეთვე, რამდენიმე გამოსახულება შევაფასოთ, ასე:

```
test Expression1 -a Expression2
```

ეს გამოსახულება ჭეშმარიტია თუ `Expression1` და `Expression2`, ორივე, ჭეშმარიტია. ან ასე:

```
test Expression1 -o Expression2
```

ეს გამოსახულება კი ჭეშმარიტია მაშინ, როდესაც ერთ-ერთი გამოსახულება მაინც არის ჭეშმარიტი (ან `Expression1`, ან `Expression2`, ან ორივე ერთად).

მაგალითები:

```
achiko@debian:~$ [ 39 -ge 12 -a -f Tux.jpg ]; echo $?  
0  
achiko@debian:~$ [ 39 -ge 12 -a -f Tux.jpgggg ]; echo $?  
1  
achiko@debian:~$ [ 39 -ge 12 -o -f Tux.jpgggg ]; echo $?  
0  
achiko@debian:~$ [ 39 -ge 12 -a -f Tux.jpg -a 3 -lt 8 ]; echo $?  
0  
achiko@debian:~$ [ 39 -ge 12 -a -f Tux.jpgggg -o 3 -lt 8 ]; echo $?  
0
```

ამ მაგალითებში ვნახეთ, თუ როგორ გამოიყენება რამდენიმე გამოსახულების შესაფასებლად `test` ბრძანება `-a` და `-o` ოპერატორებით, თუმცა ეს მეთოდი მიღებული აღარაა. მის ნაცვლად რეკომენდებულია `&&` და `||` ოპერატორების გამოყენება. მათი საშუალებით, როგორც უკვე ვიცით, შეგვიძლია პირობით გადავაბათ რამდენიმე ბრძანება. მოდით, ჩვენც

test ბრძანებებზე ვიმოქმედოთ. შედეგიც იდენტური უნდა იყოს. ვნახოთ ამ ხერხით ჩაწერილი იგივე მაგალითები:

```
achiko@debian:~$ [ 39 -ge 12 ] && [ -f Tux.jpg ]; echo $?
0
achiko@debian:~$ [ 39 -ge 12 ] && [ -f Tux.jpbggg ]; echo $?
1
achiko@debian:~$ [ 39 -ge 12 ] || [ -f Tux.jpbggg ]; echo $?
0
achiko@debian:~$ [ 39 -ge 12 ] && [ -f Tux.jpg ] && [ 3 -lt 8 ]; echo $?
0
achiko@debian:~$ [ 39 -ge 12 ] && [ -f Tux.jpbggg ] || [ 3 -lt 8 ]; echo $?
0
```

[[...]] კონსტრუქცია (გაფართოებული test) bash-ის 2.02 ვერსიიდან მოყოლებული ksh88 შელიდან გადმოიღეს და დანერგეს. იგი ძალიან ჰგავს [...] ბრძანებას, თუმცა მეტი შესაძლებლობების მატარებელია. [[...]] სიტყვა-გასაღებია და არა ბრძანება. სიტყვა-გასაღები არის დარეზერვებული სიტყვა ან ოპერატორი და ის, შიდა ბრძანებისგან განსხვავებით, არა მთლიანი ბრძანება, არამედ ბრძანების სტრუქტურის შემადგენელი ნაწილია. აღსანიშნავია ისიც, რომ სადღესოდ, ეს კონსტრუქცია POSIX ყველა შელის მიერ არ არის მხარდაჭერილი, არამედ მხოლოდ KornShell, Zsh-სა და Bash-ის მიერ. [[სიტყვა-გასაღებს „ახალ“ test-საც უწოდებენ.

```
achiko@debian:~$ type [[
[[ is a shell keyword
```

[[და [-ს სინტაქსურ ჩანაწერებში ძალიან დიდი მსგავსებაა, თუმცა მაინც არის მათ შორის გარკვეული განსხვავებები. იხილეთ შედარებითი ცხრილი მაგალითებით:

შეფასება	[[[მაგალითი
string-ების შედარება	>	> ²	[[a > b]] echo "a არ მოდის b-ს შემდეგ"
	<	< ²	[[az < za]] && echo "az არის za-ის წინ"
	=, ==	=	[[a = a]] && echo "a იდენტურია a-სი"
	!=	!=	[[a != b]] && echo "a არ არის b-ს იდენტური"
რიცხვების შედარება	-gt	-gt	[[5 -gt 10]] echo "5 მეტია 10-ზე"
	-lt	-lt	[[8 -lt 9]] && echo "8 ნაკლებია 9-ზე"
	-ge	-ge	[[3 -ge 3]] && echo "3 მეტია ან ტოლი 3-ის"
	-le	-le	[[3 -le 8]] && echo "3 ნაკლებია ან ტოლი 8-ს"
	-eq	-eq	[[5 -eq 05]] && echo "5 ტოლია 05-ის"
	-ne	-ne	[[6 -ne 20]] && echo "6 არ არის 20-ის ტოლი"
პირობითი შეფასება	&&	-a ³	[[-n \$v && -f \$v]] && echo "\$v არის ფაილი"

		-o ³	[[-b \$v -c \$v]] && echo "\$v არის მონყობილობა"
დაჯგუფება	(...)	\(...\) ³	[[\$v = img* && (\$v = *.png \$v = *.jpg)]] && echo "\$v იწყება img-ით და მთავრდება ან .jpg ან .png-ით"
თანხვედრა	=,	არ	[[\$name = a*]] echo "name არ იწყება 'a' ასო-
	==	აქვს	ნიშნით: \$name"
რეგ.	=~	არ	[[\$(date) =~^Fri ... 13]] && echo "დღეს
გამოსახულება		აქვს	პარასკევია, 13 რიცხვი!"

16.4 გადაწყვეტილების მიღება

ამ თავში ვისაუბრებთ იმაზე, თუ როგორ ხდება გადაწყვეტილების მიღება შელში. ხშირად გვხვდება ისეთი სიტუაცია, როდესაც პირობის შემოწმების შემდეგ ან ერთი გზა უნდა ავირჩიოთ, ან მეორე. პირობის შესამოწმებლად და გადაწყვეტილების მისაღებად, წინა თავში აღნიშნული, მარტივი, პირობითი ოპერატორების გარდა, შელში ორი სრულფასოვანი ჩანაწერი არსებობს. ესენია:

```
if ... fi
```

და

```
case ... esac
```

ორივე, if-ც და case-ც, სიტყვა-გასაღებს წარმოადგენს. განვიხილოთ თითოეული მათგანი დეტალურად.

16.4.1 სიტყვა-გასაღები if

if ... fi ჩანაწერის ვარიაციებია:

ა)

```
if Command;           # თუ Command ბრძანება ჭეშმარიტია
then                  # მაშინ
    cmd1;             # შესრულდეს ბრძანება cmd1
    cmd2;             # შესრულდეს ბრძანება cmd2
    ...
fi                    # if-ს ვხურავთ fi-თი
```

ბ)

```
if Command;           # თუ Command ბრძანება ჭეშმარიტია
then                  # მაშინ
    cmd1;             # შესრულდეს ბრძანება cmd1
```

²ეს არის POSIX სტანდარტის გაფართოება. ზოგი შელი შეიცავს მას, ზოგიც არა.

³მისი გამოყენება აღარ არის რეკომენდებული! მის ნაცვლად მიზანშეწონილია რამდენიმე [ბრძანების გამოყენება.

```

else      ...      # წინააღმდეგ შემთხვევაში
      cmd2;      # შესრულდეს ბრძანება cmd2
      ...
fi      # if-ს ვხუროვთ fi-თი

```

გ)

```

if Command1;      # თუ Command1 ბრძანება ჭეშმარიტია
then      # მაშინ
      cmd1; ...      # შესრულდეს ბრძანება cmd1 ...
elif Command2;      # თუ არა, შემოწმდეს Command2.
then      # თუ ის ჭეშმარიტია, მაშინ
      cmd2; ...      # შესრულდეს ბრძანება cmd2 ...
.
.
.
else      # წინააღმდეგ შემთხვევაში
      cmdN; ...      # შესრულდეს ბრძანება cmdN
fi      # if-ს ვხუროვთ fi-თი

```

ამ სინტაქსური ჩანაწერიდან კარგად ჩანს, რომ `if` ბრძანება არ არის, არამედ სიტყვა-გასაღებია, რადგან მას `fi` სუფიქსი სჭირდება დასასრულად. `if`-ის შემდეგ შესამოწმებელი ბრძანება ნებისმიერი შეიძლება იყოს, მათ შორის ჩვენთვის უკვე კარგად ნაცნობი `test` ბრძანებაც. `if ... fi` სიტყვა-გასაღების კონსტრუქცია, როგორც ვხედავთ, მრავალ საზღვ არის ჩაწერილი. ასეთი ფორმა კოდის უკვე კითხვადობას უწყობს ხელს, განსაკუთრებით მაშინ, როდესაც კოდი საკმაოდ გრძელია. თუმცა, მისი ჩაწერა ერთ საზღვეც არის შესაძლებელი. სწორედ, ამ დროს არის აუცილებელი წერტილ-მძიმეს (;) ბოლოში მიწერა. მრავალ საზიან ჩანაწერში წერტილ-მძიმეს გამოყენება აუცილებელი აღარ არის!

მაგალითები:

```

achiko@debian:~$ vi script.sh
#!/bin/bash

file=/etc/passwd

if [[ -e $file ]]
then
  echo "პაროლების ფაილი არსებობს!"
fi

achiko@debian:~$ chmod u+x script.sh
achiko@debian:~$ ./script.sh
პაროლების ფაილი არსებობს!

```

ეს კოდი ერთ საზღვე ასე ჩაიწერება:

```
achiko@debian:~$ file=/etc/passwd; if [[ -e $file ]]; then echo "პაროლების  
ფაილი არსებობს!"; fi  
პაროლების ფაილი არსებობს!
```

მოვიყვანოთ სხვა მაგალითები:

```
#!/bin/bash  
x=8  
y=17  
  
if [ $x -eq $y ]  
then  
    echo "$x ტოლია $y-ის"  
else  
    echo "$x არ არის $y-ის ტოლი!"  
fi  
  
achiko@debian:~$ ./script_XYZ.sh  
8 არ არის 17-ის ტოლი!
```

```
#!/bin/bash  
x=18  
  
if [ $x -eq 100 ]  
then  
    echo "x 100-ის ტოლია"  
elif [ $x -gt 100 ]  
then  
    echo "x 100-ზე მეტია"  
else  
    echo "x 100-ზე ნაკლებია"  
fi  
  
achiko@debian:~$ ./script_XYZ.sh  
x 100-ზე ნაკლებია
```

if ... fi კონსტრუქციაში თავისი თავის გამოყენებაც არის შესაძლებელი:

```
#!/bin/bash  
file=/bin  
if [ -e $file ]  
then  
    if [ -h $file ]  
    then  
        echo "$file მალსახმობია."  
    elif [ -d $file ]  
    then
```

```

        echo "$file დირექტორიაა."
    elif [ -f $file ]
    then
        echo "$file ჩვეულებრივი ტიპის ფაილია."
    else
        echo "$file სხვა ტიპის ფაილია!"
    fi
else
    echo "$file არ არსებობს!"
fi
achiko@debian:~$ ./script_XYZ.sh
/bin დირექტორიაა.

```

ეს სკრიფტი ერთ ხაზში ასე ჩაიწერება:

```

achiko@debian:~$ file=/bin; if [ -e $file ]; then if [ -h $file ]; then echo
"$file მალსახმოზია."; elif [ -d $file ]; then echo "$file დირექტორიაა.";
elif [ -f $file ]; then echo "$file ჩვეულებრივი ტიპის ფაილია."; else echo
"$file სხვა ტიპის ფაილია!"; fi; else echo "$file არ არსებობს!"; fi
/bin დირექტორიაა.

```

ცალსახაა, რომ სკრიფტის წასაკითხად და მასში ცვლილების შესტანად მისი მრავალ ხაზზე ჩაწერა გაცილებით მოსახერხებელია!

16.4.2 სიტყვა-გასაღები case


if ... elif ... fi კონსტრუქცია შეგვიძლია, ბევრჯერ გამოვიყენოთ, როდესაც მრავალი პირობის შემოწმება გვჭირდება. თუმცა ეს მიდგომა არაა საუკეთესო, მითუმეტეს იმ შემთხვევაში, როდესაც ერთი ცვლადის მნიშვნელობაზეა დამოკიდებული ეს პირობები და ამ ცვლადთან შედარება მრავალჯერ გვიწევს. ასეთ სიტუაციებშია მართებული case ... esac კონსტრუქციის გამოყენება. მისი სინტაქსური ჩანაწერი შემდეგნაირია:

case VARIABLE in	# თუ VARIABLE-ის მნიშვნელობა ემთხვევა:
pattern1)	# pattern1-ს, მაშინ
cmd1 ...	# შესრულდეს ბრძანება cmd1 ...
;;	# პირველი შედარების დასასრული
pattern2)	# pattern2-ს, მაშინ
cmd2 ...	# შესრულდეს ბრძანება cmd2
;;	# მეორე შედარების დასასრული
.	
.	
.	
*)	# წინააღმდეგ შემთხვევაში
cmdN ...	# შესრულდეს ბრძანება cmdN
;;	# ბოლო შედარების დასასრული
esac	# if-ს ვხურავთ fi-თი

pattern-ში უფრო მეტი მოქნილობისთვის შესაძლებელია, გამოვიყენოთ მაგენერირებელი სიმბოლოები. ეს ის სიმბოლოებია, რომლებსაც ფაილის დასახელების ზოგადად ჩაწერაში ვიყენებთ ხოლმე. დახურული ფრჩხილი „)“ **pattern**-ის დასასრულს ნიშნავს. **pattern**-ის სწორი ჩანაწერის მაგალითებია:

Pattern	აღნიშნავს შემდეგ ასოთა წყობას
[[:digit:]]	ერთ ციფრს.
f*[:upper:]	იწყება f-ით და მთავრდება დიდი ასოთი.
*.txt)	იწყება ნებისმიერი ასოთა წყობით და მთავრდება .txt-ით.
???)	ნებისმიერი სამ ასო-ნიშნის წყობა.
*)	ნებისმიერ ასოთა წყობა. მისი ჩაწერა ბოლო Pattern-ად სასარგებლოა. თუ, ზედა არცერთი შემთხვევა არ გვხვდება, იგი ყოველთვის შესრულდება. ფაქტობრივად, ეს Pattern-ის ნაგულისხმევ მნიშვნელობას წარმოადგენს.

ვნახოთ **case** სიტყვა-გასაღების გამოყენების ერთ-ერთი მაგალითი. შევიყვანოთ კლავიატურიდან ერთ ასო-ნიშანი და სკრიფტმა გამოიძნოს ის პატარა ასოა თუ დიდი, ციფრია თუ პუნქტუაციის ნიშანია და ა.შ.

მოდით, ჯერ ვნახოთ, როგორ ხდება კლავიატურიდან მონაცემის შეყვანა შელში. ამისთვის გამოიყენება **read** ბრძანება. ის იღებს კლავიატურიდან შეტანილ მონაცემს და ინახავს არგუმენტად გადაცემულ ცვლადში. კლავიატურიდან მონაცემების შეტანა მანამ გრძელდება, სანამ შეყვანის ლილაკს არ დავაჭერთ (**Enter** , ).

```
achiko@debian:~$ read x
Data
achiko@debian:~$ echo $x
Data
```

თუ **read** ბრძანებას არგუმენტი არ გადავცით, კლავიატურიდან შეტანილ მონაცემს, ნაგულისხმევი მნიშვნელობით, **REPLY** ცვლადში შეინახავს.

```
achiko@debian:~$ read
Data
achiko@debian:~$ echo $REPLY
Data
```

ახლა კი, **case** სიტყვა-გასაღები განვიხილოთ სკრიფტში:

```
#!/bin/bash

echo; echo "შემოიტანეთ ერთი სიმბოლო:"
read Key

case "$Key" in
```

```

[[[:lower:]]])
    echo "პატარა ასო"
;;
[[[:upper:]]])
    echo "დიდი ასო"
;;
[[[:digit:]]])
    echo "ციფრი"
;;
[[[:punct:]]])
    echo "პუნქტუაციის ნიშანი"
;;
[[[:xdigit:]]])
    echo "თექვსმეტობითი თვლის სისტემის ასო-ნიშანი"
;;
*)
    echo "სხვა"
;;
esac

```

```
achiko@debian:~$ ./script_XYZ.sh
```

შემოიტანეთ ერთი სიმბოლო:

h

პატარა ასო

უფრო კომპაქტურად ეს კოდი ასეც ჩაიწერება:

```

#!/bin/bash

echo; echo "შემოიტანეთ ერთი სიმბოლო:"
read Key

case "$Key" in
    [[[:lower:]]]) echo "პატარა ასო";;
    [[[:upper:]]]) echo "დიდი ასო";;
    [[[:digit:]]]) echo "ციფრი";;
    [[[:punct:]]]) echo "პუნქტუაციის ნიშანი";;
    [[[:xdigit:]]]) echo "თექვსმეტობითი სისტემის ასო";;
    *)            echo "სხვა";;
esac

```

ამ მაგალითებიდან ჩანს, რომ თუ ცვლადის მნიშვნელობა ერთ-ერთ შესაძლო მნიშვნელობაზე კრიტერიუმს დამთხვევა, მხოლოდ მისი შესაბამისი ბრძანებები შესრულდება და სიტყვა-გასაღებში ჩამოთვლილი სხვა პირობები აღარ შეამოწმდება. მაგალითად, თუ გამოსაცნობად შევიტანთ „F“ ასო-ნიშანს, სკრიფტი გვეტყვის, რომ ის დიდი ასოა.

```
achiko@debian:~$ ./script_XYZ.sh
```

შემოიტანეთ ერთი სიმბოლო:

F

დიდი ასო

მართალია, „F“ დიდი ასოა, თუმცა ის, ამავდროულად, წარმოადგენს თექვსმეტობითი თვლის სისტემის ასო-ნიშანსაც. ინტერპრეტატორი ამის შემოწმებამდე აღარ მიდის, რადგან მის წინ მოცემული ერთი-ერთი კრიტერიუმი უკვე დაკმაყოფილდა. თუ გვსურს, რომ **case ... esac** კონსტრუქციაში მრავალი კრიტერიუმი შემოწმდეს ერთდროულად, მაშინ მასში „;““-ის ნაცვლად „;;&“ ჩანაწერი უნდა გამოვიყენოთ, ასე:

```
#!/bin/bash
```

```
echo; echo "შემოიტანეთ ერთი სიმბოლო"
```

```
read Key
```

```
case "$Key" in
```

```
  [[[:lower:]]]) echo "პატარა ასო";;&
```

```
  [[[:upper:]]]) echo "დიდი ასო";;&
```

```
  [[[:digit:]]]) echo "ციფრი";;&
```

```
  [[[:punct:]]]) echo "პუნქტუაციის ნიშანი";;&
```

```
  [[[:xdigit:]]]) echo "თექვსმეტობითი სისტემის ასო-ნიშანი";;&
```

```
esac
```

```
achiko@debian:~$ ./script_XYZ.sh
```

შემოიტანეთ ერთი სიმბოლო:

F

დიდი ასო

თექვსმეტობითი თვლის სისტემის ასო-ნიშანი

ამჯერად, **case** ყველა ჩამოთვლილ კრიტერიუმს შეადარებს ცვლადს.

case სტრუქტურაში ცვლადის მნიშვნელობის ერთხელ შემოწმებისას შესაძლებელია, აგრეთვე, **pattern**-ის ალტერნატივები გადავცეთ „|“ სიმბოლოს გამოყენებით, ასე:

```
#!/bin/bash
```

```
NOW=$(date +"%a")
```

```
case $NOW in
```

```
  Mon|Tue|Wed|Thu|Fri)
```

```
    echo "ორშაბათი-პარასკევი";;
```

```
  Sat|Sun)
```

```
    echo "შაბათი ან კვირა";;
```

```
*)
```

```
  echo "რეგ. პარამეტრები ინგლისურ ენაზე არაა ან სხვა!";;
```

```
esac
```

სანამ სხვა საკითხზე გადავალთ, მოდით, `read` ბრძანებას დავუბრუნდეთ და მისი გამოყენების სხვა მაგალითები ვნახოთ.


მას შეუძლია მონაცემები ერთზე მეტ ცვლადში გადაანაწილოს. მართლაც, თუ კლავიატურიდან შეტანილი მონაცემი გამოტოვებებს შეიცავს, მაშინ შესაძლებელია, `read` ბრძანებას რამდენიმე ცვლადი გადავცეთ არგუმენტებად.

```
achiko@debian:~$ read a b c
Data1 Data2 Data3
achiko@debian:~$ echo $a
Data1
achiko@debian:~$ echo $b
Data2
achiko@debian:~$ echo $c
Data3
```


```
achiko@debian:~$ read x y
Data11 Data22 Data33
achiko@debian:~$ echo $x
Data11
achiko@debian:~$ echo $y
Data22 Data33
```

-p ოფციით შესაძლებელია, გარკვეული ტექსტი წავუმძღვართ კლავიატურიდან მონაცემის შეტანას, ასე:

```
achiko@debian:~$ read -p "შემოიტანე შენი სახელი: "x
შემოიტანე შენი სახელი: Achiko
achiko@debian:~$ echo $x
Achiko
```

-n ოფციით მხოლოდ მოცემული რაოდენობის სიმბოლო შეგვყავს. მართალია, შესაძლებელია, უფრო მეტი სიმბოლო ავკრიფოთ კლავიატურაზე, თუმცა `read` ავტომატურად შეწყვეტს ზღვარს მიღმა მითითებული სიმბოლოების მიღებას.  ღილაკზე დაჭერა საჭირო არ არის.

```
achiko@debian:~$ read -n3 -p "შემოიტანე სამნიშნა რიცხვი: "x
შემოიტანე სამნიშნა რიცხვი: 123achiko@debian:~$ echo $x
123
```

ვცადოთ და მეტი სიმბოლო შევიყვანოთ, მაგალითად 12345. ასე, დანარჩენი სიმბოლოები (ჩვენს შემთხვევაში - 45) შელის მოსაწვევზე ჩაიწერება და თუ  ღილაკსაც დავაჭერთ, ის ბრძანებად აღიქმება:

```
achiko@debian:~$ read -n3 -p "შემოიტანე სამნიშნა რიცხვი: "x
შემოიტანე სამნიშნა რიცხვი: 123achiko@debian:~$ 45
-bash: 45: command not found
```

როგორც წინა მაგალითებში ვნახეთ, კლავიატურიდან ჩვენ მიერ შეტანილი მონაცემი, ვიზუალურად - ეკრანზე იბეჭდება. -s ოფციით კი, ეს მონაცემი უჩინარი ხდება და ეკრანზე აღარეფერი ჩანს. ეს ოფცია, ძირითადად, გამოიყენება კლავიატურიდან პაროლის შეტანის დროს.

```
achiko@debian:~$ read -s -p "შემოიტანე პაროლი: "x
შემოიტანე პაროლი: achiko@debian:~$
achiko@debian:~$ echo $x
Password
```

ამ მაგალითში პაროლად სიტყვა Password ავიღეთ. მისი კლავიატურიდან შეტანისას, როგორც ვნახეთ, ეკრანზე აკრეფილი პაროლი არ გამოჩნდა.

16.5 ციკლი

შელში, როგორც, ზოგადად, დაპროგრამების ყველა ენაში, ციკლი გამოიყენება გარკვეული ბრძანებების დაჯგუფების გამეორებისთვის, იტერაციისთვის. ციკლის ჩასაწერად არსებობს რამდენიმე საშუალება: **for**, **while** და **until** სიტყვა-გასაღებები.

განვიხილოთ, დეტალურად თითოეული მათგანი.

16.5.1 ციკლი for

for-ის სინტაქსი ასეთია:

```
for Variable in list;
do
    cmd1;
    cmd2;
    ...
done
```

for ციკლი ელემენტებს სათითაოდ იღებს **list**-დან, ანიჭებს **Variable** ცვლადს და თითოეულისთვის თანმიმდევრობით ასრულებს **do ... done** სტრუქტურაში მოცემულ ბრძანებებს. იტერაციების რაოდენობა დამოკიდებულია იმაზე, თუ რამდენ ელემენტს გადავცემთ **list**-ში. იგი შეიძლება იყოს გამოტოვებით ან ახალი ხაზით გამოყოფილი ნებისმიერი მონაცემის ერთობლიობა, ბრძანების შედეგი ან ცვლადის მნიშვნელობა. **list**-ში, აგრეთვე, შესაძლებელია მაგენერირებული სიმბოლოების გამოყენება. სადემონსტრაციოდ, დავწეროთ სკრიფტი, რომელიც გამოიტანს **list**-ში ჩაწერილ **a**, **b** და **c** მონაცემს და ვნახოთ სინტაქსური ჩანაწერის სხვადასხვა ვარიაცია:

```
#!/bin/bash
for i in a b c
```

```
do
    echo $i
done
```

```
#!/bin/bash
x="a b c"
for i in $x
do
    echo $i
done
```

```
#!/bin/bash
x="a
b
c"
for i in $x
do
    echo $i
done
```

```
#!/bin/bash
for i in $(echo a b c)
do
    echo $i
done
```

```
#!/bin/bash
for i in {a..c}
do
    echo $i
done
```

შედეგი ყველა შემთხვევაში ერთი იქნება:

```
achiko@debian:~$ ./script_XYZ.sh
a
b
c
```

ეს მაგალითი ერთ ხაზში ასე ჩაიწერება:

```
achiko@debian:~$ for i in a b c; do echo $i; done
a
b
c
```

ერთ ხაზში ჩაწერისას წერტილ-მიძიმეს გამოყენება აუცილებელია! როგორც ეს if-ის შემთხვევაში იყო.

როდესაც list-ში მაგენერირებული სიმბოლო შედის, მის ქვეშ იგულისხმება ფაილების ერთობლიობა. ეს ის ფაილებია, რომლებიც განთავსებულია მიმდინარე დირექტორიაში, საიდანაც ვუშვებთ სკრიფტს.

```
#!/bin/bash
for i in *
do
    echo $i
done
achiko@debian:~$ ./script_XYZ.sh
...
Desktop
distros.txt
Documents
Downloads
GPL-3
...
```

მოდით, ახლა ამ ფაილების შესახებ დეტალური ინფორმაცია გამოვიტანოთ:

```
#!/bin/bash
for i in *
do
    ls -ld $i
done
achiko@debian:~$ ./script_XYZ.sh
...
drwxr-xr-x 2 achiko achiko 4096 Oct 6 2017 Desktop
-rw-r--r-- 1 achiko achiko 263 Jun 11 2018 distros.txt
drwxr-xr-x 2 achiko achiko 4096 Oct 6 2017 Documents
drwxr-xr-x 2 achiko achiko 4096 Oct 10 2017 Downloads
-rw-r--r-- 1 achiko achiko 35147 Jun 26 2018 GPL-3
...
```

ეს მარტივი სკრიფტი, ერთი შეხედვით, სწორად მუშაობს. მოდით ახლა, ერთი ახალი ფაილი შევქმნათ, რომლის დასახელებამაც გამოტოვება შედის და ვნახოთ, როგორ იმუშავებს ჩვენი სკრიფტი.

```

achiko@debian:~$ touch "erti ori"
achiko@debian:~$ ./script_XYZ.sh
...
drwxr-xr-x 2 achiko achiko 4096 Oct 10 2017 Downloads
ls: cannot access 'erti': No such file or directory
ls: cannot access 'ori': No such file or directory
-rw-r--r-- 1 achiko achiko 35147 Jun 26 2018 GPL-3
...

```

ეკრანზე შეცდომის შეტყობინება გამოვიდა, რომელიც გვეუბნება, რომ არ არსებობს არც ფაილი „erti“ და არც ფაილი „ori“. მოლით, დეტალურად მივყვეთ და გავიგოთ რა, როგორ ხდება. „*“-ს მითითებით `for`-ს გადავცით მიმდინარე დირექტორიაში არსებული ფაილების სია. მათი დასახელებები გამოტოვებითაა მოცემული. `for` პირველ იტერაციაზე `i` ცვლადს პირველი ფაილის სახელს მიანიჭებს და `ls -ld` ბრძანებას გადასცემს არგუმენტად, მეორე იტერაციაზე `i` ცვლადს მეორე ფაილის სახელი მიენიჭება და ა.შ. იმ იტერაციაზე, როდესაც `i` ცვლადში `erti ori` ფაილის სახელი წერია, ის `ls -ld` ბრძანებას გადაეცემა. ეს ბრძანება მას ორ სხვადასხვა არგუმენტად აღიქვამს. ასეთი დასახელების ცალკე ფაილები კი არ გაგვანია. შესაბამისად, შედეგიც ლოგიკურია. ამ ყველაფრის თავიდან ასაცილებლად, შეგვიძლია, ცვლადი ბრჭყალებში ჩავსვათ, მარტივად. ასე:

```

#!/bin/bash
for i in *
do
    ls -ld "$i"
done
achiko@debian:~$ ./script_XYZ.sh
...
drwxr-xr-x 2 achiko achiko 4096 Oct 10 2017 Downloads
-rw-r--r-- 1 achiko achiko 0 May 30 12:04 erti ori
-rw-r--r-- 1 achiko achiko 35147 Jun 26 2018 GPL-3
...

```

ასე უკვე, ყველაფერი თავის ადგილზე დადგა.

დაიმახსოვრეთ!

თუ ფაილების დასახელება ცვლადში გაქვთ შენახული, ამ ცვლადის გამოყენებისას, ის ყოველთვის ბრჭყალებში ჩასვით! ამგვარად, ფაილის დასახელების არაანბანური ასო-ნიშანი მართებულად იქნება აღქმული ინტერპრეტატორის მიერ.

`for`-ს ციკლის ჩანაწერს მეორე ფორმაც აქვს. ისეთი, რომელსაც დაპროგრამების C ენაში იყენებენ ხოლმე. მას სამი გამოსახულება გადაეცემა (დაწყება, პირობა და განახლება). ეს ფორმა სინტაქსურად ასე გამოიყურება:

```

for ((exp1; exp2; exp3));
do

```



```

cmd1;
cmd2;
...
done

```

მაგალითებით მარტივად მივხვდებით ამ გამოსახულებების არსს:

```

#!/bin/bash
for ((i=1;i<=20;i=i+2)) # i-ვანიჭებთ 1-ს. სანამდე i<=20, ამ ცვლადს
do                      # 2-ით ვზრდით ყოველი იტერაციის დროს
    echo -n "$i "
done                    # როგორც კი i>20 for ციკლი დასრულდება და
echo                   # შესრულდება done-ის შემდეგ ჩაწერილი ბრძანება
achiko@debian:~$ ./script_XYZ.sh
1 3 5 7 9 11 13 15 17 19

```

```

#!/bin/bash
read -p "შემოიტანე რიცხვი 1-დან 10-მდე: " a
for (( i=1; i<=$a; i++ ))
do
    for (( j=1; j<=$i; j++ ))
    do
        echo -n $j
    done
    echo
done
achiko@debian:~$ ./script_XYZ.sh
შემოიტანე რიცხვი 1-დან 10-მდე: 5
1
22
333
4444
55555

```

16.5.2 break

ციკლის წერისას, ხშირ შემთხვევებში, სასარგებლოა **break** ბრძანების გამოყენება. ის იხმარება ციკლის შიგნით და ახდენს ციკლიდან გამოსვლას **break**-მდე ჩაწერილი ინსტრუქციების შესრულების შემდეგ. ვნახოთ მისი გამოყენების მარტივი მაგალითი:

```

#!/bin/bash

for outerloop in 1 2 3 4 5
do
    echo -n "გარე ციკლი $outerloop => შიდა: "

```

```

for innerloop in 1 2 3 4 5
do
    echo -n "$innerloop "
    if [ "$innerloop" -eq 3 ]
    then
        break
    fi
done
echo
done
exit 0

```

achiko@debian:~\$./script_XYZ.sh

გარე ციკლი 1 => შიდა: 1 2 3
 გარე ციკლი 2 => შიდა: 1 2 3
 გარე ციკლი 3 => შიდა: 1 2 3
 გარე ციკლი 4 => შიდა: 1 2 3
 გარე ციკლი 5 => შიდა: 1 2 3

მისი ზოგადი სინტაქსი ასეთია: **break N**, სადაც **N** დადებითი რიცხვია. როდესაც ციკლში სხვა ციკლი რამდენჯერმე გვაქვს ჩაშენებული, სასარგებლოა რიცხვის არგუმენტად გადაცემა და ის მიუთითებს, თუ რამდენი ციკლიდან უნდა მოხდეს გამოსვლა. ვნახოთ მაგალითი:

```

#!/bin/bash

for outerloop in 1 2 3 4 5
do
    echo -n "გარე ციკლი $outerloop => შიდა: "
    for innerloop in 1 2 3 4 5
    do
        echo -n "$innerloop "
        if [ "$innerloop" -eq 3 ]
        then
            break 2
        fi
    done
    echo
done
exit 0

```

achiko@debian:~\$./script_XYZ.sh

გარე ციკლი 1 => შიდა: 1 2 3

იმისთვის, რომ ვნახოთ ყოველ იტერაციაზე რა ცვლადს რა მნიშვნელობა ენიჭება და სკრიფტი რა დროს რა ბრძანებას ასრულებს, შეგვიძლია, გამოვიყენოთ პროგრამის გამმართველი (ე.წ. debugger). გამმართველს ზშირად კოდში შეცდომების აღმოსაჩენად და მათ აღმოსაფხვრელად იყენებენ. ამ ფორმით **bash**-ის სკრიფტების გაშვება **bash -x** ბრძანებით ხორციელდება. მოდით, ჩვენი მაგალითი გავუშვათ გამმართველით.

```
#!/bin/bash

for outerloop in 1 2 3 4 5
do
    echo -n "გარე ციკლი $outerloop => შიდა: "
    for innerloop in 1 2 3 4 5
    do
        echo -n "$innerloop "
        if [ "$innerloop" -eq 3 ]
        then
            break
        fi
    done
    echo
done
exit 0

achiko@debian:~$ bash -x script_XYZ.sh
+ for outerloop in 1 2 3 4 5
+ echo -n 'გარე ციკლი 1 => შიდა: '
გარე ციკლი 1 => შიდა: + for innerloop in 1 2 3 4 5
+ echo -n '1 '
1 + '[' 1 -eq 3 ']'
+ for innerloop in 1 2 3 4 5
+ echo -n '2 '
2 + '[' 2 -eq 3 ']'
+ for innerloop in 1 2 3 4 5
+ echo -n '3 '
3 + '[' 3 -eq 3 ']'
+ break 2
+ echo

+ exit 0
```

დეტალურად გამოჩნდა თუ რომელ ბრძანებას, როდის უშვებს ინტერპრეტატორი და რა შედეგი გამოდის ეკრანზე ყოველ ნაბიჯზე.

