საქართველოს საპატრიარქოს წმ. ანდრია პირველწოდებულის სახელობის
ქართული უნივერსიტეტი

ფიზიკა-მათემატიკურ და კომპიუტერულ მეცნიერებათა სკოლა(ფაკულტეტი)

კომპიუტერული ტექნოლოგიები და მათემატიკური მოდელირება

ბენიძე ნანა

ინფორმატიკის დოქტორის აკადემიური ხარისხის მოსაპოვებლად წარმოდგენილი

სადისერტაციო ნაშრომი

კომპიუტერული მეცნიერებები - 04.01.04

ალგორითმებისა და პროგრამების ვერიფიკაციის (კორექტულობის) კრიტერიუმების დადგენა ავტომატების თეორიის მეთოდებით

სამეცნიერო ხელმძღვანელი: ფიზიკა მათემატიკის მეცნიერებათა დოქტორი, პროფესორი გ. ცერცვაძე

თბილისი

2012

სარჩევი

ანოტაცია

საინფორმაციო ტექნოლოგიების სწრაფი და ინტენსიური ზრდის პირობებში განსაკუთრებით აქტუალური გახდა საიმედოდ გამართული პროგრამული უზრუნველყოფის შემუშავება. ცხადია, რაც უფრო ნაკლებია შეცდომების დაშვების შემთხვევები შესაბამის პროგრამულ უზრუნველყოფაში, მით უფრო საიმედოა ასეთი პროგრამული უზრუნველყოფის პირობებში მომუშავე კომპიუტერული ტექნიკა. საინფორმაციო ტექნოლოგიების პროგრამული უზრუნველყოფის შექმნის პრაქტიკამ აჩვენა, რომ სისტემის შექმნაზე დახარჯული დროის 2/3 მოდის მის გამართვაზე, ანუ იმის შემოწმებაზე, თუ რამდენად კორექტულია შესაბამისი პროგრამა. დღეისათვის ალგორითმებისა და შესაბამისი პროგრამების კორექტულობის შემოწმების ყველაზე მარტივ საშუალებას წარმოადგენს ტესტირება. თუმცა პროგრამული ტექნოლოგიების განვითარების კვალდაკვალ ტესტირების ალტერნატივად მოიაზრება პროგრამების მათემატიკური ვერიფიცირება(კორექტულობა).

საკითხები, რომლებიც წარმოდგენილ სადისერტაციო ნაშრომშია განხილული, ეხება ალგორითმულ ენებზე დაწერილი თანამედროვე კომპიუტერული პროგრამების ვერიფიკაციის ამოცანას ამ პროგრამებისათვის დამახასიათებელი თავისებურებებით. ვინაიდან პროგრამების ვერიფიკაციის ამოცანის გადაწყვეტა არსებითად დაკავშირებულია ამ პროგრამების მიმართ წაყენებულ არაფორმალურ მოთხოვნებთან, ამიტომ განსაკუთრებულ მნიშვნელობას იძენს მათემატიკურად კორექტული ამოცანის დასმა ზემოთ აღნიშნული არაფორმალური მოთხოვნათა პირობებში.

ამ საკითხის გადაჭრის მიზნით დისერტაციაში განვითარებულია მიდგომა, რომელიც არსებითად ეფუძნება სასრული ავტომატის ფორმალურ მოდელს, რომლის ფარგლებში მოხერხებულია სასრულ-ავტომატური ენების აღწერა და ანალიზი.

წარმოადგენს რა ალგორითმებისა და მათი შესაბამისი კომპიუტერული პროგრამების ტესტირების ალტერნატივას, პროგრამების ვერიფიკაციის სასრულ–ავტომატური მეთოდი არსებითად ემყარება ზუსტ სპეციფიკაციას როგორც პროგრამის მიმართ წაყენებული არაფორმალური მოთხოვნების გამოხატვის შესაძლებლობას ავტომატების თეორიის ენაზე.

Algorithms and software verification (correctness) criteria determination by methods of automata theory

Annotation

Under the condition of rapid and intense growth of the information technology that securing the smooth development of the software has become particularly important. Obviously, in the case of fewer mistakes, the computer techniques are more reliable in terms of software-based computers. The practice to create the information technology software has shown that 2/3 spent time on system comes from its setup, or to check on how correct is the relevant program. Today, testing is the simplest means for checking the correctness of the algorithms and corresponding programs. However, following the development of software technology, mathematical (formal) verification is considered as an alternative to testing.

Issues that are discussed in the presented thesis refer to the verification task of the modern computer programs written in algorithmical language by characteristic features of these programs.

According to the fact that the programs verification tasks is substantially related to the informal requirements against these programs, therefore, particular importance given the mathematically correct setting of the task in above mentioned informal requirements conditions.

In order to solve this issue in the thesis is developed an approach that is essentially based on a formal model of the finite-state automata, in the framework of which the description and analysis of finite-state language is convenient.

Performs the testing alternative of algorithms and their appropriate computer programs, complete automatic method is essentially based on the exact specification of the program verification, as the opportunity to express requirements on finite-state language toward programs.

შესავალი

პროგრამირების სამყაროში, ყოველთვის აქტუალური იყო პროგრამების კორექტულობის, ანუ ჭეშმარიტების პრობლემა. ნებისმიერმა პროგრამისტმა, საკუთარი მწარე გამოცდილებიდან გამომდინარე იცის, თუ რა რთულია კორექტული პროგრამის დაწერა, ანუ ისეთი პროგრამის, რომელიც ზუსტად იმას აკეთებს, რა მოთხოვნებიც ამ პროგრამის წინაშე დგას.

ობიექტურად - ორიენტირებული ტექნოლოგიის დამკვიდრების შემდეგ, იქმნება რა საკმაოდ სერიოზული და შთამბეჭდავი მასშტაბების პროგრამული პროდუქტი, ამ პრობლემის სიმწვავე განსაკუთრებით მნიშვნელოვანია. ასეთ სერიოზულ პროექტებში ხშირია შეცდომები, რომელთა შედეგები სხვადასხვა ხასიათის შეიძლება იყოს: ზოგიერთი მათგანი უმნიშვნელო და უწყინარია, ზოგიერთი მათგანი უბრალოდ შემაწუხებელია, ხოლო ზოგიერთი კი სასიცოცხლოდ მნიშვნელოვანიც შეიძლება იყოს. პროგრამული უზრუნველყოფის შექმნის პრაქტიკამ აჩვენა, რომ სისტემის შექმნაზე დახარჯული დროის 2/3 მის გამართვაზე მოდის, ანუ იმის შემოწმებაზე, თუ რამდენად სწორია პროგრამა. ყველაზე მარტივი საშუალება კონკრეტული ალგორითმისა და შესაბამისი პროგრამის კორექტულობის შესამოწმებლად არის ტესტირება.

ტესტირების მეთოდი არსი საკმაოდ მარტივი და ნათელია: თუ პროგრამა კორექტულად მუშაობს სპეციალურად შერჩეული საწყის მონაცემთა საკმაოდ დიდი სიმრავლისათვის, მაშინ ალბათობა იმისა, რომ ეს პროგრამა იმუშავებს კორექტულად ნებისმიერი საწყისი მონაცემებისათვის ინტუიციურად საკმაოდ მაღალია. მაგრამ არანაკლებ დიდია ალბათობა იმისა, რომ თუნდაც ერთი რომელიმე საწყისი მონაცემთა ნაკრებისათვის პროგრამა მცდარია.

დეიქსტრა, თავის სტატიაში "შენიშვნები სტრუქტურულ პროგრამირებაზე"[1] აღნიშნავს, რომ ტესტი შეიძლება სასარგებლო იყოს შეცდომების არსებობის დემონსტრირებისათვის, მაგრამ მათი არ არსებობის დამტკიცება მხოლოდ გამონაკლის შემთხვევაში შეუძლია. (ეს გამონაკლისები, თავის მხრივ, მხოლოდ ბედნიერ შემთხვევითობას უნდა უმადლოდეს).

პროგრამების ტესტირების ალტერნატივად მოიაზრება ამ პროგრამების მათემატიკური ვერიფიცირების შესაძლებლობა.

ვერიფიკაცია[1] განსაზღვრავს, აკმაყოფილებს თუ არა პროგრამა და მისი კომპონენტები (ქვეპროგრამები) იმ მოთხოვნებს, რომელიც მათ წინაშე დგას. ვერიფიცირების ძირითადი მიზანია იმის დამტკიცება, რომ პროგრამა შეესაბამება. ამ მოთხოვნებს, ხოლო მისი დამატებითი მისიაა პროგრამის დეფექტების და შეცდომების აღმოჩენა – რეგისტრაცია.

პროგრამის ვერიფიცირებისა და გამართვის ცნებები, აბსოლუტურად სხვადასხვა რამეა. ორივე ეს პროცესი მიმართულია პროგრამის შეცდომების შემცირებისაკენ. მაგრამ პროგრამის გამართვა გულისხმობს პროგრამაში შეცდომების ლოკალიზებას და მის აღმოფხვრას, ხოლო ვერიფიკაცია კი გულისხმობს შეცდომების არსებობის დემონსტრირებას და მათი აღმოჩენის შესაძლებლობებს და ასევე იმ მიზეზების აღმოჩენა, რა პირობებში შეიძლება წარმოიშვას ეს შეცდომები.

ვერიფიკაცია მოიცავს თავის თავში შეცდომების აღმოცენების მიზეზების ანალიზს, ასევე იმ შეცდომების ანალიზს, რაც მოსალოდნელია მათი გამოსწორების შემდეგ, შეცდომების ძიების პროცესის უზრუნველყოფას და მათ აღმოფხვრას, მიღებული შედეგების ანალიზს.

პროგრამას, რომელიც რეალიზებულია რომელიმე დაპროგრამების ენის საშუალებებით, გააჩნია ცალსახა მათემატიკური ახსნა. ანალოგიურად, პროგრამებისადმი მოთხოვნებიც შეიძლება გამოხატული იყოს მათემატიკისა და ლოგიკის ცნებების ფარგლებში, როგორც ზუსტი სპეციფიკაცია. ასეთი ფორმალიზაცია საშუალებას იძლევა დამტკიცდეს შესაბამისობა პროგრამებსა და მათემატიკურ სპეციფიკაციას შორის. ამრიგად, პროგრამების ვერიფიკაცია ეყრდნობა ამ პროგრამის მათემატიკურ სპეციფიკაციას. ზოგჯერ შესაძლებელია აღმოჩნდეს, რომ ეს მათემატიკური სპეციფიკაცია ზუსტად არ შეესაბამება იმ მოთხოვნებს, რომლებიც პროგრამის წინაშე დგას. სწორედ ამიტომ ყველაზე რთულ ამოცანას წარმოადგენს პროგრამის არაფორმალური მოთხოვნების მათემატიკურად კორექტული ვერსიის ჩამოყალიბება.

შექმნილი ვითარება სქემატურად შეიძლება შემდეგნაირად გამოისახოს

[1] ლათინურად *verus* — "ჭეშმარიტი" და *facere* — "კეთება"

საჭიროა დიაგრამის ზედა და ქვედა ელემენტებს შორის კავშირის დადგენა, ანუ პროგრამისადმი მოთხოვნებსა და პროგრამის რეალიზაციას შორის კავშირის დადგენა. ამ მიზნის მისაღწევად საჭიროა ორივე მათგანის ფორმალიზაცია. ზემოთ მოყვანილ დიაგრამაზე მხოლოდ ერთ კავშირს, კავშირს პროგრამის მათემატიკურ მოდელსა და მოთხოვნების მათემატიკური სპეციფიკაციას შორის, გააჩნია ზუსტი მათემატიკური რაობა(შინაარსი), მაშინ როდესაც ყველა სხვა კავშირი არაფორმალური ხასიათისაა. ამ დროს აუცილებელია შენარჩუნებული იყოს მაქსიმალური გამჭირვალობა და სიმარტივე არაფორმალურ და ფორმალურ მოთხოვნებს შორის.

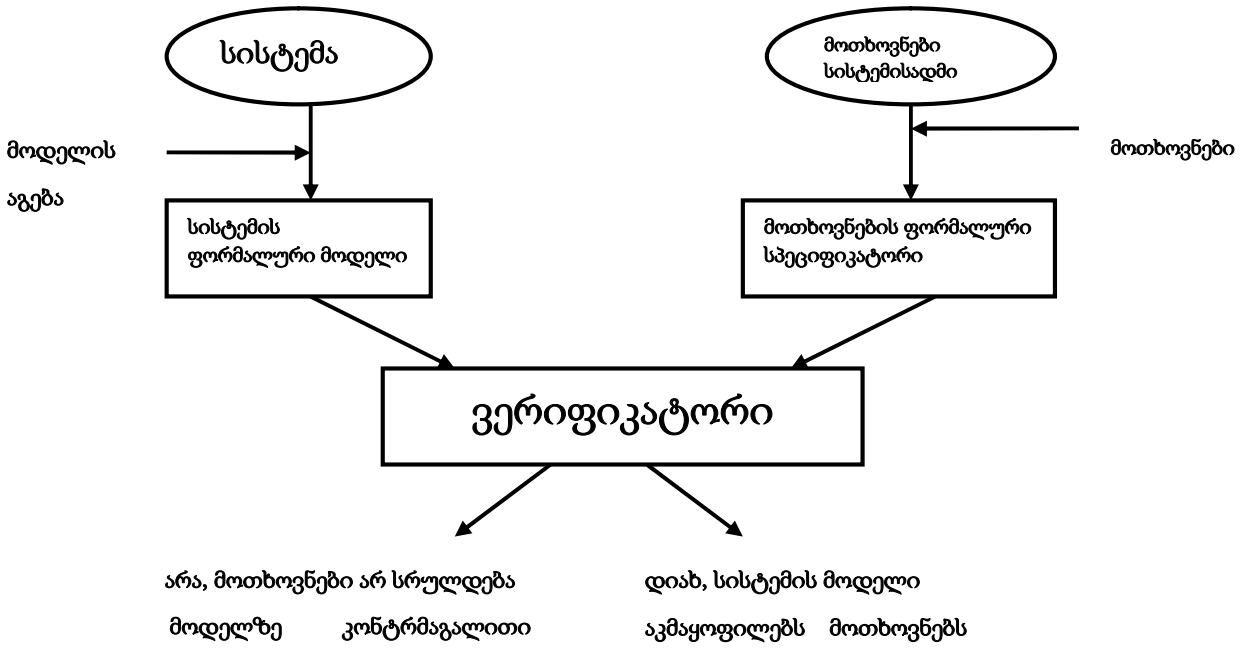
მიუხედავად, ზემოთ აღნიშნული ფორმალური და არაფორმალური სპეციფიკაციების არსებობისა (განსაკუთრებით დამაბრკოლებელი ფაქტორია არაფორმალური მოთხოვნები), ვერიფიკაცია ფლობს უპირატესობას ტესტირებასთან შედარებით: ის ერთხელ და საბოლოოდ ადგენს პროგრამის კორექტულობას და ეს შესაძლებელია მოხდეს საკმაოდ ფართო კლასის ამოცანებისათვის. უფრო მეტიც, ვერიფიკაციის პროცესის ანალიზური ხასიათი განაპირობებს (წარუმატებლობის შემთხვევაშიც კი) არა მარტო პროგრამის უფრო ღრმად გაგებას, არამედ დასმული ამოცანის და პრობლემის უფრო სერიოზულ ანალიზს.

ვერიფიკაციის სქემა მოყვანილია 1.1 ნახაზზე.

საზოგადოდ, პრობლემა მოიცავს ორ ასპექტს:

- ჭეშმარიტია თუ არა ჩვენს მიერ დაწერილი პროგრამა.
- როგორ დავწეროთ კორექტული პროგრამა.

ამ პრობლემებზე მუშაობდნენ და მუშაობენ ამერიკაში, ევროპაში, რუსეთში: არსებობს უამრავი მოსაზრება, მიმართულება, მეთოდი, მაგრამ ჯერ-ჯერობით არ



ნახ1. ვერიფიკაციის საერთო სქემა

არსებობს ერთი უნივერსალური საშუალება, რომელიც ნებისმიერი პროგრამისათვის დასვამდა უტყუარ დიაგნოზს : კორექტულია იგი თუ არა.

შენიშვნა 1. ნაშრომში, ზოგიერთ ადგილას შეგხვდებათ ტერმინი პროგრამის კორექტულობა, ზოგიერთ ადგილას კიდევ ალგორითმისა და მისი შესაბამისი პროგრამის სამართლიანობა. იმდენად, რამდენადაც პროგრამა წარმოადგენს ალგორითმის ჩაწერის ერთ-ერთ ხერხს, ამიტომ ასეთი ტერმინოლოგიური აღრევა სავსებით გუნებრივად ჩავთვალე და არ შევეცადე ამ ორი ტერმინის ერთმანეთისაგან გამიჯვნა.
შენიშვნა 2. ეს შენიშვნაც ტერმინოლოგიური ხასიათისაა. პროგრამის კორექტულობა, სამართლიანობა, ჭეშმარიტება ექვივალენტური ცნებებია და იმის მიხედვით თუ კონტექსტი რას მოითხოვს, შეიძლება გამოყენებული იყოს კორექტულობაც, სამართლიანობაც და ჭეშმარიტებაც.

ისეთი პროგრამების შექმნის სურვილმა, რომლებისთვისაც კორექტულობის დამტკიცება შესაძლებელი იქნებოდა ჩვენი ყურადღება გადაიტანა ისეთი საკითხების შეაწავლაზე, რომლებიც უშუალოდ დაკავშირებული არიან პროგრამების კონსტრუირებისა და დაპროგრამების ენების შექმნის პრობლემატიკასთან.

ნიდერლანდური წარმომავლობის გენიალური ადამიანის და პროგრამისტის დეიქსტრას[2] ნაშრომები იყო პირველი იმ ნაშრომთაგანი, რომლებიც ეხება პროგრამების კონსტრუიურებისა და მათი კორექტულობის დამტკიცების პრობლემას. თავის ნაშრომებში დეიქსტრა ყოველთვის ხაზს უსვამს იმ ფაქტს, რომ ენის შექმნის საკითხები უცილობელ კავშირშია კორექტულობის პრობლემებთან და ლამის „ჯვაროსნული ბრძოლა“ აქვს გამართული goto ოპერატორის საწინააღმდეგოდ.

არსებობს მთელი რიგი ნაშრომებისა, რომლებიც შეეხება სტრუქტურული პროგრამირების პრინციპებს და სადაც პროგრამების შექმნის პრობლება დაკავშირებულია მათი სამართლიანობის პრობლემასთან. ძირითადი ყურადღება გამახვილებულია პროგრამების შექმნის ისეთ მეთოდებზე, რომლებიც პროგრამისტს აიძულებს თვალი მიადევნოს პროგრამის კორექტულობის საკითხებს, მისი შექმნის ყველა ეტაპზე.