დაიმახსოვრეთ!

break 1 იგივე შედეგს გვაძლევს, რასაც **break** ბრძანება.

16.5.3 continue

ციკლის წერისას **break** ბრძანების გარდა, ხშირად გამოყენებენ **continue** ბრძანებასაც. თუ **break** მთლიანი ციკლიდან გამოდის, **continue** მხოლოდ ციკლის მიმდინარე იტერაციას გამოტოვებს. ამ ბრძანებას მიმართავენ ისეთ სიტუაციაში, როდესაც სკრიფტის გაშვებისას შეცდომა ხდება და გვსურს, რომ ინტერპრეტატორი ამ შეცდომას „გადაახტეს“ და სკრიფტის გაშვება შემდეგ იტერაციაზე გაგრძელდეს. დავბეჭდოთ 1-დან 10-მდე რიცხვები 3-სა და 6-ის გამოტოვებით:

```
#!/bin/bash

i=0
while [ $i -lt 10 ]
do
    i=`expr $i + 1`
    if [ $i -eq 3 ] || [ $i -eq 6 ]
    then
        continue
    fi
    echo -n "$i "
done
echo
exit 0

achiko@debian:~$ ./script_XYZ.sh
1 2 4 5 7 8 9
```

N არგუმენტი **continue**-საც შეგვიძლია გადავცეთ. **continue** N ბრძანება გამოტოვებს მიმდინარე ციკლის ყველა დარჩენილ იტერაციას და გადავა N დონით წინა ციკლის შემდეგ იტერაციაზე. მაგალითი:

```
#!/bin/bash

for outerloop in 1 2 3 4 5
do
    echo; echo -n "გარე ციკლი $outerloop => შიდა: "
    for innerloop in {1..10}
    do
        if [[ "$innerloop" -eq 7 && "$outerloop" = "3" ]]
        then
            continue 2
        fi
        echo -n "$innerloop "
    done
done
echo
exit 0

achiko@debian:~$ ./script_XYZ.sh
გარე ციკლი 1 => შიდა: 1 2 3 4 5 6 7 8 9 10
გარე ციკლი 2 => შიდა: 1 2 3 4 5 6 7 8 9 10
გარე ციკლი 3 => შიდა: 1 2 3 4 5 6
გარე ციკლი 4 => შიდა: 1 2 3 4 5 6 7 8 9 10
გარე ციკლი 5 => შიდა: 1 2 3 4 5 6 7 8 9 10
```

ამ სკრიფტში **continue 2**-ის ნაცვლად, **continue 1** (ან მისი ექვივალენტი - **continue**) რომ გამოგვეყენებინა, ასეთ შედეგს მივიღებდით:

```
#!/bin/bash

for outerloop in 1 2 3 4 5
do
    echo; echo -n "გარე ციკლი $outerloop => შიდა: "
    for innerloop in {1..10}
    do
        if [[ "$innerloop" -eq 7 && "$outerloop" = "3" ]]
        then
            continue 1
        fi
        echo -n "$innerloop "
    done
done
echo
exit 0
```

achiko@debian:~\$./script_XYZ.sh

გარე ციკლი 1 => შიდა: 1 2 3 4 5 6 7 8 9 10
 გარე ციკლი 2 => შიდა: 1 2 3 4 5 6 7 8 9 10
 გარე ციკლი 3 => შიდა: 1 2 3 4 5 7 8 9 10
 გარე ციკლი 4 => შიდა: 1 2 3 4 5 6 7 8 9 10
 გარე ციკლი 5 => შიდა: 1 2 3 4 5 6 7 8 9 10

დაიმახსოვრეთ!

`continue 1` იგივე შედეგს გვაძლევს, რასაც `continue` ბრძანება.

16.5.4 ციკლი while

`while`-ის სინტაქსი ასეთია:

```
while Command;           # სანამდე Command ქეშმარიტია, მანამდე
do
    cmd1;                 # შესრულდეს ბრძანება cmd1
    cmd2;                 # შესრულდეს ბრძანება cmd2
    ...
done                     # do-ს ვხურავთ done-ით
```

ვნახოთ მისი გამოყენების მაგალითები:

```
#!/bin/bash
x=1
while [ $x -le 5 ]
do
    echo $x
    x=$((x+1))
done
```

```
done
achiko@debian:~$ ./script_XYZ.sh
1
2
3
4
5
```

ამ მაგალითში მოდით, დეტალურად გავშიფროთ $x=$((x+1))$ ჩანაწერი. x ცვლადში დამახსოვრებული გვაქვს რიცხვი - 1. შესაბამისად, $x+1$ არითმეტიკულ გამოსახულებას წარმოადგენს, რომელსაც გამოთვლა სჭირდება. ამისთვის, ეს გამოსახულება $((...))$ ან $[...]$ სტრუქტურაში უნდა მოვაქციოთ. საბოლოოდ, x ცვლადში ვწერთ მის წინა მნიშვნელობაზე 1-ით მეტ მნიშვნელობას. ანუ, ვაკეთებთ ცვლადის ინკრიმენტს 1-ით.

`while` ხშირად გამოიყენება სკრიფტიდან ფაილის შიგთავსის ხაზ-ხაზ ამოღებისას. ასე:

```
#!/bin/bash
File=$HOME/countries.txt
while read line
do
    echo $line
done<"$File"
achiko@debian:~$ ./script_XYZ.sh
Georgia 1
France 2
USA 3
Germany 4
Great Britain 5
```

ცხადია, ეს სკრიფტი ერთ ხაზშიც შეიძლება ჩავწეროთ:

```
achiko@debian:~$ while read line; do echo $line; done<countries.txt
Georgia 1
France 2
USA 3
Germany 4
Great Britain 5
```

16.5.5 ციკლი until

`until`-ის სინტაქსი `while`-ის იდენტურია:

```
until Command;          # სანამდე Command მცდარია, მანამდე
do
    cmd1;                # შესრულდეს ბრძანება cmd1
    cmd2;                # შესრულდეს ბრძანება cmd2
    ...
```

```
done
```

```
# do-ს ვხურავთ done-ით
```

მათ შორის განსხვავება შემდეგია: **while**-ის შემთხვევაში **do ... done** ბლოკში ჩაწერილი ბრძანებები სრულდება მანამდე, სანამდეც შესამოწმებელი **Command** ჭეშმარიტია, ანუ, წარმატებით სრულდება. როგორც კი, მისი გამოსახველი მცდარი გახდება, ციკლი დასრულდება. **until**-ის შემთხვევაში კი პირიქითაა. ციკლი გრძელდება მანამდე, სანამდეც შესამოწმებელი **Command** მცდარია. როგორც კი, ის ჭეშმარიტი გახდება, ციკლი დასრულდება.

მაგალითი:

```
#!/bin/bash
x=1
until [ $x -gt 5 ]
do
    echo $x
    x=$((x+1))
done
achiko@debian:~$ ./script_XYZ.sh
1
2
3
4
5
```

რა თქმა უნდა, პირობა ისე შეიძლება შევარჩიოთ, რომ ციკლი უსასრულოდ გაგრძელდეს. **while**-ის შემთხვევაში, ნებისმიერი მუდმივად ჭეშმარიტი ბრძანება უნდა გამოვიყენოთ, **until**-ის შემთხვევაში კი - მუდმივად მცდარი ბრძანება. მუდმივი ციკლიდან გამოსვლა **Ctrl^c** კლავიშების კომბინაციით შეიძლება.

მოვიყვანოთ მუდმივი ციკლის მაგალითები:

```
#!/bin/bash
while true                # true ბრძანება ყოველთვის ჭეშმარიტია
do
    echo TEST; sleep 1
done
achiko@debian:~$ ./script_XYZ.sh
...
TEST
TEST
...
```

```
#!/bin/bash
while [ 1 ]               # უბრალოდ გამოსახულება "1" ყოველთვის ჭეშმარიტია
do                         # ისევე როგორც, ნებისმიერ ასო-ნიშანთა ერთობლიობა
    echo TEST; sleep 1
done
```

```
#!/bin/bash
while [ მიშკა ]           # მიშკა ყოველთვის ჭეშმარიტია
do
    echo TEST; sleep 1
done
```

```
#!/bin/bash
while sleep 1
do
    echo TEST
done
```

```
#!/bin/bash
while :           # ":" ყველაზე მარტივი ფორმაა მუდმივი ციკლის ჩასაწერად
do
    echo TEST; sleep 1
done
```

```
#!/bin/bash
for ((;;))           # ცარიელი პირობებით for ყოველთვის ჭეშმარიტია
do
    echo TEST; sleep 1
done
```

```
#!/bin/bash
until false
do
    echo TEST; sleep 1
done
```

უამრავი მსგავსი ვარიანტის მოფიქრება შეგვიძლია. ერთ-ერთ ვარიანტში გამოვიყენეთ ჩანაწერი „:“. ავხსნათ, თუ რას წარმოადგენს ის. : ბაშის შიდა ბრძანებაა და ცარიელ ბრძანებას ნიშნავს (null command ან NOP, no op, do-nothing ოპერაცია). ანუ, ეს ბრძანება არაფერს აკეთებს და მისი გამოსასვლელი კოდი არის 0. ის შეიძლება true ბრძანების სინონიმადაც მივიჩნიოთ.

შელის სკრიფტებში ხშირად იყენებენ exit ბრძანებას. ეს ბრძანება პირდაპირ შელში რომ გავუშვათ, დავინახავთ, რომ შელი დაიხურება. ის შელიდან გამოსვლას ნიშნავს. სკრიფტში მისი გამოყენება კი ამ სკრიფტის დახურვას გამოიწვევს. მას შეგვიძლია პარამეტრად გადავცეთ რიცხვი - exit N. ეს იმას ნიშნავს, რომ შელის დახურვის გამოსასვლელი კოდი სწორედ N რიცხვი იქნება. თუ ეს პარამეტრი არ გადავცით ბრძანებას, ის ბოლო შესრულებული ბრძანების გამოსასვლელ კოდს დაგვიბრუნებს. გამოსასვლელი კოდების განსაზღვრა კი მნიშვნელოვანია, რამეთუ ისინი შეიძლება გამოყენებულ იქნეს სხვა

ბრძანებებისა თუ სკრიფტების მიერ.

```
achiko@debian:~$ exit
```

დავინახავთ, რომ ტერმინალი დაიხურა და ეს იმიტომ, რომ შელიდან გამოვედით. შესაბამისად, გამოსასვლელი კოდის ნახვა გაგვიჭირდება, რადგან ბრძანებათა ზაზი აღარ არსებობს, რომ შესატყვისი ბრძანება გავუშვათ. ამიტომ, ჯერ გაშვებულ bash-ში თავად bash გავუშვათ შვილ პროცესად:

```
achiko@debian:~$ bash
achiko@debian:~$ ps -jH
  PID  PGID   SID TTY          TIME CMD
 11168 11168 11168 pts/0    00:00:00 bash
 11880 11880 11168 pts/0    00:00:00  bash
 11885 11885 11168 pts/0    00:00:00   ps
achiko@debian:~$ exit
achiko@debian:~$ echo $?
0
achiko@debian:~$ bash
achiko@debian:~$ exit 15
achiko@debian:~$ echo $?
15
```

16.5.6 ციკლი select

select ბაშში სიტყვა-გასაღებს წარმოადგენს და ის ძალიან სასარგებლოა მენიუების შესაქმნელად. მისი სინტაქსი შემდეგნაირია:

```
select Variable in list;
do
    cmd1;
    cmd2;
    ...
done
```

ის სრულად ემთხვევა **for**-ის სინტაქსურ ჩანაწერს, მაგრამ **for**-ისგან განსხვავებით, **select** სიის ყველა ელემენტზე თანმიმდევრულად არ იმოქმედებს **do ... done** მოცემული ბრძანებებით, არამედ შემოგვთავაზებს მენიუს სიის ელემენტებით და გვთხოვს ჩვენ შევიყვანოთ, თუ რომელი ელემენტის არჩევა გვსურს.

მოვიყვანოთ მარტივი მაგალითი:

```
#!/bin/bash

echo "Select a country where do you want to spend this summer ?"

select country in France Germany Georgia Spain USA
```

```
do
    echo "You choose: $country"
done
achiko@debian:~$ ./script_XYZ.sh      # ასარჩევად მიუთითეთ შესაბამისი ნომერი
Select a country where do you want to spend this summer ?
1) France
2) Germany
3) Georgia
4) Spain
5) USA
#? 3
You choose: Georgia
#? 1
You choose: France
#? 4
You choose: Spain
...
#? ^C
```

მენიუდან არჩევანის გაკეთების შემდეგ **select** კვალავ შემოგვთავაზებს სიას და ეს პროცესი მუდმივად გაგრძელდება. ამიტომ Ctrl^C კლავიშების კომბინაცია დაგჭირდებათ გამოსასვლელად. თუ მხოლოდ ერთი არჩევანის გაკეთება გვსურს, შეგვიძლია, ციკლიდან გამოსასვლელად **break** ბრძანება გამოვიყენოთ. ასევე შესაძლებელია PS3 ცვლადის გამოყენება, რომლითაც შევცვლით მენიუს მოსაწვევის ტექსტს (#?-ის ნაცვლად), ასე:

```
#!/bin/bash

PS3="Input number: "
echo "Select a country where do you want to spend this summer ?"

select country in France Germany Georgia Spain USA
do
    echo "Your choice is $country"
    break
done
achiko@debian:~$ ./script_XYZ.sh
Select a country where do you want to spend this summer ?
1) France
2) Germany
3) Georgia
4) Spain
5) USA
Input number: 3
Your choice is Georgia
achiko@debian:~$
```

select ძალიან ხშირად გამოიყენება **case**-თან კომბინაციაში. ბუნებრივიცაა, რადგან მენიუდან ცვლადის არჩევისას კოდში შემდეგი განვითარება უნდა მოხდეს. მოვიყვანოთ

შემდეგი მაგალითი, თან select-ში თავად select გამოვიყენოთ.

```
#!/bin/bash

PS3="Select country: "
echo "Select a country where do you want to spend this summer ?"

select country in France Georgia Spain
do
    PS3="Select City/Region/Province: "
    echo "What region do you prefer to visit in $country ?"
    case $country in
        France)
            select region in Paris Bretagne Normandie ...
            do
                break
            done
            ;;
        Georgia)
            select region in Tbilisi Kakheti Imereti ...
            do
                break
            done
            ;;
        Spain)
            select region in Madrid Barcelona Valencia ...
            do
                break
            done
            ;;
    esac
    break
done

echo
echo "Perfect choice: $country => $region. Have a nice trip!"
echo
achiko@debian:~$ ./script_XYZ.sh
Select a country where do you want to spend this summer ?
1) France
2) Georgia
3) Spain
Select country number: 2
What City/region/province do you prefer to visit in Georgia ?
1) Tbilisi
2) Kakheti
3) Kartli
4) Imereti
5) ...
```

```
Select City/Region/Province: 2
```

```
Perfect choice: Georgia => Kakheti. Have a nice trip!
```

16.6 არითმეტიკული გამოთვლები

სკრიფტების წერისას მარტივი არითმეტიკული გამოთვლები მთელ რიცხვებზე, როგორც ვნახეთ, ზშირად გვჭირდება. მართალია, ისინი მე-10 თავში ჩვენ უკვე მიმოვიხილეთ, თუმცა, დავუბრუნდეთ მათ უფრო დეტალურად განსახილველად. გამოთვლებისთვის ძირითადად ვიყენებდით შემდეგ ჩანაწერებს: `$((expression))` ან `$(expression)`, სადაც `expression` ნამდვილი არითმეტიკული გამოსახულებაა.

Bash-ში ორმაგი ფრჩხილი აგრეთვე წარმოადგენს მთელი რიცხვების მანიპულირების C-ის სტილის მექანიზმს. მაგალითად, ასეთი ჩანაწერი `((var++))` სავსებით სამართლიანია. ორმაგი ფრჩხილის კონსტრუქცია ჭეშმარიტია, თუ მასში ჩაწერილი არითმეტიკული გამოსახულების მნიშვნელობა 0-სგან განსხვავებულია. შესაბამისად, მისი გამოსასვლელი კოდი 0 იქნება. წინააღმდეგ შემთხვევაში, თუ ორმაგ ფრჩხილში 0 ან 0-ის ტოლი გამოსახულება წერია, გამოსასვლელი კოდი 0-სგან განსხვავებული იქნება. დავრწმუნდეთ:

```
achiko@debian:~$ ((8))
achiko@debian:~$ echo $?
0
achiko@debian:~$ ((-1)); echo $?
0
achiko@debian:~$ ((0)); echo $?      # 0-ზე გამოსახულება მცდარია!
1                                     # შესაბამისად, გამოსასვლელი კოდი 0-სგან განსხვავდება.
```

ორმაგ ფრჩხილებში შესაძლებელია, ჩვენთვის ცნობილი ნებისმიერი ოპერაცია გამოვიყენოთ: + მიმატება, - გამოკლება, * გამრავლება, / გაყოფა, ** ხარისხში აყვანა, % ნაშთიანი გაყოფა, = მინიჭება.

```
achiko@debian:~$ ((a=2**10))
achiko@debian:~$ echo $a
1024
```

ორმაგი ფრჩხილი, bash-სა და zsh-ში, ksh-დან გადმოღებული ფუნქციონალია. ჯერჯერობით, ეს კონსტრუქცია არ წარმოადგენს POSIX სტანდარტის ფორმას. ის არ არის bash-ის საკუთარი ბრძანება და მასში დასაშვებია გამოტოვებების გამოყენება სურვილისამებრ, ასე:

```
achiko@debian:~$ (( a=5+ 3 )); echo $a
8
achiko@debian:~$ (( a = 7 % 5 )); echo $a
2
achiko@debian:~$ (( a=8      - 5)); echo $a
3
```

ორმაგი ფრჩხილში სამართლიანია მინიჭების ოპერატორების ჩაწერის გამარტივებული შემდეგი ფორმები:

ჩანაწერი	მნიშვნელობა
<code>var = value</code>	უბრალო მინიჭება. <code>var</code> ცვლადს ვანიჭებთ <code>value</code> მნიშვნელობას.
<code>var += value</code>	მიმატება. <code>var</code> ცვლადს ვანიჭებთ მის ძველ მნიშვნელობას დამატებული <code>value</code> მნიშვნელობა. ექვივალენტია შემდეგი ჩანაწერის: <code>var=var+value</code>
<code>var -= value</code>	გამოკლება. ექვივალენტია შემდეგი ჩანაწერის: <code>var=var-value</code>
<code>var *= value</code>	გამრავლება. ექვივალენტია შემდეგი ჩანაწერის: <code>var=var*value</code>
<code>var /= value</code>	გაყოფა. ექვივალენტია შემდეგი ჩანაწერის: <code>var=var/value</code>
<code>var++</code>	ინკრიმენტი ⁴ . <code>var</code> ცვლადის პოსტ-ინკრიმენტი ⁵ . ექვივალენტია შემდეგი ჩანაწერის: <code>var=var+1</code>
<code>var--</code>	დეკრიმენტი ⁶ . <code>var</code> ცვლადის პოსტ-დეკრიმენტი ⁷ . ექვივალენტია შემდეგი ჩანაწერის: <code>var=var-1</code>
<code>++var</code>	<code>var</code> ცვლადის პრე-ინკრიმენტი ⁸ . ექვივალენტია შემდეგი ჩანაწერის: <code>var=var+1</code>
<code>--var</code>	<code>var</code> ცვლადის პრე-დეკრიმენტი ⁹ . ექვივალენტია შემდეგი ჩანაწერის: <code>var=var-1</code>

მოვიყვანოთ მაგალითი პოსტ/პრე-ინკრიმენტ/დეკრიმენტს შორის განსხვავების დასაფიქსირებლად.

```
#!/bin/bash
i=1
while :
do
    echo $((i++))
    sleep 1
done
achiko@debian:~$ ./script_XYZ.sh
1
2
3
^C
```

⁴ინკრიმენტი ნიშნავს ცვლადის მნიშვნელობის 1-ით გაზრდას.

⁵პოსტ-ინკრიმენტი ნიშნავს ცვლადის მნიშვნელობის 1-ით გაზრდას მისი პირველად გამოყენების შემდეგ.

⁶დეკრიმენტი ნიშნავს ცვლადის მნიშვნელობის 1-ით შემცირებას .

⁷პოსტ-დეკრიმენტი ნიშნავს ცვლადის მნიშვნელობის 1-ით შემცირებას მისი პირველად გამოყენების შემდეგ.

⁸პრე-ინკრიმენტი ნიშნავს ცვლადის მნიშვნელობის 1-ით გაზრდას მის გამოყენებამდე.

⁹პრე-დეკრიმენტი ნიშნავს ცვლადის მნიშვნელობის 1-ით შემცირებას მის გამოყენებამდე.

```
#!/bin/bash
i=1
while :
do
    echo $((++i))
    sleep 1
done
achiko@debian:~$ ./script_XYZ.sh
2
3
4
^C
```

პირველ სკრიფტში ჯერ ვბეჭდავთ *i* ცვლადში არსებულ მნიშვნელობას - 1-ს, ხოლო შემდეგ ვახდენთ მის ინკრიმენტს. მეორე სკრიფტში კი ჯერ ვახდენთ *i* ცვლადის ინკრიმენტს და შემდეგ ვბეჭდავთ ეკრანზე მის მნიშვნელობას.

ორმაგი ფრჩხილით რიცხვების შედარებაც უფრო მარტივად და კომპაქტურად ჩაიწერება, ასე:

```
#!/bin/bash
a=37
if (($a>100))
then
    x=10
else
    x=20
fi
echo $x
exit 0
achiko@debian:~$ ./script_XYZ.sh
20
```

ცვლადი *\$*-ის გარეშეც, ავტომატურად ცვლადად აღიქმება, რადგან ორმაგ ფრჩხილში მხოლოდ არითმეტიკული გამოსახულება იწერება. შესაბამისად, ეს სკრიფტი ასეც დაიწერება:

```
#!/bin/bash
a=37
if ((a>100))
then
    x=10
else
    x=20
fi
echo $x
exit 0
achiko@debian:~$ ./script_XYZ.sh
```

ეს მაგალითი გაცილებით მარტივად ჩაიწერება სამკომპონენტური (ტერნარული - ternary) ოპერატორით. ამ ოპერატორს ზოგჯერ შედარების ოპერატორსაც უწოდებენ. მისი სინტაქსური ჩანაწერი ასეთია:

```
expr1?expr2:expr3
```

თუ **expr1** არითმეტიკული გამოსახულება ჰქმნარს, ანუ არ არის ნული, მაშინ შესრულდეს **expr2** გამოსახულება. წინააღმდეგ შემთხვევაში - **expr3** გამოსახულება.

ჩვენი სკრიფტის ჩანაწერი ამ ოპერატორით შემდეგნაირად გამარტივდება:

```
achiko@debian:~$ a=37; ((x = a>100?10:20))
achiko@debian:~$ echo $x
20
```

ორმაგ ფრჩხილში შეიძლება შემდეგი ლოგიკური ოპერატორები გამოვიყენოთ:

ოპერ.	მნიშვნელობა	ნიმუში	შედეგი
>	მეტია	a=37; ((x = a>12?10:20)); echo \$x	10
>=	მეტია ან ტოლი	a=37; ((x = a>=12?10:20)); echo \$x	10
<	ნაკლებია	a=37; ((x = a<12?10:20)); echo \$x	20
<=	ნაკლებია ან ტოლი	a=37; ((x = a<=12?10:20)); echo \$x	20
==	უდრის	a=37; ((x = a==37?10:20)); echo \$x	10
!=	არ უდრის	a=37; ((x = a!=37?10:20)); echo \$x	20
&&	ლოგიკური და	a=37; ((x = a>20&&a<30?10:20)); echo \$x	20
	ლოგიკური ან	a=37; ((x = a>20 a<30?10:20)); echo \$x	10

ზემოთაღნიშნული სამკომპონენტური ოპერატორი მხოლოდ ორმაგ ფრჩხილში გამოიყენება. დავრწმუნდეთ! შევეცადოთ, პირდაპირ გავუშვათ ის როგორც ბრძანება:

```
achiko@debian:~$ x = a>100?10:20
-bash: x: command not found
```

ეკრანზე შეცდომა მივიღეთ. ავხსნათ: შელის მიერ **x** ბრძანებად აღიქმება, რადგან მის შემდეგ გამოტოვების ნიშანია. ხოლო ბრძანება **x** დასახელებით არ არსებობს და შედეგიც გასაგებია. ახლა გამოტოვებები წავშალოთ და ისე გავუშვათ:

```
achiko@debian:~$ x=a>100?10:20
achiko@debian:~$
```

ამჯერად შეცდომა ეკრანზე აღარ გამოვიდა. ბრძანება წარმატებით გაეშვა. რა ბრძანებაზეა საუბარი? დავაკვირდეთ ყურადღებით! ბრძანების ჩანაწერში გადამისამართების ნიშანი (>) გვაქვს გამოყენებული. ეს ნიშნავს, რომ მის წინ მდგომი ჩანაწერი წარმოადგენს ბრძანებას და ამ ბრძანების სტანდარტულ გამოსასვლელზე მიღებული შედეგი გადაგვაქვს ფაილში სახელად „100?10:20“. თავად ბრძანება მეტი არაფერია, თუ არა **x** ცვლადის შექმნა, სადაც **a** მნიშვნელობას ვინახავთ. შესაბამისად, მის სტანდარტულ გამოსასვლელზე არაფერი გამოვა და ფაილიც ცარიელი იქნება. გადავამოწმოთ:

```
achiko@debian:~$ ls -l
-rw-r--r-- 1 achiko achiko      0 Jun  5 13:12 100?10:20
...
```

აღნიშნული ფაილი მართლაც შეიქმნა (ან არსებობდა და მის შიგთავსს ახალი ბრძანების შედეგი გადაგაწერეთ) და მისი ზომა არის ნული ბაიტი.

სამკომპონენტიანი ოპერატორის ჩაწერისას უნდა გავითვალისწინოთ, რომ მის გამოსახულებებში მინიჭების ოპერაციის (=) პირდაპირ გამოყენება არ შეიძლება. ის შეცდომით დასრულდება bash-ში. ამის თავიდან ასაცილებლად, გამოსახულება ფრჩხილებში უნდა მოვათავსოთ. მოდით, მაგალითი მოვიყვანოთ:

```
achiko@debian:~$ i=0
achiko@debian:~$ ((i<1?++i:-i)); echo $i
1
achiko@debian:~$ ((i<1?++i:-i)); echo $i
0
achiko@debian:~$ ((i<1?++i:-i)); echo $i
1
achiko@debian:~$ ((i<1?++i:-i)); echo $i
0
```

ეს ბრძანება, ყოველ გაშვებაზე, **i** ცვლადის მნიშვნელობის 0-დან 1-ზე გადართვას ახორციელებს და პირიქით. მოდით, ახლა აქ გამოყენებული ინკრიმენტისა და დეკრიმენტის ოპერაცია, მინიჭების ოპერაციის დახმარებით, სხვაგვარად ჩავწეროთ, ასე:

```
achiko@debian:~$ ((i<1?i+=1:i-=i))
-bash: ((: i<1?i+=1:i-=1: attempted assignment to non-variable (error token is "-=1")
```

მართალია, ++i და --i, i+=1 და i-=1 ჩანაწერების სრული ექვივალენტია, მაგრამ, თუ სამკომპონენტიანი ოპერატორში მეორე ვერსიის ჩანაწერი ფრჩხილებში არ მოვაქციეთ, ისე არ იმუშავებს:

```
achiko@debian:~$ ((i<1?(i+=1):(i-=i))); echo $i
1
achiko@debian:~$ ((i<1?(i+=1):(i-=i))); echo $i
0
```


ასე, კი ყველაფერი რიგზეა.

ორმაგი ფრჩხილები აგრეთვე გამოიყენება თვლის სხვადასხვა სისტემაში ჩაწერილი რიცხვების ათობით სისტემაში წარმოსადგენად. მისი სტრუქტურა მარტივია.

`base#number`

თვლის რვაობითი და თექვსმეტობითი სისტემების რიცხვებისათვის სპეციალური ჩანაწერები არსებობს, რადგან ისინი შელში მეტ-ნაკლებად ხშირად გამოიყენება. თუმცა, შელში ზოგადი ჩანაწერის გაკეთებაც შეიძლება. ვნახოთ ისინი:

ჩანაწერი	მნიშვნელობა
<code>0xnumber</code>	არითმეტიკულ გამოსახულებაში რიცხვი, რომელსაც წინ ნული აქვს მიწერილი, აღიქმება როგორც რვაობითი თვლის სისტემის რიცხვი.
<code>0xnumber</code>	ასეთი ჩანაწერი კი თექვსმეტობითი თვლის სისტემის რიცხვია.
<code>number</code>	ჩვეულებრივად ჩაწერილი რიცხვი, ნაგულისხმევი მნიშვნელობით, ათობითი სისტემის რიცხვია.
<code>base#number</code>	ეს ზოგადი აღნიშვნაა. ასე მოიცემა <code>base</code> თვლის სისტემის <code>number</code> რიცხვი.

ავიღოთ თვლის სხვადასხვა სისტემაში ერთი რიცხვი, მაგალითად 21 და ვნახოთ, რა იქნება მისი შესაბამისი რიცხვი თვლის ათობით სისტემაში.

```
achiko@debian:~$ echo $((21))      # 2110 ⇔ 2110
21
achiko@debian:~$ echo $((021))      # 218 ⇔ 1710
17
achiko@debian:~$ echo $((0x21))      # 2116 ⇔ 3310
33
achiko@debian:~$ echo $((11#21))     # 2111 ⇔ 2310
23
achiko@debian:~$ echo $((16#21))     # 2116 ⇔ 3310
33
achiko@debian:~$ echo $((2#21))      # 212 ⇔ შეცდომა
-bash: 2#21: value too great for base (error token is "2#21")
```

ბოლო ბრძანება სწორად არ არის ჩაწერილი, რადგან თვლის ორობითი სისტემაში მხოლოდ ორი ცალი ციფრი გვაქვს: 0 და 1. ჩვენს მაგალითში კი შედის სხვა ციფრიც - 2 (რიცხვში 21).