დეიქსტრას სტატიები : “შენიშვნები სტრუქტურულ პროგრამირებაზე” [1], “GOTO ოპერატორის საფრთხეები”(GOTO considered harmful) და წიგნი “პროგრამირების დისციპლინა”[2] წარმოადგენს კლასიკურ ნაშრომებს სტრუქტურული პროგრამირების დარგში. გასული საუკუნის 70-იან წლებში ტონი ხოართან[3] და ნიკალაუს ვირტთან[4] ერთად მან შეიმუშავა სტრუქტურული პროგრამირების ძირითადი პრინციპები.

დეიქსტრა ავითარებდა იმ აზრს, რომ პროგრამირებისადმი აუცილებელია მათემატიკური მიდგომა, რაც თავისთავად გულისხმობს დასმული ამოცანის და მისი გადაჭრის მათემატიკური სპეციფიკაციის აღწერას, არჩეული ალგორითმის სამართლიანობის ფორმალურ დამტკიცებას და მხოლოდ ამის შემდეგ, ჭეშმარიტი ალგორითმის რეალიზაციას მაქსიმალურად მარტივი, სტრუქტურირებული პროგრამის სახით, რომლის კორექტულობის დამტკიცებაც ასევე აუცილებელია. დეიქსტრას

---

[2] ედგარ ვიბე დეიქსტრა (ინგლისურად Edsger Wybe Dijkstra) დაიბადა 1930 წელს როტერდამში, ჰოლანდია, გარდაიცვალა 2002წელს). ჰოლანდიელი მეცნიერი, რომლის იდეებმა მნიშვნელოვანი გავლენა იქონია კომპიუტერულ ინდუსტრიაზე. 1972 წელს ის გახდა ტიურინგის პრემიის ლაურიატი.

[3] სერ ჩარლზ ენტონი რიჩარდ ხოარი (ინგ. Charles Antony Richard Hoare, დაიბადა 1934 წელს, ინგლისის იმპერია, კოლომბო, ცეილონი, დღევანდელი შრილანკა) — ინგლისელი მეცნიერი, ინფორმატიკისა და გამოთვლითი ტექნიკის სფეროში.

[4] ნიკალაუს ვირტი (გერმანულად Niklaus Wirth, დაიბადა 1934 წელს) — შვეიცარელი მეცნიერი, ინფორმატიკის დარგში, ერთერთი ცნობილი თეორეტიკოსი დაპროგრამების ენების შექმნის სფეროში, კომპიუტერულ მეცნიერებათა პროფესორი, 1984 წლის ტიურინგის პრემიის ლაურიატი.

აზრით, იმ წლებში გაბატონებული ტესტირების მიდგომა მანკიერია და პროგრამირებას განიხილავს როგორც ხელობას და არა ხელოვნებას.

სწორედ ამ წლებს ეკუთვნის დეიქსტრას ცნობილი თეზა:

ტესტი შეიძლება სასარგებლო იყოს შეცდომების არსებობის თვალსაზრისით, მაგრამ მათი არ არსებობის დამტკიცება მხოლოდ გამონაკლის შემთხვევაში შეუძლია.

დეიქსტრას მიმდევრები და თანამედროვეები იყვნენ ჩარლზ ენტონ რიჩარდ ხოარი, ნიკალაუს ვირტი, რობერტ ფლოიდი[5] . . .

1969 წელს ინფორმატიკისა და მათემატიკურ ლოგიკის ინგლისელმა მეცნიერმა ხოარმა შეიმუშავა ე.წ. ხოარის ლოგიკის პრინციპები (Hoare logic, ანუ Hoare rules), რომელიც წარმოადგენდა კომპიუტერული პროგრამების კორექტულობის დასამტკიცებელ ფორმალურ სისტემას ლოგიკურ წესების თანხლებით. თუმცა, სამართლიანობის გამო უნდა ავღნიშნოთ რომ პირველად იდეა ეკუთვნოდა ფლოიდს (რომელიც ასევე მუშაობდა ფორმალური ვერიფიკაციის საკითხებზე), რომელმაც აღწერა მსგავსი სისტემა, მხოლოდ ბლოკ-სქემებთან მიმართებაში. ამიტომაც, ხოარის ლოგიკას ხშირად მოიხსენიებენ ხოლმე როგორც ფლოიდი - ხოარის ლოგიკას (Floyd — Hoare logic).

ამავე წლებში დაედო საფუძველი დონალდ კნუტის[6] ეპოქალურ ოთხ ტომეულს ”პროგრამირების ხელოვნება”[3], სადაც განხილულია უამრავი ალგორითმი, მათი კორექტულობის დამტკიცების შესაბამისი მათემატიკური აპარატის მოყვანით. წიგნის სარედაქციო კოლეგიაში ასევე მუშაობდა რობერტ ფლოიდი.

ამავე წლების მონაპოვარია ნიკალაუს ვირტის ძალიან საინტერესო წიგნი “ალგორითმები + მონაცემთა სტრუქტურები = პროგრამებს ” [4], თუმცა ამ წიგნში ვერიფიკაციის საკითხები უკანა პლანზე დგას და იმ საკითხებზეა გამახვილებული ყურადღება, თუ როგორ უნდა დავწეროთ ე.წ. წიგნიერი პროგრამები.

გავიდა წლები და პროგრამირებაში ფეხი მოიკიდა და დამკვიდრდა ფორმალური ვერიფიკაციის ტერმინი.

---

[5] რობერტ ვ ფლოიდი (ინგლისურად Robert W Floyd, დაიბადა 1936 წელს, ნიუ-იორკი, აშშ — გარდაიცვალა 2001 წელს, სტენფორდი, აშშ) — ამერიკელი მეცნიერი გამოთვლითი სისტემების თეორიის სფეროში.

[6] დონალდ ერვინ კნუტი (ინგლისურად KnuthDonald Ervin Knuth, დაიბადა 1938 წელს) — ამერიკელი მეცნიერი, სტენფორდის უნივერსიტეტის საპატიო პროფესორი, ასევე რამდენიმე სხვადასხვა ქვეყნის საპატიო პროფესორი. 1974 წელს გახდა ტიურინგის პრემიის ლაურიატი გამოთვლითი მათემატიკისა და პროგრამირების დარგში გაწეული წვლილისათვის.

ფორმალური ვერიფიკაცია წარმოადგენს ფორმალური მეთოდების საშუალებით პროგრამების სამართლიანობის, თუ არაკორექტულობის დამტკიცების მეთოდს, პროგრამის თვისებების ფორმალურ აღწერასთან ერთად.

ფორმალური ვერიფიკაცია შეიძლება გამოყენებული იყოს ისეთი სისტემების შესამოწმებლად, როგორიცაა პროგრამული უზრუნველყოფა, კრიპტოგრაფიული პროტოკოლები, კომბინატორული ლოგიკური სქემები, ციფრული სქემები შიდა მეხსიერებით და ა.შ.

ვერიფიკაცია წარმოადგენს ფორმალურ დამტკიცებას სისტემის აბსტრაქტულ მათემატიკურ მოდელზე.

სისტემებისა და პროგრამების მოდელირებასა და ფორმალური ვერიფიკაციაში ხშირად გამოყენებადი მათემატიკური ობიექტების მაგალითებია:

- დაპროგრამების ენების ფორმალური სემანტიკა, მაგალითად აქსიომატიკური სემანტიკა (ხოარის ლოგიკა)
- სასრული ავტომატი
- პეტრას ქსელი
- დროითი ავტომატი
- ჰიბრიდული ავტომატი
- სტრუქტურირებული ალგორითმები
- სტრუქტურირებული პროგრამები

დღევანდელი გადასახედიდან ფორმალური ვერიფიკაციის მეთოდებია:

- ხოარის აქსიომატური სემანტიკის მეთოდი [7]
- ფლოიდის ინდუქციურ დაშვებათა მეთოდი [8]
- მოდელების შემოწმება(Model cheking) [9]

---

[7] რიჩარდ ენტონი ხოარი 1980 წელს გახდა ტიურინგის პრემიის ლაურიატი "დაპროგრამების ენების განსაზღვრასა და დიზაინში გაწეული დამსახურებისათვის"

[8] რობერტ ფლოიდი 1978 წელს გახდა ტიურინგის პრემიის ლაურიატი "ეფექტური და საიმედო პროგრამული უზრუნველყოფის მეთოდოლოგიის შემუშავების საკითხში მისი უცილობელი გავლენის გამო და მის მიერ გაწეული ღვაწლის გამო კომპიუტერული მეცნიერების ისეთ სფეროებში როგორიცაა დაპროგრამების ენების სემანტიკა, პროგრამების ავტომატური ვერიფიცირება, პროგრამების ავტომატური სინტეზირება და ალგორითმების ანალიზი."

[9] ედმუნდ კლარკმა, ალენ ემერსონთან და იოსებ სიფაკისთან ერთად 2007 წელს მიიღო ტიურინგის პრემია "მოდელების შემოწმების (model cheking) - პარალელური პროგრამირების ვერიფიცირების მაღალეფექტური ტექნოლოგიის განვითარებაში შესრულებული როლის გამო"

- Proofing programming – მტკიცებულებითი პროგრამირება
- Theorem proving – თეორემების ავტომატური დამტკიცება.
- Symbolic execution – სიმბოლური შესრულება .
- Abstract Interpretation - აბსტრაქტული ინტერპრეტაცია.

ფორმალური ვერიფიცირების საკითხები დღესაც არ კარგავს თავის აქტუალობას. ამაზე მეტყველებს თუნდაც ის ფაქტი, რომ 2007 წელს ტიურინგის პრემია მიენიჭა ედმუნდ კლარკს, ალენ ემერსონათან და იოსებ სიფაკისთან ერთად "მოდელების შემოწმების" (model cheking) ტექნოლოგიის დამუშავებისათვის. თუმცა სამართლიანობისათვის უნდა ავღნიშნოთ, რომ ამ საკითხებზე მუშაობა დაიწყო გაცილებით ადრე: გასული საუკუნის 80-იან წლებში კლარკისა და ემერსონის მიერ ამერიკაში, ხოლო მათგან დამოუკიდებლად კვაილისა და სიფაკისის მიერ საფრანგეთში შემუშავებული იყო ვერიფიცირების ტექნიკა, რომელმაც შემდეგ თავი დაიმკვიდრა, როგორც ვერიფიცირება მოდელებზე, ან შემოწმება მოდელებზე დროითი (ანუ ტემპოლარული) ლოგიკის დახმარებით.

მოდელებზე შემოწმება წარმოადგენს სასრული რაოდენობა მდგომარეობების მქონე პარალელური სისტემების ავტომატურ ვერიფიცირების მეთოდს. ეს მეთოდი წარმატებით იყო გამოყენებული მიმდევრობითი ელექტრონული სქემებისა და რთული საკომუნიკაციო ე.წ. პროტოკოლების (ოქმების) გამართულობის შესამოწმებლად.

არსებობს უამრავი მზა ინსტრუმენტი, რომელიც სწორედ ამ მეთოდს იყენებს, მაგალითად:

BLAST — C პროგრამების სტატისტიკური ანალიზატორი

CADP (Construction and Analysis of Distributed Processes) — დანაწილებული სისტემებისა და ოქმების პროექტირების ინსტრუმენტი

CHESS — მრავალპროგრამული (მრავალნაკადური) .Net და Win32, 64 ბიტიანი პროგრამების ტესტირების ინსტრუმენტი

ISP— MPI პროგრამების კოდის ვერიფიკატორი.

Java Pathfinder — მულტინაკადური ჯავა პროგრამების შემოწმების თავისუფალი ინსტრუმენტი

MoonWalker — .Net პროგრამების შემოწმების თავისუფალი ინსტრუმენტი

MRMC (Markov Reward Model Checker)

NuSMV — შემოწმების სიმბოლური მოდელი

PRISM — შემოწმების ალბათური სიმბოლური მოდელი

Rabbit — რეალურ დროში მომუშავე სისტემების შემოწმების მოდელი

SPIN — განაწილებული (დაყოფილი, პარალელური) პროგრამების ვერიფიკაციის შემოწმების ზოგადი დანიშნულების მოდელი

Vereofy — კომპონენტური სისტემების პროგრამების შემოწმების მოდელი

μCRL2— თავისუფალი ინსტრუმენტი დაფუძნებული ASP - ზე

UPPAAL — რეალური დროის სისტემების (რომლებიც მოდელირებულია როგორც დროებითი ავტომატების ქსელი) მოდელირების, ვერიფიკაციის და ვალიდაციის ინსტრუმენტი.

ასევე 2008 წელს ბარბარა ლისკოვმა[10] მიიღო ტიურინგის პრემია დაპროგრამების ენების და სისტემური დიზაინის პრაქტიკული და თეორიული საფუძვლების დამუშავების სფეროში შეტანილი ღვაწლისათვის. კერძოდ, შეცდომებისადმი მდგრადი აბსტრაქტული მონაცემებისა და დაყოფილი (პარალელური) გამოთვლების გამოკვლევის საკითხებში (Liskov Substitution Principle, LSP).

წარმოდგენილი სადისერტაციო ნაშრომი მოიცავს პროგრამების ვერიფიკაციის პრობლემასთან დაკავშირებული ამოცანების შესწავლას და მისი გადაწყვეტის თანამედროვე მეთოდების დამუშავებას. როგორც, ზემოთ აღვნიშნეთ, პროგრამების ვერიფიკაცია არსებითად დაკავშირებულია არაფორმალურ მოთხოვნებთან, ეს ართულებს მათემატიკურად კორექტული ამოცანის დასმას და საკითხის მთელი სირთულე სწორედ ამ პრობლემის გადაჭრაზე მოდის.

საკითხის აქტუალურობა განპირობებულია იმით, რომ თანამედროვე ცივილიზებული სამყარო არა მარტო სამეცნიერო თემატიკასთან დაკავშირებით, არამედ ჩვეულებრივ ყოფით ცხოვრებაში აქტიურად იყენებს კომპიუტერულ ტექნიკას. ეს თავისთავად დაკავშირებულია შესაბამისი პროგრამული უზრუნველყოფის გამართულობასთან. ცხადია, რომ რაც უფრო ნაკლებია შეცდომები პროგრამულ

[10] ბარბარა ლისკოვა (ინგლისურად Barbara Liskov, დაიბადა 1939 წელს) — გამოთვლითი სისტემების თეორიის სფეროში მომუშავე მეცნიერი.

უზრუნველყოფაში, მით უფრო საიმედოა ასეთი პროგრამული უზრუნველყოფის ქვეშ მომუშავე კომპიუტერული ტექნიკა.

წარმოდგენილი დისერტაციის მიზანია პროგრამების საიმედო ფუნქციონირების კრიტერიუმების დადგენა სასრულ - ავტომატური მეთოდის გამოყენებით, ბუნებრივია გარკვეული კლასის ამოცანებისათვის.

დისერტაცია შედგება შესავალისაგან, სამი თავისაგან და დანართისაგან.

შესავალში ფორმულირებულია დისერტაციაში განხილული საკითხების ისტორიული წინასვლები და საკითხის მეცნიერული და მეთოდოლოგიური პრინციპები.

ნაშრომის პირველი თავი შეეხება პრობლემასთან დაკავშირებული არსებული სამეცნიერო ლიტერატურის მიმოხილვას და ჩატარებულია მიღებული შედეგების შედარებითი ანალიზი.

მეორე თავი მოიცავს ფორმალური ენების, გრამატიკებისა და სასრულ-ავტომატური თეორიის იმ მეთოდების მიმოხილვას, რომელიც გამოყენებულია დისერტაციის თემატიკის დამუშავების დროს.

ამავე თავში დამტკიცებულია ორი ახალი თეორემა, რომელიც შეეხება სინტაქსური ანალიზის შესაბამისი ალგორითმების კორექტულობის პრინციპს და მოყვანილია თავად ეს პრინციპი.

დისერტაციის მესამე თავში განხილულია ერთი კრიპტოლოგიური ამოცანა. განხილულია ბლოკური დაშიფვრის სიმეტრიული კრიპტოგრაფიული სისტემის ამოცანის ალგორითმი და დადგენილია მისი ჭეშმარიტების კრიტერიუმი.

დანართში მოყვანილია

1. დაპროგრამების ენის სინტაქსის აღწერა ბეკუს–ნაურის ფორმალიზმით.
2. დისერტაციის მესამე თავში განხილული კრიპტოგრაფიული ამოცანის ალგორითმის შესაბამისი პროგრამული კოდები.

თავი 1. ვერიფიკაციის მეთოდების მიმოხილვა

როგორც შესავალში ავღნიშნე, დღევანდელ სინამდვილეში პროგრამების სამართლიანობის დამტკიცების სფეროში საკმაოდ ინტენსიური მუშაობა მიმდინარეობდა და მიმდინარეობს. პირველ თავში განვიხილავ სამი ძირითადი მიმართულებას:

1. ხოარის აქსიომატური სემანტიკის მეთოდი
2. ფლოიდის ინდუქციურ დაშვებათა მეთოდი
3. მოდელების შემოწმება (model cheking)

ბუნებრივია, ამ მიმართულებებს გააჩნიათ შეხების წერტილები, ზოგიერთ საკითხებში კი შეიძლება მოხდეს მათი ურთიერთგადაფარვა. მიუხედავად ამისა თითოეულ მათგანს გააჩნია თვითმყოფადი თავისებურებები და ფილოსოფია, რაც კვლევისა და განსჯის საკმაოდ ნოყიერ ნიადაგს ქმნის.

განვიხილოთ ცალ-ცალკე თითოეული მათგანი და შეძლებისდაგვარად გავაკეთოთ შესაბამისი დასკვნები.

1.1 ხოარის აქსიომატური სემანტიკის მეთოდი

ხოარის აქსიომატური სემანტიკის მეთოდი დამყარებულია ე.წ. დეკომპოზიციის პრინციპზე, რომლის არსი შემდეგში მდგომარეობს:

ნაბიჯი 1. საერთო ამოცანის დაყოფა (დეკომპოზიცია) ზუსტად განსაზღვრულ ქვეამოცანებად და იმის დამტკიცება, რომ თუ თითოეული ქვეამოცანა გადაჭრილია კორექტულად და მიღებული შედეგები დაკავშირებულია ერთმანეთთან გარკვეული თანადობით, მაშინ საწყისი ამოცანა ამოხსნილია კორექტულად (ან რომელიმე ქვეამოცანა არაკორექტულია).

ნაბიჯი 2. "დეკომპოზიციისა და მიღებული ქვეამოცანების კორექტულობის" დამტკიცების პროცესის გამეორება ყოველი ქვეამოცანისათვის.

ნაბიჯი 3. პირველი და მეორე ნაბიჯების გამეორება მანამ სანამ არ მიიღება ისეთი მარტივი ქვეამოცანები, რომელთა გადაწყვეტა შეიძლება მოცემული იყოს კონკრეტული დაპროგრამების ენის რამდენიმე სტრიქონში.

ზემოთ მოყვანილ სამ ნაბიჯს შემდეგში მოვიხსენიებთ, როგორც დეკომპოზიციის პრინციპს.

ბუნებრივია, კონკრეტულად რომელი დაპროგრამების ენა იქნება გამოყენებული, არ არის არსებითი: იგი გამოიყენება როგორც ინსტრუმენტი კონკრეტული ალგორითმის ჩასაწერად. რთული პრობლემების გადაწყვეტის საწყისს ეტაპზე ძირითადი ყურადღება უნდა იყოს მიმართული არა კონკრეტული ენის სპეციფიური დეტალებისადმი, არამედ გლობალური პრობლემებისადმი. პრობლემის გადაწყვეტის პროცესში, იგი უნდა იყოს ფორმულირებული რაც შეიძლება მაღალი დონის ბუნებრივ ენაზე. ამოცანის ასეთ ფორმულირებას ეწოდება აბსტრაქტული ალგორითმები, ხოლო მის შემადგენელ ნაწილებს აბსტრაქტული ოპერატორები და აბსტრაქტული მონაცემები.

დავუბრუნდეთ ისევ დეკომპოზიციის პრინციპს და გავაკეთოთ მისი ფორმულირება ცოტა სხვანაირად, ამისათვის შევთანხმდეთ შემდეგ აღნიშვნებზე:

დავუშვათ S არის პროგრამა, რომლის სამართლიანობა უნდა დამტკიცდეს, P არის დამოკიდებულება განსაზღვრული საწყისს მონაცემებზე, ანუ S პროგრამის ე.წ. წინაპირობა, ხოლო Q დამოკიდებულება განსაზღვრული საშედეგო მონაცემებზე, S პროგრამის ე.წ. შემდეგი პირობა.

ზემოთ ნათქვამი წარმოადგენს ე.წ. ხოარის ლოგიკის ქვაკუთხედს რომელიც შემდეგნაირად ჩაიწერება:

$$\{P\}\ S\ \{Q\} \quad (1)$$

P, S, Q ეწოდება ხოარის სამეული (ინგლისურად Hoare triple) და იგი აღწერს თუ როგორ ცვლის გამოთვლის მდგომარეობას პროგრამის კოდის შესრულება.

(1) ნიშნავს შემდეგს: თუ P წინაპირობა სამართლიანია S პროგრამის შესრულებამდე და S კორექტული პროგრამაა, მაშინ Q შემდეგი პირობა აუცილებლად სამართლიანი იქნება S პროგრამის შესრულების შემდეგ.

მაგალითი:

{ A არის მთელი რიცხვების მასივი } $Sort(N, A)$ $\{A_i < A_{i+1} \quad i = 1, \ldots, N-1\}$

სადაც

$P \quad \cong \quad$ { A მთელი რიცხვების მასივი }

$S \cong Sort(N, A)$

$Q \cong \{ A_i < A_{i+1}, i = 1,..., N \}$

თუ $S \cong Sort(N, A)$ ალგორითმი სამართლიანია, მაშინ P წინაპირობიდან გამომდინარე პირობა $Q \cong \{ A_i < A_{i+1}, i = 1,..., N \}$ აუცილებლად სამართლიანი უნდა იყოს.

ზემოთ მოყვანილ აღნიშვნებში დეკომპოზიციის პრინციპი შეიძლება ფორმულირებული იყოს შემდეგნაირად:

ნაბიჯი 1. S პროგრამა შეიძლება განხილული იყოს, როგორც S_i $(i = 1,.., N)$ ქვეპროგრამების (ან ფრაგმენტების) ერთობლიობა. შესაბამისად P_i და Q_i იქნება S_i – ის წინა და შემდეგი პირობა

$$\{P_i\}\ S_i\ \{Q_i\}$$

თუ შესაძლებელია დამტკიცდეს ყოველი S_i - ის სამართლიანობა, მაშინ პროგრამა, რომელიც S_i ბლოკებისაგან შედგება, ასევე სამართლიანი იქნება.

ნაბიჯი 2. გავიმეოროთ პირველი ნაბიჯი თითოეული S_i ბლოკისათვის.

ნაბიჯი 3. გავიმეოროთ პირველი და მეორე ნაბიჯი, მანამ სანამ არ მივიღებთ ისეთ S_i ბლოკებს (საწყისი პროგრამის ფრაგმენტებს), რომელთა კორექტულობის დამტკიცება ძალიან მარტივია (აქ უკვე შეიძლება გამოყენებული იყოს ტესტირება).

ხოარის ლოგიკაში განსაზღვრულია აქსიომები და ე.წ. გამოყვანის წესები დაპროგრამების ენების ყველა კონსტრუქციისათვის.

შემოვიღოთ აღნიშვნა

$\frac{D1, D2, \ldots, Dn}{D}$ ეს აღნიშვნა ნიშნავს, რომ თუ $D_1, D_2, \ldots, D_n$ კორექტულია, მაშინ D დამოკიდებულებაც სამართლიანი იქნება.

აქსიომა 1.

$$\frac{\{ P\}\ S\{ R \},\ R \supset Q}{\{ P\}\ S\{ Q \}}$$

თუ S პროგრამა უზრუნველყოფს R დამოკიდებულების სამართლიანობას, მაშინ ის ასევე განაპირობებს ყველა იმ მტკიცებულების სამართლიანობას, რაც გამომდინარეობს R – დან.

აქსიომა 2.

$$\frac{P \supset R \ \{ R\}\ S\{ Q \}}{\{ P\}\ S\{ Q \}}$$

თუ R წარმოადგენს S პროგრამის წინაპირობას, ხოლო Q შემდეგ პირობას, მაშინ იგივე სამართლიანია, ყველა იმ P - თვის, საიდანაც გამომდინარეობს R.

1-2 აქსიომას ეწოდება კონსეკვენციის აქსიომები. ზემოთ საუბარი იყო აბსტრაქტულ ალგორითმებზე და მის შემადგენელ აბსტრაქტულ ოპერატორებსა და მონაცემებზე. ყოველი აბსტრაქტული ოპერატორისათვის ხოარის ლოგიკაში არსებობს ე.წ. გამოყვანის წესები – ხოარი ამ წესებს **ვერიფიკაციის წესებს** უწოდებს.

ვერიფიკაციის წესები აბსტრაქტული ოპერატორებისათვის:

1. ცარიელი ოპერატორის შესაბამის წესს აქვს სახე:

$$\{P\}\ ;\ \{Q\}$$

2. შედგენილი ოპერატორი:

$$\frac{\{ P \ \}\ S_1\{ R\}, \{ R\}\ S_2\{ Q\}}{\{ P \ \}\ begin\ S_1, S_2\ , end\{ Q \ \}}$$

ზოგადად შედგენილი ოპარატორის შესაბამისს ვერიფიკაციის წესს აქვს სახე:

$$\frac{\{ P_{i-1} \}\ S_i\{ P_i \}, i = 1..n}{\{ P_0 \}\ begin\ S_1, S_2, ... S_n\ , end\{ P_n \}}$$

3. წესები პირობითი გადასვლის ოპერატორისათვის:

3.1

$$\frac{\{ P \& B \}\ S_1\{ Q \}, \{ P \& \sim B \}\ S_2\{ Q \}}{\{ P \ \}\ if\ B\ then\ S_1\ else\ S_2\{ Q \}}$$

3.2

$$\frac{\{ P \& B \}\ S\ \{ Q \}, P \& \sim B \supset Q}{\{ P \ \}\ if\ B\ then\ S\ \{ Q \}}$$

4. ამორჩევის ოპერატორი:

$$\frac{\{P \,\&\, (x = k_1) S_i \{Q\}, i = 1..n}{\{P \,\&\, (x \in [k_1,...,k_n]) caseof k_1 : S_1;...;k_n : S_n; end \{Q\}}$$

5. while - do ციკლის შესაბამისი წესი:

$$\frac{\{\ P \,\&\, B\ \}\ S\ \{\ P \,\&\, \sim B\ \}}{\{\ P \,\&\, B\ \}\, while\ B\ do\ S\, \{\ P \,\&\, \sim B\ \}}$$

6. repeat - until ციკლი

$$\frac{\{\ P\ \}\ S\ \{\ Q\},\ Q \,\&\, \sim B \supset P}{\{\ P\ \}\, repeat\ \ S\ until\ B\ \{\ \ Q \,\&\, B\ \}}$$

7. ბოლოს კიდევ ერთი საჭირო წესი

$$\frac{\{\ P1\}\ S\{Q\}, \{\ P2\ \}S\{Q\}}{\{\ P1 \vee\ \ P2\ \}\ S\{Q\}}$$

ვერიფიკაციის წესების გამოყენებით, დავამტკიცოთ ევკლიდეს ალგორითმის შესაბამისი პროგრამის კორექტულობა.

. . .

```
int a,b,x,y;
cin>>a>>b;
while(a!=b)                    ( 2 )
      if(a>b)  a-=b;
    else b-=a;
cout<<" usg="<<a;
```

. . .

$P \cong \{\ a \geq 0 \,\&\, b \geq 0 \,\&\, a \neq b\}$ წარმოადგენს *(2)* პროგრამის წინაპირობას, ხოლო $Q \cong \{\ a \geq 0 \,\&\, b \geq 0 \,\&\, a = b\}$ შემდეგ პირობას.

ამ აღნიშვნებისს გათვალისწინებით *(2)* პროგრამა გადავწეროთ ცოტა სხვა სახით:

// $\{\ a \geq 0 \,\&\, b \geq 0 \,\&\, a \neq b\}$ P წინაპირობა

```
while (a!=b)
if(a>b)  a-=b;
    else b-=a;
```

S პროგრამა (3)

// $\{ a \geq 0 \& b \geq 0 \& a = b \}$ *Q შემდეგი პირობა*

უნდა დავამტკიცოთ *S* პროგრამის სამართლიანობა.

ავღნიშნოთ

$P_1 \cong \{ a \geq 0 \& b \geq 0 \}$

S1 ≅ if(a>b) a-=b; (4)
else b-=a;

$Q_1 \cong \{a \neq b\}$

თუ მოვახერხებთ *(4)* პროგრამის სამართლიანობის დამტკიცებას, მაშინ ვერიფიკაციის მე-5 წესის თანახმად გამოვა, რომ *(3)* სამართლიანია, რაც ფაქტიურად ნიშნავს რომ ჩვენს მიერ მოყვანილი ევკლიდეს ალგორითმის შესაბამისი პროგრამა სამართლიანია.

ე.ი. დაგვრჩა დავამტკიცოთ *S* -ის სამართლიანობა. შემოვიღოთ ახალი აღნიშვნები:

$P_2 \cong \{ a \geq 0 \& b \geq 0 \}$

S2 ≅ a-=b;

S3 ≅ b-=a;

Q2 ≅ {a>b}

ვერიფიკაციის 3.1 წესის თანახმად

$\{ a \geq 0 \& b \geq 0 \}$

if B then S1

else S2;

$\{ a \geq 0 \& b \geq 0 \}$

აქედან გამომდინარე (3) პროგრამა სამართლიანია.

1.1.1 არარეკურსიული და რეკურსიული ფუნქციებისათვის ვერიფიკაციის წესი

ვთქვათ მოცემულია ფუნქციის აღწერა

function F(L) : T

S;

სადაც F ფუნქციის სახელია, L ფორმალური პარამეტრების სპეციფიკაცია, T ფუნქციის დასაბრუნებელი ტიპი, ხოლო S ფუნქციის ტანი.

დავუშვათ S–ისთვის დამტკიცებულია

$$\{P\}\ S\ \{Q\}$$

ასევე ვთქვათ X არის F ფუნქციის ფორმალური პარამეტრების სია. ამ აღნიშვნებიდან გამომდინარე ფუნქციის ვერიფიკაციის წესს აქვს სახე:

$$\frac{\{P\}\ S\ \{Q\}}{\forall X\ \left(\ P \supset Q\ {}^{f}_{f(x)}\right)}$$

ახლა განვიხილოთ რეკურსიული ფუნქციების საკითხი. ვთქვათ მოცემული გვაქვს ფაქტორიალის დასათვლელი ფუნქცია:

```
int  fact ( int n )
{ if ( n==0 ) return 1;
        return  n * fact( n-1 );}
```

იმისათვის რომ დავამტკიცოთ, *fact* ფუნქცია აკმაყოფილებს სასურველ პირობას, საჭიროა დავამტკიცოთ, რომ ამავე პირობას აკმაყოფილებს ფუნქციის ტანი. ამრიგად, ფორმალურად იმისათვის რომ დავამტკიცოთ

$$\{\ n \geq 0\ \}\ \ fact\ (n)\ \{\ \ fact = n!\ \}\quad (1)$$

სამართლიანია ნებისმიერი n–თვის, საჭიროა დავამტკიცოთ, რომ

$$\{\ n \geq 0\ \}\ \ S_{fact}\ \ \{\ \ fact = n!\ \}\ \ (2)$$

რადგან ფუნქცია რეკურსიულია, ამიტომ მისი ტანი ი მოიცავს *fact(n-1)* გამოძახებას. სწორი სტრატეგია მდგომარეობს იმაში, რომ ვივარაუდოთ: ყველა შიდა გამოძახება აკმაყოფილებს *(2)*, ხოლო შემდეგ დავამტკიცოთ, რეკურსიული ფუნქციის შესაბამისი ვერიფიკაციის წესი:

$$\frac{\forall X\ (\{\ P\ \}\ F(X)\ \{\ Q\ \})\ \ |-\ \{\ P\ \}\ S\ \{\ Q\ \}}{\forall X\ (\{\ P\ \}\ F(X)\ \{\ Q\ \})}\ \ (3)$$

სადაც S რეკურსიული F ფუნქციის ტანია.

(3) –დან გამომდინარეობს

$$\frac{\forall X\ (\{\ P\ \}\ F(X)\ \{\ Q\ \})}{\{\ P_a^x\ \}\ F(a)\ \{\ Q_a^x\ \}}\ \ (4)$$

(3) დამტკიცება ხდება შემდეგნაირად:

წინაპირობაა

$$1.\ \forall X\ (\{\ P\ \}\ F(X)\ \{\ Q\ \})\ \ |-\ \{\ P\ \}\ S\ \{\ Q\ \}$$

უნდა დავამტკიცოთ, რომ

$$2.\ \forall X\ (\{\ P\ \}\ F(X)\ \{\ Q\ \})$$

დამტკიცება ხდება ინდუქციის მეთოდით, რეკურსიის იმ დონეების მიხედვით, რომლებიც ჩნდებიან *F(X)* ფუნქციის გამოძახებისას.

საბაზისო ნაბიჯი: დავუშვათ ვდგევართ რეკურსიის დასრულების დონეზე, ე.ი. ფუნქციის ახალი გამოძახება არ ხდება, ე.ი. *F (X)* ფუნქციის გამოძახებისას არ ხდება მისი ტანის გააქტიურება პარამეტრების ახალი მნიშვნელობისათვის. მაშინ თეორემის წინაპირობიდან გამომდინარეობს, რომ *F* ფუნქციის გამოძახება აკმაყოფილებს

$$\forall X\ (\{\ P\ \}\ F(X)\ \{\ Q\ \})$$

ამიტომ ამ შემთხვევაში თეორემა სამართლიანია.

ინდუქციის ნაბიჯი: ახლა დავუშვათ, რომ მე-2 სამართლიანია, თუ *F* ფუნქციის გამოძახებას მივყავართ არა უმეტეს *n* დონიან რეკურსიასთან, ანუ *S* –ში ყოველი შიდა რეკურსიულ გამოძახებას მივყავართ არა უმეტეს *n* რაოდენობა *F* ფუნქციის გამოძახებასთან (ინდუქციის ჰიპოთეზა), მაშინ *1*-ლი წინაპირობა საშუალებას გვაძლევს დავასკვნათ, რომ სამართლიანია

$$\{P\}\ \ S\ \ \{Q\}$$

და ფუნქციის ვერიფიკაციის წესის მიხედვით სამართლიანია 2, სადაც *F(X)* ფუნქციას მივყავართ რეკურსიის *n+1* დონემდე. ამრიგად ინდუქციური ნაბიჯი დამტკიცებულია.

1.1.2 პროგრამების შექმნა, მათი კორექტულობის დამტკიცების პარალელურად

როგორც 1.2-ში იყო აღნიშნული რთული ამოცანის განხილვა კონკრეტული დაპროგრამების ენის თვალსაზრისით არ ღირს. დაპროგრამების ყოველ ენას გააჩნია თავისებურებები, რომელთა გათვალისწინებამ შეიძლება ძირითადი პრობლემისაგან დააშოროს პროგრამისტი. ამიტომ, ბუნებრივია პირველად ყურადღება კონცენტრირებული უნდა იყოს ე.წ. აბსტრაქტულ ალგორითმზე.

აბსტრაქტული ალგორითმის შემუშავებისას, პროგრამის შექმნის პროცესი წარმოადგენს ეტაპობრივი დაზუსტების შედეგს. ყოველ ეტაპზე(ნაბიჯზე)

აბსტრაქტული ოპერატორებისა და მონაცემების დეკომპოზიცია ხორციელდება ისე, რომ მიღებული შედეგი გამოისახება ცნებებში, რომლებიც ახლოს არის არჩეული დაპროგრამების ენის საშუალებებთან.

ამგვარად, შეიძლება ითქვას, რომ ყოველ დაზუსტებას მივყავართ უფრო დაბალი აბსტრაქციის პროგრამამდე. ეს პროცესი გრძელდება მანამ სანამ მთელი პროგრამა არ იქნება წარმოდგენილი კონკრეტული დაპროგრამების ენაზე, ანუ სანამ არ დაიწერება პროგრამა კონკრეტული პროგრამირების ენაზე.

პროგრამის ეტაპობრივი დაზუსტების გზით შექმნის ერთ-ერთი უპირატესობა მდგომარეობს იმაში, რომ შედეგად ვღებულობთ პროგრამას მოდულირების უმაღლესი ხარისხით. ეს ნიშნავს იმას, რომ მთელი პროგრამა ჩაიწერება შედარებით ნაკლები აბსტრაქტული ცნებების (მოდულების) ტერმინებში და ეს მოდულები პროგრამაში აღმოცენდება სხვადასხვა კონტექსტურ სიტუაციებში.

პროგრამების ეტაპობრივი დაზუსტების მეორე მნიშვნელოვანი უპირატესობა მდგომარეობს იმაში, რომ ის შეიძლება პროგრამის კორექტულობის დამტკიცებასთან პარალელურად მიმდინარეობდეს. სავსებით ნათელია, რომ დიდი პროგრამებისათვის გარკვეული ხარისხით საჭიროა კორექტულობის დადგენა. ამ პრობლემის ერთ-ერთი გადაწყვეტაა პროგრამის ერთობლივი ეტაპობრივი შექმნა და კორექტულობის დამტკიცება.

აბსტრაქციის ყოველ ეტაპზე, მისი კორექტულობის დამტკიცება ხდება ისე, რომ აბსტრაქციის უფრო დაბალ დონეზე დაზუსტებისას კორექტულობის სასურველი კრიტერიუმის შენარჩუნება გარანტირებულია. ჩვენ ვიღებთ პროგრამის პირველ, ყველაზე უფრო აბსტრაქტულ ვერსიას და ვამტკიცებთ მის კორექტულობას, იმის გათვალიწინებით, რომ აბსტრაქტული ოპერატორები და ოპერაციები, რომლის ტერმინებშიც დაწერილია პროგრამა, ზუსტდება კორექტულად და თანდათანობით.