თვლის ორობითი სისტემისთვის შემდეგი მაგალითი გამოდგება:

```
achiko@debian:~$ echo $((2#111110000)) # 1111100002 ⇔ 49610
496
```

111110000 ორობითი რიცხვისგან მივიღეთ 496 ათობითი (რიგით მესამე

სრულყოფილი ¹⁰⁾ რიცხვი.

ადამიანი ყველაზე მარტივად აღიქვამს და იაზრებს თვლის ათობით სისტემას. ზემოთ მოყვანილ მაგალითებში ვნახეთ, რომ თვლის ნებისმიერ სისტემაში ჩაწერილი რიცხვი გამოსასვლელზე, ნაგულისხმევი მნიშვნელობით, ათობით სისტემაში წარმოდგინდებოდა. როგორ მოვიქცეთ, თუ გვსურს ერთი სისტემის რიცხვი, ათობითის გარდა სხვა სისტემაში ვნახოთ. ამ დროს შეგვიძლია, გამოვიყენოთ ჩვენთვის უკვე კარგად ნაცნობი **bc** ბრძანება. ამ ბრძანებაში სპეციალური ცვლადებით შეგვიძლია განვსაზღვროთ დასამუშავებელი რიცხვი რა სისტემაში იგულისხმოს (**ibase** ცვლადით) და უკვე დამუშავებული რიცხვი თვლის რა სისტემაში გამოიტანოს (**obase** ცვლადით) შელმა. ნაგულისხმევი მნიშვნელობით, ეს ცვლადები ათობით სისტემას წარმოადგენს. მაგალითი:

```
achiko@debian:~$ echo "ibase=2; 496" | bc      # 49610 ⇔ 1111100002
111110000
achiko@debian:~$ echo "ibase=16; obase=2; 496" | bc  # 49616 ⇔ 100100101102
10010010110
```

16.6.1 ბიტური ოპერაციები

კომპიუტერები ინფორმაციას იმავე ხერხით ვერ აღიქვამენ, როგორითაც ადამიანები. მაგალითად, მათ არ ესმით სიტყვები, ხმა. თუ ამ ინფორმაციას ისეთ სისტემაში გადავიყვანთ, რომელიც კომპიუტერისთვის გასაგები იქნება, ცხადია ისინიც გაიგებენ ამ ყველაფერს. როგორც უკვე ვიცით, ასეთი სისტემა ორობითი სისტემაა. თანამედროვე გამოთვლელ სისტემებში ნებისმიერი ინფორმაცია წარმოდგენილია ელექტრული სიგნალით, რომელსაც ორი მდგომარეობა შეიძლება ჰქონდეს: ჩართული ან გამორთული. ეს კი კარგად შეესაბამება თვლის ორობით სისტემას, რომელშიც მხოლოდ ორი ციფრი გვაქვს 1 და 0.

თვლის ორობით სისტემაზე ოპერაციების ჩატარების ცოდნა მნიშვნელოვანია, რადგან კომპიუტერში ინფორმაცია სწორედ ორობით სისტემაშია წარმოდგენილი. ასეთი ოპერაციები გამოიყენება ჩაშენებულ სისტემებში, დაბალი დონის დაპროგრამების ენებში, ვიდეო დეკოდერების პროგრამებში, შეკუმშვის პროგრამებში და სხვა. რიცხვებზე ასეთი მანიპულაციების ჩატარება ჩვენთვის უჩვეულო ხერხია, მაგრამ კომპიუტერებისთვის ეს ბუნებრივი მეთოდია. ოპერაციები ბიტებზე ხორციელდება. შესაბამისად მათ ბიტურ ოპერაციებს ეძახიან. ცხადია, შელში შესაძლებელია ორობით რიცხვებზე ოპერაციების ჩატარება. მოდით, ვნახოთ ისინი დეტალურად:

ოპერატორი	მნიშვნელობა
&	ბიტური და (AND). ამ ოპერაციის დროს ოპერანდად ორი რიცხვი აიღება და მათ თითოეულ ბიტზე ხორციელდება და ოპერაცია.
	ბიტური ან (OR). ამ ოპერაციის დროს ოპერანდად ორი რიცხვი აიღება და მათ თითოეულ ბიტზე ხორციელდება ან ოპერაცია.
^	ბიტური გამორიცხვა (XOR). ამ ოპერაციის დროს ოპერანდად ორი რიცხვი აიღება და მათ თითოეულ ბიტზე ხორციელდება ეს ოპერაცია.
~	ბიტური უარყოფა (NOT). ამ ოპერაციის დროს ოპერანდად ერთი რიცხვი აიღება და მისი თითოეული ბიტის ორობითი მნიშვნელობის შეტრიალება ხდება. თუ არის 0 გახდება 1, თუ არის 1 გახდება 0.

¹⁰სრულყოფილია რიცხვი, თუ ის ყველა საკუთარი გამყოფის ჯამის ტოლია.

>> მარჯვნივ დაძვრა. ამ ოპერაციის დროს ოპერანდად ორი რიცხვი აიღება და პირველი რიცხვის ორობით ვერსიაზე ზდება ბიტების მარჯვნივ დაძვრა იმდენი ბიტით, რამდენიც მეორე ოპერანდად არის მითითებული. უკიდურესი მარჯვენა ბიტები დაძვრის შედეგად იკარგება, ხოლო ცარიელი ბიტები 0-ით ივსება.

<< მარცხნივ დაძვრა. ამ ოპერაციის დროს ოპერანდად ორი რიცხვი აიღება და პირველი რიცხვის ორობით ვერსიაზე ზდება ბიტების მარცხნივ დაძვრა იმდენი ბიტით, რამდენიც მეორე ოპერანდად არის მითითებული. უკიდურესი მარცხენა ბიტები არ იკარგება, მხოლოდ დაიძვრება მარცხნივ და ცარიელი ბიტები 0-ებით შეივსება. მოკლედ რომ ვთქვათ, ამ რიცხვის ორობით ჩანაწერს ბოლოში, მარცხნივ, იმდენივე 0 მიეწერება, რამდენიც მეორე ოპერანდად არის მოცემული.

ბიტური ოპერაციები ორობით რიცხვებზე ხორციელდება. მართალია, შელში შეტანისას ოპერანდებად აღებული რიცხვები ათობით რიცხვებს წარმოადგენს, თუმცა ისინი ოპერაციის დროს თვლის ორობით სისტემაში კონვერტირდება და ოპერაციაც მათზე ზდება. მიღებული შედეგი კი კვლავ ათობით სისტემაში წარმოდგინდება და ისე მოგვეწოდება ჩვენ ეკრანზე.

მოლით, გავიხსენოთ ბიტური ოპერაციები:

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

მაგალითები:

```
achiko@debian:~$ echo $(( 11 & 12 ))
8
```

მოლით, დავრწმუნდეთ. გადავიყვანოთ ათობითი სისტემის რიცხვები 11 და 12 ორობითში. მივიღებთ 1011-სა და 1100-ს.

```
achiko@debian:~$ echo "obase=2; 11" | bc      # 1110 ⇔ 10112
1011
achiko@debian:~$ echo "obase=2; 12" | bc      # 1210 ⇔ 11002
1100
```

ახლა კი ღა ოპერაცია შევასრულოთ მიღებულ ორობით რიცხვებზე. მივიღებთ 1000-ს. ეს ორობითი რიცხვი კი ათობითში 8-ს ტოლია.

```
achiko@debian:~$ echo $((2#1000))      # 10002 ⇔ 810
8
```

ახლა უკვე გასაგებია შედეგი. ასეთივე ხერხით შეგიძლიათ შემდეგი ბრძანებების სისწორეში დარწმუნდეთ.

```
achiko@debian:~$ echo $(( 11 | 12 ))
15
```

```
achiko@debian:~$ echo $(( 11 ^ 12 ))
7
```

```
achiko@debian:~$ echo $(( ~126 ))
-127
```

ავსნათ, თუ რატომ მივიღეთ ბოლო მაგალითში უარყოფითი რიცხვი -127. როგორც უკვე ვახსენეთ, ნებისმიერი მონაცემი, მათ შორის რიცხვი, კომპიუტერში ორობითი ჩანაწერით მოიცემა ანუ რიცხვი წარმოდგენს ბიტების (0 და 1) თანმიმდევრობას. ავიღოთ უარყოფითი რიცხვი. მათემატიკაში უარყოფით რიცხვს წინ „-“ ნიშანი უწერია. კომპიუტერებში კი ბიტების თანმიმდევრობას წინ ნიშანი არ აქვს. ასეთ დროს ნიშანის განსაზღვრა დამატებითი ბიტის შემოტანით ხდება. არსებობს რამდენიმე მეთოდი ორობით სისტემაში უარყოფითი რიცხვების წარმოსადგენად. ერთ-ერთი მათგანია „ორის დამატება“ (two's complement). სწორედ ამ მეთოდს იყენებს bash ნიშნის რიცხვების წარმოსადგენად. ვნახოთ 8 ბიტის რიცხვის წარმოდგენა:

ორობითი მნიშვნელობა	ორის დამატებით წარმოდგენა	უნიშნო წარმოდგენა
00000000	0	0
00000001	1	1
⋮	⋮	⋮
01111110	126	126
01111111	127	127
10000000	-128	128
10000001	-127	129
10000010	-126	130
⋮	⋮	⋮
11111110	-2	254
11111111	-1	255

ამ ცხრილით უკვე ნათლად ჩანს, თუ რატომ მივიღეთ ეს შედეგი. ცხრილიდან ისიც ჩანს, რომ არითმეტიკულ გამოთვლებისთვის აღქმადი რიცხვი გარკვეულ დიაპაზონში $[-x, x]$ ვარირებს. Bash-ში არითმეტიკისთვის გამოიყენება `intmax_t` ცვლადი. ჩვენს სისტემებში ის 64 ბიტის სიგრძისაა და Bash-ში გასაგები რიცხვების დიაპაზონი არის $[-2^{63}, 2^{63}]$.

დავრწმუნდეთ:

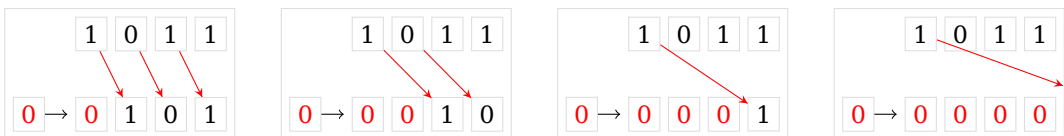
```
achiko@debian:~$ echo $(( 2**63 -1 ))
9223372036854775807
achiko@debian:~$ echo $(( 2**63 ))
-9223372036854775808
```

ჩვენს მაგალითში 2^{63} -ს 1 იმიტომ გამოვაკელით, რომ რაოდენობის ათვლა 0-დან იწყება და მაქსიმალური რიცხვი პირველი უარყოფით რიცხვზე გადადის (დიაპაზონის უმცირესი რიცხვი). 1-ის გამოკლებით დიაპაზონის უდიდეს რიცხვს მივიღებთ.

ახლა, ბიტების მარჯვნივ და მარცხნივ დაძვრის მაგალითები ვნახოთ:

```
achiko@debian:~$ echo $(( 11>>1 ))
5
achiko@debian:~$ echo $(( 11>>2 ))
2
achiko@debian:~$ echo $(( 11>>3 ))
1
achiko@debian:~$ echo $(( 11>>4 ))
0
achiko@debian:~$ echo $(( 11>>5 ))
0
```

ავხსნათ, თუ რატომ მივიღეთ ასეთი შედეგი. მივყვეთ დეტალურად: 11-ის ორობითი მნიშვნელობაა 1011. მარჯვნივ 1 ბიტის წანაცვლებით მივიღებთ 0101-ს. მისი ათობითი მნიშვნელობა არის 5. მარჯვნივ 2 ბიტის წანაცვლებით მივიღებთ 0010-ს. მისი ათობითი მნიშვნელობა არის 2. 3 ბიტის წანაცვლებით მივიღებთ 0001-ს, ათობითში - 1. 4 ბიტის წანაცვლებით (და შემდეგ) მივიღებთ ყოველთვის 0000-ს, ათობითში 0-ს. გამოდის, მარჯვნივ დაძვრით რიცხვის მნიშვნელობა ნელ-ნელა კლებულობს და ბოლოს 0-ზე დადის. ამ ოპერაციების უკეთ გასაგებად [16.11](#) ცხრილი დაგვეხმარება:



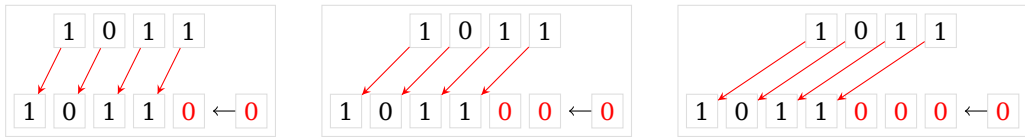
ცხრილი 16.11: დაძვრა მარჯვნივით 1 ბიტით, 2 ბიტით, 3 ბიტით და 4 ბიტით

ახლა, მარცხნივ დაძვრის მაგალითი ვნახოთ:

```
achiko@debian:~$ echo $(( 11<<1 ))
22
achiko@debian:~$ echo $(( 11<<2 ))
44
achiko@debian:~$ echo $(( 11<<3 ))
88
```

ავხსნათ, რა მოხდა: 11-ის ორობითი მნიშვნელობაა 1011. მარცხნივ 1 ბიტის წანაცვლებით მივიღებთ 10110-ს. მისი ათობითი მნიშვნელობა არის 22. მარცხნივ 2 ბიტის

წანაცვლებით მივიღებთ 101100-ს. მისი ათობითი მნიშვნელობა არის 44. 3 ბიტის წანაცვლებით მივიღებთ 1011000-ს, ათობითში - 88. გამოდის, მარცხნივ დაძვრით რიცხვის ორბითი მნიშვნელობას 0-ებს ვუმატებთ ბოლოში და მნიშვნელობა ამით ორჯერ იზრდება. ამ ოპერაციების უკეთ გასაგებად [16.12](#) ცხრილი დაგვეხმარება:



ცხრილი 16.12: დაძვრა მარცხნივ 1 ბიტით, 2 ბიტით და 3 ბიტით

რადგანაც მარცხნივ დაძვრა წინა რიცხვის ორმაგ მნიშვნელობას გვაძლევს, ამ ოპერაციითაც შეგვიძლია, ვიპოვოთ Bash-ის არითმეტიკულ გამოთვლებში უდიდესი რიცხვი, ასე:

```
achiko@debian:~$ echo $(( (1<<63) -1 ))
9223372036854775807
```

იმის გამო, რომ Bash-ში გასაგები რიცხვების დიაპაზონი 64 ბიტით არის განსაზღვრული, 64-ის ჯერადი ნებისმიერი რიცხვი გამოგვადგება უდიდესი საპოვნელად, ასე:

```
achiko@debian:~$ echo $(( (1<<127) -1 ))
9223372036854775807
achiko@debian:~$ echo $(( (1<<1023) -1 ))
9223372036854775807
```

შეგვიძლია უდიდესი რიცხვის საძიებელ პროცესს დინამიკურად დავაკვირდეთ შემდეგი სკრიფტის საშუალებით:

```
#!/bin/bash
a=1
while ((a>0))
do
    echo $((a<=1))
    sleep 0.2
done
exit 0
achiko@debian:~$ ./script_XYZ.sh
2
4
8
16
32
64
```

```

128
256
512
1024
...
8388608
16777216
...
8796093022208
17592186044416
...
4611686018427387904
-9223372036854775808

```

მხოლოდ უდიდესი რიცხვის გამოსატანად საკმარისია, სკრიფტით მიღებული ბოლო რიცხვს 1 გამოვაკლოთ:

```

achiko@debian:~$ echo $((($a=1;while((a>0));do((a<=1));done;echo $a)-1))
9223372036854775807

```

როგორც ბევრ მაგალითში ვნახეთ, `((expression))` კონსტრუქცია მოსახერხებელ მეთოდს წარმოადგენს რიცხვებთან ურთიერთობისას. `bash`-ში გვაქვს ამ ჩანაწერის ზუსტი ექვივალენტური სხვა სტრუქტურაც. მისი სინტაქსი ასეთია: `let expression`. უკეთესი კითხვადობისთვის გამოსახულებას ბრჭყალებში სვამენ ხოლმე, ასე: `let "expression"`, თუმცა ეს აუცილებელი არ არის. `let` ბაზის შიდა ბრძანებას წარმოადგენს. ვნახოთ მისი გამოყენების რამდენიმე მაგალითი:

```

achiko@debian:~$ let a=11          # იგივეა, რაც a=11
achiko@debian:~$ let a=a+5
achiko@debian:~$ echo "11 + 5 = $a"
11 + 5 = 16
achiko@debian:~$ let "a<=3"        # იგივეა, რაც let "a = a < 3"
achiko@debian:~$ echo $a
128
achiko@debian:~$ let "a /= 4"       # იგივეა, რაც let "a = a / 4"
achiko@debian:~$ echo $a
32
achiko@debian:~$ let a++
achiko@debian:~$ echo $a
33
achiko@debian:~$ let "x = a>9?11:22"
achiko@debian:~$ echo $x
11

```

ერთ-ერთ წინა მაგალითში, უდიდესი რიცხვის პოვნის პროცესმა გარკვეული დრო წაიღო. ვნახოთ რამდენს ხანს გაგრძელდა ის და ზოგადად, როგორ შეგვიძლია გაშვებული სკრიფტის მუშაობის ხანგრძლივობის ნახვა. ამისთვის შეგვიძლია `SECONDS` ცვლადი

გამოვიყენოთ. მასში მოცემულია პროცესის ხანგრძლივობა წამებში. მისი გამოძახებისას ვნახულობთ თუ რამდენი წამია გასული გამოძახების მომენტამდე. SECONDS-ში გაგრძელდება დროის ათვლა და მისი მნიშვნელობის ნახვა ნებისმიერ სხვა მომენტშიც შეგვიძლია.

```
#!/bin/bash
a=1
while ((a>0))
do
    $((a<=1))
    sleep 0.03
done
echo "უდიდესი რიცხვია: $((a-1))"
echo "სკრიფტი გაგრძელდა $SECONDS წამი"
exit 0

achiko@debian:~$ ./script_XYZ.sh
უდიდესი რიცხვია: 9223372036854775807
სკრიფტი გაგრძელდა 2 წამი
```

ამ სკრიფტში ხელოვნურად გვაქვს დაყოვნება მოცემული, თორემ თანამედროვე პროცესორები ამ ამოცანას წამიერად ითვლის.

თუ დიდხანს გრძელდება პროცესი, მისი ხანგრძლივობის მოცემა წამებში კარგად აღქმადი აღარ იქნება. ამიტომ, შეგვიძლია ეს დრო საათებში, წუთებსა და წამებში თავად გადავიყვანოთ. მაგალითისთვის, შევამოწმოთ, რა დრო გავიდა შელის გაშვებიდან, სადაც მუშაობას ვაგრძელებთ.

```
achiko@debian:~$ echo $SECONDS
26829
achiko@debian:~$ t=$SECONDS
achiko@debian:~$ echo "$(($t/3600)) სთ $(($t%3600/60)) წთ $(($t%60)) წმ"
7 სთ 27 წთ 34 წმ
```

ასე, დრო უფრო გასაგებად გამოიყურება. თუ დარწმუნებული ხართ, რომ SECONDS ერთ დღესაც კი არ ითვლის (86400 წამს), მაშინ date ბრძანებასაც შეუძლია მისაღები ფორმატით დროის მოცემა, ასე:

```
achiko@debian:~$ date -ud "@$SECONDS" "+გასული დრო: %H:%M:%S"
გასული დრო: 07:29:39
```

ბრძანების/სკრიფტის ხანგრძლივობის სანახავად, SECONDS ცვლადის გარდა, შელში, როგორც წიგნის პირველ ნაწილში განვიხილეთ, არსებობს ბრძანება time. ის უფრო ზუსტ და ტექნიკური ხასიათის ინფორმაციას გვაძლევს. ვნახოთ, თუ რა რესურსები დაიხარჯა, მაგალითად vi-ში მუშაობის დროს:

```
achiko@debian:~$ time vi test.txt
```



```
real    0m3.496s
user    0m0.096s
sys     0m0.124s
```

vi-დან გამოსვლის შემდეგ ვიგებთ, რომ 3.496 წამი გასულა, რაც ამ რედაქტორში ვმუშაობთ. ამასთან, დეტალური ინფორმაციაც გვაქვს იმის შესახებ, თუ რა დრო (CPU time) დახარჯა პროცესორმა ამ პროცესის შესრულებისთვის მომხმარებლის გარემოში და რა დრო დახარჯა ბირთვის გარემოში (ანუ ბირთვმა რა დრო გამოიყენა ამ პროცესისთვის). time-ის გამოსასვლელის ფორმატი შეგვიძლია დავაკონფიგურიროთ, სურვილისამებრ, TIMEFORMAT ცვლადის საშუალებით, ასე:

```
achiko@debian:~$ export TIMEFORMAT='${\nreal %3lR \tuser %3lU\tsys %3lS'
achiko@debian:~$ time vi test.txt

real 0m2.067s   user 0m0.052s   sys 0m0.108s
```

time ბრძანება, სინამდვილეში, bash-ში არის ინტეგრირებული და მის სიტყვა-გასაღებს წარმოადგენს. მის გარდა, არსებობს GNU-ს ვერსიის ბრძანებაც იმავე დასახელებით (განთავსებულია /usr/bin/ დირექტორიაში). /usr/bin/time ბრძანებით უფრო მეტი ინფორმაციის მიღება შეიძლება გამოსასვლელზე. ამავედროულად, მისი კონფიგურირებაც შეგვიძლია -f ან --format ოფციით, თან გაცილებით მრავალფეროვნად, ვიდრე სიტყვა-გასაღებს. აღსანიშნავია, რომ GNU-ს ვერსიის time ბრძანება, ნაგულისხმევი მნიშვნელობით, სისტემაში არ არის დაყენებული. ის ხელით უნდა დავაინსტალიროთ.

```
achiko@debian:~$ type -a time
time is a shell keyword
time is /usr/bin/time
achiko@debian:~$ /usr/bin/time vi test.txt
0.10user 0.03system 0:03.17elapsed 4%CPU (0avgtext+0avgdata 7528maxresident)k
0inputs+64outputs (0major+798minor)pagefaults 0swaps
achiko@debian:~$ /usr/bin/time -f"რეალური დრო: %E" vi test.txt
რეალური დრო: 0:04.58
```

დეტალური ინფორმაციის სანახავად მიმართეთ სახელმძღვანელო გვერდს, [man time](#).

16.6.2 რიცხვების მწკრივები

რიცხვების სწრაფად დასაგენერირებლად ხშირად გამოიყენება seq ბრძანება. მას არგუმენტად რიცხვები გადაეცემა. ერთი არგუმენტის შემთხვევაში, ის 1-დან ამ რიცხვის ჩათვლით ყველა რიცხვს ჩამოწერს (ნაგულისხმევი მნიშვნელობით, 1-ის ბიჯით). ორი არგუმენტის შემთხვევაში, პირველი რიცხვიდან მეორეს ჩათვლით, ხოლო თუ სამი არგუმენტი გადავეცით, ის გამოიტანს პირველი რიცხვიდან მესამე რიცხვის ჩათვლით, მეორე რიცხვის ბიჯით. ვნახოთ, მისი გამოყენების მაგალითები:

```

achiko@debian:~$ seq 5
1
2
3
4
5
achiko@debian:~$ seq 5 10
5
6
7
8
9
10
achiko@debian:~$ seq -w 5 10
05
06
07
08
09
10
achiko@debian:~$ seq -w -s " " 5 10
05 06 07 08 09 10
achiko@debian:~$ seq -50 15 50
-50
-35
-20
-5
10
25
40
achiko@debian:~$ seq 0 0.02 1
0.00
0.02
0.04
...
0.98
1.00

```

`seq` ხშირად გამოიყენება ციკლების არგუმენტების დასაგენერირებლად. მას ასევე შეუძლია `-f` ოფციით ათწილადების `printf` ბრძანების ფორმატით გამოტანა.

```

achiko@debian:~$ seq -f "%3g" 1 0.1 5
...
3.7
3.8
3.9
4
4.1

```

```
4.2
4.3
4.4
4.5
4.6
4.7
4.8
4.9
5
```

რიცხვებთან ურთიერთობისას საინტერესოა **factor** ბრძანებაც. ის მოცემული რიცხვს მარტივ მამრავლებად ყოფს და ეკრანზე გამოაქვს.

```
achiko@debian:~$ factor 120
120: 2 2 2 3 5
achiko@debian:~$ factor 135798642
135798642: 2 3 3 7 41 97 271
```

16.6.3 შემთხვევითი რიცხვები

შელში მუშაობისას აუცილებლად გექნებათ შემთხვევა, როდესაც შემთხვევითი მონაცემის დაგენერირება დაგვირდებათ - ეს იქნება შემთხვევითი რიცხვი, შემთხვევითი სიმბოლოების ერთობლიობა, შემთხვევითი ფაილის სახელი თუ შემთხვევითი პაროლი. რადგან ამ ნაწილში რიცხვებზე ვმუშაობთ, ვახსენოთ, თუ როგორ შეგვიძლია შემთხვევითი რიცხვების შექმნა.

ერთ-ერთი უმარტივესი გზა **RANDOM** ცვლადის გამოყენებაა. ის ყოველ ჯერზე, 0-დან 32767-ის ჩათვლით, სხვადასხვა რიცხვს მოგვცემს.

```
achiko@debian:~$ echo $RANDOM
9240
achiko@debian:~$ echo $RANDOM
30960
achiko@debian:~$ echo $RANDOM
10671
```

ამ ცვლადის გარდა, შელში არსებობს **shuf** ბრძანებაც, რომელიც მოცემული რიცხვების ინტერვალში შემთხვევით გადანაცვლებებს აკეთებს და ეკრანზე გამოაქვს ისინი. ეს ინტერვალი **-i** ოფციით მოიცემა. შესაძლებელია, აგრეთვე, ამ ინტერვალიდან საწყისი რამდენიმე რიცხვის ამორჩევა **-n** ოფციით.

```
achiko@debian:~$ shuf -i 100-300
196
259
177
273
236
```

```

281
283
246
259
...
achiko@debian:~$ shuf -i 100-300 -n3
188
122
290
achiko@debian:~$ shuf -i 100-300 -n3
250
271
246
achiko@debian:~$ shuf -i 100-300 -n1      # მხოლოდ ერთი შემთხვევითი რიცხვი
154
achiko@debian:~$ shuf -i 100-300 -n1
234
achiko@debian:~$ shuf -i 100-300 -n1
249

```

შემთხვევითობის ხარისხის გასაზრდელად შეგვიძლია `/dev/urandom` ან, უკეთესი შედეგისთვის, `/dev/random` ფაილის აღება და მისი შიგთავსის `od` ბრძანებით დამუშავება. ასე:

```

achiko@debian:~$ od -An -N1 -i /dev/random
49
achiko@debian:~$ od -An -N1 -i /dev/random
158

```

`-N1` ოფციით `/dev/random` ფაილიდან მხოლოდ 1 ბაიტის ტოლ მონაცემს ვიღებთ. შედეგად, მისი ათობით ფორმატში მოცემისას, შეგვეძლება მივიღოთ რიცხვი `[0-256]` ინტერვალში. უფრო დიდი რიცხვის მისაღებად, შეგვიძლია 2 ან 3 ბაიტი ამოვიღოთ დასამუშავებლად და მივიღებთ `[0-65536]` ან `[0-16777216]` ინტერვალიდან ერთ-ერთ რიცხვს.

```

achiko@debian:~$ od -An -N2 -i /dev/random
4198
achiko@debian:~$ od -An -N1 -i /dev/random
19241
achiko@debian:~$ od -An -N3 -i /dev/random
1719087
achiko@debian:~$ od -An -N3 -i /dev/random
10228736

```

16.7 სპეციალური ცვლადები

ცვლადების შემოტანა სკრიფტში და მონაცემების მათში შენახვა, გაცილებით გვიმარტივებს ამ მონაცემების დამუშავებას კოდის წერის დროს. ამასთან, ნებისმიერ დროს შეგვიძლია შევცვალოთ მათი მნიშვნელობა. შელის სკრიფტში ცვლადში მონაცემის

შემოსატანად რამდენიმე გზა არსებობს. ერთი, პირდაპირ განვსაზღვროთ ეს ცვლადი და მივანიჭოთ სასურველი მნიშვნელობა. მეორე, `read` ბრძანება გამოვიყენოთ. არსებობს მესამე, უფრო მოსახერხებელი ხერხიც. ჩვენ შეგვიძლია სკრიფტს მონაცემები პირდაპირ არგუმენტებად, პარამეტრად¹¹ გადავცეთ. ამ შემთხვევაში ჩვენი სკრიფტი მათ სპეციალურ ცვლადებში ავტომატურად დაიმასსოვრებს. ეს სპეციალური ცვლადებია `$1` პირველი არგუმენტისთვის, `$2` მეორე არგუმენტისთვის, `$3` მესამესთვის და ა.შ. მე-10 არგუმენტიდან ცვლადების სახელები ფიგურულ ფრჩხილებში უნდა მოვაქციოთ, ასე: `${10}`, `${11}`, `${12}` ...

სწორედ, ამ ცვლადებს ვუწოდებთ პოზიციურ პარამეტრებს. მოვიყვანოთ მარტივი მაგალითი:

```
#!/bin/bash
echo "პირველი არგუმენტია: $1"
echo "მეორე არგუმენტია: $2"
echo "მესამე არგუმენტია: $3"
achiko@debian:~$ ./script_XYZ.sh მიშკა ნინო იო
პირველი არგუმენტია: მიშკა
მეორე არგუმენტია: ნინო
მესამე არგუმენტია: იო
```

შემდეგ გაშვებისას სკრიფტს თუ სხვა მონაცემებს გადაცევით, ცვლადის მნიშვნელობები შეიცვლება. თუ გადაცემული არგუმენტები საკმარისი არაა, მაშინ „ზედმეტი“ ცვლადები ცარიელი იქნება.

```
achiko@debian:~$ ./script_XYZ.sh ფიზიკა ქიმია
პირველი არგუმენტია: ფიზიკა
მეორე არგუმენტია: ქიმია
მესამე არგუმენტია:
achiko@debian:~$ ./script_XYZ.sh ფიზიკა ქიმია მათემატიკა ბიოლოგია
პირველი არგუმენტია: ფიზიკა
მეორე არგუმენტია: ქიმია
მესამე არგუმენტია: მათემატიკა
```

პოზიციური პარამეტრების გარდა, შელში სხვა სპეციალური ცვლადებიც არსებობს.

სპეც. ცვლადი	მნიშვნელობა
<code>\$0</code>	გაშვებული სკრიფტის დასახელება, გზა - იმ ფორმით (აბსოლუტური ან ფართობითი), რა ფორმითაც იქნა გაშვებული.
<code>\$1, \$2, ... \$N</code>	პოზიციური პარამეტრები.
<code>\$#</code>	გადაცემული არგუმენტების რაოდენობა.
<code>\$*</code>	ყველა არგუმენტი. თუ მას ბრჭყალებში მოაქცევთ - <code>"\$"</code> , მაშინ ყველა პოზიციური პარამეტრი ერთობლივად აღიქმება როგორც ერთი სიტყვა.

¹¹ ტერმინები - „არგუმენტი“ და „პარამეტრი“ სკრიფტის წერისას ერთი და იმავე მნიშვნელობით იხმარება.

\$@	იგივეა რაც, \$*. თითოეული პარამეტრი არგუმენტთა სიაში აღიქმება როგორც ცალკეული სიტყვა. ბრჭყალებში ჩასმის დროსაც - "\$@" თითოეული პარამეტრი ცალკეულ სიტყვად აღიქმება (თუ ხელოვნურად არ გვაქვს გაერთიანებული რამდენიმე პარამეტრი). მათ შორის დეტალური განსხვავების დასახნაად იხილეთ მაგალითი 16.1 .
\$?	სკრიფტის გამოსასველი კოდი.
\$\$	სკრიფტის შესაბამისი პროცესის იდენტიფიკატორი (PID). სკრიფტში, ქვეშელის შიგნით გამოყენებული \$\$ მაინც სკრიფტის PID-ს ნიშნავს, და არა ქვეშელის. იხილეთ მაგალითი 16.2
#!	სკრიფტში ფონურ რეჟიმში გაშვებული ბოლო პროგრამის PID. იხილეთ მაგალითი 16.3 .
\$_	წინა გაშვებული ბრძანების ბოლო არგუმენტი. იხილეთ მაგალითი 16.4

მაგალითი 16.1: სკრიფტის არგუმენტების სია: \$@ და \$*

```
#!/bin/bash

E_BADARGS=85

if [ ! -n "$1" ]
then
    echo "გამოყენის წესი: `basename $0` არგ1 არგ2 და ა.შ."
    exit $E_BADARGS
fi
echo

index=1                # ათვლის დანყება

echo "არგუმენტების გამოტანა \"\$*\"-ს საშუალებით:"
for arg in "$*"        # არ იმუშავებს კარაგად ბრჭყალების გარეშე
do
    echo -e "\tArg #$index = $arg"
    let "index+=1"
done                  # "$*" ყველა არგუმენს ერთ სიტყვად ხედავს

echo -e "\tარგუმენტთა სრული სია აღიქმება როგორც ერთი სიტყვა."

echo

index=1                # ათვლის თავიდან დანყება.
                        # რა მოხსდება თუ ამის გაკეთება დაგავინწყდათ?

echo "არგუმენტების გამოტანა \"\$@\"-ს საშუალებით:"
for arg in "$@"
do
```

```

        echo -e "\tArg #$index = $arg"
        let "index+=1"
done          # $@ თითოეულ არგუმენტს ცალკეულ სიტყვებად ხედავს.

echo -e "\tარგუმენტები აღიქმება როგორც ცალკეული სიტყვები."

echo

index=1      # ათვლის თავიდან დანწყება.

echo "არგუმენტების გამოტანა \${*-}ს საშუალებით (ბრჭალების გარეშე):"
for arg in $*
do
    echo -e "\tArg #$index = $arg"
    let "index+=1"
done          # $* თითოეულ არგუმენტს ცალკეულ სიტყვებად ხედავს.

echo -e "\tარგუმენტები აღიქმება როგორც ცალკეული სიტყვები."

echo

index=1      # ათვლის თავიდან დანწყება.

echo "არგუმენტების გამოტანა \${@}ს საშუალებით (ბრჭალების გარეშე):"
for arg in $@
do
    echo -e "\tArg #$index = $arg"
    let "index+=1"
done          # $@ თითოეულ არგუმენტს ცალკეულ სიტყვებად ხედავს.

echo -e "\tარგუმენტები აღიქმება როგორც ცალკეული სიტყვები."

exit 0
achiko@debian:~$ ./script_XYZ.sh
გამოყენის წესი: script\_XYZ.sh არგ1 არგ2 და ა.შ.
achiko@debian:~$ echo $?
85
achiko@debian:~$ ./script_XYZ.sh მიშკა ნინო იო

არგუმენტების გამოტანა "$*" -ს საშუალებით:
Arg #1 = მიშკა ნინო იო
არგუმენტთა სრული სია აღიქმება როგორც ერთი სიტყვა.

არგუმენტების გამოტანა "$@" -ს საშუალებით:
Arg #1 = მიშკა
Arg #2 = ნინო
Arg #3 = იო
არგუმენტები აღიქმება როგორც ცალკეული სიტყვები.

```

არგუმენტების გამოტანა \$*-ს საშუალებით (ბრჭალების გარეშე):

Arg #1 = მიშკა

Arg #2 = ნინო

Arg #3 = იო

არგუმენტები აღიქმება როგორც ცალკეული სიტყვები.

არგუმენტების გამოტანა \$@-ს საშუალებით (ბრჭალების გარეშე):

Arg #1 = მიშკა

Arg #2 = ნინო

Arg #3 = იო

არგუმენტები აღიქმება როგორც ცალკეული სიტყვები.

achiko@debian:~\$./script_XYZ.sh მიშკა "ნინო იო"

არგუმენტების გამოტანა "\$*" -ს საშუალებით:

Arg #1 = მიშკა ნინო იო

არგუმენტთა სრული სია აღიქმება როგორც ერთი სიტყვა.

არგუმენტების გამოტანა "\$@" -ს საშუალებით:

Arg #1 = მიშკა

Arg #2 = ნინო იო

არგუმენტები აღიქმება როგორც ცალკეული სიტყვები.

არგუმენტების გამოტანა \$*-ს საშუალებით (ბრჭყალების გარეშე):

Arg #1 = მიშკა

Arg #2 = ნინო

Arg #3 = იო

არგუმენტები აღიქმება როგორც ცალკეული სიტყვები.

არგუმენტების გამოტანა \$@-ს საშუალებით (ბრჭყალების გარეშე):

Arg #1 = მიშკა

Arg #2 = ნინო

Arg #3 = იო

არგუმენტები აღიქმება როგორც ცალკეული სიტყვები.

შევვიძლია ვთქვათ, რომ \$@ და \$* ჩანაწერები იდენტურია, ხოლო "\$@" და "\$*" კი განსხვავდება, ამასთან "\$@"-ის ქცევა დამოკიდებულია, იმაზეც, თუ როგორი ფორმით გადავცემთ არგუმენტებს.

სკრიფტზე პოზიციური პარამეტრების გადაცემისას ხშირად გამოიყენებენ shift ბრძანებას. ეს ბრძანება, როგორც თვითონ სიტყვის მნიშვნელობიდან ჩანს, დაძრავს არგუმენტებს ისე, რომ მე-2 პარამეტრის მნიშვნელობა პირველ პარამეტრში ჩაიწეროს, მე-3 მე-2-ში, მე-4 მე-3-ში და ბოლო - ბოლოს წინაში. პირველი პარამეტრის მნიშვნელობა კი, ამ შემთხვევაში, იკარგება. მოდით, მარტივი მაგალითი მოვიყვანოთ:

```
#!/bin/bash
```

```
# გამოვიდახოთ სკრიფტი შემდეგნაირად ./script_XYZ 1 2 3 4 5
```

```
echo "$1"           # "$@" = 1 2 3 4 5
```



```

shift
echo "$1"          # "$@" = 2 3 4 5
shift
echo "$1"          # "$@" = 3 4 5
# ყოველი shift-ის გამოძახებით იკარგება $1-ის წინა მნიშვნელობა.
achiko@debian:~$ ./script_XYZ.sh 1 2 3 4 5
1
2
3

```

სხვა მაგალითიც მოვიყვანოთ:

```

#!/bin/bash

while (( $# ))
do
    echo -n "არგუმენტების რაოდენობა: $# - "
    echo "$@"
    shift
done
achiko@debian:~$ ./script_XYZ.sh 1 2 3 4 5
არგუმენტების რაოდენობა: 5 - 1 2 3 4 5
არგუმენტების რაოდენობა: 4 - 2 3 4 5
არგუმენტების რაოდენობა: 3 - 3 4 5
არგუმენტების რაოდენობა: 2 - 4 5
არგუმენტების რაოდენობა: 1 - 5

```

შესაძლებელია, **shift** ბრძანებას არგუმენტად რიცხვი გადავცეთ, ასე: **shift N** და, ამ შემთხვევაში, ის 1-ის ნაცვლად N ბიჯით დაძრავს არგუმენტებს.

მაგალითი 16.2: სკრიფტის PID, \$\$

```

#!/bin/bash

echo "ეს ხაზი გამოჩნდება"
kill $$
echo "ეს ხაზი აღარ გამოჩნდება"
exit 0      # ნორმალურ გამოსვლაზე აღარ მივა სკრიფტი
achiko@debian:~$ ./script_XYZ.sh
ეს ხაზი გამოჩნდება
Terminated
achiko@debian:~$

```

მაგალითი 16.3: სკრიფტში ფონურ რეჟიმში ბოლო ბრძანების PID, \$!

```

#!/bin/bash

```

```

xlogo &
gedit &
xclock &      # ეს არის ფონურ რეჟიმში ბოლო გაშვებული პროგრამა

echo "$!"

achiko@debian:~$ ./script_XYZ.sh
14551
achiko@debian:~$ ps      # შევამოწმოთ PID-ები
  PID TTY          TIME CMD
 14305 pts/1        00:00:00 bash
 14549 pts/1        00:00:00 xlogo
 14550 pts/1        00:00:00 gedit
 14551 pts/1        00:00:00 xclock
 14558 pts/1        00:00:00 ps

```

მაგალითი 16.4: წინა ბრძანების ბოლო არგუმენტი, \$-

```

#!/bin/bash

echo $-      # ./script_XYZ.sh, გააჩნია რა ფორმით უშვებთ სკრიფტს

du >/dev/null      # ბრძანების სტ. გამოსასვლელს ეკრანზე არ ვუშვებთ
echo $-      # du

ls -al >/dev/null  # ბრძანების სტ. გამოსასვლელს ეკრანზე არ ვუშვებთ
echo $-      # -al (ბოლო არგუმენტი)

mkdir -p dira/dirb/dirc
echo $-      # dira/dirb/dirc (ბოლო არგუმენტი)

:
echo $-      # :
achiko@debian:~$ ./script_XYZ.sh
./script_XYZ.sh
du
-al
dira/dirb/dirc
:

```

16.8 ფუნქციები

როგორც დაპროგრამების ყველა „ნამდვილ“ ენაშია, bash-შიც არსებობს ფუნქციები. ფუნქცია - ეს არის გარკვეული ინსტრუქციების ერთობლიობა, დაჯგუფება და ის ძალიან სასარგებლო შეიძლება აღმოჩნდეს მაშინ, როდესაც ამოცანების განმეორებითი გაშვებაა საჭირო. გარდა ამისა, ფუნქცია კოდს უფრო ადვილად წაკითხვადს ხდის და საშუალებას იძლევა, რომ პროგრამის გარკვეული ნაწილები ცალ-ცალკე გავუშვათ და ასე ვატესტიროთ.

ფუნქცია შეიძლება ორი ხერხით განისაზღვროს:

```
function name {  
    command1;  
    command2;  
    ...  
}
```

ან

```
name () {  
    command1;  
    command2;  
    ...  
}
```

ის შეიძლება კომპაქტურად ერთ ხაზზეც ჩაიწეროს. მარტივი მაგალითი მოვიყვანოთ:

```
achiko@debian:~$ func () { echo "ეს არის ფუნქცია"; echo;}
```

მისი გამოძახება კი უბრალოდ სახელის ჩაწერით ხდება:

```
achiko@debian:~$ func  
ეს არის ფუნქცია
```

შესაძლებელია, ერთი ფუნქციიდან მეორე ფუნქცია გამოიძახოთ. ასევე, შესაძლებელია რეკურსიული¹² ფუნქციის (ფუნქციის თავის თავში გამოძახება) შექმნა.

დაიმახსოვრეთ!

- 1) ფუნქციის განსაზღვრა მის გამოძახებამდე უნდა მოხდეს, ერთმნიშვნელოვნად!
- 2) ფუნქცია არ შეიძლება იყოს ცარიელი!

```
achiko@debian:~$ test ()  
> {  
> }  
-bash: syntax error near unexpected token `}'
```

¹²რეკურსია არის ამოცანის გადაწყვეტის მეთოდი, როდესაც ამონახსენი დამოკიდებულია იმავე ამოცანის წინა წევრებისთვის მიღებულ შედეგზე. კომპიუტერულ მეცნიერებებში რეკურსია ერთ-ერთ მთავარ პარადიგმას წარმოადგენს.

```
achiko@debian:~$ test () { }  
-bash: syntax error near unexpected token `}'
```

თუ ახალ ფუნქციას შეგვქმნით იგივე დასახელებით, ძველი მნიშვნელობა დაიკარგება:

```
achiko@debian:~$ func ()  
> {  
> echo "პირველი ვერსია"  
> }  
achiko@debian:~$ func  
პირველი ვერსია  
achiko@debian:~$ func () { echo "მეორე ვერსია"; echo;}  
achiko@debian:~$ func  
მეორე ვერსია
```

ფუნქციის მთლიანად წაშლა კი ასე შეგეძლება:

```
achiko@debian:~$ unset -f func  
achiko@debian:~$ func  
-bash: func: command not found
```

ფუნქციის გაშვებისას შესაძლებელია, მას არგუმენტები გადავცეთ. ამ შემთხვევაში, ფუნქცია მათ აღიქვამს როგორც, ჩვეთნთვის უკვე კარგად ცნობილ, პოზიციურ პარამეტრებს და ისინი ფუნქციაში იმავენაირად წარმოდგინდება, როგორც სკრიფტში - \$1, \$2 და ა.შ. ცვლადებით.

```
achiko@debian:~$ func ()  
> {  
> echo "პირველი არგუმენტი: $1"  
> echo "მეორე არგუმენტი: $2"  
> echo "მესამე არგუმენტი: $3"  
> }  
achiko@debian:~$ func მიშკა ნინო იო  
პირველი არგუმენტი: მიშკა  
მეორე არგუმენტი: ნინო  
მესამე არგუმენტი: იო
```

თუ ფუნქცია სკრიფტში გაქვთ განსაზღვრული, მაშინ მასში გამოყენებული \$1, \$2 ... ცვლადები ფუნქციის პოზიციური პატარემტრებია, ხოლო ფუნქციის გარეთ გამოყენებული \$1, \$2 ... ცვლადები სკრიფტზე გადაცემულ არგუმენტებად იქნება აღქმული. ვნახოთ მაგალითი:

```
#!/bin/bash
```

```
func () {
    echo "ფუნქციის პირველი არგუმენტი: $1"
    echo "ფუნქციის მეორე არგუმენტი: $2"
    echo "ფუნქციის მესამე არგუმენტი: $3"
}
```

func ერთი ორი სამი

```
echo
echo "სკრიფტის პირველი არგუმენტი: $1"
echo "სკრიფტის მეორე არგუმენტი: $2"
echo "სკრიფტის მესამე არგუმენტი: $3"
```

```
achiko@debian:~$ ./script_XYZ.sh one two three
```

ფუნქციის პირველი არგუმენტი: ერთი

ფუნქციის მეორე არგუმენტი: ორი

ფუნქციის მესამე არგუმენტი: სამი

სკრიფტის პირველი არგუმენტი: one

სკრიფტის მეორე არგუმენტი: two

სკრიფტის მესამე არგუმენტი: three

როგორ გავნსაზღვროთ ფუნქციის გამოსასვლელი კოდი? აქ ჩვენთვის ნაცნობი **exit** N ბრძანება აღარ გამოგვადგება, რადგან ის, არა მხოლოდ ფუნქციიდან, არამედ მთლიანად სკრიფტიდან გამოგვიყვანს. მხოლოდ ფუნქციის დასრულება **return** N ბრძანებით ხორციელდება. ვნახოთ:

```
#!/bin/bash
```

```
func () {
    echo "ფუნქციის პირველი არგუმენტი: $1"
    echo "ფუნქციის მეორე არგუმენტი: $2"
    echo "ფუნქციის მესამე არგუმენტი: $3"
    exit 12
}
```

func ერთი ორი სამი

```
echo $?
```

```
echo
echo "სკრიფტის პირველი არგუმენტი: $1"
echo "სკრიფტის მეორე არგუმენტი: $2"
echo "სკრიფტის მესამე არგუმენტი: $3"
```

```
achiko@debian:~$ ./script_XYZ.sh one two three
```

ფუნქციის პირველი არგუმენტი: ერთი

ფუნქციის მეორე არგუმენტი: ორი

ფუნქციის მესამე არგუმენტი: სამი

```
achiko@debian:~$ echo $?
```

ახლა `exit`-ის ნაცვლად `return` გამოვიყენოთ და დავრწმუნდეთ ნათქვამში:

```
#!/bin/bash

func () {
    echo "ფუნქციის პირველი არგუმენტი: $1"
    echo "ფუნქციის მეორე არგუმენტი: $2"
    echo "ფუნქციის მესამე არგუმენტი: $3"
    return 12
}

func ერთი ორი სამი
echo $?

echo
echo "სკრიფტის პირველი არგუმენტი: $1"
echo "სკრიფტის მეორე არგუმენტი: $2"
echo "სკრიფტის მესამე არგუმენტი: $3"
achiko@debian:~$ ./script_XYZ.sh one two three
ფუნქციის პირველი არგუმენტი: ერთი
ფუნქციის მეორე არგუმენტი: ორი
ფუნქციის მესამე არგუმენტი: სამი
12

სკრიფტის პირველი არგუმენტი: one
სკრიფტის მეორე არგუმენტი: two
სკრიფტის მესამე არგუმენტი: three
achiko@debian:~$ echo $?
0
```

რეკურსიული ფუნქციის მაგალითი მოვიყვანოთ. სკრიფტს არგუმენტად რიცხვი გადავცეთ და ეკრანზე ამდენივე ფიბონაჩის¹³ რიცხვი გამოვიტანოთ (ფიბონაჩის მიმდევრობაში წევრის მნიშვნელობა მისი წინა 2 წევრის მნიშვნელობების ჯამის ტოლია):

```
#!/bin/bash

if [ ! -n "$1" ]
then
    echo "გამოყენის წესი: `basename $0` N (N ცალი ფიბონაჩის რიცხვის გამოტანა)"
```

¹³ფიბონაჩის რიცხვები მჭიდრო კავშირშია ბუნებასთან. ამ რიცხვებს ვხვდებით ხეების დატოტვისას, ტოტებზე ფოთლების მოწყობის დროს, მზესუმზირაში მარცვლების განლაგებისას, ნიჟარაზე სპირალის აგებულებისას და ა.შ. საოცარია, მაგრამ ფიბონაჩის ერთი წევრის მის წინაზე გაყოფით (ზღვარში) ვიღებთ ოქროს კვეთის მნიშვნელობას. ოქროს პროპორცია, ოქროს შუალედი კი არის პარმონიული გაყოფა მთელისა ისეთ ორ არატოლ ნაწილად, როდესაც მცირე ნაწილი იხე შეეფარდება დიდს, როგორც დიდი მთელს ($\approx 1,61803\dots$).

```

        exit 85
fi

fibonacci () {
    index=$1
    if (( $index == 0 ))
    then
        echo -n "0 "
    elif (( $index == 1 ))
    then
        echo -n "1 "
    else
        ((--index)); x=$(fibonacci $index)
        ((--index)); y=$(fibonacci $index)
        echo -n "$((x+y)) "
    fi
}

for i in `seq 0 $((($1-1))`
do
    fibonacci $i
done; echo
achiko@debian:~$ ./script_XYZ.sh 17
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

```

თუ კარგად დააკვირდებით, შენიშავთ, რომ სკრიფტის შესრულებამ გარკვეული დრო წაიღო. რეკურსია მეტ-ნაკლებად ანელებს სკრიფტის პროცესს. ის ზომ თავის თავს იძახებს შესრულების პროცესში. შეგვიძლია სკრიფტის გაშვების პარალელურად, სხვა ტერმინალიდან პროცესების სიას დავაკვირდეთ და დავრწმუნდეთ, რომ ის რამდენჯერმე იქნება გამოძახებული დროის კონკრეტულ მომენტში. ამაში `watch` ბრძანება დაგვეხმარება, რომელიც მოცემული ბრძანების გამოსასვლელს გარკვეული პერიოდულობით განახლებს ეკრანზე:

```

achiko@debian:~$ watch -n 0.3 pstree -A
Every 0.3s: pstree -A

...
|-login---bash---script_XYZ.sh---script_XYZ.sh---script_XYZ.sh ...
...

```

შეგვიძლია, აგრეთვე, `top` ბრძანება გამოვიყენოთ (ჩამოვიდეთ მიმართულების ისრის გამოყენებით ჩვენი პროცესის სანახავად და მასში გავააქტიუროთ ხესებრი სტრუქტურის რეჟიმი `V` ინსტრუქციით) და, სკრიფტის მუშაობის პარალელურად, დინამიკურად დავაკვირდეთ მას.

```

achiko@debian:~$ top
...

```

```

... 0:00.00      \- script_XYZ.sh
... 0:00.00      \- script_XYZ.sh
... 0:00.00      \- script_XYZ.sh
... 0:00.00      \- script_XYZ.sh
... 0:00.00      \- script_XYZ.sh
... 0:00.00      \- script_XYZ.sh
... 0:00.00      \- script_XYZ.sh
... 0:00.00      \- script_XYZ.sh
...

```

ფიბონაჩის მიმდევრობის სკრიფტი კომპაქტურად შეგვიძლია ასეც ჩავწეროთ:

```

#!/bin/bash

([ ! -n "$1" ] && echo "გამოყენის წესი: `basename $0` N (N ცალი
ფიბონაჩის რიცხვის გამოტანა)"; exit 85)

f(){((($1<2))&&echo $1||echo ${f ${1-1}}+${f ${1-2}});}

for i in `seq 0 $1`; do f $i; done
achiko@debian:~$ ./script_XYZ.sh 9
0
1
1
2
3
5
8
13
21
34

```

შენიშნეთ, ალბათ, რომ პირველი ზაზი ფრჩხილებშია მოქცეული. ახლა მოაშორეთ ისინი და თავიდან სცადეთ. რა შედეგს მიიღებთ? ამ საკითხს მოგვიანებით დავუბრუნდებით.

ამჯერად, რეკურსიის გარეშე დავწეროთ ეს პროგრამა. ამგვარად, სკრიფტი გაცილებით სწრაფქმედი გახდება. ვცადოთ:

```

#!/bin/bash

n=$1
a=0; b=1; i=2
echo $a; echo $b
while ((i<n))
do
    i=$((i+1)); c=$((a+b))
    echo $c
    a=$b; b=$c
done

```



```

done
achiko@debian:~$ ./script_XYZ.sh 90
0
1
1
2
3
5
8
...
987
1597
2584
...
6557470319842
10610209857723
17167680177565
...
420196140727489673
679891637638612258
1100087778366101931
1779979416004714189

```

წინა სკრიფტებში რეკურსიულობა ფუნქციაში ფუნქციის გამოძახებით შევქმენით. თუმცა, შეგვიძლია ფუნქციას თავი ავარიდოთ და, მის ნაცვლად, თავად სკრიფტი გამოვიძახოთ სკრიფტში. მაგალითის სანახავად წარმოვიდგინოთ შემდეგი სცენარი: ჩამოვიართ ყველა ფაილი (ქვედირექტორიების ჩათვლით) და ეკრანზე მხოლოდ ჩვეულებრივი ტიპის ფაილი გამოვიტანოთ. ამ ამოცანის გადასაჭრელად ასე შეგვიძლია მოვიქცეთ - მიმდინარე დირექტორიიდან სკრიფტს მხოლოდ ჩვეულებრივი ტიპის ფაილი გამოვატანინოთ, ხოლო თუ დირექტორია შეგვხვდებათ, მაშინ საკმარისია, მასში გადავიდეთ და ჩვენი სკრიფტი იქ გავუშვათ. შემდეგ იქიდან გამოვიდეთ, რათა სხვა დირექტორიებში ჩამოვლა არ გამოგვრჩეს. მოდით, განვახორციელოთ ეს იდეა.

```

#!/bin/bash

for i in *
do
    [ -f "$i" ] && echo "$i"
    if [ -d "$i" ]
    then
        cd "$i"
        /home/achiko/script_XYZ.sh
        cd ..
    fi
done

achiko@debian:~$ ./script_XYZ.sh
cities.txt

```

```
countries.txt
GPL-3
script_XYZ.sh
Tux.jpg
...
```

სკრიფტში ამავე სკრიფტის გამოძახებისას მისი სრული გზა უნდა მიეუთითოთ.

16.9 სკრიფტის ოფციები

შელში ბრძანების გაშვებისას ვიცით, რომ შესაძლებელია მისთვის ოფციის გადაცემა. ნუთუ იგივეს გაკეთება შეგვიძლია სკრიფტზეც? რა თქმა უნდა, დიახ. სრულიად შესაძლებელია, რომ გავაანალიზოთ სკრიფტზე გადაცემული არგუმენტები და თუ რომელიმე მათგანი ტირეთი იწყება, შეგვიძლია, ის ჩვენთვის წინასწარ განსაზღვრულ ოფციად მივიჩნიოთ და შესაბამისი ქმედებები მოვაყოლოთ. ამ კოდის დაწერა რთული არ არის, თუმცა შელში უკვე არსებობს ასეთი საშუალება. ამიტომ, სჯობს თავი ავარიდოთ „ველოსიპედის ხელახლა გამოგონებას“. ბრძანება `getopts`-ს შეუძლია გააანალიზოს სკრიფტის მთლიანი ხაზი და გადაარჩიოს გადაცემული ოფციები და მისი არგუმენტები. ის იყენებს ორ ცვლადს: `OPTARG`-ს (`OPT`ion `ARG`ument), რომელშიც ინახება ოფციაზე გადაცემული არგუმენტი თუ ასეთი არსებობს და `OPTIND`-ს (`OPT`ion `IND`ex), რომელიც არგუმენტის მიმთითებელს წარმოადგენს. ის წარმოადგენს რიცხვს - შემდეგი დასამუშავებელი არგუმენტის ინდექსს (ინდექსის ნომერი იწყება 1-ით). `getopts` ბრძანებას, როგორც წესი, `while` ციკლში წერენ ასე: `while getopts "ab:cd:e"....` ასეთი ჩანაწერი ნიშნავს, რომ სკრიფტს შეიძლება გადაეცეთ შემდეგი ოფციები: `a`, `b`, `c`, `d` ან/და `e`. ორწერტილი იმაზე მიუთითებს, რომ მის წინ მდგომ ოფციას არგუმენტი უნდა გადაეცეთ, რომელიც `OPTARG` ცვლადში ჩაიწერება (ჩვენს შემთხვევაში ასეთებია `b` და `d` ოფცია). მოვიყვანოთ მარტივი მაგალითი:

```
#!/bin/bash

while getopts "ab:cd:e" Option
do
    case $Option in
        a|c) echo "გადაცემულია $Option ოფცია";;
        b) echo "გადაცემულია $Option ოფცია. მისი არგუმენტია $OPTARG";;
        d) echo "გადაცემულია $Option ოფცია";;
        e) echo "გადაცემულია $Option ოფცია. მისი არგუმენტია $OPTARG";;
    esac
done

achiko@debian:~$ ./script_XYZ.sh -a
გადაცემულია a ოფცია
achiko@debian:~$ ./script_XYZ.sh -a -c
გადაცემულია a ოფცია
გადაცემულია c ოფცია
achiko@debian:~$ ./script_XYZ.sh -ac
გადაცემულია a ოფცია
გადაცემულია c ოფცია
achiko@debian:~$ ./script_XYZ.sh -acbTEST1
```

```

გადაცემულია a ოფცია
გადაცემულია c ოფცია
გადაცემულია b ოფცია. მისი არგუმენტია TEST1
achiko@debian:~$ ./script_XYZ.sh -ac -b TEST1
გადაცემულია a ოფცია
გადაცემულია c ოფცია
გადაცემულია b ოფცია. მისი არგუმენტია TEST1
achiko@debian:~$ ./script_XYZ.sh -acb
გადაცემულია a ოფცია
გადაცემულია c ოფცია
./script_XYZ.sh: option requires an argument -- b
achiko@debian:~$ ./script_XYZ.sh -acdbTEST1 -eTEST2
გადაცემულია a ოფცია
გადაცემულია c ოფცია
გადაცემულია d ოფცია
გადაცემულია b ოფცია. მისი არგუმენტია TEST1
გადაცემულია e ოფცია. მისი არგუმენტია TEST2

```

იმ შემთხვევაში თუ **b** და **e** ოფციებს არგუმენტს არ გადავცემთ, ეკრანზე შეტყობინება გამოვა, რომელიც მიგვითითებს, რომ მას არგუმენტი სჭირდება. ამის თავიდან ასარიდებლად საკმარისია `getopts`-ს ოფციების სია ორწერტილით დავიწყოთ. ამ ქმედებით ჩავრთავთ ე.წ. შეცდომების ჩუმი გამოტანის რეჟიმს (silent error reporting). ამ დროს არასწორად გადაცემული ოფცია უგულებელყოფილი იქნება.

```

#!/bin/bash

while getopts ":ab:cd:e" Option
do
    case $Option in
        a|c|e) echo "გადაცემულია $Option ოფცია";;
        b|d) echo "გადაცემულია $Option ოფცია არგუმენტით $OPTARG";;
        esac
    done
achiko@debian:~$ ./script_XYZ.sh -acb
გადაცემულია a ოფცია
გადაცემულია c ოფცია
achiko@debian:~$ ./script_XYZ.sh -acbTEST1
გადაცემულია a ოფცია
გადაცემულია c ოფცია
გადაცემულია b ოფცია არგუმენტით TEST1

```

ვნახოთ, რას ნიშნავს `OPTIND` ცვლადი. ის არის არა მიმდინარე, არამედ მომდევნო დასამუშავებელი არგუმენტის ინდექსის ნომერი. მოვიყვანოთ მაგალითები:

```

#!/bin/bash

```

```
while getopts "abc:" Option
do
    echo "$Option" $OPTIND $OPTARG
done
achiko@debian:~$ ./script_XYZ.sh -a -b -c TEST
a 2
b 3
c 5 TEST
```

ავხსნათ, თუ რატომ მივიღეთ ეს შედეგი. ამ სკრიფტს გადავცით 4 არგუმენტი: არგუმენტი 1 - a, არგუმენტი 2 - b, არგუმენტი 3 - c, არგუმენტი 4 - TEST. რადგან ოფციისთვის OPTIND არის შემდეგი არგუმენტის ინდექსი, a-სთვის ის იქნება 2, b ოფციისთვის 3, ხოლო c-სთვის 5. თუ სკრიფტს ასეთი ფორმით გავუშვებთ - ./script_XYZ.sh -a -b -cTEST, მაშინ სულ 3 არგუმენტი გვექნება და c ოფციისთვის OPTIND ცვლადში 5-ის ნაცვლად 4 ჩაიწერება. დავწმუდებთ:

```
achiko@debian:~$ ./script_XYZ.sh -a -b -cTEST
a 2
b 3
c 4 TEST
```

```
achiko@debian:~$ ./script_XYZ.sh -ab -c TEST
a 1
b 2
c 4 TEST
```

ბოლო მაგალითში პირველი არგუმენტია ab, მეორე c, მესამე კი TEST. a ოფციის შემდეგ გამოძახებული ოფციის (ანუ b ოფციის) არგუმენტი კვლავ 1 არის. ამიტომ გამოვიდა a-სთვის 1. b ოფციის შემდეგ გამოძახებული ოფციის არგუმენტი უკვე არის 2, ხოლო c-სთვის კი იქნება 4, იმიტომ, რომ TEST c ოფციის შემადგენელი ნაწილია და ეს ორი პარამეტრი მთლიანობაში აღიქმება. რადგან TEST მესამე არგუმენტია, მისი მომდევნო ნომერი 4 გამოდის.

ვნახოთ სხვადასხვა ვარიაციაც:

```
achiko@debian:~$ ./script_XYZ.sh -abc TEST
a 1
b 1
c 3 TEST
achiko@debian:~$ ./script_XYZ.sh -abcTEST
a 1
b 1
c 2 TEST
```

როგორც უკვე ვნახეთ, OPTIND-ის საწყისი მნიშვნელობა სკრიფტის ყოველი გაშვებისას 1-ით განისაზღვრება. თუმცა, თუ რამდენჯერმე მოხდა getopt-ის გამოყენება ერთ პროცესში, შელი ამ ცვლადს საწყისი მნიშვნელობაზე ავტომატურად არ დააყენებს. ამ დროს ის ხელით უნდა გადავიყვანოთ პირველად მნიშვნელობაზე. ამაში, შემდეგი ბრძანება

დაგვეხმარება: `shift "$((OPTIND-1))"`. არ უნდა დაგვავიწყდეს ის, რომ, თუ ეს ეტაპი სკრიფტში გამოგვრჩა და ოფციების გარდა, სკრიფტზე გადაცემული გვაქვს არგუმენტები, მათი იდენტიფიცირება გავიჭირდება. დავრწმუნდეთ ამაში:

```
#!/bin/bash

NO_ARGS=0
E_OPTERROR=85

if [ $# -eq "$NO_ARGS" ]
then
    echo "გამოყენება: `basename $0` ოფციები [-abcde]"
    exit $E_OPTERROR
fi

while getopts ":ab:cd:e" Option
do
    case $Option in
        a|c|e) echo "გადაცემულია $Option ოფცია";;
        b|d) echo "გადაცემულია $Option ოფცია არგუმენტით $OPTARG";;
        esac
    done

    shift $(( $OPTIND - 1 ))

    echo; echo "ოფციების გარდა სკრიფტზე გადაცემული $# არგუმენტი. ესენია:"
    for i in $@
    do
        echo "$i"
    done
    exit $?

achiko@debian:~$ ./script_XYZ.sh -a -c -d TEXT arg1 arg2 arg3
გადაცემულია a ოფცია
გადაცემულია c ოფცია
გადაცემულია d ოფცია არგუმენტით TEXT

ოფციების გარდა სკრიფტზე გადაცემული 3 არგუმენტი. ესენია:
arg1
arg2
arg3
```

ახლა `shift "$((OPTIND-1))"` ბრძანების გარეშე ვცადოთ:

```
#!/bin/bash

NO_ARGS=0
E_OPTERROR=85
```

```

if [ $# -eq "$NO_ARGS" ]
then
    echo "გამოყენება: `basename $0` ოფციები [-abcde]"
    exit $E_OPTERROR
fi

while getopts ":ab:cd:e" Option
do
    case $Option in
        a|c|e) echo "გადაცემულია $Option ოფცია";;
        b|d) echo "გადაცემულია $Option ოფცია არგუმენტით $OPTARG";;
        esac
    done

    #shift $((OPTIND - 1))

echo; echo "ოფციების გარდა სკრიფტზე გადაცემული $# არგუმენტი. ესენია:"
for i in @$@
do
    echo "$i"
done
exit $?

achiko@debian:~$ ./script_XYZ.sh -a -c -d TEXT arg1 arg2 arg3
გადაცემულია a ოფცია
გადაცემულია c ოფცია
გადაცემულია d ოფცია არგუმენტით TEXT

ოფციების გარდა სკრიფტზე გადაცემული 7 არგუმენტი. ესენია:
-a
-c
-d
TEXT
arg1
arg2
arg3

```

ნათელი გახდა, რომ სკრიფტის არგუმენტების წანაცვლება აუცილებელია `getopts` ბრძანების მიერ ოფციებისთვის აღქმული არგუმენტების რაოდენობით.