იმდენად რამდენადაც აბსტრაქტული პროგრამები, ბუნებრივია, უფრო კომპაქტურია, ვიდრე რეალური პროგრამები, ამიტომ აბსტრაქტული პროგრამების კორექტულობის დადგენა უფრო მარტივი და ნაკლებად დამღლელი პროცესია.

1.2 ფლოიდის ინდუქციურ დაშვებათა მეთოდი

განმარტება 1.2.1 ვთქვათ A – რაიმე დამოკიდებულებაა, რომელიც აღწერს მონაცემთა მოსალოდნელ თვისებებს პროგრამაში, ხოლო C – დამოკიდებულება აღწერს იმას, რაც მოლოდინების მიხედვით უნდა მივიღოთ პროგრამის მუშაობის შედეგად. ვიტყვით, რომ პროგრამა ნაწილობრივ კორექტულია (ჭეშმარიტია), თუ მისი ყოველი შესრულებისას იმ მონაცემებზე, რომლებიც აკმაყოფილებს A დამოკიდებულებას, C დამოკიდებულება იქნება სამართლიანი, იმ პირობით რომ პროგრამა დასრულდება. სხვა სიტყვებით, თუ შევასრულებთ პროგრამას იმ მონაცემებზე, რომლებიც აკმაყოფილებს A დამოკიდებულებას, ის ან არ დასრულდება, ან დასრულდება და საშედეგო მონაცემები დააკმაყოფილებს C დამოკიდებულებას.

ამრიგად, პროგრამა შეიძლება იყოს ნაწილობრივ კორექტული მაშინაც კი, თუ იგი რომელიმე საწყისი მონაცემებისათვის შეიძლება არც კი დასრულდეს. მხოლოდ აუცილებელია, თუ პროგრამა დასრულდება, მაშინ აუცილებლად სამართლიანი უნდა იყოს C დამოკიდებულება იმ მონაცემებზე, რომლებიც აკმაყოფილებენ A დამოკიდებულებას.

განმარტება 1.2.2. პროგრამას ეწოდება სრულად სამართლიანი (კორექტული, A და C დამოკიდებულების მიმართ), თუ ის ნაწილობრივ სამართლიანია A და C დამოკიდებულების მიმართ და აუცილებლად ასრულებს მუშაობას ყველა იმ მონაცემებისათვის, რომლებიც აკმაყოფილებს A დამოკიდებულებას.

პროგრამის ნაწილობრივი სამართლიანობის დასამტკიცებლად, საჭიროა მოვიქცეთ შემდეგნაირად: დავუკავშიროთ A დამოკიდებულება პროგრამის საწყისს წერტილს, ხოლო C დამოკიდებულება კი პროგრამის საბოლოო წერტილს. ამასთან ერთად საჭიროა გამოვავლინოთ კანონზომიერება, რომელიც დაკავშირებულია პროგრამის ცვლადების მნიშვნელობებთან და დავაკავშიროთ შესაბამისი დამოკიდებულებები პროგრამის დანარჩენ წერტილებთან. უკიდურეს შემთხვევაში დავაკავშიროთ ასეთი დამოკიდებულებები პროგრამის ციკლებთან. პროგრამაში ყოველი ფრაგმენტისათვის, რომელიც იწყება i წერტილიდან (ამ წერტილთან დაკავშირებულია A_i დამოკიდებულება) და გრძელდება j წერტილამდე (ამ წერტილთან დაკავშირებულია A_j დამოკიდებულება), დავამტკიცებთ, რომ თუ

მოვხვდით *i* წერტილში და სამართლიანია A_i დამოკიდებულება, შემდეგ გავიარეთ რა გზა *i* წერტილიდან *j* წერტილამდე, *j* წერტილში მოხვედრისას A_j დამოკიდებულება აუცილებლად იქნება სამართლიანი. ციკლებისათვის *i* და *j* შეიძლება იყოს ერთი და იგივე წერტილი.

ის, რომ ზემოთ მოყვანილი წესით ნამდვილად შეიძლება დამტკიცდეს პროგრამის ნაწილობრივი კორექტულობა, საჭიროა შემდეგი თეორემის დამტკიცება.

თეორემა 1.2.1 თუ, რომელიმე პროგრამისათვის, შეიძლება შევასრულოთ ყველა მოქმედება, რაც გათვალისწინებულია ინდუქციური დაშვების მეთოდის მიხედვით, მაშინ ეს პროგრამა ნაწილობრივ სამართლიანია (*A* და *C* დამოკიდებულების მიმართ).

დამტკიცება. დავუშვათ, რომ დავამტკიცეთ პროგრამის კორექტულობა ინდუქციური დაშვების მეთოდით. დავიწყოთ პროგრამის შესრულება საწყისი მონაცემებისათვის(რომლებიც თავის მხრივ აკმაყოფილებენ *A* დამოკიდებულებას). ჩვენ უნდა დავამტკიცოთ, რომ როცა პროგრამის შესრულება დასრულდება, მაშინ *C* დამოკიდებულება იქნება სამართლიანი. ჩვენ ვაჩვენებთ, რომ ყოველთვის, როცა მოვხვდებით პროგრამის რომელიმე წერტილში, მაშინ მასთან დაკავშირებული დამოკიდებულება აუცილებლად სამართლიანი იქნება. ეს კი ნიშნავს, რომ თუ ჩვენ მივაღწევთ პროგრამის ბოლო წერტილს(ანუ პროგრამამ უკვე დაასრულა მუშაობა), მაშინ *C* დამოკიდებულება აუცილებლად იქნება სამართლიანი. დამტკიცება მოვახდინოთ ინდუქციის მეთოდით პროგრამის *n* წერტილის გავლით.

1. დავუშვათ, რომ პროგრამის შესრულების პროცესში, მოვხვდით პროგრამის პირველ წერტილში *(n=1).* საჭიროა ვაჩვენოთ, რომ ამ წერტილთან დაკავშირებული დამოკიდებულება სამართლიანია. მაგრამ პირველი წერტილი ემთხვევა პროგრამის საწყისი წერტილის და მასთან დაკავშირებულია საწყისი *A* დამოკიდებულება. ჩვენ კი ვიცით, რომ ის ჭეშმარიტია, რადგან ლაპარაკია იმ პროგრამის შესრულებაზე, ისეთი საწყისი მონაცემებისათვის, რომლისთვისაც ის სრულდება.

2. დავუშვათ, რომ ჩვენ მივაღწიეთ პროგრამის რომელიღაც *n*-ურ წერტილამდე, რომელთანაც დაკავშირებულია A_{in} დამოკიდებულება. დავუშვათ, რომ A_{in} ჭეშმარიტია (ინდუქციის ჰიპოტეზის მიხედვით). საჭიროა დავამტკიცოთ, რომ თუ პროგრამის შესრულება გრძელდება და ჩვენ მოვხვდებით *n*-ური წერტილიდან *(n+1)* წერტილში, მაშინ A_{in+1} დამოკიდებულება იქნება სამართლიანი. ცხადია, რომ *n*-ური და *(n+1)*

წერტილებს შორის არსებობს გარკვეული გზა. იმდენაც, რამდენადაც ლაპარაკია ინდუქციურ დაშვებათა მეთოდზე, მაშინ ბუნებრივია ჩვენ განვიხილეთ ეს გზა და დავამტკიცეთ, რომ თუ ვართ *n*-ურ წერტილში და შესაბამისად A_{in} ჭეშმარიტია, ხოლო შემდეგ გადავედით *n*-დან *(n+1)* წერტილში, ან წერტილში A_{in+1} დამოკიდებულება აუცილებლად სამართლიანი იქნება. ამ ფაქტიდან და იქედან, რომ A_{in} სამართლიანია, გამომდინარეობს, რომ *n+1* წერტილში მოხვედრისას A_{in+1} დამოკიდებულება სამართლიანია. ეს სწორედ ისაა, რისი დამტკიცებაც გვსურდა.

ამრიგად, ჩვენ ინდუქციით დავამტკიცეთ, რომ თუ პროგრამა აკმაყოფილებს ინდუქციური დაშვების მეთოდის პირობებს, მაშინ პროგრამის ნებისმიერი წერტილიდან მოხვედრისას მასთან დაკავშირებული შესაბამისი დამოკიდებულება სამართლიანია. აქედან გამომდინარე, თუ პროგრამის ბოლო წერტილამდე მივედით, მაშინ სამართლიანი იქნება ამ წერტილთან დაკავშირებული დამოკიდებულება. ე.ი. განხილული პროგრამა ნაწილობრივ ჭეშმარიტია.

პროგრამის სრული ჭეშმარიტების დასამტკიცებლად, პირველად საჭიროა ინდუქციური დაშვების მეთოდის დახმარებით მისი ნაწილობრივი ჭეშმარიტების დამტკიცება, ხოლო შემდეგ იმის დამტკიცებაა საჭირო, რომ პროგრამა ოდესმე აუცილებლად დასრულდება.

მაგალითი. განვიხილოთ მთელი რიცხვის განაყოფის მთელი ნაწილისა და ნაშთის პოვნის პროგრამა და ვეცადოთ მისი სამართლიანობის დამტკიცება.

```
main()
    { int x, y, k, q;
      cin>>x>>y;
      k=0;
     q=x;
     while ( q >= y )
     {k++;
      q-=y;}
     cout<<"mteli="<<k<<" nashti="<<q;
           }
```

გადავწეროთ ეს პროგრამა სტრუქტურული ელემენტის (ციკლის) გარეშე

```
main()
      { int x, y, k, q ;
        cin>>x>>y;      //x>=0, y>=1        -> A1
        k = 0;
        q = x ;
2 :     if ( q < y )  goto 4 ;
      // x == k*y+q  &&  q >= 0            -> A2
        k++;
        q -= y;
        goto 2 ;
4 :     cout<<"mteli="<<k<<" nasti="<<q;
        // x == k*y  &&  k >= 0  &&  q <= y       -> A3
            }
```

ამ პროგრამაში განვიხილოთ შემდეგი ტრაექტორიები :

გზა საწყისი მონაცემების წაკითხვიდან *if* ოპერატორამდე. დავუშვათ, რომ წაკითხვა უკვე შესრულდა და სამართლიანია ის მტკიცებულება, რომელიც კომენტარის სახით არის მიწერილი. ამის შემდეგ თანმიმდევრობით სრულდება ოპერატორები *if* ოპერატორამდე. საჭიროა დავამტკიცოთ, რომ სამართლიანია მტკიცებულება:

$x == k*y+q \;\&\&\; q >= 0$

თუ ჩვენ მივედით ამ წერტილამდე, გვაქვს $k = 0;\ q = x,\ 0 <= x \;\&\&\; 1 <= y$

ამრიგად

$x = k*y + q = 0*y + x \;\&\&\; 0 <= x = q$

აქედან გამომდინარე A_2 დამოკიდებულება ჭეშმარიტია.

(პროგრამის ძირითადი ციკლი). დავუშვათ, რომ ჩვენ *if* ოპერატორს ვასრულებთ პირველად და სამართლიანია ლოგიკური გამოსახულება $x == k*y+q \;\&\&\; q >= 0,$ შემდეგ შესრულდება ციკლი და ისევ ვბრუნდებით *if* ოპერატორთან. აუცილებელია დავამტკიცოთ, რომ ზემოთ ხსენებული ლოგიკური გამოსახულება ისევ ჭეშმარიტია. დავუშვათ k და q ციკლის შესრულებამდე ღებულობდნენ მნიშვნელობა k_n და q_n -ს. მაშინ $x == k_n*y+ q_n \;\&\&\; q_n >= 0.$ *if* ოპერატორთან დაბრუნებისას, ციკლის გავლის შემდეგ $k_{n+1}= k_n+1$ da $q_{n+1}= q_n-y$, ხოლო x და y უცვლელი რჩება. ამრიგად,

$k_{n+1} * y + q_{n+1} = (k_n + 1) * y + (q_n - y) = k_n * y + y + q_n - y = k_n * y + q_n = x$

ამის გარდა, ცნობილია, რომ თუ ვმოძრაობთ ციკლზე $q_n <= y$ პირობა იყო მცდარი, ე.ი. q > y. აქედან გამომდინარეობს, რომ *if* ოპერატორთან დაბრუნებისას $0 <= q_n - y = q_{n+1}$.

გზა *if* ოპერატორიდან მონაცემების გამოტანამდე.

დავუშვათ, რომ ჩვენ შევასრულეთ *if* ოპერატორი, სამართლიანია შესაბამისი მტკიცებულება და გადავდივართ მონაცემების გამოტანაზე. საჭიროა დავამტკიცოთ, რომ სამართლიანია შემდეგი ლოგიკური გამოსახულება

$x == k*y + q \;\&\&\; k >= 0 \;\&\&\; q<=y.$

if ოპერატორიდან მოვხვდებით მონაცემების გამოტანაზე, თუ $q < y$. ამ გადასვლის დროს არც ერთი ცვლადის მნიშვნელობა არ იცვლება და შედეგად გამოტანის წერტილიდან მოსვლისას გვაქვს

$x = k*y + q \;,\;\; 0 <= q \;\&\&\; q<=y.$

ამრიგად, ჩვენ დავამტკიცეთ ამ პროგრამის ნაწილობრივი კორექტულობა.

ახლა დაგვრჩა დავამტკიცოთ, რომ პროგრამა დროში სასრულია, ანუ ადრე თუ გვიან იგი დაასრულებს მუშაობას. ჩვენი პროგრამა მუშაობას დაასრულებს, მაშინ როცა ბოლოს და ბოლოს ციკლი დასრულდება. აქედან გამომდინარე საჭიროა ვაჩვენოთ, რომ პირობა $q < y$ (რომელიც ფაქტიურად ციკლის დასრულების პირობას წარმოადგენს) გახდება ჭეშმარიტი.

იმდენად, რამდენადაც *q* ცვლადის მნიშვნელობა ციკლის ყოველი გავლის შემდეგ მცირდება *y*-ით, ხოლო თვითონ *y* რჩება უცვლელი და თან $1 <= y$, ამიტომაც შეიძლება დავასკვნათ, რომ *q* ცვლადის მნიშვნელობა ციკლის ყოველი გავლისას, უკიდურეს შემთხვევაში, *1*-ით მაინც მცირდება და ოდესმე გახდება *y*-ზე ნაკლები. ამ მომენტში $q <= y$ და რა თქმა უნდა ციკლი შეწყდება და შესაბამისად პროგრამაც დაასრულებს მუშაობას.

აქედან გამომდინარე, ჩვენი პროგრამა სრულად კორექტულია.

1.2.1 რეკურსიული პროგრამების სამართლიანობის დამტკიცება

ახლა ვნახოთ, ინდუქციურ დაშვებათა მეთოდი როგორ გაართმევს თავს რეკურსიული პროგრამების კორექტულობის დამტკიცების პრობლემას.

რეკურსიული პროგრამები, როგორც წესი აგებულია შემდეგი წესით: პირველად ცხადად განისაზღვრება შედეგის გამოთვლის წესი რეკურსიული ფუნქციის პარამეტრების უმარტივესი მნიშვნელობებისათვის, შემდეგ განისაზღვრება შედეგის გამოთვლის წესი, პარამეტრების უფრო რთული მნიშვნელობებისათვის და ამ დროს გამოიყენება ფუნქციის მუშაობის წინა შედეგი.

სავსებით ლოგიკურია რეკურსიული პროგრამების, თუ ფუნქციების კორექტულობის დასამტკიცებლად მოვიქცეთ შემდეგნაირად:

1. დავამტკიცოთ, რომ პროგრამა კორექტულად მუშაობს უმარტივესი არგუმენტისათვის.

2 დავამტკიცოთ, რომ პროგრამა სწორად მუშაობს უფრო რთული არგუმენტებისათვის, იმ ვარაუდით, რომ ის სწორად მუშაობს მარტივი არგუმენტებისათვის.

ყურადსაღებია ის ფაქტი, რომ 1-ლი და მე-2 ეტაპი ფაქტიურად წარმოადგენს ინდუქციის მეთოდით დამტკიცებას, მხოლოდ ამ შემთხვევაში ინდუქცია უნდა განხორციელდეს იმ მონაცემთა სტრუქტურის მიმართ, რომელსაც ამუშავებს ეს რეკურსიული ფუნქცია. სწორედ ამიტომაც ამ მეთოდს ზოგჯერ სტრუქტურული ინდუქციის მეთოდსაც უწოდებენ.

მაგალითი

```
int   Fact( int x )
    { if ( x == 1 )  return 1;
                  else return x * Fact( x-1 );
                        }
```

დავუშვათ, რომ ეს ფუნქცია ითვლის ფაქტორიალს. საჭიროა დავამტკიცოთ, რომ:

$Fact = 1 * 2 * 3 * \ldots * (n-1) * n = n!$

ნებისმიერი დადებითი n-ისათვის.

სტრუქტურული ინდუქციით, ზემოთ მოყვანილი, ფუნქციის კორექტულობის დასამტკიცებლად გამოვიყენებთ მარტივ ინდუქციას:

1. დავამტკიცოთ, რომ $Fact = 1!$, მართლაც $Fact(1) = 1 = 1!$
2. დავამტკიცოთ, რომ თუ

$$Fact(n) = 1 * 2 * 3 * \ldots * (n-1) * n = n!$$

მაშინ

$$Fact(n+1) = 1 * 2 * 3* . . . * (n-1) * n * (n+1) = (n+1)!$$

სამართლიანი იქნება.

რა თქმა უნდა, ჩვენ ვთვლით, რომ n დადებითია და $Fact(n) = n!$ – ინდუქციის ჰიპოთეზაა. რადგან n დადებითი რიცხვია, ამიტომ პირობა $n+1==n$ მცდარია და გავყვებით რა ფუნქციის ტანს, მივიღებთ

$$Fact(n +1) = (n+1) * Fact((n+1)-1) = (n+1) * Fact(n) = (n+1) * (n!) = (n+1) * (1 * 2 * 3* . . .* n) = 1 * 2 * 3* . . .* n * (n+1) = (n+1)!$$

ინდუქციის ჰიპოტეზა

რისი დამტკიცებაც იყო საჭირო, ე.ი.

$$Fact(n) = n!$$

ნებისმიერი დადებითი n-ისათვის.