თავი 17

ტერმინალები

ტერმინალზე მუშაობისას გაშვებული ბრძანებების შედეგი ეკრანზე შავ-თეთრი ფერებით გამოდის. არის კი შესაძლებელი ტექსტი შელში სხვა ფერებით გამოვიტანოთ? ალბათ, გაგახსენდებათ ბევრი შემთხვევა, როდესაც შელის მოსაწვევში ან სკრიფტების წერის დროს ტერმინალზე მონაცემები სხვადასხვა ფერში გამოისახება. მაგალითად, vi-სა თუ emacs ტექსტურ რედაქტორში კოდის გარკვეული ფრანგმენტები. გამოდის, რომ ტერმინალს ფერების აღქმა შეუძლია. მაშ, როგორ შეგვიძლია, სხვადასხვა ფერით თავად დავბეჭდოთ ტექსტი ტერმინალში? სანამ ამ საკითხს დეტალურად განვიხილავთ, გავისხენოთ თუ რა არის ტერმინალი.

17.1 ტერმინალის საკონტროლო კოდები

ისტორიულად, ტერმინალი წარმოადგენდა ფიზიკურ მოწყობილობას, რომელიც გადასცემდა და იღებდა მონაცემებს და მათ გამოსახავდა ეკრანზე. უმრავლეს შემთხვევაში ტერმინალი დაკავშირებული იყო მთავარ კომპიუტერთან და მისი საშუალებით ახდენდა მომხმარებელი კომპიუტერთან ინტერაქციას. მონაცემების გაცვლა ხდებოდა მიმდევრობითი (serial) პორტის საშუალებით, სადაც ისინი ბაიტ-ბაიტ გადაიცემოდა. ინტერპრეტატორში, მაგალითად, შელში მუშაობისას, კლავიატურიდან აკრეფილი ყოველი ასო-ნიშანი ასეთი ფორმით, ბაიტ-ბაიტ მიედინება კომპიუტერთან და შემდეგ ეკრანზე გამოისახება. თუმცა ყოველი ბაიტის გამოტანისას ყოველთვის ასო-ნიშანი რომ გამოისახებოდეს, მაშინ ასო-ნიშნების გამოსახვის გარდა არაფერი მოხდებოდა დისპლეიზე. ამიტომ, საჭირო გახდა ტერმინალისთვის „ეჭვავლებინათ“ რომ, თუ ზოგიერთი ბაიტი, კლავიატურიდან აკრეფილი სიმბოლო, ან კონკრეტული სიმბოლოთა ერთობლიობა თანმიმდევრობით გადაიცა, მაშინ ეკრანზე გამოსახვის ნაცვლად მოხდეს გარკვეული ქმედების შესრულება. ეს ქმედება შეიძლება იყოს კურსორის გადატანა ეკრანის ერთი პოზიციიდან მეორეზე და სხვა მსგავსი მოქმედებები. მათ ტერმინალის საკონტროლო სიმბოლოებს, მართვის კოდებს უწოდებენ. ეს კოდები ზილული სიმბოლოების გარდა, უზილავ სიმბოლოებსაც (რაც ეკრანზე არ გამოისახება) შეიცავენ.

კომპიუტერული მოწყობილობების გაჩენის შემდეგ, შეიქმნა სხვადასხვა სახის ტერმინალები, როგორიცაა Dumb terminals¹, Graphic GUI Capabilities of Text Terminals², Text terminals. სწორედ ტექსტურ ტერმილაში გადაიცემა მონაცემი ბაიტ-ბაიტ. ისინი შემდეგი სახელწოდებებითაც გვხვდება: General purpose terminal, General display terminal, Serial

¹ის ერთ-ერთი ყველაზე პრიმიტიული, ნელი ტერმინალი იყო. dumb terminal-ს მხოლოდ პროცესორისგან მიღებული მონაცემებს გამოტანა შეეძლო ეკრანზე

²გრაფიკულ ტერმინალს ტექსტის გარდა შეუძლია სურათის გამოტანაც. არსებობს როგორც ვექტორული, ასევე რასტრული რეჟიმში მომუშავე ტერმინალები.

monitor, Serial console, Serial terminal, Character-cell terminal, Character terminal, ASCII/ANSI terminal, Asynchronous terminal, Data terminal, Video terminal, Video display terminal (VDT), Green terminal (მას შემდეგ, რაც მწვანე დისპლეის გამოყენება დაიწყო). არსებობენ ისეთი Text terminal-ებიც, რომლებიც მონაცემებს არა ბაიტ-ბაიტ, არამედ ბლოკურად გადასცემენ. აქ შეყვანილი ასო-ნიშნების გარკვეული რაოდენობა დროებით ინახება ტერმინალის მეხსიერებაში და შემდეგ ერთბაშად, ბლოკურად, გადაეცემა კომპიუტერს. ასეთ რეჟიმში მომუშავე ტერმინალები არ არიან მხარდაჭერილი ლინუქსში.

თავდაპირველად, ტერმინალების მწარმოებელი თითქმის ყველა კომპანია საკუთარ ტერმინალზე მორგებულ საკონტროლო სიმბოლოებს ქმნიდა. დროთა განმავლობაში მათი რიცხვი გაიზარდა. DEC კომპანიამ შექმნა შემდეგი ცნობილი ვიდეო ტერმინალები - VT100, VT101 და ა.შ. IBM-მა - IBM 2250, IBM 2260 და სხვა. მრავალ ტერმინალებს შორის თავსებადობის შესანარჩუნებლად ზოგიერთი ტერმინალი თუ ტერმინალის ემულატორები ფართოდ გავრცელებულ, უკვე შემქნილ კოდებს იყენებდნენ. საყოველთაოდ გავრცელებული ერთ-ერთი ასეთი სპეციალური სიმბოლოების ნაკრებია ANSI³ Control sequences. მიუხედავად იმისა, რომ Unix-ის მსგავსი სისტემებისთვის შექმნეს ბიბლიოთეკების ნაკრები termcap, შემდეგ terminfo (termcap-ის გაუმჯობესებული ვარიანტი), რომლებიც შეიცავენ სხვადასხვა ტერმინალის შესაძლებლობების მონაცემთა ბაზას, ამ სისტემებში მაინც ყოველთვის იყო ინტეგრირებული ANSI Control sequences-ის მხარდაჭერა. სწორედ ამან გამოიწვია ANSI სპეციალური სიმბოლოების ფართოდ გამოყენება. ბევრი პროგრამა, მათ შორის vi და GNU emacs, იყენებს terminfo ან curses ბიბლიოთეკებს (curses თავის მხრივ იყენებს terminfo-ს). შესაბამისად, თეორიულად, ამ პროგრამებს შეუძლიათ ANSI-ის გარდა სხვა ტერმინალებში მუშაობაც, თუმცა დღეს, ეს ძალიან იშვიათად ხდება, რადგანაც ANSI კოდები თითქმის ყველა ტერმინალებისა და ემულატორების მიერ არის მხარდაჭერილი.

სადღეისოდ, ნამდვილი ტერმინალების ნაცვლად ტერმინალის ემულატორებსა და ვირტუალურ ტერმინალებს ვიყენებთ. ისინი ბევრ ოპერაციულ სისტემაში არსებობენ. Unix-ის მსგავს სისტემებში ცნობილია შემდეგი ტერმინალის ემულატორები. ზოგი მათგანი გრაფიკულ გარემოშია (X Window System და Wayland) ინტეგრირებული, ზოგიც პირდაპირ ტექსტური გარემოდან ეშვება.

ჩამოვთვალოთ ზოგიერთი ცნობილი ტერმინალის ემულატორი:

GNOME terminal	ნაგულისხმევი მნიშვნელობის ტერმინალის ემულატორი GNOME გრაფიკულ გარემოში.
Konsole	ნაგულისხმევი მნიშვნელობის ტერმინალის ემულატორი KDE გრაფიკულ გარემოში.
Guake	python-ზე დაფუძნებული ჩამოსაშლელი ტერმინალის ემულატორი GNOME-ში.
Xfce4-terminal	სტანდარტული ტერმინალის ემულატორი Xfce გრაფიკულ გარემოში.
XTerm	სტანდარტული ტერმინალის ემულატორი X11 გრაფიკულ გარემოში.
Terminator	მძლავრი და სრულად მორგებადი ტერმინალის ემულატორი X11 გრაფიკულ გარემოში.
Eterm	ძალიან მსუბუქი ტერმინალის ემულატორი. შექმნეს XTerm-ის შემცველად.

³American National Standards Institute (ANSI) ამერიკის ეროვნული სტანდარტების ინსტიტუტი არის არამომგებიანი ორგანიზაცია, რომელიც ხელს უწყობს შესაბამისობის სტანდარტების განვითარებას შეერთებულ შტატებში. ANSI აგრეთვე წარმოადგენს შეერთებულ შტატებს საერთაშორისო სტანდარტების ორგანიზაციებში და ეხმარება მათ შექმნან საყოველთაოდ მისაღები პრინციპები მრავალ ინდუსტრიაში

Rxvt	XTerm-ის შემცვლელი ემულატორი. დასახელება მოდის გაფართოებული ვირტუალური ტერმინალიდან (extended virtual terminal).
Rxvt-unicode	მსუბუქი ტერმინალის ემულატორი, რომელსაც უნიკოდის წყალობით სხვადასხვა ენების მხარდაჭერა აქვს.
Tilda	თანამედროვე, ჩამოსაშლელი ტერმინალის ემულატორი, რომელიც დაფუძნებულია GTK+ ბიბლიოთეკებზე. ერთ კლავიშზე დაჭერით შეგიძლიათ ახალი ფანჯრის გახსნა ან არსებულის დამალვა.
GNU screen	ტექსტური გარემოს ტერმინალის მულტიპლექსერი ⁴ VT100/ANSI ტერმინალის ემულაციით.
tmux	ტექსტური გარემოს გაუმჯობესებული ტერმინალის მულტიპლექსერი.

Apple macOS სისტემებში ცნობილია Terminal, iTerm2 და სხვა. Microsoft Windows სისტემებში PUTTY, mintty (Cygwin ტერმინალი) და ა.შ.

თითოეული ტერმინალის ემულატორი კონკრეტული საკონტროლო სიმბოლოების ნაკრებს აღიქვამს. ლინუქსში TERM გარემოს ცვლადი შეიცავს იდენტიფიკატორს - სახელს ტერმინალის ტექსტური ფანჯრის შესაძლებლობების შესახებ.

ვნახოთ, რა მნიშვნელობაა მოცემული ჩვენი gnome-terminal-ის TERM ცვლადში?

```
achiko@debian:~$ echo $TERM
xterm-256color
```

TERM გარემოს ცვლადის დანიშნულებაა უთხრას ტერმინალში გაშვებულ პროგრამებს თუ როგორ უნდა იურთიერთონ მათ ტერმინალთან. ეს პროგრამები კითხულობენ ცვლადში ჩაწერილი ტერმინალის ტიპს და გამოითხოვენ Terminfo-ს ბაზიდან ამ ტიპის ტერმინალის შესაძლებლობებს, სპეციალურ სიმბოლოთა კომბინაციებს (ისინი შეიძლება შეიცავდნენ უხილავ ასო-ნიშნებსაც). მაგალითად, მათი შესრულების შედეგად ეკრანზე შეიძლება დავინახოთ, რომ კურსორი გადაადგილდა ერთი პოზიციიდან მეორეზე ან გასუფთავდა ეკრანის ნაწილი, ჩვენ მიერ შეყვანილი ტექსტი მწვანე ფერით გამოვიდა ან სხვა.

ტერმინალის ემულატორის ტექსტური ფანჯარა, როგორც წესი, მინიმუმ 8 ფერს არჩევს. თუმცა, სშირ შემთხვევაში, გარჩევადი ფერების რაოდენობა იდენტიფიკატორშივეა მოცემული, როგორც ეს ჩვენს შემთხვევაშია ნაჩვენები (xterm-256color - 256 სხვადასხვა ფერი). ამაში მარტივად დასარწმუნებლად შეგვიძლია msgcat --color=test ბრძანება გავუშვათ (ეს ბრძანება შედის gettext საინსტალაციო პაკეტში. შესაბამისად, ის დაყენებული უნდა იყოს!). ეს ბრძანება უყურებს TERM გარემოს ცვლადის მნიშვნელობას და იყენებს მასში ჩაწერილი ტერმინალის ტიპის შესაძლებლობებს.

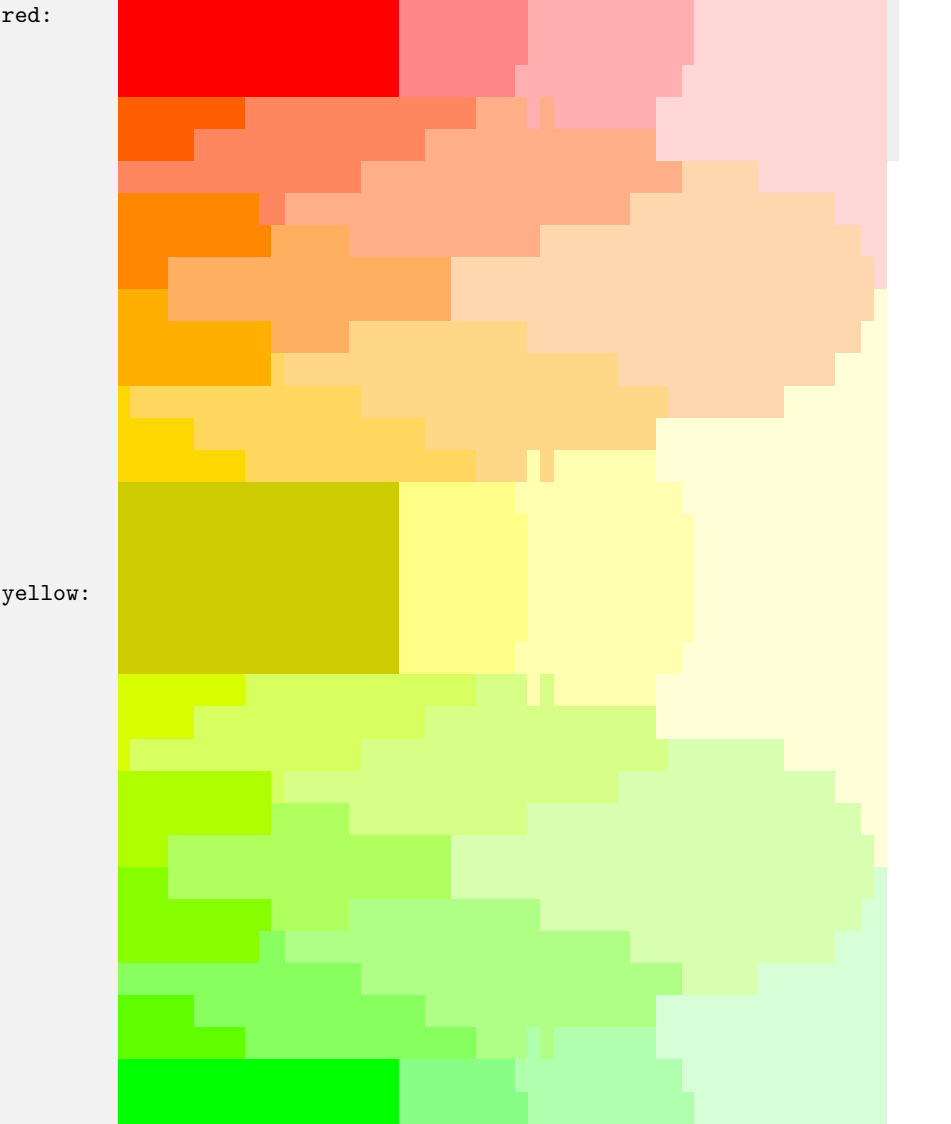
```
achiko@debian:~$ msgcat --color=test
```

⁴მულტიპლექსერი არის მოწყობილობა, სადაც ხდება არხების შემჭიდროება/გაერთიანება. მისი საშუალებით შესაძლებელია ერთი და იმავე არხით ერთდროულად რამდენიმე სიგნალის გაგზავნა. ტერმინალის მულტიპლექსერი წარმოადგენს პროგრამას, რომელიც ტერმინალში მსგავსი მოწყობილობის ემულაციას აკეთებს.

Colors (foreground/background):

	black	blue	green	cyan	red	magenta	yellow	white	default
black		Words	Words	Words	Words	Words	Words	Words	Words
blue	Words		Words	Words	Words	Words	Words	Words	Words
green	Words	Words		Words	Words	Words	Words	Words	Words
cyan	Words	Words	Words		Words	Words	Words	Words	Words
red	Words	Words	Words	Words		Words	Words	Words	Words
magenta	Words	Words	Words	Words	Words		Words	Words	Words
yellow	Words	Words	Words	Words	Words	Words		Words	Words
white	Words	Words	Words	Words	Words	Words	Words		Words
default		Words	Words	Words	Words	Words	Words	Words	

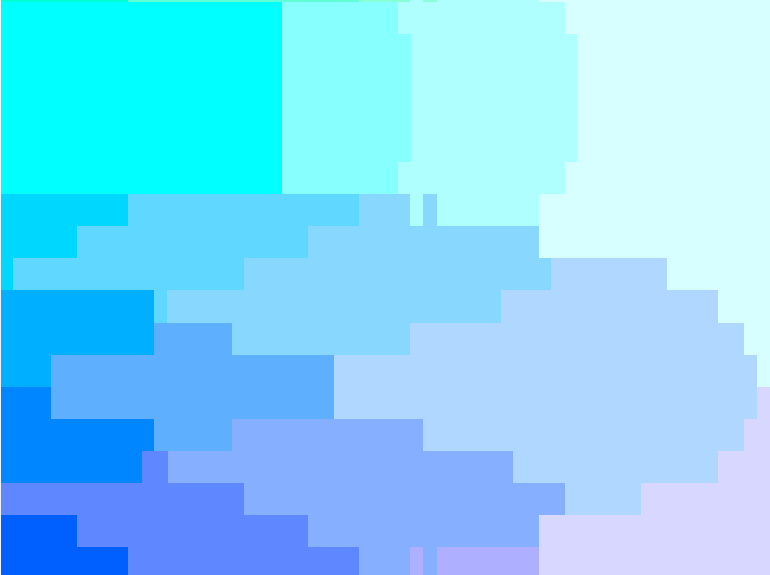
Colors (hue/saturation):



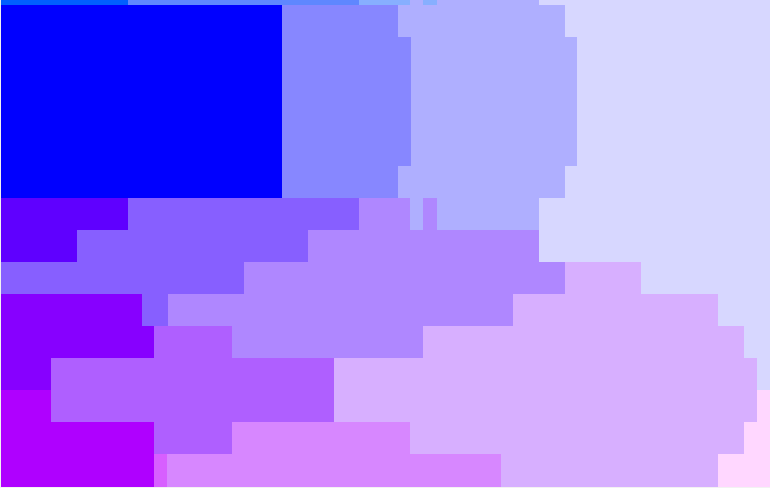
green:



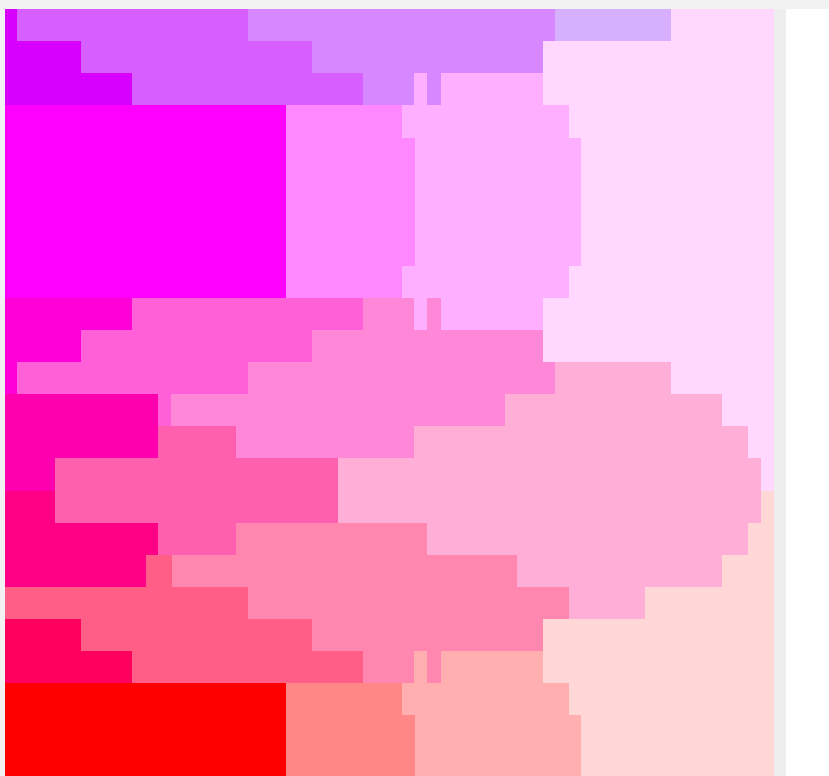
cyan:



blue:



magenta:



red:

Weights:

normal, bold, default

Postures:

normal, *italic*, default

Text decorations:

normal, underlined, default

Colors (foreground) mixed with attributes:

black	normal bold normal <i>italic</i> normal <u>underlined</u> normal
	normal <i>bold+italic</i> normal <u>bold+underl</u> normal <i><u>italic+underl</u></i> normal
blue	normal bold normal <i>italic</i> normal <u>underlined</u> normal
	normal <i>bold+italic</i> normal <u>bold+underl</u> normal <i><u>italic+underl</u></i> normal
green	normal bold normal <i>italic</i> normal <u>underlined</u> normal
	normal <i>bold+italic</i> normal <u>bold+underl</u> normal <i><u>italic+underl</u></i> normal
cyan	normal bold normal <i>italic</i> normal <u>underlined</u> normal
	normal <i>bold+italic</i> normal <u>bold+underl</u> normal <i><u>italic+underl</u></i> normal
red	normal bold normal <i>italic</i> normal <u>underlined</u> normal
	normal <i>bold+italic</i> normal <u>bold+underl</u> normal <i><u>italic+underl</u></i> normal
magenta	normal bold normal <i>italic</i> normal <u>underlined</u> normal
	normal <i>bold+italic</i> normal <u>bold+underl</u> normal <i><u>italic+underl</u></i> normal
yellow	normal bold normal <i>italic</i> normal <u>underlined</u> normal
	normal <i>bold+italic</i> normal <u>bold+underl</u> normal <i><u>italic+underl</u></i> normal
white	normal bold normal <i>italic</i> normal <u>underlined</u> normal

```

|normal|bold+italic|normal|bold+underl|normal|italic+underl|normal|
default|normal|bold|normal|italic|normal|underlined|normal|
|normal|bold+italic|normal|bold+underl|normal|italic+underl|normal|

Colors (background) mixed with attributes:
black
blue
green
cyan
red
magenta
yellow
white
default

```

`xterm-256color` ტიპის ტერმინალში კარგად ჩანს ფერების სიუხვე (256 ფერი).
შევცვალოთ TERM გარემოს ცვლადის მნიშვნელობა და ვნახოთ რა შედეგს მივიღებთ:

```

achiko@debian:~$ export TERM=dumb
achiko@debian:~$ msgcat --color=test
Colors (foreground/background):
|black|blue|green|cyan|red|magenta|yellow|white|default
black|Words|Words|Words|Words|Words|Words|Words|Words|Words
blue|Words|Words|Words|Words|Words|Words|Words|Words|Words
green|Words|Words|Words|Words|Words|Words|Words|Words|Words
cyan|Words|Words|Words|Words|Words|Words|Words|Words|Words
red|Words|Words|Words|Words|Words|Words|Words|Words|Words
magenta|Words|Words|Words|Words|Words|Words|Words|Words|Words
yellow|Words|Words|Words|Words|Words|Words|Words|Words|Words
white|Words|Words|Words|Words|Words|Words|Words|Words|Words
default|Words|Words|Words|Words|Words|Words|Words|Words|Words

Colors (hue/saturation):
red:

```

yellow:

green:

cyan:

blue:

magenta:

red:

Weights:
normal, bold, default

Postures:
normal, italic, default

Text decorations:
normal, underlined, default

Colors (foreground) mixed with attributes:

black	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
blue	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
green	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
cyan	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
red	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
magenta	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
yellow	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
white	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
default	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal

Colors (background) mixed with attributes:

black	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
blue	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
green	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
cyan	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
red	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
magenta	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
yellow	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
white	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal
default	normal bold normal italic normal underlined normal
	normal bold+italic normal bold+underl normal italic+underl normal

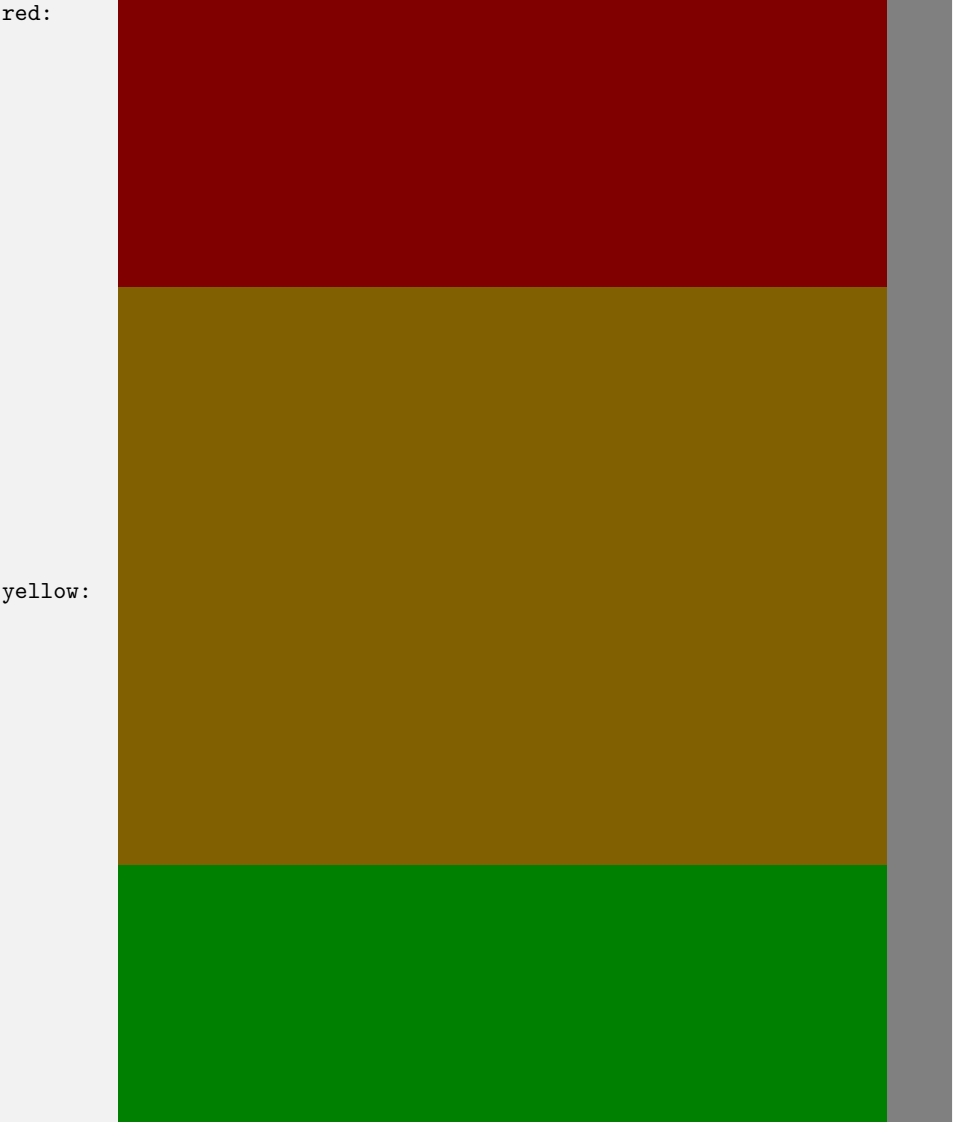
Dumb ტერმინალებზე არათუ ფერის, ტექსტის ფორმატირების ნახვაც შეუძლებელია.