1.3 მოდელებზე შემოწმება (Model cheking)

როგორც შესავალში აღვნიშნეთ, მოდელებზე შემოწმება წარმოადგენს სასრული რაოდენობა მდგომარეობების მქონე პარალელური სისტემების ავტომატურ ვერიფიცირების მეთოდს. ეს მეთოდი ფლობს მთელ რიგ უპირატესობას ვერიფიკაციის ამოცანისადმი ტრადიციულ მიდგომებთან შედარებით, რომლებიც დაფუძნებულია მოდელირებაზე, ტესატირებაზე და დედუქციურ ანალიზზე. ყველაზე მნიშვნელოვანია ის ფაქტი, რომ ეს მეთოდი არის სრულად ავტომატური: ის ან ამთავრებს თავის მუშაობას პასუხით *true*, რაც ნიშნავს რომ მოდელი სრულად შეესაბამება დასაპროექტებელ სისტემას. ან აგებს რაიმე მაგალითს, კონტრმაგალითის როლში, სადაც მიუთითებს, თუ რატომ არ მუშაობს მოდელი (შესაბამისი ფორმულა). მოდელებზე შემოწმების მეთოდი წარმატებით იყო გამოყენებული მიმდევრობითი ელექტრონული სქემებისა და რთული საკომუნიკაციო ე.წ. პროტოკოლების (ოქმების) გამართულობის შესამოწმებლად. ძირითადი სირთულე, რასაც ვაწყდებით მოდელებზე შემოწმების პროცესის დროს, არის ე. წ. "კომბინატორული აფეთქების" ეფექტი, მდგომარეობათა სივრცეში. ეს პრობლემა წარმოიქმნება ისეთ სისტემებში, რომლებიც შედგება ერთმანეთთან ურთიერთმოქმედ ბევრი კომპონენტისაგან, ასევე ისეთ სისტემებში, რომლებიც მუშაობენ დიდი სიდიდის მნიშვნელობათა მქონე მონაცემთა სტრუქტურებთან (მაგალითად ასეთი შეიძლება იყოს მონაცემთა გადაცემის მარშრუტები ლოგიკურ სქემებში). ასეთ შემთხვევაში სისტემის მდგომარეობათა რაოდენობა შეიძლება აღმოჩნდეს უზარმაზარი, თუმცა მაინც სასრული.

1.3.1 პროგრამების მოდელების ვერიფიკაციის პროცესი

Model cheking მეთოდის გამოყენება შედგება რამდენიმე ეტაპისაგან: მოდელირება, სპეციფიკაცია და ბოლოს ვერიფიკაცია.

მოდელირება

პირველი ამოცანა მდგომარეობს იმაში, რომ დასაპროექტებელი სისტემა უნდა მივიყვანოთ ისეთ ფორმალურ სახემდე, რომელიც მისაღები იქნება მოდელებზე შემოწმების ვერიფიკაციის ინსტრუმენტისათვის. მოდელის როლში გამოიყენება ე.წ. კრიპკეს სტრუქტურა.

კრიპკეს[11] მოდელი(*Kripke structure*) თავისთავად არის არადეტერმინირებული სასრულ ავტომატი. მოდელი წარმოდგინდება ორიენტირებული გრაფის სახით, რომლის თითოეული წვერო აღწერს სისტემის მიღწევად მდგომარეობებს, ხოლო წიბოები კი გადასვლებს ერთი მდგომარეობიდან მეორეში. თითოეულ წვეროსთან მიმაგრებულია თვისებათა (სპეციფიკაციების) სიმრავლე, რომელიც სრულდება სისტემისათვის შესაბამისს მდგომარეობაში. ამისათვის განიმარტება ე.წ. დაჭდევების (მონიშვნათა) ფუნქცია.

კრიპკეს სტრუქტურა ფორმალურად აღიწერება შემდეგნაირად:

ვთქვათ AP არის ატომარული გამონათქვამების (პრედიკატების) სიმრავლე. კრიპკეს მოდელი[5] ეწოდება $M = (S, I, R, L)$ ოთხეულს, რომელიც შედგება:

- მდგომარეობათა სასრული სიმრავლე S;
- საწყის მდგომარეობათა სიმრავლე $I \subseteq S$;
- გადასვლათა სპეციფიკაცია $R \subseteq S \times S$, სადაც $\forall s \in S, \exists s' \in S$, ისეთი, რომ $(s, s') \in R$;
- დაჭდევების (მონიშვნათა) ფუნქცია $L : S \to 2^{AP}$;

პირობა, რომელიც R გადასვლათა სპეციფიკაციაზეა დადებული, უზრუნველყოფს იმას, რომ ყოველ მდგომარეობას, მოსდევს შემდეგი მდგომარეობა. თუ

[11] კრიპკეს მოდელი(ინგ. Kripke structure) --- წარმოადგენს არადეტერმინირებული სასრული ავტომატის ერთერთ სახეობას, რომელიც წარმოდგენილი იყო სოლომ კრიპკეს მიერ. კრიპკეს მოდელი წარმოადგენს მარტივ აბსტრაქტულ მანქანას, რომელიც საშუალებას იძლევა გამოთვლითი მანქანის იდეების განხორციელებას რაიმე განსაკუთრებული გართულებების გარეშე.

სოლ აარონ კრიპკე (ინგ. Saul Aaron Kripke, დაიბადა 1940 წლის 13 ნოემბერს) — ამერიკელი ფილოსოფოსი და ლოგიკის პროფესორი. არის ჰარვარდის უნივერსიტეტის საპატიო პროფესორი, ამჟამად მუშაობს ნიუ–იორკის საქალაქო უნივერსიტეტის უმაღლესი სკოლასა და საუნივერსიტეტო ცენტრში საპატიო პროფესორად.

საჭიროა ურთიერთბლოკირების ემულაცია, კრიპკეს სტრუქტურაში უბრალოდ უნდა დაემატოს კიდევ ერთი წიბო, იმ $s \in S$ მდგომარეობიდან, რომელიც აკეთებს ბლოკირებას თავის თავში.

მონიშვნათა L ფუნქცია, ყოველი $s \in S$ მდგომარეობისათვის განსაზღვრავს $L(s)$ სიმრავლეს, რომელიც თავის მხრივ წარმოადგენს s მდგომარეომაში სამართლიან ყველა ატომარულ მტკიცებულებათა სიმრავლეს.

სპეციფიკაცია

ვერიფიკაციის პროცესის დაწყებამდე საჭიროა ყველა იმ არსებითი თვისებების (სპეციფიკაციების) ფორმულირება, რომლებსაც უნდა ფლობდეს დასაპროექტებელი სისტემა. როგორც წესი, სპეციფიკაცია განიმარტება ფორმალური ლოგიკის ენის დახმარებით. მაგრამ ტრადიციული ლოგიკის საშუალებები ამ შემთხვევაში არასაკმარისია, ამიტომაც გამოიყენება ე.წ. დროითი, ანუ ტემპორალური ლოგიკა, რომელიც საშუალებას იძლევა აღიწეროს, თუ როგორ იცვლება დროში სისტემის მდგომარეობა.

განმარტება: დროითი ლოგიკა არის ისეთი ლოგიკა, სადაც ლოგიკური ფორმულების ჭეშმარიტება დამოკიდებულია დროის იმ მომენტზე, რომელშიც სრულდება ეს ფორმულები.

დროით ლოგიკებში არსებობს ორი სახის ოპერატორი : ლოგიკური და მოდალური.

ლოგიკური ოპერატორების როლში, როგორც წესი გამოიყენება კლასიკური ლოგიკის $(\neg, \wedge, \vee, \rightarrow)$ ოპერატორები.

დროითი ლოგიკის ერთ-ერთი წინამორდები გახლდათ ე.წ. ტენზორული ლოგიკა(*Tense Logic)*, რომელიც შემუშავებული იყო ინგლისელი მეცნიერის არტურ პრიორის*(Arthur Pryor)*[12] მიერ. სწორედ მან შემოიტანა ორი მოდალური ოპერატორი: F (*Future)* – "ოდესმე, მომავალში, იქნება ჭეშმარიტი" და P *(Past)* – "ოდესღაც , წარსულში, იყო ჭეშმარიტი".

[12] Arthur Norman Prior (დაიბადა 1914წელს მასტერტონში, ახალი ზელანდია, გარდაიცვალა 1969წელს ტრონდაიმში, ნორვეგია) – ფილოსოფოსი და ლოგიკის სპეციალისტი. ტენზორული ლოგიკის ფუძემდებელი (1957), რომელიც დღეს ცნობილია როგორც ტემპორალური ლოგიკა.

ტენზორულ ლოგიკაში ასევე შემოღებული იყო ორი დუალური ოპერატორი: *G(Globally)* და *H(History)*, რომლებისთვისაც განმარტებულია შემდეგი ექვივალენტობა:

$$Gq \equiv \neg F \neg q, \quad Hq \equiv \neg P \neg q$$

სადაც q რაიმე ლოგიკური გამონათქვამია. ამ გამონათქვამების სამართლიანობა ცხადია. მაგალითად, პირველი ნიშნავს: "მომავალში q ყოველთვის ჭეშმარიტია, ეს იგივეა ვამტკიცოთ, რომ არასწორია – ოდესმე q მომავალში გახდება მცდარი".

F და P ტემპორალური ოპერატორების დახმარებით შეიძლება განვმარტოთ უფრო რთული, დროზე დამოკიდებული თვისებები, მაგალითად q ლოგიკური ფორმულა იქნება ჭეშმარიტი:

- ყოველთვის მომავალში Gq
- ერთხელ მაინც მომავალში Fq
- არასდროს მომავალში $\neg Fq$
- უამრავჯერ მომავალში GFq
- მუდმივად, რაღაცა მომენტიდან დაწყებული FGq

ტემპორალურ ლოგიკაში განმარტებულია კიდევ ორი ოპერატორი: *X(NextTime)* და *U(Until).*

X(NextTime) ოპერატორის სემანტიკა: Xq ჭეშმარიტია დროის t მომენტში, თუ იგი ჭეშმარიტია დროის მომდევნო $t+1$ მომენტში.

U(Until) ოპერატორი ბინარულ ოპერატორს წარმოადგენს და მისი სემანტიკა ასეთია: pUq ჭეშმარიტია დროის t მომენტში, თუ q ჭეშმარიტია მომავალში, რომელიმე დროის მომენტში, ხოლო დროის $[t, t')$ მონაკვეთში ჭეშმარიტია p.

Model cheking-ში განიხილება შემდეგი ტემპორალური ლოგიკები: წრფივი დროითი ტემპორალური ლოგიკა *LTL* და დროში განშტოებადი გაფართოებული ტემპორალური ლოგიკა *CTL.*

განმარტება. წრფივი დროითი ლოგიკის *LTL* ფორმულა არის :

- ატომარული პრედიკატები : $p, q, \ldots;$
- ან *LTL* ფორმულები, დაკავშირებული ერთმანეთთან ლოგიკური ოპერატორებით $\neg, \vee$;
- ან *LTL* ფორმულები, დაკავშირებული ერთმანეთთან ტემპორალური ოპერატორებით $X, U;$

LTL ლოგიკაში წარსული დრო არ განიხილება.

უფრო ფორმალურად *LTL* ლოგიკის ფორმულების აგების ყველა შესაძლო სტრუქტურა მოიცემა შემდეგი გრამატიკით:

$$\varphi :: p \mid \neg \varphi \mid \varphi \vee \varphi \mid X\varphi \mid \varphi U \varphi$$

1.3.2 კრიპკეს სტრუქტურა. ვერიფიკაციის პაკეტი *SPIN*

მოდელებზე შემოწმების მეთოდზე ინტენსიური მუშაობის შედეგად, ამ მეთოდის პრაქტიკული რეალიზაცია მნიშვნელოვნად მარტივია, ვიდრე ამ მეთოდის თეორიული საფუძვლები.

ვერიფიკაციის თანამედროვე პაკეტები ავტომატიზაციას უკეთებს ვერიფიკაციის ყველა ეტაპს, ის პროგრამისტს ათავისუფლებს იმ სირთულეებისაგან, რაც თან სდევს ვერიფიცირებადი სისტემის შესაბამისი კრიპკეს სტრუქტურის აგებას. ვერიცირებადი სისტემის მოდელი აღიწერება პაკეტის საწყისს ენაზე, რომელიც ფაქტობრივად წარმოადგენს დაპროგრამების მაღალი დონის ენას და იგი საკმაოდ მოხერხებულად აღწერს პარალელურად ურთიერთმოქმედ პროცესების სიმრავლეს. ზემოთ აღწერილი *CTL* და *LTL* ალგორითმების შესაბამისად ვერიფიცირების პაკეტი ავტომატურად აგებს მოცემული სისტემის ყველა მოდელების შესაბამისს კრიპკეს სტრუქტურას(გადასვლების სისტემას), ააგებს მათ კომპოზიციას და ავტომატურად შეასრულებს ვერიფიცირებას – ტემპორალური ლოგიკის ფორმულების სამართლიანობის შემოწმებას კონკრეტული კრიპკეს სტრუქტურისათვის,. სწორედ ასეთი ვერიფიკაციის პაკეტს წარმოადგენს – პაკეტი *SPIN.*

ვერიფიცირების პაკეტი *SPIN.* *SPIN* – არის ვერიფიცირების სისტემა, რომელიც გულისხმობს სასრულ რაოდენობა მდგომარეობის მქონე პარალელური სისტემების კორექტულობის ანალიზს, სისტემის თვისებათა სპეციფიკაცია წარმოდგენილი უნდა იყოს *LTL* ფორმულების დახმარებით. *SPIN* პაკეტი ორიენტირებულია ასინქრონული მოდელებზე, რომელიც ასე დამახასიათებელია პროგრამირების სიტემებისათვის. *SPIN* პაკეტი მომხმარებელს თავაზობს შემდეგ საშუალებებს:

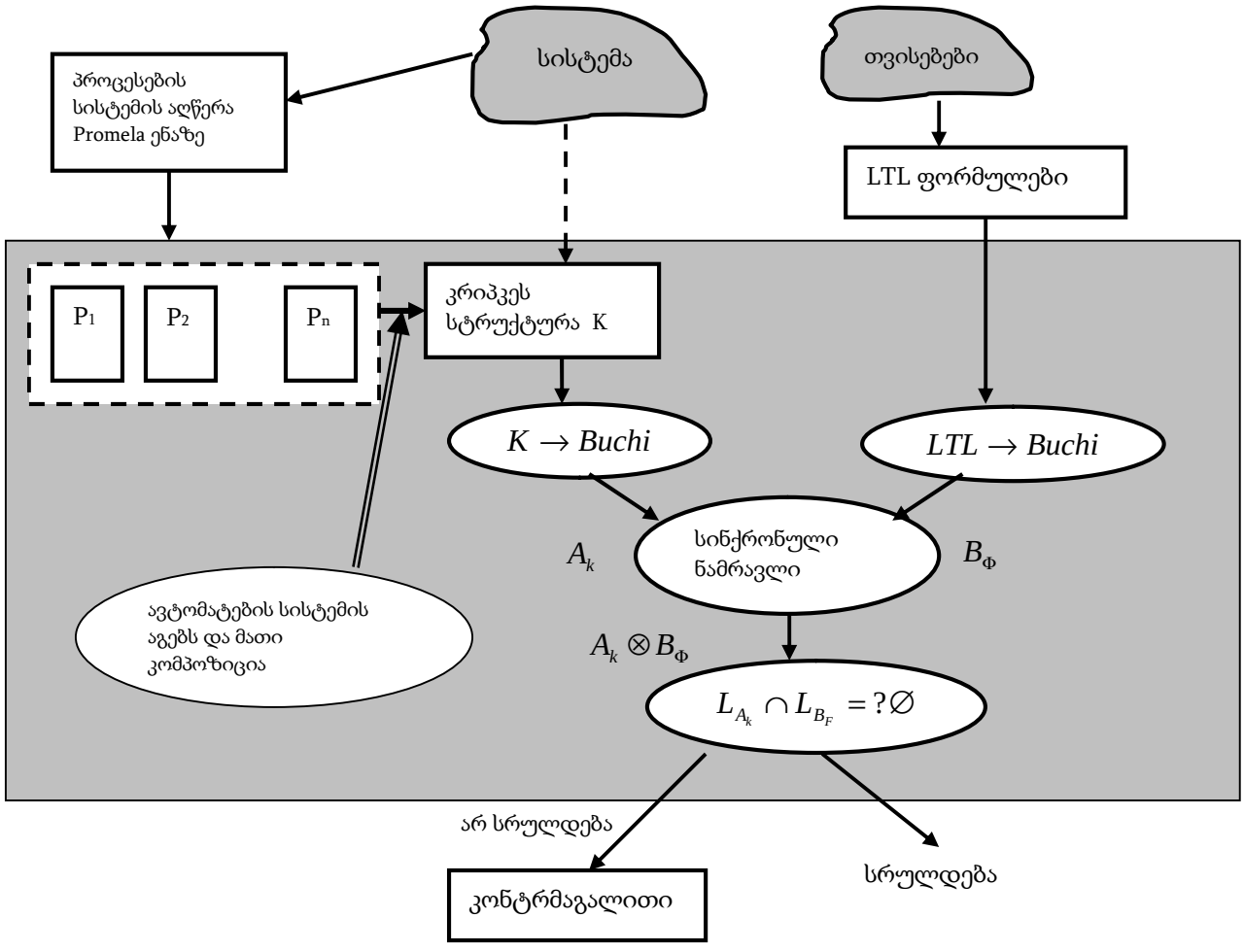
1. ენა *Promela(Protocol Meta Language)* – *C*–ის მსგავსი ენა მოდელის სპეციფიკაციებისათვის.

2. სისტემის კორექტულობის მოთხოვნათა გამოხატვის მოსახერხებელი სისტემა, რაც დაფუძნებულია წრფივ ტემპორალურ ლოგიკაზე(X (Next Time) ოპერატორის გარეშე).

აბრევიატურა *SPIN* იშიფრება როგორც *Simple Promela Interpreter.* ენა *Promela* მომხმარებელს აძლევს საშუალებას ააგოს პარალელური სისტემების შესაბამისი მოდელი, რომელიც ასახავს პარალელური პროცესების ურთიერთობას. ამიტომაც ამ ენაში არსებობს ობიექტების სპეციფიკაციის სამი ტიპი:

- პროცესები;
- არხები, რომელთა დახმარებით პროცესები ურთიეთობენ;
- მარტივი ტიპის(bit, bool, byte(8 ბიტი),short(ნიშნიანი მოკლე მთელი, 16 ბიტი), int(ნიშნიანი მთელი, 32 ბიტი)) ცვლადები;
- შესაძლებელია აიგოს ფიქსირებული ზომის მასივები, ასვე შეიძლება განსაზღვრული იყოს ჩანაწერიც.

SPIN პაკეტის ზოგადი სტრუქტურა ასეთია:



განვიხილოთ მარტივი პარალელური პროგრამა:

process Inc = *while* true *do if* x < 10 *then* x := x + 1 *od*

process Dec = *while* true *do if* x > 0 *then* x := x - 1 *od*

process Reset = *while* true *do if* x = 10 *then* x := 0 *od*

შეიძლება თუ არა იმის მტკიცება, რომ პროცესების შესრულებისას ყოველთვის სრულდება პირობა $0 \le x \le 10$? შევამოწმოთ ეს Promela-ზე დაწერილი შემდეგი პროგრამის მეშვეობით:

```
1   x=0;
2
3   proctype Inc() {
4         do  :: true -> if :: x<10  -> x=x+1   if   od  }
5   proctype Dec() {
6         do  :: true -> if :: x>0  -> x=x-1   if   od  }
7
8   proctype Reset() {
9         do  :: true -> if :: x==10  -> x=0   if   od  }
10
11  proctype Check() {
12        assert  (  x>=0  && x<=10  }
13
14  init  {
15       atomic {   run   Inc(); run Dec(); run Reset(); Run Check();
16    }
```

ამ მაგალითში პარალელური პროგრამის ვერიფიცირებისათვის ჩვენ შევქმენით პროცესი – მონიტორი Check(), რომელიც სხვა პროცესების პარალელურად მუშაობს და შეიცავს ერთადერთ assert ოპერატორს. ლოგიკური პირობები, რომელიც ამ ოპერატორის ტანს წარმოადგენს, შემოწმდება პარალელური პროგრამის ყველა შესაძლო მდგომარეობისათვის. Spin–ის გაშვების შემდეგ, იგი მოძებნის შეცდომას – სად დაირღვა assert ოპერატორში მითითებული პირობის ჭეშმარიტება. პროგრამის ასეთი ყოფაქცევის

მიზეზი ერთია: ყველა პროცესში – Inc, Dec, reset – if ოპერატორები სრულდება არა ატომარულად. დავუშვათ x=10 . მას შემდეგ რაც Dec პროცესში შემოწმებდა პირობა x>0, პროცესების ასინქრონულობის გამო, პარალელურ პროცესში შეიძლება შესრულდეს Reset პროცესის if ოპერატორი, რომელიც x ცვლადს მიანიჭებს 0–ს, მანამდე შეამოწმებს რა, რომ x=10.

სისტემის ვერიფიცირების მეორე შანსი არის ოპერატორი never. never ოპერატორში აღწერილია შესამოწმებელი სისტემის არასასურველი ტრაექტორია. ფუნქცია *never* განმარტავს ბიუხეს ავტომატს – საკონტროლო ავტომატს – რომელიც შესამოწმებელი სისტემის სინქრონულად მუშაობს. თუ ეს ავტომატი მოდის საბოლოო მდგომარეობაში, ეს ნიშნავს, რომ სისტემაში არასწორი გამოთვლები არსებობს.

მაგალითად, იმისათვის რომ შევამოწმოთ ვერიფიცირებადი სისტემისათვის ყოველთვის სრულდება თუ არა პირობა *GF(x>3),* პროგრამისტმა უნდა ააგოს შემდეგი პროცესი *never:*

```
1  never  {     /* FG(x<=3), თუ ეს პირობა არასდროს არ შესრულდება, ეს ნიშნავს, რომ
                GF(x>3)  */
2  To_init:
3   if
4     :: ( x<=3 )  ->  goto accept
5     :: true      ->  goto To_init
6   fi;
7  accept
8  if
9    :: ( x<=3 )   ->  goto accept
10  fi;
11  }
```

Spin-ს ასევე შეუძლია, შეამოწმოს პროცესების სისტემის ყოფაქცევა, განმარტებული წრფივი ტემპორალური ლოგიკის ფორმულების საშუალებით. Spin სისტემის სინტაქსისი საშუალებას იძლევა დროითი *LTL* ლოგიკის ოპერატორები ჩაიწეროს შემდეგი სახით: ოპერატორი *G* ჩაიწერება, როგორც კვადრატული ფრჩხილების წყვილი [], ოპერატორი *F* – როგორც კუთხური ფრჩხილების წყვილი <>, ხოლო ოპერატორი *Until* - უბრალოდ *U.*

იმისათვის, რომ შესრულდეს ასეთი ვერიფიკაცია, *Spin* პაკეტი *LTL* ლოგიკის ფორმულების შინაარსის მიხედვით, აგებს მათ უარყოფას, რომელიც ბუნებრივია განსაზღვრავს სისტემის არასასურველ ქცევას.

Promela ენაზე წარმოდგენილი ყველა მოდელი აუცილებლად ექვემდებარება რაღაც შეზღუდვებს. ამიტომაც ბუნებრივია, კორექტულობის საკითხი ფორმალურად განმარტებულია ამ შეზღუდვების ფარგლებში. ცხადია, რომ ვერიფიცირების ყველა სისტემას გააჩნია ფიზიკური შეზღუდვები, განპირობებული მეხსიერების მოცულობის, პროცესორის სწრაფქმედების პარამეტრებით. სისტემა Spin-ში გამოიყენება რამდენიმე ხრიკი, რომელიც საშუალებას იძლევა გავუმკლავდეთ "მდგომარეობათა რაოდენობის აფეთქების" პრობლემას და გაზრდილი იყოს ისეთი სისტემების რაოდენობა, რომლისთვისაც შესაძლებელია ამომწურავი ანალიზის ჩატარება.

თავი 2. დაპროგრამების ენების განსაზღვრა სასრული ავტომატების საშუალებით.

პროგრამირების ენების ზოგადი თვისებების აღწერისას მოსახერხებელია, ეს ენები განვიხილოთ, როგორც ფორმალური ენების კლასი. თავისთავად, ფორმალური ენა შედგება სიტყვებისა და წინადადებების უსასრულო სიმრავლისაგან, მაგრამ შესაძლებელია იგი განმარტებული იყოს სასრული მეთოდით – ე.წ. გრამატიკების დახმარებით. ამ თავის მიზანია განვიხილოთ ფორმალური ენებისა და ენის ელემენტების სინტაქსური თვისებების აღწერა გრამატიკების დახმარებით. პროგრამირების ენებისათვის ასეთ ელემენტებად გვევლინება კონკრეტული პროგრამები, ამიტომაც ამ თავში განხილულია პროგრამების სინტაქსური თვისებების საერთო სტრუქტურა და საკითხი, წარმოადგენს თუ არა ალფავიტის სიმბოლოთა მიმდევრობა ენის წინადადებებს.

ზემოთ მოხსენებული საკითხების ანალიზი ბუნებრივად მოითხოვს ავტომატების თეორიის საკითხების განხილვას, რომლის შესახებაც საუბარი იქნება მომდევნო ქვეთავში.

2.1 გრამატიკები და ფორმალური ენები.

სანამ უშუალოდ გრამატიკების და ფორმალური ენების განმარტებას შევუდგებოდეთ, საჭიროა რამდენიმე ცნების განმარტება – მაგალითად, რა არის ალფავიტი.

ალფავიტს ჩვენ დავუძახებთ სიმბოლოთა ნებისმიერ სიმრავლეს. იგულისხმება, რომ სიმბოლოს ინტუიციური არსი გასაგებია და ის თავისთავად არ საჭიროებს განმარტებას.

არ არის სავალდებულო, რომ ალფავიტი იყოს სასრული, ან თვლადი, მაგრამ ყველა პრაქტიკულ გამოყენებაში ის იქნება სასრული. ალფავიტის მაგალითია: სიმრავლე, რომელიც შედგება 26 დიდი და 26 პატარა ლათინური სიმბოლოსაგან (ლათინური ალფავიტი), სიმრავლე $\{ 0, 1 \}$, რომელსაც ბინარული, ანუ ორობითი ალფავიტი ეწოდება.

სიმბოლოების მიმდევრობას ეწოდება ჯაჭვი. მაგალითად 01011 წარმოადგენს ჯაჭვს ბინარულ ალფავიტში. ტერმინები სიტყვა და წინადადება ხშირად გამოიყენება როგორც ჯაჭვის სინონიმი.

არსებობს ე.წ. ცარიელი ჯაჭვი, რომელიც ხშირად გამოიყენება და აქვს თავისი სპეციალური აღნიშვნა – £ . ცარიელი ჯაჭვი არ შეიცავს არც ერთ სიმბოლოს.

განმარტება 2.1.1. ჯაჭვი Σ ალფავიტში ფორმალურად განიმარტება შემდეგნაირად:

(*) £ – ცარიელი ჯაჭვია ალფავიტში,

(**) თუ x წარმოადგენს ჯაჭვს Σ ალფავიტში, ხოლო $a \in \Sigma$ მაშინ xa არის ჯაჭვი Σ ალფავიტში,

(***) y წარმოადგენს ჯაჭვს Σ ალფავიტში, მაშინ და მხოლოდ მაშინ, თუ ის აკმაყოფილებს *(1)* და *(2)* პირობებს.

(ჯაჭვის სიგრძე აღინიშნება ასე $/X/$ და უდრის ჯაჭვში სიმბოლოების რაოდენობას).

დავუშვათ, მოცემულია ტერმინალური სიმბოლოების, ტერმინალების სასრული, არაცარიელი Σ სიმრავლე, ანუ ალფავიტი. Σ ალფავიტის ელემენტების ბუნების შესახებ არაფერი არ არის ცნობილი. თუ ჩვენ ვაპირებთ Σ ალფავიტის გამოყენებას დაპროგრამების ენის აღსაწერად, მაშინ ცხადია ამ ალფავიტში გვხვდება ისეთი ჯაჭვები (სიტყვები), როგორიცაა: *if, while, case, switch, do, for* და ა. შ., ასევე ისეთი ჯაჭვები, საიდანაც იგება იდენტიფიკატორები, კონსტანტები და ენის სხვა ელემენტები.

Σ^* არის სიმრავლე, რომელიც შეიცავს Σ ალფავიტის სიმბოლოთა შესაძლო ყველა კომბინაციას (ჯაჭვს), ერთეულოვანი £ ჯაჭვის ჩათვლით. ასეთ კომბინაციას ეწოდება Σ^* -ის სიტყვები, ან ფრაზები (პროგრამირების ენებში მათ შეესაბამებათ პროგრამის ფრაგმენტები).

N არის არატერმინალური სიმბოლოების - სიმბოლოები, რომლებიც განმარტებას საჭიროებენ - არაცარიელი, სასრული სიმრავლე, ისეთი, რომ $N \cap \Sigma = \varnothing$. N^* არის N სიმრავლეზე განსაზღვრული სიმბოლოთა ყველა შესაძლო კომბინაციათა სიმრავლე, ერთეულოვანი £ ჯაჭვის ჩათვლით.

ყველა ტერმინალური და არატერმინალური სიმბოლოთა სიმრავლეს ეწოდება ფორმალური ენის ლექსიკონი და აღინიშნება ასე $V = \Sigma \cup N$. V^*არის Vსიმრავლეზე განსაზღვრული სიმბოლოთა ყველა შესაძლო კომბინაციათა სიმრავლე, ერთეულოვანი £ ჯაჭვის ჩათვლით - $V^* = (\Sigma \cup N)^*$.

შემოვიღოთ ასეთი აღნიშვნებიც:

$$\Sigma^+ = \Sigma^* - £\,, \qquad N^+ = N^* - £\,, \quad V^+ = V^* - £\,.$$

$¶ \in V^* \times N \times V^*$ წარმოადგენს ტერმინალური და არატერმინალური სიმბოლოებისაგან შედგენილ ჯაჭვს.

ავღწეროთ შემდეგი სიმრავლე:

$$\Im = \{\ (¶,\ \ ) \mid ¶ \in V^* \times N \times V^* \wedge \ \ \in V^+\ \}$$

ყოველ დალაგებულ (¶,) წყვილს $\Im$ სიმრავლიდან ეწოდება ე.წ. შექმნის წესები და ჩაიწერება ასე: ¶ → . მაგალითად, შექმნის წესი შეიძლება იყოს შემდეგი წყვილი *(AB, CDE)*. თუ ¶ წარმოადგენს ჯაჭვს - *FGABH*, რომელიც შეიცავს *AB* –ს, როგორც ქვეჯაჭვს, მაშინ შეიძლება გაჩნდეს ახალი ჯაჭვი – *FGCDEH*, სადაც *AB* შეცვლილი იქნება *CDE*– თი.

განმარტება 2.1.2 ფორმალური გრამატიკა G განმარტებით არის შემდეგი ოთხეული

$$G = (N,\ \Sigma,\ P,\ S)\ , \qquad (1)$$

სადაც N არის არატერმინალური სიმბოლოების სიმრავლე, Σ არის ტერმინალური სიმბოლოების სიმრავლე, P არის $\Im$ სიმრავლის არაცარიელი, სასრული ქვესიმრავლე (სინტაქსური წესების ერთობლიობა) და ბოლოს $S \in N$ და მას ეწოდება საწყისი სიმბოლო, ანუ ენის აქსიომა.

ფორმალური გრამატიკა წარმოადგენს ფორმალური ენის აღწერის მათემატიკურ საფუძველს. თავის მხრივ გრამატიკის ბირთვს წარმოადგენს შექმნის წესების P არაცარიელი, სასრული ქვესიმრავლე, ანუ სინტაქსური წესების ერთობლიობა. სხვა სიტყვებით, წესი არის ჯაჭვების უბრალო წყვილი, სადაც წესის

პირველი კომპონენტი არის ნებისმიერი ჯაჭვი, რომელიც შეიცავს ერთ მაინც არატერმინალურ სიმბოლოს, ხოლო მეორე კომპონენტი კი არის ნებისმიერი ჯაჭვი.

შესაძლებელია გრამატიკების კლასიფიკაცია შექმნის წესის მიხედვით. ვთქვათ, $G=(N,\ \Sigma,\ P,\ S)$ გრამატიკაა.

განმარტება 2.1.3. G გრამატიკას ეწოდება

(1) წრფივი, თუ P სიმრავლიდან ნებისმიერი წესი განიმარტება ასე: $\sim \to x$, ან $\sim \to x$, სადაც $\sim,\ \in N,\ x\in \Sigma^*$; (მე-3 ტიპის გრამატიკა)

(2) კონტექსტიდან თავისუფალი, თუ P სიმრავლიდან ნებისმიერ წესს აქვს შემდეგი სახე: $\sim \to$, სადაც $\sim \in N,\ \in (N \cup \Sigma)^*$; (მე-2 ტიპის გრამატიკა)

(3) კონტექსტზე დამოკიდებული, თუ P სიმრავლიდან ნებისმიერი წესს აქვს ასეთი სახე: $\sim \to$, სადაც $|\sim| \le |\ |$; (პირველი ტიპის გრამატიკა)

გრამატიკას, რომელიც არც ერთ ზემოთ განმარტებულ ტიპს არ მიეკუთვნება, ეწოდება ზოგადი სახის გრამატიკა, ანუ გრამატიკა შეზღუდვების გარეშე (ნულოვანი ტიპის გრამატიკა).

გრამატიკების ზემოთ მოყვანილი კლასიფიკაცია გაკეთებულია ხომსკის[13] ნაშრომებზე დაყრდნობით და ხშირად ამ გრამატიკებს ე.წ. ხომსკის გრამატიკები ეწოდება.

განმარტებიდან გამომდინარეობს, რომ ნებისმიერი წრფივი გრამატიკა წარმოადგენს კონტექსტიდან თავისუფალ, კონტექსტზე დამოკიდებულ და შეზღუდვების გარეშე გრამატიკას. თავის მხრივ ნებისმიერი კონტექსტიდან თავისუფალი გრამატიკა წარმოადგენს კონტექსტზე დამოკიდებულ და შეზღუდვების გარეშე გრამატიკას და ა.შ. ფაქტიურად მივიღეთ ხომსკის გრამატიკების იერარქია, რომელსაც შეესაბამება ფორმალური ენების იერარქია. ფორმალური ენები განიმარტება შემდეგი სამი ნაბიჯის საშუალებით.

---

[13] *Avram Noam Chomsky* (ავრაამ ნოემ ხომსკი) – ამერიკელი ლინგვისტი, მასაჩუსეტის ტექნოლოგიური ინსტიტუტის ლინგვისტიკის პროფესორი, ფორმალური ენების კლასიფიკაციის ავტორი, რომელიც ცნობილია ხომსკის იერარქიის სახელით.

- დამოკიდებულება $\underset{G}{\Rightarrow}$, რომელიც მოცემულია V^* სიმრავლეზე განისაზღვრება შემდეგნაირად: $¥_1 \underset{G}{\Rightarrow} ¥_2$ (სადაც $¥_1, ¥_2 \in V^+$), თუ არსებობს $¢_1, ¢_2 \in V^*$ და შექმნის წესი $¶ \to \in P$, ისეთი რომ

$$¥_1 = ¢_1 ¶ ¢_2 \quad \wedge \quad ¥_2 = ¢_1 \quad ¢_2 \qquad (2)$$

ფაქტიურად, საქმე გვაქვს ჯაჭვების ე.წ. გარდაქმნასთან: ჯაჭვი $¥_1$ გარდაიქმნება $¥_2$ ჯაჭვში, თუ $¥_1$ ჯაჭვში შევცვლით ერთ ან რამდენიმე სიმბოლოს(ანუ $¶$ ქვეჯაჭვს) ჯაჭვით. სიმბოლოები, რომლებიც ესაზღვრება $¶$ ქვეჯაჭვს (ანუ $¢_1, ¢_2$) რჩება უცვლელი.

- დამოკიდებულება $\overset{*}{\underset{G}{\Rightarrow}}$, რომელიც მოცემულია V^* სიმრავლეზე, განიმარტება ასე:

$$¥_0 \overset{*}{\underset{G}{\Rightarrow}} ¥_n, \quad \text{თუ } ¥_0, ¥_1, ..., ¥_n \in V^* \ (n \geq 0) \text{ და } ¥_{i-1} \underset{G}{\Rightarrow} ¥_i \qquad (i = 1,2,...,n) \quad (3)$$

$¥_0, ¥_1, ..., ¥_n$ მიმდევრობას ეწოდება n სიგრძის გამოყვანა, ვინაიდან $¥_n$ გამომდინარეობს $¥_0$-დან n ოპერაციის განხორციელებით.

უკვე შესაძლებელია განმარტოთ ფორმალური ენა.

განმარტება 2.1.4. G გრამატიკაზე დაყრდნობით შექმნილი $L(G)$ ფორმალური ენა განიმარტება, როგორც Σ^* სიმრავლის შემდეგი ქვესიმრავლე:

$$L(G) = \left\{ \quad x \mid S \overset{*}{\underset{G}{\Rightarrow}} x \ \wedge \ x \in \Sigma^* \ \wedge \ S \in N \ \right\}. \qquad (4)$$

შენიშვნა 1. თუ $S \Rightarrow \underset{G}{\overset{*}{¥}} \in V^*$ და $¥$ შეიცავს არატერმინალებს, მაშინ ასეთ $¥$-ს ეწოდება ფრაზული ფორმა. თუ $¥$ არ შეიცავს არატერმინალებს, მაშინ მას ეწოდება ენის ფრაზა, ან სიტყვა.

შენიშვნა 2. თუ S-დან ყველა გამოყვანა ქმნის მხოლოდ ფრაზულ ფორმებს, მაშინ $L(G)$ წარმოადგენს ცარიელ ენას.

ენა, რომელიც შექმნილია შესაბამის გრამატიკაზე დაყრდნობით, წარმოადგენს ან წრფივ ენას (მე-3 ტიპის ენა), ან კონტექსტიდან თავისუფალ ენას (მე-2 ტიპის ენა), ან კონტექსტზე დამოკიდებულ ენას (1-ლი ტიპის ენა), ან ენას შეზღუდვების გარეშე (ნულოვანი ტიპის ენა). ბუნებრივია, რომ ნებისმიერი წრფივი ენა წარმოადგენს კონტექსტიდან თავისუფალ, კონტექსტზე დამოკიდებულ და შეზღუდვების გარეშე ენას. თავის მხრივ ნებისმიერი კონტექსტიდან თავისუფალი ენა წარმოადგენს კონტექსტზე დამოკიდებულ და შეზღუდვების გარეშე ენას და ა.შ.