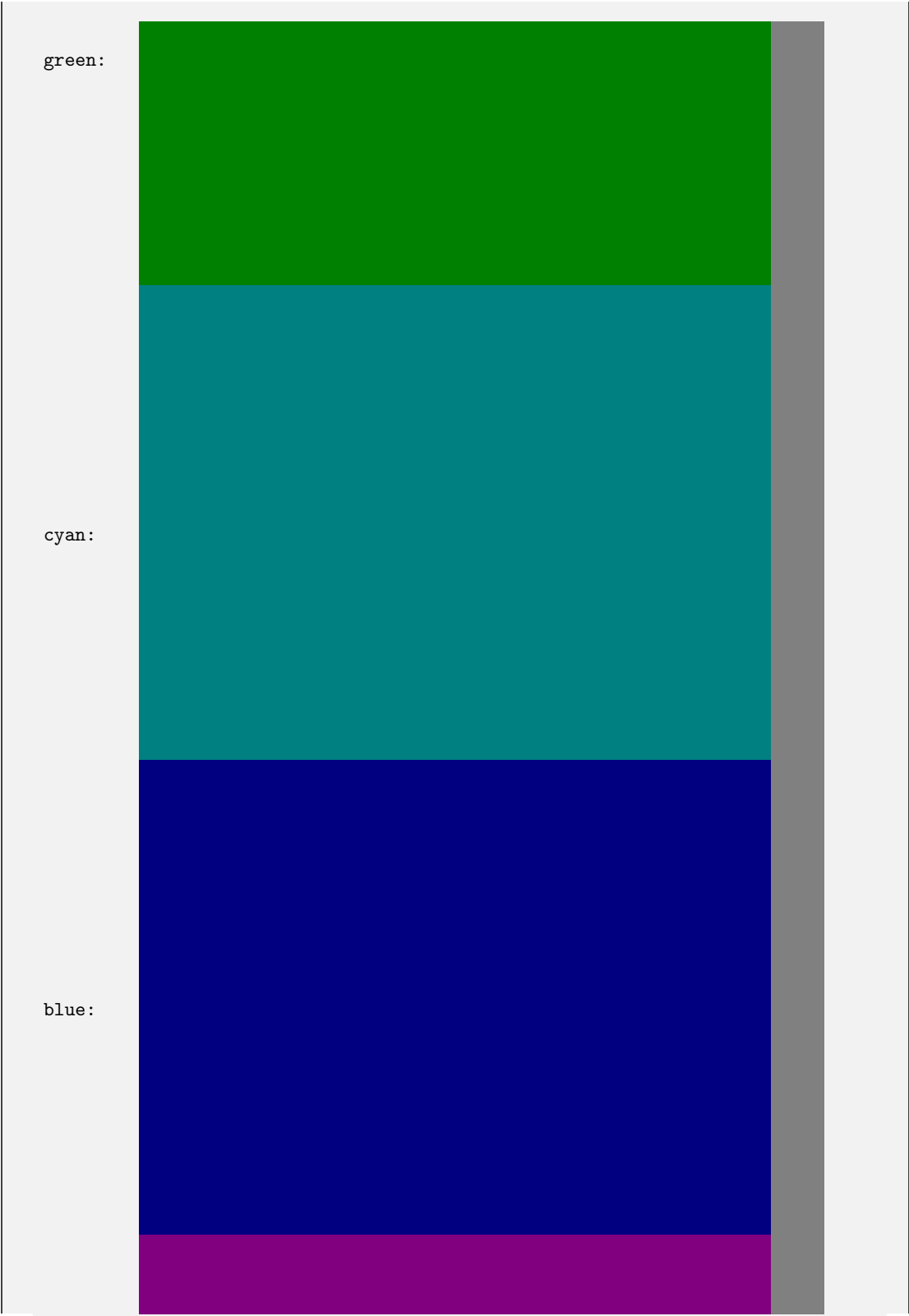
```
achiko@debian:~$ export TERM=xterm
achiko@debian:~$ msgcat --color=test
```


Colors (foreground/background):

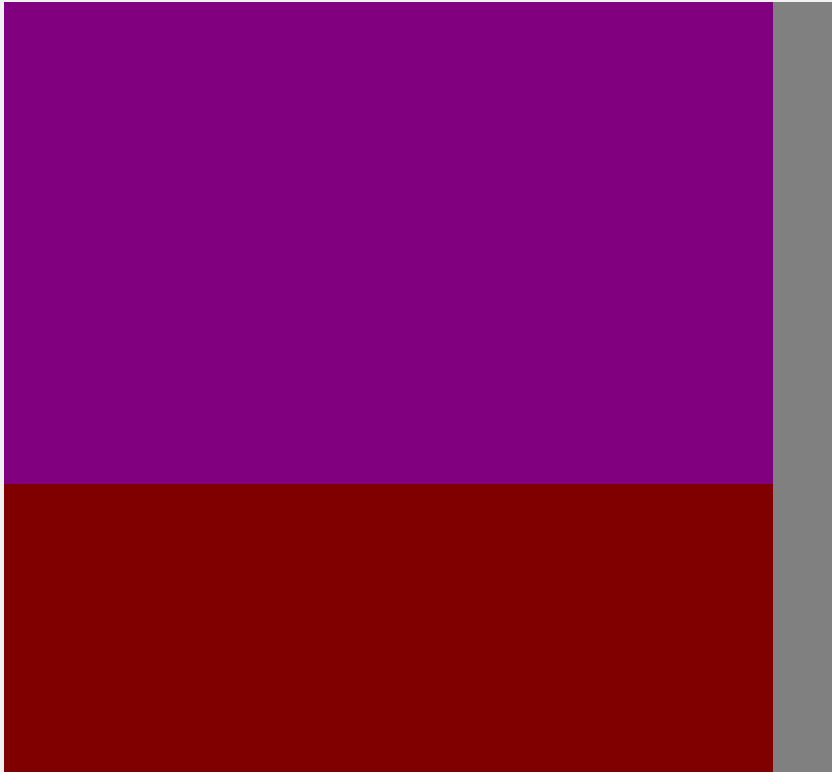
	black	blue	green	cyan	red	magenta	yellow	white	default
black		Words	Words	Words	Words	Words	Words	Words	Words
blue	Words		Words	Words	Words	Words	Words	Words	Words
green	Words	Words		Words	Words	Words	Words	Words	Words
cyan	Words	Words	Words		Words	Words	Words	Words	Words
red	Words	Words	Words	Words		Words	Words	Words	Words
magenta	Words	Words	Words	Words	Words		Words	Words	Words
yellow	Words	Words	Words	Words	Words	Words		Words	Words
white	Words	Words	Words	Words	Words	Words	Words		Words
default		Words	Words	Words	Words	Words	Words	Words	

Colors (hue/saturation):





magenta:



red:

Weights:

normal, bold, default

Postures:

normal, *italic*, default

Text decorations:

normal, underlined, default

Colors (foreground) mixed with attributes:

black	normal bold normal <i>italic</i> normal <u>underlined</u> normal
	normal <i>bold+italic</i> normal <u>bold+underl</u> normal <i><u>italic+underl</u></i> normal
blue	normal bold normal <i>italic</i> normal <u>underlined</u> normal
	normal <i>bold+italic</i> normal <u>bold+underl</u> normal <i><u>italic+underl</u></i> normal
green	normal bold normal <i>italic</i> normal <u>underlined</u> normal
	normal <i>bold+italic</i> normal <u>bold+underl</u> normal <i><u>italic+underl</u></i> normal
cyan	normal bold normal <i>italic</i> normal <u>underlined</u> normal
	normal <i>bold+italic</i> normal <u>bold+underl</u> normal <i><u>italic+underl</u></i> normal
red	normal bold normal <i>italic</i> normal <u>underlined</u> normal
	normal <i>bold+italic</i> normal <u>bold+underl</u> normal <i><u>italic+underl</u></i> normal
magenta	normal bold normal <i>italic</i> normal <u>underlined</u> normal
	normal <i>bold+italic</i> normal <u>bold+underl</u> normal <i><u>italic+underl</u></i> normal
yellow	normal bold normal <i>italic</i> normal <u>underlined</u> normal
	normal <i>bold+italic</i> normal <u>bold+underl</u> normal <i><u>italic+underl</u></i> normal
white	normal bold normal <i>italic</i> normal <u>underlined</u> normal

```

|normal|bold+italic|normal|bold+underl|normal|italic+underl|normal|
default|normal|bold|normal|italic|normal|underlined|normal|
|normal|bold+italic|normal|bold+underl|normal|italic+underl|normal|

Colors (background) mixed with attributes:
black
blue
green
cyan
red
magenta
yellow
white
default

```

როგორც ჩანს, **xterm** ტიპის ტერმინალში სულ 8 ფერია 256-ის ნაცვლად.

ცხადია, ტერმინალის იმ ტიპების გარდა, რაც ზემოთ ვახსენეთ, ბევრი სხვაც არსებობს. სისტემაში მხარდაჭერილი ტერმინალის ტიპები შეგიძლიათ /etc/terminfo დირექტორიაში, /lib/terminfo-ში ან /usr/share/terminfo-ში ნახოთ. თუმცა, თუ მომხმარებელის საკუთარ დირექტორიაში არსებობს \$HOME/.terminfo ფაილი, მასში ჩაწერილი ტერმინალის ტიპები იქნება უპირველეს ყოვლისა მხედველობაში მიღებული.

17.2 ANSI კოდები

ANSI სიმბოლოების კომბინაციების ერთობლიობა (ANSI escape sequences) დე-ფაქტო სტანდარტს წარმოადგენს ტექსტური ტერმინალებისა და ტერმინალის ემულატორებზე in-band signaling რეჟიმში მართვის სიგნალების გადაცემისას. ამ დროს საკონტროლო სიმბოლოები იგივე არხში გადაიცემა, რომელშიც თავად მონაცემები. ამით განსხვავდება ეს რეჟიმი out-of-band signaling-სგან, რომელშიც მართვის სიგნალების გადაცემა, არათუ სხვა არხით, რიგ შემთხვევებში, სხვა ქსელის გამოყენებითაც ხდება. ANSI სიმბოლოების გამოყენებით ტერმინალზე შესაძლებელია განისაზღვროს კურსორის ადგილმდებარეობა, ფერი და მისი სხვა პარამეტრი. მართალია, ტექსტური ტერმინალების მოწყობილობები დღეს იშვიათ მოვლენას წარმოადგენს, ტექსტური ემულატორების უმრავლესობისთვის ANSI სტანდარტი (ძირითადად, ფერების თვალსაზრისით) კვლავ აქტუალური რჩება. იმ ტერმინალებს შორის, რომლებიც დაფუძნებულია ANSI ფერების სტანდარტზე, გამორჩეულია VT100 ტიპის ტერმინალი. DEC (Digital Equipment Corporation) კორპორაციამ ის 1978 წელს შეიქმნა და ერთ-ერთი პირველი იყო ვიდეო ტერმინალთაგან, რომლებშიც სრულად მხარდაჭერილი იყო ANSI კოდები (მათ გარდა,

საკუთარი კოდებიც ჰქონდა). VT სერიის ტერმინალები ძალიან პოპულარული გახდა, რასაც, თავის მხრივ, ANSI კოდების ფართო გამოყენება მოჰყვა. შედეგად, ANSI კოდები სტანდარტად ჩამოყალიბდა ტერმინალის ემულატორებში. ამის გამო, ANSI კოდებს ხშირად ANSI/VT100 ტერმინალის მართვის კოდებად მოიხსენიებენ.

ამ კოდების სრული დასახელებაა ANSI escape sequences და როგორც დასახელებიდან ჩანს, ისინი <Esc> სიმბოლოს შეიცავენ. ეს კოდები გენერირება **Esc** ლილაკის საშუალებით ხდება. შელში კი მისი გამოძახება `echo` ბრძანებით არის შესაძლებელი. ამისთვის `echo`-ს მისთვის გასაგები Backspace-escaped characters-სიმბოლოებიდან ერთ-ერთი, `\e` უნდა გადავცეთ. სწორედ ასე ხდება შელში Escape სიმბოლოს გაგება. Bash-ში მის მისაღებად `\e`-ს ნაცვლად შეგვიძლია გამოვიყენოთ `\033` ან `\x1B` ჩანაწერიც. სამივე **Esc** ლილაკის კოდს წარმოადგენს სხვადასხვა წარმოდგენაში - პირველი სიმბოლურში, მეორე რვაობითში, ხოლო მესამე თექვსმეტობითში.

მოდით, პრაქტიკულ ღონისძიებებზე გადავიდეთ და ვცადოთ ტექსტის ფერში გამოტანა. ფერის მისაღებად შემდეგი ფორმატის ჩანაწერია საჭირო:

```
achiko@debian:~$ echo -e "\e[ფერში ტექსტი\e[0m"
```

აქ „`\e[`“ კოდი ფერის შემოტანის დასაწყისს აღნიშნავს, „`ფერში`“ ჩანაწერში ფერი და ფორმატი მოიცემა შესაბამისი ნომრებით (`m` სიმბოლო დაფორმატირებული ჩანაწერის ნაწილია. არ გამოგრჩეთ მისი მითითება!), ხოლო „`\e[0m`“ კოდი ფორმატირების ყველა ატრიბუტის დასასრულს ნიშნავს. ცხადია, შესაძლებელია Escape-ის კოდის სხვა ფორმატით მოცემაც.

```
achiko@debian:~$ echo -e "\033[ფერში ტექსტი\033[0m"
achiko@debian:~$ echo -e "\x1B[ფერში ტექსტი\x1B[0m"
```

მოვიყვანოთ მაგალითი:

```
achiko@debian:~$ echo -e "\e[31mწითელი ტექსტი\e[0m"
წითელი ტექსტი
achiko@debian:~$ echo -e "\e[42mმწვანე ფონი\e[0m"
მწვანე ფონი
achiko@debian:~$ echo -e "\e[4mხაზგასმული\e[0m"
ხაზგასმული
```

შეგვიძლია ტექსტისა და ფონის ფერისა და ფორმატის ერთდროულად გამოყენებაც:

```
$ echo -e "\e[4;31;42mწითელი ხაზგასმული ტექსტი მწვანე ფონზე\e[0m"
წითელი ხაზგასმული ტექსტი მწვანე ფონზე
$ echo -e "\e[33;40;1m გამარჯობა \e[30;43;1m საქართველო \e[0m"
გამარჯობა საქართველო
```

სხვადასხვა ტერმინალის ემულატორები სხვადასხვა რაოდენობის ფერებს არჩევენ.









მათი უმრავლესობა მინიმუმ 8 ფერს მაინც აღიქვამს. ვნახოთ რა კოდი რა ფერს შეესაბამება, აგრეთვე, ფონისა და ფორმატის კოდები.

17.2.1 ძირითადი 8 ფერი

ტექსტის ფერი:

კოდი	ფერი	ნიმუში	შედეგი
39	ნაგულის- ხმევი ფერი	<code>echo -e "Default \e[39mDefault\e[0m"</code>	Default Default
30	შავი	<code>echo -e "Default \e[30mBlack\e[0m"</code>	Default Default
31	წითელი	<code>echo -e "Default \e[31mRed\e[0m"</code>	Default Red
32	მწვანე	<code>echo -e "Default \e[32mGreen\e[0m"</code>	Default Green
33	ყვითელი	<code>echo -e "Default \e[33mYellow\e[0m"</code>	Default Yellow
34	ლურჯი	<code>echo -e "Default \e[34mBlue\e[0m"</code>	Default Blue
35	მეწამული	<code>echo -e "Default \e[35mMagenta\e[0m"</code>	Default Magenta
36	ცისფერი	<code>echo -e "Default \e[36mCyan\e[0m"</code>	Default Cyan
37	ღია რუხი	<code>echo -e "Default \e[37mLight gray\e[0m"</code>	Default Light gray

ტექსტის ფონის ფერი:

კოდი	ფერი	ნიმუში	შედეგი
49	ნაგულის- ხმევი ფერი	<code>echo -e "Default \e[49mDefault\e[0m"</code>	Default Default
40	შავი	<code>echo -e "Default \e[40mBlack\e[0m"</code>	Default 
41	წითელი	<code>echo -e "Default \e[41mRed\e[0m"</code>	Default 
42	მწვანე	<code>echo -e "Default \e[42mGreen\e[0m"</code>	Default 
43	ყვითელი	<code>echo -e "Default \e[43mYellow\e[0m"</code>	Default 
44	ლურჯი	<code>echo -e "Default \e[44mBlue\e[0m"</code>	Default 
45	მეწამული	<code>echo -e "Default \e[45mMagenta\e[0m"</code>	Default 
46	ცისფერი	<code>echo -e "Default \e[46mCyan\e[0m"</code>	Default 
47	ღია რუხი	<code>echo -e "Default \e[47mLight gray\e[0m"</code>	Default 

თუ ტერმინალს 16 ფერის აღქმა შეუძლია, მაშინ დამატებითი კოდების ნომრები იქნება [90,97] ინტერვალში. ამ შემთხვევაში, ზემოთ ჩამოთვლილი ფერები უფრო ღია ტონალობაში გამოისახება. ფონის შესაცვლელად კი [100,107] ინტერვალში არსებული კოდები უნდა გამოვიყენოთ.

```

achiko@debian:~$ for i in `seq 90 97`
> do
> echo -e "\e[${i}m ტექსტი \e[0m"
> done; echo
ტექსტი ტექსტი ტექსტი ტექსტი ტექსტი ტექსტი ტექსტი
achiko@debian:~$ for i in `seq 100 107`
> do
> echo -e "\e[${i}m ტექსტი \e[0m"
> done; echo
ტექსტი ტექსტი ტექსტი ტექსტი ტექსტი ტექსტი ტექსტი ტექსტი

```

ტექსტის ფორმატი:

კოდი	აღწერა	ნიმუში	შედეგი
1	ადაპტირებული (Bold)	<code>echo -e "Normal \e[1mBold\e[0m"</code>	Normal Bold
2	ბაცი (Dim)	<code>echo -e "Normal \e[2mDim\e[0m"</code>	Normal Dim
3	კურსივი (Italic)	<code>echo -e "Normal \e[3mItalic\e[0m"</code>	Normal <i>Italic</i>
4	ხაზგასმული (Underline)	<code>echo -e "Normal \e[4mUnderline\e[0m"</code>	Normal <u>Underline</u>
5	ციმციმა (Blink)	<code>echo -e "Normal \e[5mBlink\e[0m"</code>	Normal Blinking
7	შებრუნებული (Inverted)	<code>echo -e "Normal \e[7mInverted\e[0m"</code>	Normal Inverted
8	დაფარული (Hidden)	<code>echo -e "Normal \e[8mHidden\e[0m"</code>	Normal
0	ფორმატირების მოხსნა	<code>echo -e "Normal \e[0mNormal\e[0m"</code>	Normal Normal

შებრუნებული ფორმატი ნიშნავს, რომ ტექსტისა და ფონის ფერები გადანაცვლდება. დაფარული ფორმატი გამოიყენება პაროლების შეტანის დროს.

მუქ (bold) ფორმატთან კომბინაციაში ტექსტის ფერი უფრო ღია ელფერს იღებს. დავწეროთ მარტივი სკრიფტი და ვნახოთ განსხვავება:

```

#!/bin/bash
for i in 0 1
do
  for j in {30..37}
  do
    echo -en "\e[${i}];${j}m ტექსტი \e[0m"
  done
done

```

```
achiko@debian:~$ ./script_XYZ.sh
```

```
ტექსტი ტექსტი ტექსტი ტექსტი ტექსტი ტექსტი ტექსტი ტექსტი  
ტექსტი ტექსტი ტექსტი ტექსტი ტექსტი ტექსტი ტექსტი ტექსტი
```

17.2.2 256 ფერი

როგორც უკვე ვნახეთ, ჩვენი ტერმინალი (`gnome-terminal`) 256 სზვადასხვა ფერს არჩევს. ვნახოთ, ANSI-ის რომელი კოდები უზრუნველყოფენ ამ ფერებით ტექსტის ფორმატირებას. თავად, ტექსტისთვის ფერის მისაცემად შემდეგი სპეციალური სინტაქსი უნდა გამოვიყენოთ:

```
achiko@debian:~$ echo -e "\e[38;5;Xm ტექსტი \e[0m"
```

ამ სინტაქსურ ჩანაწერში `X`-ის ადგილზე მხოლოდ ფერის კოდია მისათითებელი. ფერი კი `[0-256]` ინტერვალში ვარირებს.

მაგალითი:

```
achiko@debian:~$ echo -e "\e[38;5;200m ტექსტი \e[0m"  
ტექსტი
```

ყველა ფერზე წარმოდგენა რომ შეგვექმნას, სათითაოდ ვნახოთ 256-ვე ფერი და შესაბამისი კოდები:

```
achiko@debian:~$ for i in `seq 0 256`  
> do  
> echo -e "\e[38;5;${i}m ${i}\t\e[0m"  
> done; echo
```


0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109
110	111	112	113	114	115	116	117	118	119
120	121	122	123	124	125	126	127	128	129
130	131	132	133	134	135	136	137	138	139
140	141	142	143	144	145	146	147	148	149
150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169
170	171	172	173	174	175	176	177	178	179
180	181	182	183	184	185	186	187	188	189
190	191	192	193	194	195	196	197	198	199
200	201	202	203	204	205	206	207	208	209
210	211	212	213	214	215	216	217	218	219
220	221	222	223	224	225	226	227	228	229
230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249
250	251	252	253	254	255				

ტექსტის ფონის ფერის დასაყენებლად (256 ვარიანტიდან) შემდეგი ბრძანება უნდა გავუშვათ:

```
achiko@debian:~$ echo -e "\e[48;5;Xm ტექსტი \e[0m"
```

აქაც X [0-256] ინტერვალში მდებარეობს:

```
achiko@debian:~$ for i in `seq 0 256`
> do
> echo -e "\e[48;5;${i}m ${i} \t\e[0m"
> done; echo
```

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109
110	111	112	113	114	115	116	117	118	119
120	121	122	123	124	125	126	127	128	129
130	131	132	133	134	135	136	137	138	139
140	141	142	143	144	145	146	147	148	149
150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169
170	171	172	173	174	175	176	177	178	179
180	181	182	183	184	185	186	187	188	189
190	191	192	193	194	195	196	197	198	199
200	201	202	203	204	205	206	207	208	209
210	211	212	213	214	215	216	217	218	219
220	221	222	223	224	225	226	227	228	229
230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249
	251	252	253	254	255	256			

სკრიფტების გასაფორმებლად შეგვიძლია ფერის სხვადასხვა გრადაციების თანმიმდევრობა გამოვიტანოთ. ასე, მაგალითად:

```
achiko@debian:~$ for i in {16..21} {21..16}
> do
> echo -en "\e[48;5;${i}m \e[0m"
> done; echo
```



ამ მაგალითის გარდა, კიდევ ბევრი საინტერესო კომბინაციების მოფიქრება შეიძლება.

შელში კოდების ჩანაწერებს \ სიმბოლო ახლავს ხოლმე. როგორ მოვიქცეთ თუ კოდებთან ერთად ისეთი ტექსტის გამოტანა გვსურს ეკრანზე, რომელშიც თავად ეს სიმბოლო (ბექსლემ ასო-ნიშანი) იქნება მოცემული ერთხელ ან რამდენჯერმე. ვცადოთ:

```
achiko@debian:~$ echo -e "\ ასო-ნიშანი"
\ ასო-ნიშანი
achiko@debian:~$ echo -e "\\ ასო-ნიშანი"
\ \ ასო-ნიშანი
achiko@debian:~$ echo -e "\\ \ ასო-ნიშანი"
\ \ \ ასო-ნიშანი
achiko@debian:~$ echo -e "\\ \ \ ასო-ნიშანი"
\ \ \ \ ასო-ნიშანი
achiko@debian:~$ echo -e "ორი \\ \ \ \ \ ასო-ნიშანი"
ორი \ \ \ \ \ ასო-ნიშანი
```

ერთი შესხედვით დამაბნეველი, თუმცა ლოგიკური შედეგი მივიღეთ. მოდით, გავარკვეოთ, თუ რატომ გამოვიდა ასე. გავიხსენოთ, რომ `\` და `"` სიმბოლოები უკარგავენ სხვა ასო-ნიშნებს თავიანთ ფუნქციას (`\`-ის შემთხვევაში გამონაკლისია ახალ ხაზზე გადასვლა - `<newline>`, ხოლო ბრჭყალის შეთხვევაში `$`, ``` და `\` სიმბოლოები). ბრჭყალებში მოცემული `\` სიმბოლო ინარჩუნებს ფუნქციას, როდესაც მას მოსდევს ერთ-ერთი ამ სიმბოლოთაგანი: `$`, ```, `\` ან `<newline>`. აქედან გამომდინარეობს, რომ `\` სიმბოლოს ყოველთვის სპეციალური დანიშნულება არ აქვს. ამის გარდა, `echo` ბრძანება `e` ოფციით `\` სიმბოლოთი დაწყებულ ასო-ნიშნების გარკვეულ თანმიმდევრობას თავისებურად აღიქვამს. (`\n` ახალ ხაზზე გადასვლაა, `\\` კი თავად `\`). დავაკვირდეთ ზემოთმოყვანილ მაგალითებს და უკვე აღარ უნდა გაგიჭირდეს მიღებული შედეგის ახსნა:

1. `echo -e "\ ასო-ნიშანი"` - არაფერი განსაკუთრებული. რჩება ერთი `\`.
2. `echo -e "\\ ასო-ნიშანი"` - პირველი `\` ბაშს ეუბნება აღიქვას მეორე `\`, როგორც სიმბოლო `\`. `echo`-საც გამოაქვს ერთი `\`.
3. `echo -e "/// ასო-ნიშანი"` - პირველი `\` ბაშს ეუბნება აღიქვას მეორე `\`, როგორც სიმბოლო `\`. `echo` იღებს `\\`-ს და `-e` ოფციის გამო `\\` აღიქმება, როგორც ერთი `\`.
4. `echo -e "//// ასო-ნიშანი"` - პირველი `\` ბაშს ეუბნება აღიქვას მეორე `\`, როგორც სიმბოლო `\`. მესამე `\` იგივეს ეუბნება მეოთხე `\`-ზე. `echo` იღებს `\\`-ს და `-e` ოფციის გამო `\\` აღიქმება, როგორც ერთი `\`.
5. `echo -e "ორი ///// ასო-ნიშანი"` - პირველი `\` ბაშს ეუბნება აღიქვას მეორე `\`, როგორც სიმბოლო `\`. მესამე `\` იგივეს ეუბნება მეოთხე `\`-ზე. ბოლო `\`-ს სპეციალური დატვირთვა არ აქვს. `echo` იღებს `///`-ს და `-e` ოფციის გამო საწყისი `\\` აღიქმება, როგორც ერთი `\`. ბოლო `\` კი რჩება ხელუხლებელი.

ასე რომ, არ გაგიკვირდეთ თუ ზოგიერთ დოკუმენტაციაში, Escape sequences ჩანაწერებში ერთი `\`-ის ნაცვლად ორი `\\` იქნება გამოყენებული.

როგორც ამ თავის დასაწყისშიც აღვნიშნეთ და ისედაც ნათლად ჩანს, ჩვენი ტერმინალის მოსაწვევი ფერებითაა მოცემული. ვნახოთ, თუ როგორ გამოიყურება PS1 ცვლადის მნიშვნელობა:

```
achiko@debian:~$ echo $PS1
\[e]0;\u@h: \w\a\}${debian_chroot:+($debian_chroot)}\[O3[01;32m\]
\u@h\[O3[00m\]:\[O3[01;34m\]\w\[O3[00m\]\$
```

აშკარაა, რომ მასში ANSI escape sequences არის გამოყენებული. გამოსასვლელზე მიღებული შედეგი გრძელი ჩანაწერია და მის გასარჩევად უმჯობესია, ის ნაწილ-ნაწილ დავყოთ. მოდით, შემდეგი სამი ფრაგმენტი ავიღოთ:

1. `\[e]0;\u@h: \w\a\`
2. `${debian_chroot:+($debian_chroot)}`
3. `\[O3[01;32m\]\u@h\[O3[00m\]:\[O3[01;34m\]\w\[O3[00m\]\$`

პირველ ფრაგმენტში გამოყენებულია `<ESC>]` კოდი (შელში, `\e`“). მისი მოკლე დასახელებაა OSC (Operating System Command) და ის გარკვეული სტრინგის გადაცემას იწვებს ოპერაციული სისტემისთვის. მისი დასრულება ხდება `<ESC> \` კოდით (შელში `\e\`“). მოკლე დასახელებაა ST (String Terminator). `xterm`-ში OSC-ს დასრულება BELL კოდითაც

ზდება (შელში „\a“). ფანჯარაში გაშვებულ `xterm`-ში, ამ ფანჯრის დასახელების განსაზღვრა შეგვიძლია შემდეგი კოდების გაშვებით: `OSC 0; სათაურის ტექსტი BEL` (შელში მისი ექვივალენტია „\e]0;ახალი სათაური\a“).

სანამ, იმ ფანჯრის სათაურს შევცვლით, სადაც ჩვენი ტერმინალი გახსნილი, ჯერ პირველად ვარიანტს შევხედოთ, რათა დავრწმუნდეთ ჩვენი ბრძანების მართებულობაში.

```
achiko@debian:~  
  
achiko@debian:~$
```

```
ახალი სათაური  
  
achiko@debian:~$ PS1=\[\e]0;ახალი სათაური\a\${debian_chroot:+($debian_chroot)}\[\033[01;32m\]\u@\h\[\033[00m\]:\[\033[01;34m\]\w\[\033[00m\]\$  
achiko@debian:~$
```

ფანჯრის სათაურის ველში უკვე ჩვენი შეტანილი დასახელება დაფიქსირდა.

ახლა, `PS1`-ის მეორე ფრაგმენტი გავარჩიოთ. ის მარტივი გასაგებია რადგან, წიგნის პირველ ნაწილში (ცვლადის დამუშავების სექციაში) მგსავსი ჩანაწერი (`${Variable:+word}`) ახსნილი გვაქვს. თუ `Variable` არ არსებობს ან `NULL`-ია, მაშინ სტანდარტულ გამოსავლელზე არაფერი გამოვა. წინააღმდეგ შემთხვევაში, მივიღებთ `word`-ს. ახლა ჩვენი შემთხვევა განვიხილოთ - `${debian_chroot:+($debian_chroot)}`. ჯერ ვნახოთ, არსებობს თუ არა ცვლადი `debian_chroot`.

```
ახალი სათაური  
  
achiko@debian:~$ echo $debian_chroot
```

როგორც ჩანს, `debian_chroot` ცვლადი არ არის განსაზღვრული. შესაბამისად, ეს ფრაგმენტი არაფრით შეიცვლება. ამ ცვლადს რაიმე მნიშვნელობა ჰქონდა მინიჭებული, `${debian_chroot:+($debian_chroot)}` ჩანაწერი `debian_chroot`-ის მნიშვნელობით შეიცვლებოდა. ამ ცვლადის არსის უკვეთ გასაგებად, ჯერ ვთქვათ, რა არის `chroot` ოპერაცია. `Unix`-ის მსგავს სისტემებში `chroot` ბრძანება გამოიყენება პროცესის ძირეული დირექტორიის (`root` დირექტორიის) შესაცვლელად. ასეთ შეცვლილ გარემოში გაშვებული პროცესი/პროგრამა ვეღარ შეძლებს ახალი `root` დირექტორიის გარეთ არსებულ ფაილებთან წვდომას. თუ `chroot` ოპერაცია განხორციელდა, `debian_chroot` ცვლადს ახალი ძირეული დირექტორიის სახელი მიენიჭება. შესაბამისად, `PS1`-ის ამ ფრაგმენტის არსებობა უზრუნველყოფს იმას, რომ ადვილად მივხვდეთ შეზღუდულ გარემოში ვიმყოფებით თუ არა ფაილების წვდომის თვალსაზრისით.

მესამე ფრაგმენტი ჩვენთვის უკვე კარგად ნაცნობი ფერების კოდებს შეიცავს.

ახალა ზემოთაღნიშნული ფერების საშუალებით შეგვიძლია სურვილისამებრ განვსაზღვროთ შელის მოსაწვევი, რათა უფრო მეტი კომფორტი შევიტანოთ შელში მუშაობის პროცესში. მოვიყვანოთ მარტივი მაგალითები:

```
achiko@debian:~$ PS1="თქვენ უშვებთ ბრძანებას ნომრით \e[1;31m \!|\e[0m \ $ "
თქვენ უშვებთ ბრძანებას ნომრით 2010 $ echo $RANDOM > /dev/null
თქვენ უშვებთ ბრძანებას ნომრით 2011 $ ls > /dev/null
თქვენ უშვებთ ბრძანებას ნომრით 2012 $
```

გასათვალისწინებელია ის, რომ bash-ში PS1 ცვლადში escape sequences გამოყენებისას ეს კოდები \[\] სტრუქტურაში უნდა მოვათავსოთ. წინააღმდეგ შემთხვევაში ის შელი მოსაწვევის სიგრძეს სწორად ვერ აღიქვამს.

ვცადოთ, გავუშვათ ისეთი ბრძანება, რომლის სიგრძეც ტერმინალის სიგრძეს გადააჭარბებს:

[illegible]

როგორც გამოჩნდა, გაშვების დილაკის დაკერამდე ბრძანების აკრეფისას, ახალი ხაზის ნაცვლად, იგივე ხაზზე გაგრძელდა ბრძანების ტექსტის შეყვანა. სწორედ ამიტომ უნდა მოვაქციოთ ეს უხილავი კოდები [\] სტრუქტურაში. ვცადოთ ახლებურად:

თქვენი უშვებო ბრძანებას ნომრით 2014 \$ PS1="... \[e[1;31m\] \!\[e[0m\] \\$ "
 თქვენი უშვებო ბრძანებას ნომრით 2015 \$ lssssssssssssssssssssssssssssssssssss
 sssssssssssssss

ახლა ყველაფერი წესრიგშია. თუ გრძელი ბრძანება ხაზზე არ დაეტივა, მისი აკრეფა ახალ ხაზზე გაგრძელდება.