განმარტება 2.1.5 $L(G_1)$ და $L(G_2)$ ენები იგივურია, თუ ისინი შეიცავენ ერთი და იგივე ფრაზებს. ამ შემთხვევაში, შესაბამისს G_1 და G_2 გრამატიკებს ეწოდება ექვივალენტური გრამატიკები.

ნებისმიერი i – ური ტიპის გრამატიკა ექვივალენტურია i – 1 (i = 1, 2, 3) ტიპის გრამატიკისა, მაგრამ i – ური ტიპის გრამატიკა ექვივალენტური არ არის არც ერთი i + 1 (i = 0, 1, 2) გრამატიკისა.

2.2* ავტომატების თეორიის ზოგიერთი ცნებები

ავტომატები შეიძლება განხილული იყოს, როგორც მექანიზმები, რომლებიც შედგება მართვის ბლოკისაგან, შემავალი და გამომავალი არხისაგან. მართვის ბლოკს შეუძლია იმყოფებოდეს სხვადასხვა მდგომარეობაში(ავტომატის ე.წ. შიდა მდგომარეობა), შემავალი არხი აღიქვამს(კითხულობს, ამოიცნობს) გარემომცველი გარემოდან შემომავალ სიგნალებს, ხოლო გამომავალი არხი იძლევა გამომავალ სიგნალებს. სიგნალებისა და მდგომარეობების ბუნება განურჩეველია: ისინი შეიძლება განხილული იყოს როგორც რაღაც სიმბოლოები(ასოები), რომლებიც თავის მხრივ ქმნიან მდგომარეობათა ალფავიტს (ანუ შიდა ალფავიტი) – Q, საწყისი ალფავიტი X და საშედეგო ალფავიტი Y. X და Y ალფავიტები ითვლება, რომ სასრულია, ხოლო Q შეიძლება იყოს თვლადიც. ავტომატის მუშაობა მიმდინარეობს დროის დისკრეტულ ტაქტებში: $t = 1, 2, 3, \ldots$ და სრულიად განისაზღვრება თანმდევი პროგრამის მიერ, ანუ ბრძანებათა სისტემით. ყოველ ბრძანებას აქვს შემდეგი სახე:

$$q_i\, x_r \rightarrow q_j\, y_s$$

სადაც q_i, q_j ავტომატის შიდა მდგომარეობაა, x_r საწყისი(შემავალი) სიმბოლო, y_s კი საბოლოო(გამომავალი) სიმბოლო. იგულისხმება, რომ ავტომატის პროგრამაში არ შეიძლება არსებობდეს ორი განსხვავებული $q_i\, x_r \to q_j\, y_s$ და $q_i\, x_r \to q_\neg\, y_r$ ბრძანება, ერთი და იგივე მარცხენა მხარეებით, მაგრამ განსხვავებული მარჯვენა მხარეებით (ცალსახობის პირობა); თუმცა არ არის სავალდებულო, რომ ნებისმიერი $q_i\, x_r$ წყვილისთვის პროგრამაში არსებობდეს ბრძანება, რომელსაც ექნებოდა ასეთი მარცხენა ნაწილი.

ვთქვათ, დროის რაიმე t_0 ტაქტში მართვის მოწყობილება მდებარეობს q_i მდგომარეობაში და შემავალი არხი ამოიცნობს x_r სიმბოლოს. თუ პროგრამაში არსებობს ბრძანება, რომლის მარცხენა მხარეა $q_i\, x_r$, მაგალითად ბრძანება $q_i\, x_r \to q_j\, y_s$,მაშინ იმავე დროის t_0 ტაქტში გამომავალი არხი იძლევა y_sსიმბოლოს, ხოლო დროის მორიგ t_0+1 ტაქტში მართვის მოწყობილობა გადადის q_j მდგომარეობაში. თუ პროგრამაში ასეთი ბრძანება არ არსებობს(სხვანაირად, რომ ვთქვათ წყვილი $q_i\, x_r$ აკრძალულია), მაშინ ავტომატი ბლოკირებულია, ანუ ის არ რეაგირებს არც ერთ სიმბოლოზე. სიმარტივისათვის შევთანხმდეთ, რომ ჩვენ განვიხილავთ ისეთ ავტომატებს, სადაც აკრძალული წყვილები არ არსებობს(სრული განსაზღვრულობის პირობა) და არ განვიხილავთ ე.წ. კერძო ავტომატებს, სადაც აკრძალული წყვილები დაშვებულია.

ვთქვათ ავტომატის მართვის მოწყობილობა დაყენებულია საწყისს $q(t_0)$ მდგომარეობაში და მისი შემავალი არხზე მიეწოდება $x(t_0), x(t_0+1), x(t_0+2),\ldots$ სიმბოლოები, მაშინ ავტომატი საკუთარი პროგრამის შესაბამისად, გამომავალ არხში $y(t_0), y(t_0+1), y(t_0+2),\ldots$ სიმბოლოებს იძლევა, ხოლო მართვის მოწყობილობაში თანმიმდევრობით აღმოცენდება $q(t_0), q(t_0+1), q(t_0+2),\ldots$ მდგომარეობები. სწორედ ამაში მდგომარეობს ავტომატის ფუნქციონირების სქემა. აქედან ჩანს, რომ გამომავალი სიმბოლო, რომელიც ავტომატის მიერ არის გამომუშავებული დროის რაღაც t მომენტში, დამოკიდებულია არა მარტო მიმდინარე შემავალ სიმბოლოზე, არამედ მანამდე შემომავალ სიმბოლოებზეც. წინა სიმბოლოები ავტომატში ფიქსირდება მისი მდგომარეობების შეცვლის გზით. ამ გაგებით ავტომატის შიდა მდგომარეობათა სიმრავლე წარმოადგენს მის შიდა მეხსიერებას. მოსახერხებელია, გარე გარემო,

საიდანაც ავტომატი იღებს საწყისს ინფორმაციას, წარმოდგენილი იყოს სასრული ან უსასრულო სიგრძის ლენტის სახით, რომელიც დაყოფილია უჯრებად და თითოეულ ამ უჯრაში ჩაწერილია საწყისი სიმბოლო. ავტომატის მუშაობის დაწყებისას მართვის ბლოკი იმყოფება საწყისს მდგომარეობაში, ხოლო შემავალი არხი(წაკითხვის თავაკი) იმყოფება იმ უჯრის საპირისპიროდ, რომელიც არჩეულია საწყისს მდგომარეობად და კითხულობს, იმ სიმბოლოს, რომელიც ამ უჯრაშია მოთავსებული. შემდეგ ყოველ ტაქტზე ლენტა გადაადგილდება ერთი უჯრით ერთი და იგივე მიმართულებით(მაგალითად, მარცხნიდან მარჯვნივ), სწორედ ასეა შესაძლებელი შემავალი სიმბოლოების თანმიმდევრობით წაკითხვა. თუ ლენტა სასრულია, მაშინ ტაქტების სასრული რაოდენობის შემდეგ წამკითხავი თავაკი გადადის ლენტიდან და ამით ავტომატის მუშაობის პროცესი სრულდება. თუ ლენტა უსასრულოა, მაშინ პროცესიც უსასრულოდ გრძელდება. შეიძლება ჩავთვალოთ, რომ არსებობს კიდევ ერთი ლენტა(გამომავალი ლენტა), რომელიც გადაადგილდება მარცხნივ, შემავალი ლენტის სინქრონულად. ავტომატის მუშაობის დაწყებისას გამომავალი ლენტის ყველა უჯრა ცარიელია და ჩამწერი თავაკი თანმიმდევრობით წერს მათში გამომავალ სიმბოლოებს.

განმარტება. ავტომატი არის შემდეგი ხუთეული $\langle Q, X, Y, \Psi, \Phi \rangle$, სადაც Q, X, Y შიდა, შემავალი და გამომავალი ალფავიტებია, Ψ გადასვლათა ფუნქცია, რომელიც $Q \times X$ ასახავს Q–ში, Φ გამოსვლების ფუნქცია, რომელიც $Q \times X$ ასახავს Y–ში. Q ალფავიტის სიმბოლოებს ეწოდება ავტომატის შიდა მდგომარეობები. ყველა შესაძლო ოთხეულებს $\langle q, x, \Psi(q,x), \Phi(q,x) \rangle$ ეწოდება ავტომატის ბრძანებები და ისინი ჩაიწერება შემდეგი სახით: $qx \to \Psi(q,x)\ \Phi(q,x)$.

დავუშვათ, დაფიქსირებულია $\P = \langle Q, X, Y, \Psi, \Phi \rangle$ ავტომატის რომელიმე q_0 მდგომარეობა, მაშინ შემდეგი რეკურენტული დამოკიდებულებები

$$\begin{aligned} q(t+1) &= \Psi[\, q(t),\ x(t)\,], \\ y(t) &= \Phi[\, q(t),\ x(t)\,] \end{aligned} \qquad (*)$$

სადაც $q(t), q(t+1) \in Q, x(t) \in X, y(t) \in Y$ და შემდეგი საწყისი პირობა

$$q(1) = q_0,$$

განსაზღვრავს ე.წ. *ოპერატორს* (აღინიშნება $T(¶, q_0)$–ით), რომელიც საწყისს სიმბოლოთა $x = x(1)\,x(2)\,x(3)\ldots x(r)$ ნებისმიერ სასრულ მიმდევრობას გარდაქმნის საბოლოო სიმბოლოთა იმავე სიგრძის რაღაც y მიმდევრობად: $y = y(1)\,y(2)\,y(3)\ldots y(r)$.

წყვილს $\langle ¶, q_0 \rangle$ ეწოდება საწყისი ავტომატი და ვიტყვით, რომ $\langle ¶, q_0 \rangle$ ავტომატს რეალიზაციას უკეთებს $T(¶, q_0)$ ოპერატორი, რაც იგივეა, რომ ოპერატორი $T(¶, q_0)$ წარმოადგენს $\langle ¶, q_0 \rangle$ საწყისი ავტომატის ქცევას. ვიტყვით, რომ ¶ ავტომატს რეალიზებას უკეთებს T ოპერატორი, თუ საწყისი მდგომარეობის q_0–ის ფიქსირებისას $T = T(¶, q_0)$.

შემოვიღოთ რამდენიმე ტერმინი.

რაიმე A ალფავიტის სიმბოლოთა არაცარიელ მიმდევრობას ეწოდება A ალფავიტის სიტყვა, ხოლო A ალფავიტის სიტყვათა ნებისმიერ სიმრავლეს – A ალფავიტზე განსაზღვრული ენა. Q, X, Y ალფავიტების სიტყვას შესაბამისად შიდა, საწყისი და საბოლო სიტყვა ეწოდება.

ანალოგიურად, ტერმინები „ზესიტყვა A ალფავიტში“, „საწყისი ზესიტყვა“, „საბოლოო ზესიტყვა“, „ზეენა“ გამოიყენება სიტყვების მაგივრად: „A ალფავიტის სიმბოლოთა უსასრულო მიმდევრობა“, „A ალფავიტის სიმბოლოთა უსასრულო მიმდევრობების ნებისმიერი სიმრავლე“ და ა.შ.

თუ $T(¶, q_0)$ ოპერატორი საწყისს $x = x(1)\,x(2)\ldots x(r)$ სიტყვას გარდაქმნის საბოლოო $y = y(1)\,y(2)\ldots y(r)$ სიტყვად, ასეთ ოპერატორს და მსგავს ოპერატორებს სიტყვითი ოპერატორები ეწოდება. ცხადია, რომ შესაძლებელია ინიცირებული ავტომატის ქცევის აღწერა ზესიტყვითი ოპერატორების დახმარებით, ანუ ისეთი ოპერატორებით, რომლებიც საწყისს ზესიტყვას გაედაქმნის საბოლოო ზესიტყვად. ასევე ადვილი გასაგებია, რომ ინიცირებული $(¶, q_0)$ ავტომატის სიტყვითი ქცევა და ზესიტყვითი ქცევა ერთმანეთთან მჭიდროდ არის დაკავშირებული. ამ მიზეზის გამო ავტომატების თეორიაში არასდროს არ არის გამახვილებული ყურადღება $T(¶, q_0)$ ოპერატორის ბუნებაზე, როგორია ის: სიტყვითი, თუ ზესიტყვითი.

2.2˚.1 ავტომატების სახეები.

როცა Q, X, Y ალფავიტებიდან ერთერთი მაინც შედგება ერთი სიმბოლოსაგან, მაშინ საქმე გვაქვს ე.წ. გადაგვარებულ შემთხვევასთან. ასეთ შემთხვევებში, მოსახერხებელია განვიხილოთ ავტომატების მოდიფიცირებული განმარტებები, რომლებიც მიიღება გადაგვარებული კომპონენტების ამოღებით $\langle Q, X, Y, \Psi, \Phi \rangle$ ხუთეულიდან და შესაბამისად სხვა კომპონენტების ჩანაცვლებით. მოვიყვანოთ ეს ცნებები.

- ავტომატი მეხსიერების გარეშე – არის სამეული $\langle X, Y, \Phi \rangle$, სადაც Φ არის X საწყისი ალფავიტის ასახვა Y საბოლოო ალფავიტზე. სხვაგვარად, რომ ვთქვათ მეხსიერების გარეშე ავტომატის ბრძანებებს აქვს სახე:

$$x_r \rightarrow y_s$$

სადაც $x_r \in X$, $y_s \in Y$. (*) დამოკიდებულებას ამ შემთხვევაში აქვს სახე

$$y(t) = \Phi[x(t)],$$

ე.ი. მოცემულ ტაქტში გამომავალი სიმბოლო დამოკიდებულია მხოლოდ შემომავალ სიმბოლოზე და სრულებით არ არის დამოკიდებული მანამდე ამოცნობილ სიმბოლოებზე. აქედან გამომდინარე ნებისმიერი მეხსიერების გარეშე ავტომატი რეალიზებას უკეთებს მხოლოდ ერთ ოპერატორს, რომელიც განახორციელებს საწყისი სიმბოლოს „ასო–ასოში გადაყვანას“ გამომავალ სიმბოლოში. ასეთ ოპერატორებს ეწოდებათ მეხსიერების გარეშე ოპერატორები.

მეხსიერების გარეშე ავტომატების საერთო რაოდენობა, რომლის ალფავიტებია $X = \{x_1, x_2, ..., x_m\}$ და $Y = \{y_1, y_2, ..., y_n\}$, n^m რიცხვის ტოლია.

- ავტონომური ავტომატი – არის ოთხეული $\langle Q, Y, \Psi, \Phi \rangle$, სადაც Ψ და Φ ასახავს Q–ს Q–ში და Q–ს Y–ში შესაბამისად. ამ ავტომატის ბრძანებებს აქვს სახე $q_i \rightarrow q_j y_s$, ხოლო (*) რეკურენტული დამოკიდებულება მიიღებს სახეს

$$q(t+1) = \Psi[q(t)],$$
$$y(t) = \Phi[q(t)]$$

$q(1) = q_0$ საწყისი მდგომარეობით.

ცხადია, რომ ავტონომური ავტომატების საერთო რაოდენობა, რომლის ალფავიტებია $Q = \{q_1, q_2, ..., q_k\}$ და $Y = \{y_1, y_2, ..., y_n\}$, $(nk)^k$ რიცხვის ტოლია.

- ავტომატი გამოსავლის გარეშე – წარმოადგენს სამეულს $\langle Q, X, \Psi \rangle$, სადაც Ψ არის $Q \times X$ –ის ასახვა Q –ში. ავტომატის ბრძანებებს აქვს სახე $q_i x_r \to q_j$, ხოლო (*) რეკურენტული დამოკიდებულებას აქვს სახე

$$q(t+1) = \Psi[q(t), x(t)].$$

- ავტომატი დაბრკოლებით. Φ ფუნქციას აქვს სახე $^{a}(q)$, ანუ ის არ არის დამოკიდებული x –ზე. ამ ავტომატისათვის t ტაქტზე გამომავალი სიმბოლო არ არის დამოკიდებული ამავე ტაქტზე შემომავალ სიმბოლოზე, ის დამოკიდებულია მხოლოდ იმ სიმბოლოზე, რომელიც ამოცნობილი იყო წინა ტაქტზე.
- მურის ავტომატი. მურის ავტომატისათვის Ψ და Φ ფუნქციებს შორის არსებობს შემდეგი კავშირი: $\Phi[q, x] = {}^{a}[\Psi(q, x)]$, სადაც ^{a} რაღაც ასახვაა Q –დან Y –ში. ამრიგად განსხვავება მურის ავტომატსა და დაბრკოლებების მქონე ავტომატს შორის მხოლოდ ის არის, რომ პირველისათვის

$$y(t) = {}^{a}[q(t)],$$

ხოლო მეორესათვის

$$y(t) = {}^{a}[q(t+1)]$$

იმ პირობით, რომ $q(t+1) = \Psi[q(t), x(t)]$.

- ალბათური ავტომატები. ალბათური ავტომატები წარმოადგენს ჩვეულებრივი(დეტერმინირებული) ავტომატების განზოგადებას. დეტერმინირებულ ავტომატებში საწყის $a \in X$ ასოს ყოველი $q_i \in Q = \{q_1, q_2, \ldots, q_k\}$ მდგომარეობა გადაჰყავს $\Psi(q_i, a) \in Q$ მდგომარეობაში. ალბათურ ავტომატში კი, საწყისს a ასოს ავტომატი q_i მდგომარეობიდან გადაჰყავს $q_j \in Q$ მდგომარეობაში $\Re(a, q_i, q_j)$ ალბათობით. ეს გადასვლითი ალბათობები არ არის დამოკიდებული დროზე და წინა მდგომარეობებზე, თუმცა ნებისმიერი ფიქსირებული $a \in X$ და $q_i \in Q$ –თვის სრულდება პირობა $\sum_{q_j \in Q} \Re(a, q_i, q_j) = 1$.

ალბათური ¶ *ავტომატი* არის სამეული $\langle Q, X, \Re \rangle$, სადაც Q და X სასრული შიდა და საწყისი ალფავიტებია, ხოლო $\Re$ (ალბათურ გადასვლების ფუნქცია) წარმოადგენს $X \times Q \times Q$ –ს ასახვას $[0,1]$ სეგმანტზე და აკმაყოფილებს პირობას

$$\sum_{q_j \in Q} \Re(a, q_i, q_j) = 1 \qquad (a \in X,\ q_i \in Q).$$

- ბიუხის ავტომატი. ბიუხის ავტომატი არის შემდეგი ხუთეული $(S, \Sigma, S_0, ¢, F)$, სადაც
    - Σ არის სიმბოლოთა სასრული სიმრავლე (ალფავიტი);
    - S არის მდგომარეობათა სიმრავლე;
    - $S_0 \subseteq S$ არის საწყისს მდგომარეობათა სიმრავლე, რომელიც დერეტმინირებული შემთხვევისათვის შეიცავს მხოლოდ ერთ ელემენტს;
    - $¢ : S \times \Sigma \to 2^s$ გადასვლათა ფუნქცია (დეტერმინირებული ავტომა–ტისათვის $¢ : S \times \Sigma \to S$);
    - $F \subseteq S$ არის შესაძლო ფინალური მდგომარეობათა სიმრავლე.