კიდევ ერთი დეტალი აღვნიშნოთ. ზემოთ ვახსენეთ, რომ ფერის კოდის შემოტანის შემდეგ, მისი დასრულების კოდიც „\e[0m“ უნდა მივუთითოთ ბოლოს. ეს ფორმატირების ყველა ატრიბუტის დასასრულს ნიშნავს. ამ კოდის გარეშე ფორმატი შენარჩუნდება და შემდეგ, ეკრანზე შეყვანილი თუ გამოტანილი მონაცემიც ამ ფორმატს მიიღებს. PS1-ის მაგალითზე კარგად გამოჩნდება ეს განსხვავება:

```
თქვენ უშვებთ ბრძანებას ნომრით 2016 $ PS1="\[\e[31m\] PS1 \[\e[0m\]"
PS1 whoami
achiko
PS1 PS1="\[\e[31m\] PS1 "          # დასრულების კოდის გარეშე
PS1 whoami
achiko
```

ფორმატის დასრულების კოდის გარეშე აკრეფილი ბრძანებისა და მისი შედეგისთვისაც შელმა „ჩარჩენილი“ ფორმატი შეინარჩუნა და ისინი ეკრანზე წითელ ფერში გამოასახა.

შელის მოსაწვევები გაკეთებული ცვლილება მხოლოდ მიმდინარე სესიაზე ვრცელდება. ახალი ტერმინალი რომ გავხსნათ, კვლავ ძველი მოსაწვევი დაგვხვდება. ცვლილების მუდმივად შესანახად PS1 ცვლადი ახალი მნიშვნელობით \$HOME/.bashrc ფაილში უნდა გავწეროთ.

```
$ echo 'PS1="\[\e[31m\] New PS1 \[\e[0m\]"' >> ~/.bashrc
```

ამის შემდეგ, ყოველ ახლად გახსნილ ტერმინალში შელის ახალი მოსაწვევი დაგვხვდება. `~/.bashrc` ფაილს დეტალურად მოგვიანებით დავუბრუნდებით.

17.3 კურსორის მართვა

ANSI კოდები, ფერების გარდა, შეიცავენ ისეთ კოდებს, რომლებიც ეხება ტერმინალის/დისპლეის პარამეტრებს, კურსორის მართვას, ტექსტის შრიფტს და სხვა. მოდით, ვნახოთ ტერმინალის დისპლეიზე კურსორის მართვის სიგნალები. სადემონსტრაციოდ, გადავიყვანოთ კურსორი არსებული პოზიციიდან 5 სტრიქონით ქვევით. ამისთვის ტერმინალს `<ESC>[NB` სიგნალი უნდა გადავცეთ, სადაც `N` გადასახტომი სტრიქონების რაოდენობაა.

```
achiko@debian:~$  
achiko@debian:~$  
achiko@debian:~$  
achiko@debian:~$  
achiko@debian:~$  
achiko@debian:~$  
achiko@debian:~$ echo -e "\e[5B"
```

```
achiko@debian:~$ █
```

`<ESC>[H` სიგნალით კი კურსორი საწყის პოზიციაზე (HOME-ში) დაბრუნდება. ვცადოთ:

```
achiko@debian:~$  
achiko@debian:~$ █  
achiko@debian:~$  
achiko@debian:~$  
achiko@debian:~$  
achiko@debian:~$  
achiko@debian:~$ echo -e "\e[5B"
```

```
achiko@debian:~$ echo -e "\e[H"
```

ნიმუშად მოყვანილი ტერმინალის კოდების გარდა, ბევრი სხვა კოდი არსებობს. უფრო მეტიც, არსებობს არაერთი ტიპის ტერმინალები. თუმცა, ამ უამრავი ტერმინალის კონტროლის ენების ცოდნა და მათი კოდების დამახსოვრება არ გვჭირდება, რადგან სისტემაში არსებობს შუალედური კომუნიკაციის ფენა

იმის ნაცვლად, რომ ამ უამრავი ტერმინალის კონტროლის სხვადასხვა ენა ვიცოდეთ და მათი კოდები დავიმახსოვროთ, როგორც წესი, სისტემას აქვს შუალედური კომუნიკაციის ფენა. რეალური კოდები იძებნება მიმდინარე ტერმინალის ტიპის მონაცემთა ბაზაში (terminfo). ჩვენ კი შეგვიძლია უკვე სტანდარტიზებული მოთხოვნები გავუგზავნოთ შელის ბრძანებით. ერთ-ერთი ასეთი ბრძანებაა `tput`. მას ესმის სიტყვათშეთანხმებები, რომლებსაც *capability names* (*cap-names*) ეწოდებათ. მათი მითითების შემთხვევაში `tput` ბრძანება გამოითხოვს შესაბამის *Escape sequences*-ებს მიმდინარე ტერმინალის ტიპის (terminfo) მონაცემთა ბაზიდან. ამიტომ, ANSI კოდების პირდაპირი გამოყენების ნაცვლად, უმჯობესია, `tput` ბრძანების გამოყენება.

ვნახოთ `tput` ბრძანებისთვის პარამეტრად გადასაცემი terminfo-ს ზოგიერთი ცნობილი შესაძლებლობა (*cupnames*) და მათი აღწერა:

კურსორის მართვა	
cup X Y	კურსორის გადაადგილება დისპლეის X Y კოორდინატზე. X წარმოადგენს დისპლეის სტრიქონის ნომერს, ხოლო Y სვეტის ნომერს. (0,0) მარცხენა ზედა კუთხეს შეესაბამება.
home	კურსორის გადაადგილება (0,0) კოორდინატზე.
civis	კურსორის გადართვა უხილავ რეჟიმში.
cvvis	კურსორის გადართვა ხილულ რეჟიმში.
cub1	კურსორის გადაადგილება ერთი პოზიციით მარცხნივ (backspace).
sc	კურსორის მიმდინარე პოზიციის შენახვა.
rc	კურსორის შენახული პოზიციის აღდგენა.
clear	ეკრანის გასუფთავება და კურსორის თავში დაყენება.
lines	სტრიქონების რაოდენობა ტერმინალის ეკრანზე. ტერმინალის ემულატორის ფანჯრის გადიდება/დაპატარავების შემთხვევაში, სტრიქონების რაოდენობა, ცხადია, შეიცვლება.
cols	სვეტების რაოდენობა ტერმინალის ეკრანზე.

ტექსტის სხვადასხვა ატრიბუტი	
bold	"bold"(მუქი) ატრიბუტის დაყენება.
dim	"dim"(ბაცი) ატრიბუტის დაყენება.

smul	"underline"(ზაზგასმული) ატრიბუტის დაყენება. მისი გამორთვა ხდება rmul ატრიბუტით.
rev	"reverse"(შებრუნებული) ატრიბუტის დაყენება. ამ დროს ტექსტის ფერი და მისი ფონის ფერი გადანაცვლდება.
invis	"hidden"(დაფარული) ატრიბუტის დაყენება.
sgr0	ყველა ატრიბუტის საწყის მნიშვნელობაზე დაყენება.

ტექსტის ფერები

setaf	0	ტექსტის მოცემა #0 ფერში (შავი - black). მისი ექვივალენტი ANSI კოდი არის [30m.
setaf	1	ტექსტის მოცემა #1 ფერში (წითელი - red). მისი ექვივალენტი ANSI კოდი არის [31m.
setaf	2	ტექსტის მოცემა #2 ფერში (მწვანე - green). მისი ექვივალენტი ANSI კოდი არის [32m.
setaf	3	ტექსტის მოცემა #3 ფერში (ყვითელი - yellow). მისი ექვივალენტი ANSI კოდი არის [33m.
setaf	4	ტექსტის მოცემა #4 ფერში (ლურჯი - blue). მისი ექვივალენტი ANSI კოდი არის [34m.
setaf	5	ტექსტის მოცემა #5 ფერში (მეწამული - magenta). მისი ექვივალენტი ANSI კოდი არის [35m.
setaf	6	ტექსტის მოცემა #6 ფერში (ცისფერი - cyan). მისი ექვივალენტი ANSI კოდი არის [36m.
setaf	7	ტექსტის მოცემა #7 ფერში (თეთრი - white). მისი ექვივალენტი ANSI კოდი არის [37m.
setaf	9	ტექსტის მოცემა ნაგულისხმევ ფერში. მისი ექვივალენტი ANSI კოდი არის [39m.

ტექსტის ფონის ფერები

setab	0	ტექსტის ფონის მოცემა #0 ფერში (შავი - black). მისი ექვივალენტი ANSI კოდი არის [30m.
setab	1	ტექსტის ფონის მოცემა #1 ფერში (წითელი - red). მისი ექვივალენტი ANSI კოდი არის [31m.
setab	2	ტექსტის ფონის მოცემა #2 ფერში (მწვანე - green). მისი ექვივალენტი ANSI კოდი არის [32m.
setab	3	ტექსტის ფონის მოცემა #3 ფერში (ყვითელი - yellow). მისი ექვივალენტი ANSI კოდი არის [33m.

setab 4	ტექსტის ფონის მოცემა #4 ფერში (ლურჯი - blue). მისი ექვივალენტი ANSI კოდი არის [34m.
setab 5	ტექსტის ფონის მოცემა #5 ფერში (მეწამული - megenta). მისი ექვივალენტი ANSI კოდი არის [35m.
setab 6	ტექსტის ფონის მოცემა #6 ფერში (ცისფერი - cyan). მისი ექვივალენტი ANSI კოდი არის [36m.
setab 7	ტექსტის ფონის მოცემა #7 ფერში (თეთრი - white). მისი ექვივალენტი ANSI კოდი არის [37m.
setab 9	ტექსტის ფონის მოცემა ნაგულისხმევ ფერში. მისი ექვივალენტი ANSI კოდი არის [39m.

ტექსტისა და ფონის ფერისთვის ANSI კოდების მხარდამჭერ ტერმინალებში სწორედ ზემოთ ჩამოთვლილი cupname-ები - **setaf** და **setab** უნდა გამოვიყენოთ. მათი ფუნქცია დასახელებიდანაც კარგად ჩანს - **setaf** (set ANSI foreground), **setab** (set ANSI background). სხვა ტერმინალების შემთხვევაში **setf** და **setb** დასახელების cupname-ების გამოყენებაა საჭირო. მათში ლურჯისა და წითელის წერის კოდები გადანაცვლებულია. შეგიძლიათ შეამოწმოთ **xterm** ტიპის ტერმინალზე. მას ორივე სახის cupname ესმის.

მოვიყვანოთ რამდენიმე მაგალითი. ვნახოთ, რამდენი სტრიქონი და სვეტი ეტევა ჩვენს ტერმინალში:

```
achiko@debian:~$ tput lines
30
achiko@debian:~$ tput cols
86
```

ახლა, კურსორი გადავაადგილოთ ტერმინალის ცენტრში და დავწეროთ სასურველი ტექსტი. ცენტრში გადასაადგილებლად ჯერ მისი კოორდინატები უნდა განვსაზღვროთ. X იქნება **tput cols** შედეგის 2-ზე განაყოფი, ხოლო Y კი **tput lines** შედეგის 2-ზე განაყოფი. ვცადოთ:

```
achiko@debian:~$ tput lines
30
achiko@debian:~$ tput cols
86
achiko@debian:~$ tput cup 15 43; echo Mishka

Mishka

achiko@debian:~$
```

ჰმმმ... ბოლომდე ის არ გამოვიდა, რასაც ველოდით. გავაანალიზოთ და გავაუმჯობესოთ შედეგი. უპირველეს ყოვლისა, ეკრანი უნდა გავასუფთავოთ, რათა აქ აკრეფილი ბრძანებები არ ჩანდეს. ამას გარდა, კურსორი ცენტრში კი გადავაადგილოთ, მაგრამ თუ ტერმინალის ფანჯრის ზომას დავაპატარავებთ ან გავზრდით ახალი ეკრანის ცენტრი აღარ დაემთხვევა ჩვენს კოორდინატებს. ამიტომ, უმჯობესია, ცენტრის კოორდინატები ფარდობითი მნიშვნელობით ავიღოთ და ეკრანის მიმდინარე ზომებს მივაბათ. ამის მიუხედავად, კურსორის ცენტრში გადაყვანის შემთხვევაშიც კი, ჩვენი სასურველი სიტყვის გამოტანა, ზუსტად ცენტრში არ განთავსდა. მართალია, Mishka-ს დაწერა ცენტრიდან დაიწყო, მაგრამ Mishka-ს სიგრძე მხედველობაში არ მიგვიღია. გარდა ამისა, Mishka-ს დაბეჭდვის შემდეგ შელის მოსაწვევი ჯობს ბოლო ხაზზე ჩავიტანოთ. მოდით, გავითვალისწინოთ ეს შენიშვნები, გავუშვათ შემდეგი ბრძანება და ვნახოთ რა გამოვა:

```
$ X=$(tput lines); Y=$(tput cols)
$ A=MISHKA
$ tput clear; tput cup $((X/2)) $((Y/2-${#A}/2)); echo $A; tput cup $X 0
```

MISHKA

achiko@debian:~\$

ახლა ყველაფერი რიგზეა. შეგვიძლია დასაბეჭდი მონაცემის სიგრძე გავზარდოთ და ფანჯრის ზომაც შევცვალოთ. ამ ბრძანებით ტექსტი მაინც ცენტრში დაიბეჭდება.

```
$ A="Mishka, NINO, IO ..."
$ tput clear; tput cup $((X/2)) $((Y/2-${#A}/2)); echo $A; tput cup $X 0
```

MISHKA, NINO, IO ...

achiko@debian:~\$

შეგვიძლია, აგრეთვე, ტექსტის ფორმატი და ფერები შემოვიყენოთ.

```
$ A="Mishka, NINO, IO ..."  
$ tput clear; tput cup $((X/2)) $((Y/2-#{A}/2)); tput bold; tput setab  
6; echo $A; tput cup $X 0; tput sgr0
```

MISHKA, NINO, IO ...

achiko@debian:~\$

```
achiko@debian:~$ echo "$(tput setaf 1)ANSI$(tput sgr0) კოდების ნაცვლად,  
უმჯობესია, მომხმარებლისთვის მარტივად გამოყენებადი - $(tput setaf  
2)tput$(tput sgr0) ბრძანება გამოიყენოთ!"  
ANSI კოდების ნაცვლად, უმჯობესია, მომხმარებლისთვის მარტივად გამოყენებადი  
- tput ბრძანება გამოიყენოთ!
```

tput ბრძანებით შესაძლებელია პატარა ხმოვანი სიგნალის გამოტანაც. ასე:

```
achiko@debian:~$ tput bel
```

შელის მოსაწყვეის განსაზღვრისას tput ბრძანებას შეგვიძლია გამოვიყენოთ ფერების მოსაცემად. ფერის კოდი [0,256] ინტერვალიდან შეგვიძლია ავიღოთ, თუ, რა თქმა უნდა, ტერმინალს ამდენი ფერის მხარდაჭერა აქვს.

```
achiko@debian:~$ PS1="\[$(tput setaf 200)\]\u@\h:\w $ \[$(tput sgr0)\]"
achiko@debian:~$
```

PS1 ცვლადში შეგიძლიათ, აგრეთვე, შელის ნებისმიერი ბრძანება (სკრიფტი, ფუნქცია ან სხვა) ჩასვათ. სასურველია, ის ბევრ მონაცემს არ შეიცავდეს გამოსასვლელზე და, ამავედროულად, დარწმუნებული იყოთ, რომ საქმე სწრაფად შესრულებად ბრძანებას ეხება, რადგან ის მოსაწვევის ყოველი გამოჩენისას გაეშვება. ასე მაგალითად, ისეთი მოსაწვევი შევქმნათ, რომელიც მიმდინარე პროცესების რაოდენობას გამოიტანს:

```
$ PS1="ამჟამად გაშვებულია \[$(tput setab 1)\]\$(ps aux | wc -l)\[$(tput sgr0)\] პროცესი "
```

გაშვებულია 159 პროცესი

გაშვებულია 157 პროცესი

ალბათ, ყურადღებას მიაქცევდით, რომ ამ ჩანაწერში ბრძანების საკუთარი შედეგით შეცვლისას \$ ნიშნის წინ „ზედმეტი“ \ სიმბოლო მიუწერეთ.

დაიმახსოვრეთ!

PS1 ცვლადის რედაქტირებისას ბრძანების საკუთარი შედეგით შეცვლის კონსტრუქციაში აუცილებელია \$ ნიშნის წინ ეწეროს \ (ბეჭდვით), ასე: \\$(...). წინააღმდეგ შემთხვევაში ეს ბრძანება, მხოლოდ ერთხელ შესრულდება და PS1-ის შიგთავსი აღარ იქნება დინამიკურად ცვალებადი.

მიმდინარე ტერმინალის ტიპის შესაძლებლობების სანახავად `infocmp` ბრძანება შეგიძლია გამოვიყენოთ. ის დეტალურად გვანახებს ამ ტერმინალის `capname`-ების ჩამონათვალს.

```
achiko@debian:~$ echo $TERM
xterm-256color
achiko@debian:~$ infocmp
# Reconstructed via infocmp from file: /lib/terminfo/x/xterm-256color
xterm-256color|xterm with 256 colors,
am, bce, ccc, km, mc5i, mir, msgr, npc, xenl,
colors#256, cols#80, it#8, lines#24, pairs#32767,
acsc=``aaffggiijjkllmmnnnooppqqrrssttuuvvwxxyyzz{|}|}~~,
bel=~G, blink=\E[5m, bold=\E[1m, cbt=\E[Z, civis=\E[?25l,
clear=\E[H\E[2J, cnorm=\E[?12l\E[?25h, cr=~M,
csr=\E[%p1%d;%p2%dr, cub=\E[%p1%dD, cub1=~H,
cud=\E[%p1%dB, cud1=~J, cuf=\E[%p1%dB, cuf1=\E[C,
cup=\E[%p1%d;%p2%h, cuu=\E[%p1%DA, cuu1=\E[A,
cvvis=\E[?12;25h, dch=\E[%p1%DP, dch1=\E[P, dim=\E[2m,
dl=\E[%p1%DM, dl1=\E[M, ech=\E[%p1%DX, ed=\E[J, el=\E[K,
el1=\E[1K, flash=\E[?5h$<100/>\E[?5l, home=\E[H,
hpa=\E[%p1%DG, ht=~I, hts=\EH, ich=\E[%p1%DO,
il=\E[%p1%DL, il1=\E[L, ind=~J, indn=\E[%p1%DS,
```

`tput` ბრძანებას, `terminfo`-ს მდიდარი მონაცემთა ბაზის წყალობით, ძალიან ბევრი შესაძლებლობა (`cupname`) აქვს. მათი სრული სიის სანახავად, მიმართეთ `terminfo`-ს სახელმძღვანელო გვერდს - `man terminfo`.

17.4 ტერმინალის მულტიპლექსერები

ტერმინალის მულტიპლექსერი არ არის ფიზიკური მოწყობილობა, არამედ წარმოადგენს პროგრამას, რომელიც ტერმინალში ეშვება. ამ პროგრამით შესაძლებელია ტერმინალის ერთი დისპლეი ან ტერმინალის ემულატორის ერთი ფანჯარა რამდენიმე, ერთმანეთისგან განცალკევებულ ფსევდო-ტერმინალის ფანჯრებად დავყოთ. ეს ბრძანება კომფორტულია მომხმარებლისათვის, როდესაც მას სხვადასხვა ბრძანების სხვადასხვა გარემოში ერთდროულად გაშვება სურს, ხოლო წვდომა მხოლოდ ერთ ტერმინალთან აქვს (მაგალითად ტერმინალზე დისტანციურად დაკავშირებისას).

ტერმინალის მულტიპლექსერით აგრეთვე შესაძლებელია ტერმინალიდან სამუშაო სესიის მოცილება და შემდეგ მიერთება. ასეთი შესაძლებლობით მომხმარებელს შეუძლია დაუკავშირდეს კომპიუტერის ტერმინალს ლოკალურად ან დისტანციურად, სესიაში გაუმვას სასურველი პროგრამა და შემდეგ გამოვიდეს ამ კომპიუტერიდან. მიუხედევად გამოსვლისა, გაშვებული პროგრამა გააგრძელებს მუშაობას და მომხმარებელი, როდესაც ხელახლა დაუკავშირდება ამ კომპიუტერს, შეძლებს დაუბრუნდეს თავის გაშვებულ წინა სესიას, საკუთარ სამუშაო გარემოს. ანუ, ტერმინალის მულტიპლექსერი უზრუნველყოფს მუდმივი სესიების (ე.წ. *Persistent session*) არსებობას.

17.4.1 Gnu screen

ერთ-ერთი ყველაზე ცნობილი ტერმინალის მულტიპლექსერია **Gnu screen**. ნაგულისხმევი მნიშვნელობით, ის არ არის დაყენებული Debian სისტემებში. გამოდის, პროგრამა ხელით უნდა დააინსტალიროთ (`apt-get install screen`). პროგრამების ინსტალაციას დეტალურად მოგვიანებით დაგუბრუნდებით.

მას შემდეგ რაც მას დავაინსტალირებთ, გავსხნათ ტერმინალი და გავუშვათ **screen** ბრძანება.

```
achiko@debian:~$ screen
```

```
GNU Screen version 4.05.00 (GNU) 10-Dec-16
```

```
Copyright (c) 2010 Juergen Weigert, Sadrul Habib Chowdhury
```

```
Copyright (c) 2008, 2009 Juergen Weigert, Michael Schroeder, Micah Cowan,  
Sadrul Habib Chowdhury
```

```
Copyright (c) 1993-2002, 2003, 2005, 2006, 2007 Juergen Weigert, Michael  
Schroeder
```

```
Copyright (c) 1987 Oliver Laumann
```

```
This program is free software; you can redistribute it and/or modify it  
under the terms of the GNU General Public License as published by the  
Free Software Foundation; either version 3, or (at your option) any later  
version.
```

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along

[Press Space for next page; Return to end.]

დააჭირეთ შეყვანის ღილაკს და ჩვეულებრივი გარემო დაგზავდება. გარეგნულად ძნელია მიხვდეთ, რომ **screen** გაშვებულია, თუმცა გაშვებული პროცესების სიაში რომ ჩაიხედოთ, ნათლად დაინახავთ მის არსებობას:

```
achiko@debian:~$ ps -ejH
...
 1086   1086   1086 ?          00:00:01      gnome-terminal-
 1092   1092   1092 pts/0    00:00:00        bash
 1098   1098   1092 pts/0    00:00:00          screen
 1099   1099   1099 ?          00:00:00          screen
 1100   1100   1100 pts/1    00:00:00            bash
 1155   1155   1100 pts/1    00:00:00            ps
...
```

შეგვიძლო პირდაპირ **TERM** ცვლადის მნიშვნელობა გადაგვემოწმებინა. თუ **screen** გაშვებულია, ის ასეთნაირად შეიცვლება:

```
achiko@debian:~$ echo $TERM
screen.xterm-256color
```

მულტიპლექსერებთან მიმართებაში გამართული ტერმინოლოგიით რომ ვისაუბროთ, **screen** ბრძანების შესრულებით გაეშვა **screen**-ის ერთი სესია, სადაც შეიქმნა ახალი, პირველი ფანჯარა. სწორედ ამ ფანჯარაში გაშვებულ შელში შეგვიძლია გავუშვათ სასურველი ბრძანებები. ასევე, შეგვიძლია ეს ფანჯარა დროებით დავტოვოთ და, მოგვიანებით, კვლავ დავუბრუნდეთ მას. მაგალითისთვის გავუშვათ რომელიმე ისეთი ბრძანება, რომელიც დიდხანს გრძელდება:

```
achiko@debian:~$ while ;; do echo "ფანჯარა 1 - $RANDOM"; sleep 1; done
ფანჯარა 1 - 19790
ფანჯარა 1 - 28242
ფანჯარა 1 - 28638
ფანჯარა 1 - 5494
ფანჯარა 1 - 7251
ფანჯარა 1 - 13115
ფანჯარა 1 - 8859
```

```
ფანჯარა 1 - 14996
ფანჯარა 1 - 6684
ფანჯარა 1 - 21119
ფანჯარა 1 - 27570
ფანჯარა 1 - 11992
ფანჯარა 1 - 1832
ფანჯარა 1 - 7085
...
```

სესიიდან გამოსასვლელად **screen**-ს უნდა გადავცეთ **Ctrl^a** + **d** კლავიშების კომბინაცია. შედეგად, ეკრანზე შემდეგი ინფორმაცია გამოვა:

```
achiko@debian:~$ screen
[detached from 1099.pts-0.mypc]
achiko@debian:~$
```

ეს იმის მიმანიშნებელია, რომ მიმდინარე სესია დავტოვეთ, შესაბამისად, მასში შექმნილი ფანჯრებიდანაც გამოვედით. ჩანაწერი **1099.pts-0.mypc** სესიის სახელია (დასახელებაში, ჩვეულებისამებრ, შედის **screen**-ის PID, ტერმინალის ინფორმაცია, საიდანაც ის გაეშვა და კომპიუტერის სახელი).

სესიაში ხელახლა დასაბრუნებლად შემდეგი ბრძანება უნდა გავუშვათ:

```
achiko@debian:~$ screen -r 1099.pts-0.mypc
...
ფანჯარა 1 - 10238
ფანჯარა 1 - 14134
ფანჯარა 1 - 18583
ფანჯარა 1 - 23894
ფანჯარა 1 - 28460
ფანჯარა 1 - 13740
ფანჯარა 1 - 12737
ფანჯარა 1 - 6717
ფანჯარა 1 - 23422
ფანჯარა 1 - 18642
ფანჯარა 1 - 4294
ფანჯარა 1 - 24691
ფანჯარა 1 - 45179
ფანჯარა 1 - 36598
...
```

ნათელია, რომ სესიის ფანჯარაში ჩვენ მიერ გაშვებული ბრძანება არ შეწყვეტილა და გრძელდება. სესიაზე დაბრუნება მოკლედ ასეც შეგვეძლო:

```
achiko@debian:~$ screen -r 1099
```

თუ **screen**-ით ერთადერთი სესიაა გახსნილი, მაშინ არგუმენტის მიწერაც აღარ არის

აუცილებელი:

```
achiko@debian:~$ screen -r
```

სახელის არგუმენტად მითითება მხოლოდ მაშინაა აუცილებელი, თუ რამდენიმე სესია გვაქვს გახსნილი. გახსნილი სესიების ჩამონათვალი `-list` ან `-ls` ოფციის გამოყენებით შეგვიძლია ვნახოთ:

```
achiko@debian:~$ screen -list      # ან screen -ls
There are screens on:
  6171.pts-0.mypc (08/21/2019 01:32:03 PM)    (Detached)
  1099.pts-0.mypc (08/21/2019 12:19:10 PM)    (Detached)
  2 Sockets in /run/screen/S-achiko.
```

ასეთნაირად შეგვიძლია იმ სესიას დაგუბრუნდეთ, რომელსაც გვსურს. მრავალი სესიის შემთხვევაში, ერთმანეთის უკეთ გასარჩევად, უმჯობესია, გაშვების დროს მათ სასურველი დასახელება მივანიჭოთ.

სესიისთვის სახელის მინიჭება მისი გაშვებისას ასე ხდება:

```
achiko@debian:~$ screen -S სახელი
```

გამოსვლის შემდეგ მასთან დასაბრუნებლად ამ სახელის მითითებითაც კმარა:

```
achiko@debian:~$ screen -r სახელი
```

`screen` მულტიპლექსერის სესიაში ახალი ფანჯრის გაშვება `Ctrl^a` + `c` კლავიშების კომბინაციით არის შესაძლებელი. ეს ფანჯარა ვიზუალურად გადაფარავს წინა აქტიურ ფანჯარას. შეგვიძლია, ბევრი ფანჯარა შევქმნათ ერთი მეორის მიყოლებით და მათში სხვადასხვა ამოცანა გავუშვათ. შემდეგ/წინა ფანჯარაზე გადასასვლელად `Ctrl^a` + `n` / `Ctrl^a` + `p` კომბინაციები უნდა გამოვიყენოთ. გაშვებული ფანჯრების სია კი `Ctrl^a` + `w` -თი შეგვიძლია ვნახოთ. ყველა ეს ფანჯარა `screen`-ის ერთ სესიაშია გაშვებული. `screen`-ის ახალი სესია მაშინ გაეშვება, როდესაც სხვა ტერმინალში `screen` ბრძანებას ხელახლა გავუშვებთ. მიმდინარე ფანჯრის დახურვა `Ctrl^a` + `k` კლავიშების კომბინაციაზე დაჭერის შედეგად ხდება.

როგორც დასაწყისშივე ვთქვით, შესაძლებელია ტერმინალის მულტიპლექსერის ეკრანის მრავალ ნაწილად დაყოფა და თითოეულ მათგანში ფანჯრის გაშვება. მაგალითისთვის, დავეოთ ჩვენი ფანჯარა ორ ვერტიკალურ ნაწილად. ამისთვის ჯერ `screen` გავუშვათ (თუ ტერმინალში ის გაშვებული არ არის), შემდეგ კი `Ctrl^a` + `|` კლავიშების კომბინაციას დავაჭიროთ. დავინახავთ, რომ ფანჯარა ორად გაიყო ვერტიკალურად:

```
ფანჯარა 1 - 13614
ფანჯარა 1 - 13614
ფანჯარა 1 - 11110
ფანჯარა 1 - 5500
```



```

ფანჯარა 1 - 20498
ფანჯარა 1 - 20357
ფანჯარა 1 - 11270
ფანჯარა 1 - 1018
ფანჯარა 1 - 29255
ფანჯარა 1 - 11032
ფანჯარა 1 - 18808
ფანჯარა 1 - 29026
ფანჯარა 1 - 587
ფანჯარა 1 - 15063
ფანჯარა 1 - 1567
ფანჯარა 1 - 3759
^C

```

```
achiko@debian:~$
```

```
0 bash
```

```
--
```

თითოეულ დანაყოფში შესაძლებელია სხვადასხვა, დამოუკიდებელი ბრძანების გაშვება. ერთი დანაყოფიდან მეორეზე გადასვლა `Ctrl^a` + `tab` კლავიშების კომბინაციით ხდება. გადავიდეთ და გავუშვათ რაიმე ბრძანება. ჩვენი აკრეფილი ეკრანზე არ გამოჩნდა. შელის მოსაწვევიც არსად ჩანს. ბუნებრივია, ამ ახალ დანაყოფში ბრძანების გაშვება არ შეგვიძლია. მასში ჯერ ახალი ფანჯრის შექმნაა საჭირო, სადაც `bash` იქნება გაშვებული. მისი უზრუნველყოფა `Ctrl^a` + `C` კლავიშების კომბინაციაზე დაჭერით არის შესაძლებელი. ამის შემდეგ უკვე შეგვიძლია ბრძანებების გაშვება. ვცადოთ:

```

ფანჯარა 1 - 13614
ფანჯარა 1 - 13614
ფანჯარა 1 - 11110
ფანჯარა 1 - 5500
ფანჯარა 1 - 20498
ფანჯარა 1 - 20357
ფანჯარა 1 - 11270
ფანჯარა 1 - 1018
ფანჯარა 1 - 29255
ფანჯარა 1 - 11032
ფანჯარა 1 - 18808
ფანჯარა 1 - 29026
ფანჯარა 1 - 587
ფანჯარა 1 - 15063
ფანჯარა 1 - 1567
ფანჯარა 1 - 3759
^C

```

```
achiko@debian:~$
```

```
0 bash
```

```
achiko@debian:~$ echo "რეგიონი 2"
```

```
რეგიონი 2
```

```
achiko@debian:~$
```

```
1 bash
```

მიმდინარე რეგიონის (დანაყოფის) დახურვა `Ctrl^a` + `X` -ით ხდება. დამატებითი ინსტრუქციების სანახავად შეგიძლიათ `Ctrl^a` + `?` კლავიშების კომბინაციით დახმარებას მიმართოთ.

Screen key bindings, page 1 of 2.

Command key: ^A Literal ^A: a

break	^B b	license	,	removebuf	=
clear	C	lockscreen	^X x	reset	Z
colon	:	log	H	screen	^C c
copy	^[[login	L	select	'
detach	^D d	meta	a	silence	_
digraph	^V	monitor	M	split	S
displays	*	next	^@ ^N sp n	suspend	^Z z
dumptermcap	.	number	N	time	^T t
fit	F	only	Q	title	A
flow	^F f	other	^A	vbell	^G
focus	^I	pow_break	B	version	v
hardcopy	h	pow_detach	D	width	W
help	?	prev	^H ^P p ^?	windows	^W w
history	{ }	quit	\	wrap	^R r
info	i	readbuf	<	writebuf	>
kill	K k	redisplay	^L l	xoff	^S s
lastmsg	^M m	remove	X	xon	^Q q

[Press Space for next page; Return to end.]

უფრო დეტალური ინფორმაციის მისაღებად მიმართეთ ბრძანების სახელმძღვანელო გვერდს - **man screen** ან/და ეწვიეთ შემდეგ ვებ-გვერდს: <https://www.gnu.org/software/screen/manual/screen.html>

17.4.2 Tmux

tmux მულტიპლექსერი GNU **screen**-თან შედარებით ახლი პროგრამაა. მას მეტი შესაძლებლობა აქვს, თუმცა ეს უპირატესობა მნიშვნელოვან ფუნქციებს არ ეხება. ორივე პროგრამა კარგად ართმევს თავს მათზე დაკისრებულ ამოცანებს. შესაბამისად, ჯერჯერობით, ერთის მეორისგან უპირატესად გამორჩევა სუბიექტურია გამოყენების თვალსაზრისით. მომხმარებელს ის პროგრამა ურჩევნია, რომელსაც იცნობს და ფლობს.

მოდით, მოკლედ მიმოვიხილოთ **tmux** მულტიპლექსერი. ნაგულისხმევი მნიშვნელობით, არც ისაა დაყენებული Debian სისტემებში. **tmux**-ის გამოსაყენებლად, GNU **screen**-ის მსგავსად, მისი ხელით დაყენება მოგიწევთ (**apt-get install tmux**).

გავუმვათ **tmux** ბრძანება და ვნახოთ, თუ როგორ გამოიყურება ეს მულტიპლექსერი.

```
achiko@debian:~$
```

```
[0] 0: bash*
```

```
"mypc" 16:19 21-Aug-19
```

ფანჯრის ქვედა, მწვანე ფერის ნაწილში განთავსებული სტატუსის ხაზი მიგვანიშნებს, რომ **tmux** გაშვებულია. მოდით, ჩამოვთვალოთ **tmux** მულტიპლექსერის გასაშვები ბრძანებების ძირითადი ოფციები ბრძანებათა ხაზისთვის. შედარებისთვის, ასევე **screen** პროგრამაში:

ოპერაცია	tmux	screen
სესიის შექმნა და მასზე მიერთება	<code>tmux</code>	<code>screen</code>
სესიის შექმნა foo სახელით და მასზე მიერთება	<code>tmux new -s foo</code>	<code>screen -S foo</code>
სესიის შექმნა მიერთების გარეშე	<code>tmux new -d</code>	<code>screen -dm</code>
სესიების ჩამონათვალი	<code>tmux ls</code>	<code>screen -list</code>
სესიაზე მიერთება	<code>tmux attach</code>	<code>screen -r</code>
foo სესიაზე მიერთება	<code>tmux attach -t foo</code>	<code>screen -r foo</code>
foo სესიის დახურვა	<code>tmux kill-session -t foo</code>	<code>screen -r foo -X quit</code>

მულტიპლექსერის ფანჯარაში კლავიშების კომბინაციები:

ოპერაცია	tmux	screen
დახმარების გამოძახება	<code>Ctrl^b</code> + <code>[?]</code>	<code>Ctrl^a</code> + <code>[?]</code>
სესიის დატოვება	<code>Ctrl^b</code> + <code>[d]</code>	<code>Ctrl^a</code> + <code>[d]</code>
ახალი ფანჯრის შექმნა	<code>Ctrl^b</code> + <code>[c]</code>	<code>Ctrl^a</code> + <code>[c]</code>
შემდეგ ფანჯარაზე გადასვლა	<code>Ctrl^b</code> + <code>[n]</code>	<code>Ctrl^a</code> + <code>[n]</code>
წინა ფანჯარაზე გადასვლა	<code>Ctrl^b</code> + <code>[p]</code>	<code>Ctrl^a</code> + <code>[p]</code>
გახსნილი ფანჯრების ნუსხა	<code>Ctrl^b</code> + <code>[w]</code>	<code>Ctrl^a</code> + <code>[w]</code>
მიმდინარე ფანჯრის დახურვა	<code>Ctrl^b</code> + <code>[&]</code>	<code>Ctrl^a</code> + <code>[k]</code>
ფანჯრის ჰორიზონტალურად დაყოფა	<code>Ctrl^b</code> + <code>["]</code>	<code>Ctrl^a</code> + <code>[S]</code>

ფანჯრის ვერტიკალურად დაყოფა	Ctrl^b + %	Ctrl^a +
ფანჯრის დანაყოფებზე გადასვლა. screen-ში დანაყოფს რეგიონი ჰქვია, tmux-ში პანელი	Ctrl^b + ← , ↓ , ↑ , →	Ctrl^a + tab
მიმდინარე დანაყოფების დახურვა	Ctrl^b + x	Ctrl^a + X

მოდით, **tmux**-ის ფანჯარა დავყოთ პანელებად (ჰორიზონტალურად და ვერტიკალურად) და თითოეულ მათგანში შელის სხვადასხვა სახალისო ბრძანება გავუშვათ (**figlet**, **toilet**, **cmatrix**, **fortune**, **cowsay**, **sl** ან სხვა). ყოველი მათგანი ხელით არის დასაყენებელი. მსგავსს სურათს მივიღებთ:

```

achiko@debian:~$ cmatrix

p i          v 3 \   k \   s L   V   a d   s   t   ( q 7   9 J & n H
2 & e        0 A ) n C |   j n   k   w [ K   X   H f o   r l { N I
v ) 5        |   & L   i b   $ L y W   \ g |   D   g   8 u R   b s 2 A ;
$ ( i        A   (   ] [   R 1 > [   E t u   G           @ E (   [ < T @ P
H d V       . g   0   V r   { J f @   e 0 }   p           v \ x S   ; X { `
B q X   A ;     "   c 5   z ` ? Q   K h P   H           H " G ]   v ` C $ @
7 ( Q   8 S     R   _ E       N 1 q   M   g   :           ( 0 # w   # _ T b E
R   Y   g       N # [ _   H u >   g   l   <           , \ | l   \ n = E n
T   0   j       y ^ t 4       M P z   y   , ^ p       ! p = H   j > | , f
Y   Z   ]       < ? i 3       j Y S           Z 6 n       # - l T v 5 6 d 4 ;
> "   : e       I | l 1           v b M           k f       L V m 8 % ; S j { x
/   8   W Q F   5 l = ,           4 I W           ) {       " &   T - i i - G r
Z   n       : Y   w & < h       7 P v m           0 ,       3 w   j . E Y 3 P i
f   0       p *   l   < G       _ j 0 6           W /       P u   g e ] n U t [
Y   5       K {   ,   _ F       g - f ]           o i       ( _   R | X b K ? i
{ 0   q T q       ^ t 0   B % ( r k   q   !           ] j   ;   } h   _ F b
P r   ' @       ! ? G   P 0   p M   S   B           q J   p   l (   s / N
g \   ( A       r Z s   r Q   m /   /   3           } "   `   '   H A [
`           y v &   h * . G f   B Z d H           N -   H   F   i Z
U           0 ^ )   U 0 G P H 8 e \ . J           E /   5 i $   x
H   U           y w H   I   " O g m @ s   J           J &   _ x U   D

achiko@debian:~$ figlet Mishka

  _ _ _ _ _
 |  \  ( )  _ _ _ | _ _ | | _ _ _ _
 |  \|  |  | /  _ _ | ' _ \ | /  _ _
 |  |  |  | ( _ _ | | | |   < ( _ |
 | _ |  | _ | \ _ _ _ | | | | \ _ _ _

achiko@debian:~$ fortune | cowsay

-----
/ The ripest fruit falls first. \
|                                  |
\ -William Shakespeare,"Richard II" /
-----

      \   ^__^
       \  (oo)\_______
          (__)\       )\/\
              ||----w |
              ||     ||

[0] 0: bash*                                     "mypc" 17:25 21-Aug-19

```

მულტიპლექსერების ერთ-ერთ მთავარ დანიშნულებას, მუდმივი სესიებისა და მრავალფანჯრული სისტემის უზრუნველყოფის გარდა, წარმოადგენს სესიების გაზიარება მრავალ მომხმარებელს შორის. ეს შესაძლებლობა მომხმარებლებს თანამშრომლობით მუშაობაში ეხმარება.

ვნახოთ, როგორ ხდება სესიის გაზიარება `tmux`-ის მაგალითზე. ამისთვის გახსნილი უნდა გვქონდეს მინიმუმ ორი სხვადასხვა ტერმინალი, საიდანაც გაზიარებულ სესიას მივუერთდებით. სიმარტივისთვის ვიგულისხმობთ, რომ ორივე ტერმინალში ერთი მომხმარებელია შესული. შევქმნათ პირველ ტერმინალში `tmux`-ის ახალი სესია `public` დასახელებით და მივუერთდეთ მას შემდეგი ბრძანებით:

```
achiko@debian:~$ tmux new -s public
```

შექმნილ სესიას მეორე ტერმინალიდან ასე შეგვიძლია შევუერთდეთ:

```
achiko@debian:~$ tmux attach -t public
```

სულ ეს არის. ახლა, რაც ერთ ტერმინალში მოხდება მეორეშიც გამოჩნდება და პირიქით. სესია დუბლირებული იქნება ორ ტერმინალში და მასში ინტერაქცია ორივე ტერმინალიდან იქნება შესაძლებელი.

ტერმინალი 1

```
$ tmux new -s public

$
$ echo "ტ1-დან შეყვანილი ტექსტი"
ტ1-დან შეყვანილი ტექსტი
$
$ echo "ტ2-დან შეყვანილი ტექსტი"
ტ2-დან შეყვანილი ტექსტი
$
```

```
[public] <h* "mypc"18:11 21-Aug-19
```

ტერმინალი 2

```
$ tmux attach -t public

$
$ echo "ტ1-დან შეყვანილი ტექსტი"
ტ1-დან შეყვანილი ტექსტი
$
$ echo "ტ2-დან შეყვანილი ტექსტი"
ტ2-დან შეყვანილი ტექსტი
$
```

```
[public] <h* "mypc"18:11 21-Aug-19
```

ახლა განვიხილოთ შემთხვევა, როდესაც სესიის გაზიარება ორ სხვადასხვა მომხმარებელს შორის ხდება. ამისთვის `tmux`-ის პროცესისთვის უნდა არსებობდეს ისეთი სოკეტი⁵, რომელშიც ორივე მომხმარებელს ექნება კითხვისა და ჩაწერის უფლება. ამავდროულად, აუცილებელია, ეს მომხმარებლები ერთსა და იმავე ჯგუფში იყვნენ გაწევრიანებული.

მაგალითისთვის შემდეგი სცენარი წარმოვიდგინოთ: გვქვს ორი ტერმინალი. ერთში `achiko` მომხმარებელი არის შესული, ხოლო მეორეში `mishka` მომხმარებელი. `achiko`-ც და `mishka`-ც joint პირველადი ჯგუფის წევრები არიან.

შევქმნათ პირველ ტერმინალში `tmux`-ის ახალი სესია `public` დასახელებით, რომელიც იყენებს `/tmp/S` სოკეტს:

⁵Unix-ის სოკეტი არის პროცესებს შორის კომუნიკაციის მექანიზმი, რომლის წყალობითაც ხდება მათ შორის მონაცემების გაცვლა. შელში Unix-ის სოკეტი `s` ტიპის ფაილით არის წარმოდგენილი. Unix-ის სოკეტის გარდა არსებობს IP სოკეტიც, რომელიც ქსელურ პროცესებს შორის კომუნიკაციისთვის არის საჭირო.

```
achiko@debian:~$ tmux -S /tmp/S new -s public
```

დავრწმუნდეთ, რომ ამ სოკეტის ფაილის მფლობელი ჯგუფი არის joint.

```
achiko@debian:~$ ls -l /tmp/S
srwxrwx--- 1 achiko joint 0 Sep  6 17:08 /tmp/S
```

წინააღმდეგ შემთხვევაში აუცილებელი იქნება ქვემოთ მოცემული ბრძანებით ამ სოკეტის ფაილის მფლობელ ჯგუფად გამოვაცხადოთ joint, რათა ამ ჯგუფის წევრ მომხმარებლებს მასში ჩაწერისა და მისი წაკითხვის უფლება ჰქონდეთ.

```
achiko@debian:~# chgrp joint /tmp/S
```

ამ ოპერაციის შესასრულებლად სისტემის ადმინისტრატორის უფლებები დაგჭირდებათ. მომხმარებლებისა და ჯგუფების ადმინისტრირებას დეტალურად მოგვიანებით დავუბრუნდებით.

ახლა mishka მომხმარებელს უკვე შეუძლია მეორე ტერმინალიდან დაუკავშირდეს achiko-ს გახსნილ სესიას მოცემული სოკეტის გამოყენებით შემდეგნაირად:

```
achiko@debian:~$ tmux -S /tmp/S attach -t public
```

ყურადღება მიაქციეთ იმას, რომ თუ joint ჯგუფში სხვა მომხმარებლებიც არიან გაწევრიანებულნი, მათაც ექნებათ გაზიარებულ სესიასთან წვდომა.

ტერმინალი 1

```
$ whoami
achiko
$ groups
joint
$ tmux -S /tmp/S new -s public
$
# chgrp joint /tmp/S
$
$ echo "ტ1-დან შეყვანილი ტექსტი"
ტ1-დან შეყვანილი ტექსტი
$
$ echo "ტ2-დან შეყვანილი ტექსტი"
ტ2-დან შეყვანილი ტექსტი
$
```

```
[public] <h* "mypc"18:11 21-Aug-19
```

ტერმინალი 2

```
$ whoami
mishka
$ groups
joint
$ tmux -S /tmp/S attach -t public
$
# chgrp joint /tmp/S
$
$ echo "ტ1-დან შეყვანილი ტექსტი"
ტ1-დან შეყვანილი ტექსტი
$
$ echo "ტ2-დან შეყვანილი ტექსტი"
ტ2-დან შეყვანილი ტექსტი
$
```

```
[public] <h* "mypc"18:11 21-Aug-19
```

ამას გარდა, mishka მომხმარებელს შეუძლია achiko-ს გაზიარებულ სესიას მხოლოდ დათვალიერების რეჟიმში შეუერთდეს, ისე, რომ ცვლილების გაკეთება არ შეეძლოს. ამისთვის

მან `-r` ოფცია უნდა გამოიყენოს მიერთებისას, შემდეგნაირად (თუმცა, ეს მხოლოდ `mishka` მომხმარებლის გადასაწყვეტია):

```
achiko@debian:~$ tmux -S /tmp/S attach -t public -r
```

ლინუქსის იმ დისტრიბუტივებში, სადაც მომხმარებლები მათი შექმნისას, ნაგულისხმევი მნიშვნელობით, ერთ ჯგუფში წევრიანდებიან, `tmux`-ით სესიის გასაზიარებლად აღარ დაგჭირდებათ ადმინისტრატორის უფლებები. `Gnu screen`-თაც შეიძლება სესიის გაზიარება, თუმცა მის გამოსაყენებლად საჭიროა `root`-ის უფლებების გამოყენება.

`tmux` მულტიპლექსერის შესახებ დეტალური ინფორმაციის მისაღებად მიმართეთ მის სახელმძღვანელო გვერდს - `man tmux`.

თავი 18

მასივები

მასივი წარმოადგენს ცვლადს, რომელიც მრავალ მნიშვნელობას შეიძლება შეიცავდეს. შესაძლებელია ნებისმიერი ცვლადი გამოვიყენოთ როგორც მასივი. მისი სიგრძე არ არის შემოსაზღვრული, რაც იმას ნიშნავს, რომ მასივში ნებისმიერი რაოდენობის ელემენტების ჩაწერა შეგვიძლია. მათი იდენტიფიკაციისთვის ელემენტების ინდექსის ნომრები გამოიყენება. ინდექსი დადებით მთელ რიცხვს წარმოადგენს და მისი ათვლა 0-დან იწყება. თუმცა, აუცილებელი არაა მასივის ელემენტების ინდექსების თანმიმდევრობით გამოყენება.

მასივის შექმნა, მასივის გამოცხადება (ამ ტერმინს ხშირად იყენებენ მასივებთან მიმართებაში) `declare` ბრძანებით არის შესაძლებელი, შემდეგნაირად:

```
achiko@debian:~$ declare -a ARRAYNAME
```

მასივის გამოცხადების შემდეგ მისი ელემენტების ერთიანად განსაზღვრა შემდეგნაირად შეგვიძლია:

```
achiko@debian:~$ ARRAYNAME=(value1 value2 ... valueN)
```

სადაც პირველი ელემენტის ინდექსის ნომერი ავტომატურად არის 0, მეორე ელემენტის 1 და ა.შ. ამ ყველაფრის გაკეთება ერთი ბრძანებითაც არის შესაძლებელი:

```
achiko@debian:~$ declare -a ARRAYNAME=(value1 value2 ... valueN)
```

მასივის ელემენტის განსაზღვრა არაპირდაპირი, ირიბი გზითაც არის შესაძლებელი:

```
achiko@debian:~$ ARRAYNAME[indexnumber]=value
```

ამ დროს, ინდექსის ნომერი შეიძლება შეარჩიოთ. მაგალითად, შეგვიძლიათ, მასივის მე-5 ელემენტი განსაზღვროთ ან მხოლოდ მე-2 და მე-10 ელემენტები დატოვოთ მასივში.

მასივის შექმნა ასეთი ფორმითაც შეიძლება:

```
achiko@debian:~$ ARRAYNAME=([1]=value1 [2]=value2 ... valueN)
```

მოდით, მოვიყვანოთ მასივის ელემენტების დამუშავების მაგალითები. ნიმუშისთვის შემდეგი მასივი შევქმნათ:

```
~$ declare -a Linux=('Debian' 'Ubuntu' 'Red hat' 'Arch Linux' 'Mint')
```

ა) მასივის ყველა ელემენტის გამოტანა:

```
achiko@debian:~$ echo ${Linux[*]}  
Debian Ubuntu Red hat Arch Linux Mint  
achiko@debian:~$ echo ${Linux[@]}  
Debian Ubuntu Red hat Arch Linux Mint
```

ბ) მასივის ელემენტების რაოდენობა:

```
achiko@debian:~$ echo ${#Linux[*]}  
5
```

გ) მასივის მე-4 ელემენტის გამოტანა:

```
achiko@debian:~$ echo ${Linux[3]}  
Arch Linux
```

დ) მასივის მე-2 ელემენტიდან დაწყებული ყველა ელემენტის გამოტანა:

```
achiko@debian:~$ echo ${Linux[*]:1}  
Ubuntu Red hat Arch Linux Mint
```

ე) მასივის მე-2 ელემენტიდან დაწყებული ორი ელემენტის გამოტანა:

```
achiko@debian:~$ echo ${Linux[*]:1:3}  
Ubuntu Red hat
```

ვ) მასივის მე-3 ელემენტის მე-5 ასო-ნიშნიდან დაწყებული ყველა ასო-ნიშნის გამოტანა:

```
achiko@debian:~$ echo ${Linux[2]:4}  
hat
```

ზ) მასივის მე-4 ელემენტის პირველი ასო-ნიშნიდან დაწყებული 4 ასო-ნიშნის გამოტანა:

```
achiko@debian:~$ echo ${Linux[3]:0:4}  
Arch
```

თ) მასივის მე-4 ელემენტზე სხვა მნიშვნელობის მინიჭება:

```
achiko@debian:~$ Linux[3]="Suse"  
Arch  
achiko@debian:~$ echo ${Linux[*]}  
Debian Ubuntu Red hat Suse Mint
```

ი) მასივის პირველი ელემენტის სიგრძე:

```
achiko@debian:~$ echo ${#Linux[0]}  
6
```

კ) მასივის მეორე ელემენტის გამოტანისას (მხოლოდ გამოტანისას! ეს ელემენტი უცვლელი რჩება) პირველი შემხვედრი სიტყვა „ა“-ს ჩანაცვლება „TEST“ სიტყვით:

```
achiko@debian:~$ echo ${Linux[1]/u/TEST}  
UbTESTntu
```

ლ) მასივის მეორე ელემენტის გამოტანისას ყველა შემხვედრი სიტყვა „ა“-ს ჩანაცვლება „TEST“ სიტყვით:

```
achiko@debian:~$ echo ${Linux[1]//u/TEST}  
UbTESTntTEST
```

მ) მასივის მეორე ელემენტის გამოტანისას პირველი შემხვედრი სიტყვა „ა“-ს ჩანაცვლება „TEST“ სიტყვით თუ ეს ელემენტი „ა“-თი იწყება:

```
achiko@debian:~$ echo ${Linux[1]/#u/TEST}  
Ubuntu
```

ნ) მასივის მეორე ელემენტის გამოტანისას პირველი შემხვედრი სიტყვა „ა“-ს ჩანაცვლება „TEST“ სიტყვით თუ ეს ელემენტი „ა“-თი მთავრდება:

```
achiko@debian:~$ echo ${Linux[1]/%u/TEST}  
UbuntTEST
```

ო) მასივის პირველი ელემენტის გამოტანისას მისი პირველი დიდი ასო-ნიშნის (თუ ასეთი შემთხვევაა) პატარით შეცვლა:

```
achiko@debian:~$ echo ${Linux[0],}  
debian
```

პ) მასივის პირველი ელემენტის გამოტანისას მისი ყველა ასო-ნიშნის დიდით შეცვლა:

```
achiko@debian:~$ echo ${Linux[0]^}
DEBIAN
```

- ჟ) მასივის პირველი ელემენტის გამოტანისას მისი პირველი ასო-ნიშნის გადანაცვლება ანუ დიდი ასო-ნიშნის პატარით შეცვლა, ან პატარა ასო-ნიშნის დიდი შეცვლა:

```
achiko@debian:~$ echo ${Linux[0]~}
DEBIAN
```

- რ) მასივის პირველი ელემენტის გამოტანისას მისი ყველა ასო-ნიშნის გადანაცვლება ანუ დიდი ასო-ნიშნების პატარით შეცვლა, ან პატარა ასო-ნიშნების დიდი შეცვლა:

```
achiko@debian:~$ echo ${Linux[0]~~}
DEBIAN
```

- ს) მასივის ელემენტების ინდექსის ნომრების გამოტანა:

```
achiko@debian:~$ echo ${!Linux[*]}
0 1 2 3 4
```

- ტ) მასივში ელემენტების დამატება:

```
achiko@debian:~$ Linux=("${Linux[*]}" 'CentOS' 'Slackware')
achiko@debian:~$ echo ${Linux[*]}
Debian Ubuntu Red hat Suse Mint CentOS Slackware
```

- უ) მასივის მე-3 ელემენტის წაშლა:

```
achiko@debian:~$ unset Linux[2]
achiko@debian:~$ echo ${Linux[*]}
Debian Ubuntu Suse Mint CentOS Slackware
```

- ფ) თავად მასივის წაშლა:

```
achiko@debian:~$ unset Linux
achiko@debian:~$ echo ${Linux[*]}
```

- ქ) ორი არსებული მასივის გაერთიანება, კონკატენაცია ახალ მასივში :

```
achiko@debian:~$ Linux=('Debian' 'Ubuntu' 'Red hat' 'Mint')
achiko@debian:~$ declare -a Unixlike=('AIX' 'HP-UX' 'OpenBSD')
```

```
achiko@debian:~$ All=("${Linux[*]}" "${Unixlike[*]}")
achiko@debian:~$ echo ${All[*]}
Debian Ubuntu Red hat Mint AIX HP-UX OpenBSD
```

18.1 ასოციაციური მასივი

ინდექსირებული მასივის გარდა, bash-ში ასოციაციური მასივაც არსებობს. თუ ინდექსირებულ მასივში ელემენტები ინდექსით, რიცხვითი მნიშვნელობით იდენტიფიცირდება, ასოციაციურ მასივში რიცხვითი მნიშვნელობის ნაცვლად, ნებისმიერი სტრინგი შეგვიძლია გამოვიყენოთ. ასოციაციური მასივის გამოცხადება ასე ხდება:

```
achiko@debian:~$ declare -A ARRAYNAME
```

```
achiko@debian:~$ ARRAYNAME=([key1]=value1 [key2]=value2 ... [keyN]=valueN)
```

სადაც პირველი ელემენტის მახასიათებელი სიტყვა (ე.წ. გასაღები) არის key1, მეორე ელემენტის key2 და ა.შ. ერთიანად შეიძლება ეს მასივი ასე შევქმნათ:

```
$ declare -A ARRAYNAME=([key1]=value1 [key2]=value2 ... [keyN]=valueN)
```

ღ) ასოციაციური მასივის შექმნა:

```
$ declare -A Numbers=([one]="ერთი" [two]="ორი" [tree]="სამი" )
achiko@debian:~$ echo ${!Numbers[@]}
two tree one
achiko@debian:~$ echo ${Numbers[@]}
ორი სამი ერთი
```

bash-ში ასოციაციური მასივები ისეთივეა, როგორც ჰეშები და ლექსიკონები დაპროგრამების სხვა ენებში. ისინი დალაგებული სახით არ იბეჭდება გამოსასვლელზე, რაც წინა მაგალითშიც ნათლად გამოჩნდა. ამ შემთხვევაში ჩვენი შესაძლებლობების მაქსიმუმში sort ბრძანების გამოყენება იქნება. ასე, ანბანის მიხედვით მოვახდენთ დახარისხებას.

ყ) ასოციაციური მასივის დახარისხება:

```
achiko@debian:~$ for key in "${!Numbers[@]}"; do
> echo $key' - '${Numbers["$key"]}
> done
two - ორი
tree - სამი
```

```

one - ერთი
achiko@debian:~$ for key in "${!Numbers[@]}"; do echo $key -
'${Numbers["$key"]}; done | sort
one - ერთი
tree - სამი
two - ორი

```

ამგვარად, თვითონ გასაღებები დავახარისხეთ ანბანის მიხედვით. შეგვიძლია შედეგი მათი მნიშვნელობების დახარისხებით ასეთი ჩანაწერით გამოვიტანო:

```

achiko@debian:~$ for key in "${!Numbers[@]}"; do echo $key -
'${Numbers["$key"]}; done | sort -k3
one - ერთი
two - ორი
tree - სამი

```

კოდის წერისას შეიძლება გქონდეთ ისეთი შემთხვევა, როდესაც არ გსურთ, შეიცვალოს მასივის ელემენტების მნიშვნელობა. ასეთ დროს უმჯობესია, მასივის გამოცხადება `declare`-ის ნაცვლად `readonly` ბრძანებით შევასრულოთ. როგორც მისი მნიშვნელობიდან ჩანს, ის უზრუნველყოფს მასივის ელემენტების მუდმივობას, მათი გამოყენების შესაძლებლობას მხოლოდ კითხვით და არა ჩაწერის რეჟიმში. `readonly`-ით შეგვიძლია როგორც ინდექსირებადი, ასევე ასოციაციური მასივი გამოვაცხადოთ.

შ) მასივი მუდმივი ელემენტებით:

```

achiko@debian:~$ readonly -a Linux=( 'Debian' 'Ubuntu' 'Red hat' )
achiko@debian:~$ Linux[2]='CentOS'
-bash: Linux: readonly variable

```

მასივების შესახებ მეტი მაგალითების სანახავად შეგიძლიათ გადახვიდეთ შემდეგ ბმულზე: <https://www.tldp.org/LDP/abs/html/arrays.html>

როგორც ზემოთ მოყვანილ მაგალითებში ჩანს `strng`-ების დასამუშავებლად გამოყენებული სინტაქსი მასივის ელემენტებისთვისაც სამართლიანია (იხილეთ 10.1).

სათაური

- | |, 53, 227
- :, 248
- ;, 53
- <, 51
- >, 48
- >>, 49
- ?, 221
- &&, 53, 227
-], 223

- ABC, 7
- ANSI კოდები, 288, 300
- ANSI/VT100, 301
- at, 214
- atq, 214
- atrm, 214

- bc, 116
- bg, 94
- break, 241
- bzip2, 200

- C, 9
- cal, 164
- cat, 29, 50
- cd, 25
- chmod, 80
- cmp, 137
- Colossus, 7
- comm, 137
- console, 23
- continue, 243
- Copyleft, 14
- Copyright, 14
- cp, 44
- cpio, 206
- cron, 209
- crontab, 209
- cupname, 311
- cut, 126

- date, 164
- debugger, 242
- declare, 329
- diff, 138
- disown, 94
- Display, 48
- du, 142

- echo, 100, 149
- ENIAC, 8
- env, 100
- environnement, 99
- exit, 248
- expand, 157
- export, 100
- expr, 108

- factor, 267
- false, 222
- fg, 93
- FHS, 31
- file, 29
- find, 139
- fmt, 160
- free, 91
- freeware, 14

- getopts, 282
- GID, 74
- GNU, 17
- GNU/Linux, 20
- GPL, 14
- grep, 167
- gzip, 198

- head, 120
- HOWTO, 21

- id, 74
- infocmp, 316

- jobs, 93
- join, 130

- kernel, 12
- kill, 95
- killall, 96

- less, 30
- let, 263
- Linux, 20
- ls, 25, 28

- mkdir, 40
- Monitor, 48
- msgcat, 289
- MULTICS, 8
- mv, 45

- nice, 97
- nl, 160
- nohup, 94

- od, 161

- passwd, 75
- paste, 128
- patch, 138
- PID, 85
- pipe, 55
- pkill, 96
- PPID, 86
- printenv, 100
- printf, 152
- ps, 86
- PS1, 307, 315
- pwd, 25

- read, 233
- readonly, 334
- regex, 167
- regexp, 167

- renice, 97
- return, 277
- Screen, 48
- sed, 178
- seq, 265
- set, 101
- SGID, 82
- shareware, 14
- shell, 12, 23
- shift, 272
- shuf, 267
- sort, 122
- Sparse file, 160
- split, 148
- stdin, 47
- stdout, 47
- sterr, 47
- Sticky bit, 82
- SUID, 82
- tac, 119
- tail, 120
- tar, 202
- tee, 57
- Terminal emulators
 - Eterm, 288
 - GNOME terminal, 288
 - GNU screen, 289
 - guake, 288
 - Konsole, 288
 - Rxvt, 289
 - Rxvt-unicode, 289
 - Terminator, 288
 - Tilda, 289
 - tmux, 289
 - Xfce4-terminal, 288
 - XTerm, 288
- Terminfo, 289
- test, 221
- time, 207, 265
- top, 90
- touch, 39
- tput, 311
- tr, 132
- true, 222
- truncate, 159
- type, 224
- UID, 74
- unexpand, 157
- Unics, 9
- uniq, 124
- Unix, 9
- Unix-like, 11
 - AIX, 11
 - Android, 12
 - BlackBerry 10, 11
 - FreeBSD, 11
 - HP-UX, 11
 - macOS, 11
 - Minix, 11
 - NetBSD, 11
 - OpenBSD, 11
 - Solaris, 11
 - Tru64, 11
- unset, 100
- uptime, 91
- vi/vim, 59
- vmstat, 91
- watch, 279
- wc, 56
- which, 82
- whoami, 73
- X/Open, 10
- xargs, 153
- Z3, 7
- აპლიკაცია, 12
- ბირთვი, 12
- ბუფერი, 92
- გარემოს ცვლადი, 99
 - HOME, 101
 - LANG, 102
 - OLDPWD, 102
 - PATH, 102
 - PWD, 101
 - SHELL, 101
 - TERM, 289
 - USER, 101
- დაშიფვრა, 133
- ზომის ერთეული, 141
 - BB, 141
 - EB, 141
 - EiB, 141
 - GB, 141
 - GiB, 141
 - GPB, 141
 - KB, 141
 - KiB, 141
 - MB, 141
 - MiB, 141
 - PB, 141
 - PiB, 141
 - TB, 141
 - TiB, 141
 - YB, 141
 - YiB, 141
 - ZB, 141
 - ZiB, 141
- ინტერპრეტატორი
 - ash, 219
 - csh, 219
 - fish, 219
 - ksh, 219
 - sh, 219
 - tcsh, 219
 - zsh, 219
- კეში, 92
- კოდირება, 133
- კონსოლი, 23
- ლიცენზია, 13
 - Affero, 15
 - Apache, 15
 - BSD, 15
 - GPL, 15
 - LGPL, 15
 - MIT, 15
 - MPL, 15
 - Public domain, 15
- ლოკალური ცვლადი, 99
- მესაკუთრეობრივი, 13
- მილი, 55
- ოფცია, 28
- პროგრამა, 12
- რეგისტრი, 92
- სამკომპონენტო
 - ოპერატორი, 255
- სიტყვა-გასაღები
 - [[, 228
 - case, 232
 - for, 237
 - if, 229
 - select, 249
 - time, 264
 - until, 246
 - while, 245

სკრიფტი, [13](#)
ტერმინალი, [23](#)
შელის ცვლადი, [99](#)

HISTFILE, [102](#)
HISTFILESIZE, [102](#)
PS1, [102](#)

პეშირება, [133](#)