ბიუხის ავტომატი ამოიცნობს $\Omega(¡)$ ზესიტყვას, თუ ამ უსასრულო სიტყვის კითხვისას ავტომატი უსასრულოდ გადის ერთი და იგივე დასაშვებ მდგომარეობას. $\Omega(¡)$ ზესიტყვების სიმრავლეს, რომელიც ამოცნობილია ბიუხის ავტომატის მიერ ეწოდება $\Omega(\Pi)$ ენა. $\Omega(\Pi)$ ენას ($L \subseteq \Sigma^{\Omega}$) ეწოდება ბიუხი – აღქმადი ენა, თუ არსებობს ბიუხის ავტომატი , რომელიც უშვებს L. ბიუხის ავტომატები ექვივალენტურია, თუ მის მიერ აღქმადი ენები ერთმანეთს ემთხვევა. რადგანა ბიუხის ავტომატი გამოიყენება უსასრულო $\Omega(\Pi)$ ენის მოსაცემად, ამიტომ მას ასევე Ω ავტომატებსაც უწოდებენ.

2.2˚.2 ავტომატები და გრაფები

ავტომატების შესწავლისას მოსახერხებელია ვისარგებლოთ გრაფთა თეორიით. ავტომატი $¶ = \langle Q, X, Y, \Psi, \Phi \rangle$ შეიძლება გამოვსახოთ გრაფის საშუალებით, რომლის წვეროები წარმოადგენს Q ალფავიტის სიმბოლოებს(ავტომატის მდგომარეობები). ამასთან ავტომატის ყოველ ბრძანებას $q_i\, x_r \to q_j\, y_s$ შეესაბამება გრაფის წიბო, რომელიც q_i წვეროს q_j წვეროსთან აკავშირებს და ეს წიბო მონიშნულია $x_r y_s$ წყვილით(x_r საწყისი ჭდე, ხოლო y_s საბოლოო ჭდე).

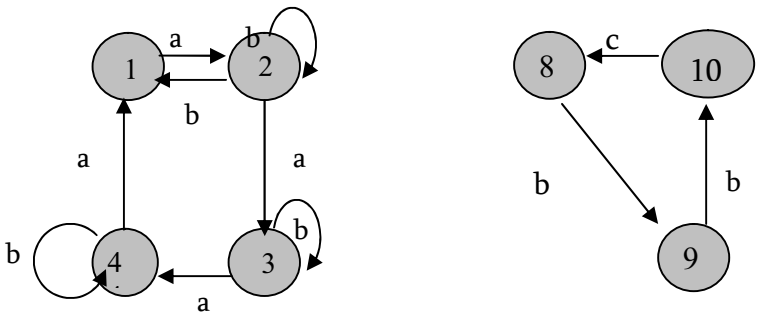
შევთანხმდეთ, რომ გრაფს ეწოდება დიაგრამა Q და A ალფავიტებში, თუ გრაფის წვეროები მონიშნულია სიმბოლოებით Q–დან(ანუ Q ალფავიტის ყოველი სიმბოლო შეესაბამება გრაფის მხოლოდ ერთ წვეროს), ხოლო წიბოები მონიშნულია A ალფავიტის სიმბოლოებით(არ არის სავალდებულო, რომ A ალფავიტის ყველა სიმბოლო

გამოყენებული იყოს წიბოების მოსანიშნად და ამასთან შესაძლებელია, რომ სრულიად სხვადასხვა წიბოები მონიშნული იყოს ერთი და იგივე სიმბოლოებით).

ამრიგად ნებისმიერი $\P = \langle Q, X, Y, \Psi, \Phi \rangle$ ავტომატი შეიძლება მოცემული იყოს დიაგრამით Q, $X \times Y$ ალფავიტების ფარგლებში, მაგრამ პირიქით არ არის სავალდებულო – ანუ შეიძლება არსებობდეს დიაგრამა Q, $X \times Y$ ალფავიტების ფარგლებში, მაგრამ ის არ წარმოადგენდეს არანაირ ავტომატს(ე.ი. ნებისმიერი დიაგრამა ავტომატს არ წარმოადგენს).

იმისათვის, რომ დიაგრამა აუცილებლად შეესაბამებოდეს რაიმე ავტომატს, საჭიროა დაცული იყოს შემდეგი პირობები:

1. დიაგრამაში არ უნდა არსებობდეს ორი წიბო, ერთი და იგივე შემავალი(საწყისი) ჭდით, რომელიც გამოდის ერთი და იმავე წვეროდან(ცალსახობის პირობა).
2. ნებისმიერი q წვეროსთვის და ნებისმიერი x შემავალი სიმბოლოსათვის, არსებობს ისეთი წიბო, რომელიც მონიშნულია x–ით და რომელიც გამოდის q წვეროდან(სასრულობის პირობა).



ა ბ

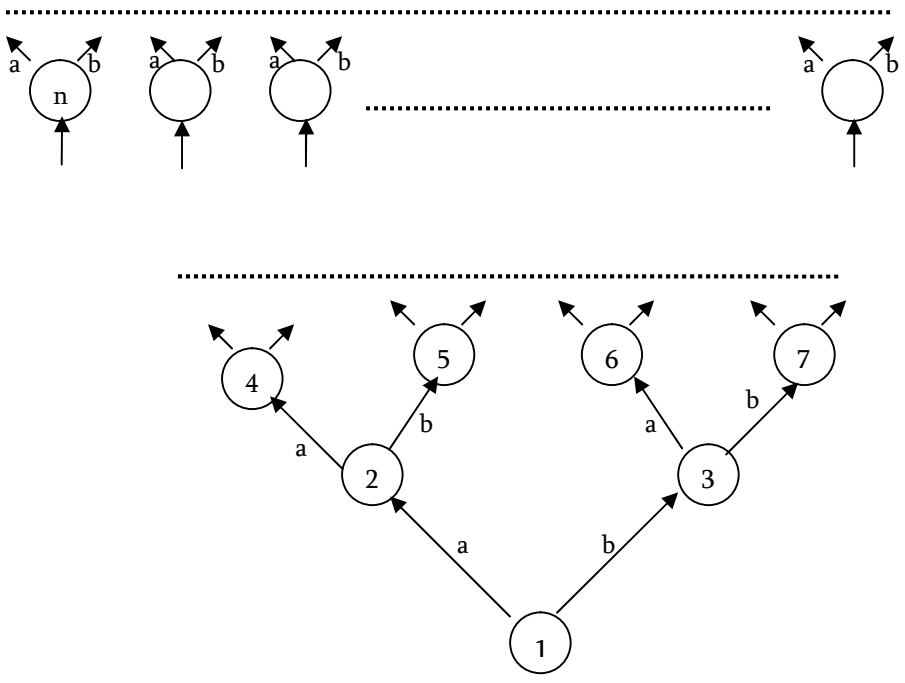
ზემოთ ნათქვამიდან ცხადია როგორ უნდა განისაზღვროს დიაგრამა ავტომატების განსაკუთრებული ვარიანტებისათვის. მაგალითად, ავტომატი გამოსავლის გარეშე მოიცემა დიაგრამით, სადაც წიბოები მონიშნულია საწყისი ალფავიტის ასოებით ავტომატურობის წესების დაცვით. ასეთ დიაგრამებს ავტომატების გრაფი ეწოდება. ა ნახაზზე ნაჩვენები დიაგრამა არ შეესაბამება ავტომატს, რადგან მასში დარღვეულია ცალსახობისა და სასრულობის პირობები. თუ 1–ლი წვეროდან რომელიმე წვეროსაკენ გავავლებთ წიბოს და მას მოვნიშნავთ b ასოთი და ამასთანავე მოვაცილებთ მე–2 წვეროდან გამოსულ რომელიმე წიბოს, რომელიც მონიშნულია b ასოთი, მაშინ

მივიღებთ ავტომატს გამოსავლის გარეშე. ანუ ასე შეცვლილ დიაგრამას შეიძლება ვუწოდოთ ავტომატური გრაფი.

დაბრკოლებებით ავტომატის შესაბამისი დიაგრამა ხასიათდება იმით, რომ ყველა წიბო, რომელიც გამოდის ერთი რომელიმე წვეროდან მონიშნულია ერთი და იგივე გამოსასვლელი ჭდით.

მურის ავტომატის შესაბამისი დიაგრამა კი ხასიათდება იმით, რომ ყველა წიბო, რომელიც შედის რომელიმე წვეროში, მონიშნულია ერთი და იგივე გამოსასვლელი ჭდით.

ვითვალისწინებთ რა კავშირს ავტომატებსა და დიაგრამებს შორის, შეიძლება გარკვეული ტერმინოლოგიური თავისუფლება გამოვიყენოთ: ზოგჯერ დიაგრამის წვეროებს მის მდგომარეობას უწოდებენ, ხოლო ყველა წვეროების სიმრავლეს დიაგრამის მაკრომდგომარეობას. ცხადია, რომ სასრული ავტომატი მოიცემა სასრული დიაგრამის მეშვეობით, ხოლო უსასრულო ავტომატი, შესაბამისად უსასრულო დიაგრამით. მაგალითად, განვიხილოთ უსასრულო დიაგრამა, რომელიც ხეს წარმოადგენს. ხის ყოველი წვეროდან გამოდის ზუსტად m ცალი წიბო და შესაბამისად ისინი მონიშნულია საწყისი $X=\{x_1, x_2, ..., x_m\}$ ალფავიტის ასოებით(ნახაზზე განხილულია ორ ასოანი $X=\{a,b\}$ალფავიტი). ეს დიაგრამა განსაზღვრავს უსასრულო ავტომატს, გამოსავლის გარეშე(მდგომარეობათა სიმბოლოებს როლში გვევლინება რიცხვები 1,2,3,. . .).



თუ დიაგრამის წიბოებს დამატებით დავტვირთავთ რომელიმე Y საბოლოო ალფავიტის სიმბოლოებით მივიღებთ დიაგრამას, რომელიც შეესაბამება ავტომატს გამოსასვლელით.

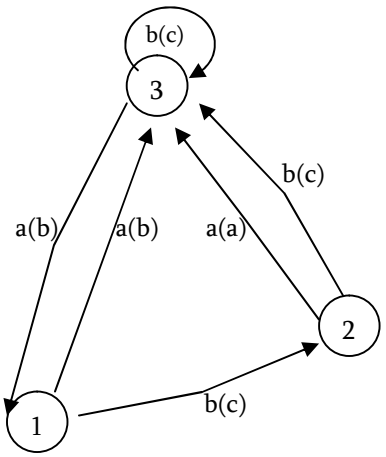
როგორც წესი, G დიაგრამაში გზა ~ წვეროდან წვერომდე, მოიცემა სასრული $\sim = \sim_1, A_1, \sim_2, A_2, \sim_3, A_3, \ldots, \sim_n, A_n, \sim_{n+1} =$ მიმდევრობით, სადაც ნებისმიერი $i \le n$ A_i არის წიბო, რომელიც $\sim_i$ წვეროს $\sim_{i+1}$ წვეროსთან აერთებს. თუ $\sim =$, მაშინ საქმე გვაქვს ციკლურ გზასთან; თუ $\sim_1, \sim_2, \sim_3, \ldots, \sim_n$ (აუცილებელი არ არის ამ მიმდევრობაში ფიგურირებდეს $\sim_{n+1}$) წვეროები წყვილ–წყვილად განსხვავებულია, მაშინ საქმე გვაქვს უბრალო გზასთან. ანალოგიურად განისაზღვრება ზეგზა G დიაგრამაში, მხოლოდ $\sim = \sim_1, A_1, \sim_2, A_2, \sim_3, A_3, \ldots, \sim_j, A_j, \sim_{j+1}, \ldots$ მიმდევრობა, ბუნებრივია უსასრულო მიმდევრობაა, ხოლო A_i ერთმანეთან აერთებს $\sim_i$ და $\sim_{i+1}$ წვეროება ($i = 1, 2, \ldots$).

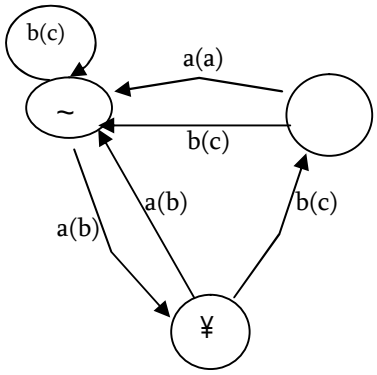
G დიაგრამაში მისი გზების(ზეგზების) ნებისმიერ სიმრავლეს ეწოდება ამ დიაგრამის ნაკადი(ზენაკადი). დიაგრამაში ყოველი ¡ გზისათვის(ზეგზისთვის) ცალსახად არის განსაზღვრული $\mu(¡)$ სიტყვა(ზესიტყვა) X ალფავიტში. ეს სიტყვა მიიღება თუკი ამოვწერთ ყველა ასოებს, რომლითაც მონიშნულია ¡ გზის(ზეგზის) ყველა წიბოები. ამათან დიაგრამის ნებისმიერი Π გზა(ზეგზა) ასოცირებულია $\mu(\Pi)$ ენასთან($\Omega(\Pi)$ ზეენასთან): ის შედგება ზუსტად იმდენი სიტყვისაგან(ზესიტყვისაგან), რომელიც შეესაბამება გზებს(ზეგზებს) Π სიმრავლიდან. ვიტყვით, რომ Π ნაკადი(ზენაკადი) წარმოადგენს $\mu(\Pi)$ ენას($\Omega(\Pi)$ ზეენას).

ორ დიაგრამას ეწოდება იზომორფული, თუ ისინი გარდაიქმნებიან ერთმანეთში წვეროების გადანომვრის შეცვლით, ბუნებრივია ამ დიაგრამების შესაბამისი ავტომატებიც იზომორფულია. $\langle Q, X, Y, \Psi, \Phi \rangle$ ტიპის ავტომატებისათვის იზომორფულობა ნიშნავს შემდეგს:

ავტომატები $¶_1 = \langle Q_1, X_1, Y_1, \Psi_1, \Phi_1 \rangle$ და $¶_2 = \langle Q_2, X_2, Y_2, \Psi_2, \Phi_2 \rangle$ იზომორფულია, თუ არსებობს Q_1 ალფავიტის Q_2 ალფავიტზე ურთიერთცალსახა ª ასახვა, რომლისთვისაც $\Psi_1(q, x) = ª^{-1}[\Psi_2[ª(q), x]]$ და $\Phi_1(q, x) = \Phi_2(ª(q), x)$.

მაგალითი: შემდეგი დიაგრამები იზომორფული დიაგრამებია, ბუნებრივია იზომორფულია შესაბამისი ავტომატებიც. ამ შემთხვევაში იზომორფიზმი განმარტებულია მდგომარეობების შემდეგი გარდაქმნის გამო: $1 \to ¥$, $2 \to$, $3 \to \sim$.





ყველა ავტომატების რიცხვი, რომლებიც მოცემულია $Q=\{q_1,...,q_k\}$, $X=\{x_1,...,x_v\}$, $Y=\{y_1,...,y_n\}$ ალფავიტებით მარტივად დათვლადია და ის ტოლია $(nk)^{vk}$ რიცხვის. ბუნებრივია, რომ ავტომატების თეორიის ბევრ ამოცანაში აზრი არა აქვს განვასხვავოთ ერთმანეთისაგან იზომორფული ავტომატები.

2.2˙3 ენების წარმოდგენა ავტომატის საშუალებით

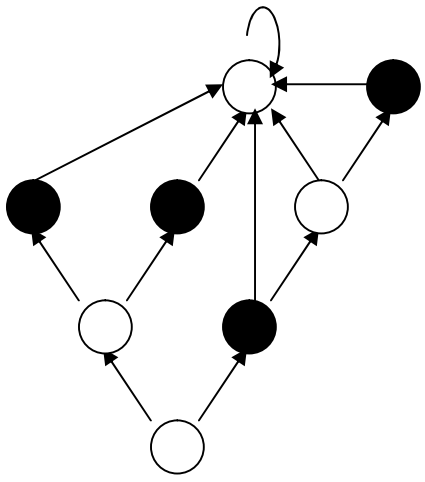
თუ გამოყენებულ ავტომატზე და მისი აწყობის წესებს არ დავადებთ არანაირ შეზღუდვებს, მაშინ ცხადზე ცხადია, რომ ნებისმიერი ენა წარმოდგენადია რაღაც ავტომატში, რომელიც შესაბამისი მოთხოვნებით არის აწყობილი. მართლაც, ვთქვათ i არის ენა, რომელიც განსაზღვრულია X ალფავიტზე, თავის მხრივ X ალფავიტში არის m ცალი სიმბოლო; შესაბამისი ინტერპრეტირებადი $\langle \P, q_0, Q` \rangle$ ავტომატი შეიძლება ავაწყოთ ასე: ავტომატის დიაგრამა წარმოადგენს უსასრულო ხეს, რომლის ყოველი წვეროდან გამოდის ზუსტად $m$ წიბო, რომლებიც თავის მხრივ მონიშნულია საწყისი ასოებით. ამ ხის ფესვს წარმოადგენს ავტომატის $q_0$ საწყისი მდგომარეობა. შემდეგ, $Q`$ სიმრავლეს ეკუთვნის ის და მხოლოდ ის q მდგომარეობები, რომლებისთვისაც i ენაში მოიძებნება ისეთი p სიტყვა, რომელსაც q_0 მდგომარეობიდან q მდგომარეობაში გადავყავართ.

გამოყენებულ ავტომატებზე და მათ მდგომარეობებზე შესაძლებელია სხვადასხვა შეზღუდვების დადება, რაც თავის მხრივ განსაზღვრავს ენებისა და ზეენების არატრივიალურ კლასებს. ასეთ კლასებს წარმოადგენს სასრულ ავტომატებში წარმოდგენილი ყველა ენების(სხვანაირად სასრულ–ავტომატური ენების) μ კლასი.

მაგალითი 1. ნებისმიერი ენა, რომელიც ერთადერთი $x = x(1)\,x(2)\ldots x(r)$ სიტყვისაგან შედგება, სასრულ–ავტომატურ ენას წარმოადგენს.

ინტერპრეტირებადი ავტომატი უნდა ავაგოთ $r+2$ მდგომარეობით: $q_1, q_2, \ldots, q_r, q_{r+1}, q_{r+2}$, სადაც q_1 საწყისი მდგომარეობაა, ხოლო q_{r+1} ერათადერთი ფინალური მდგომარეობა. ავტომატის ბრძანებებს აქვს სახე: $q_1 x(1) \to q_2, q_2 x(2) \to q_3, \ldots, q_r x(r) \to q_{r+1}$ და ამასთან ნებისმიერი სხვა $q_{\sim} x$ წყვილისათვის, რომელიც ჩამოთვლილი წყვილების მარცხენა მხარეებისაგან განსხვავებული, ადგილი აქვს ბრძანებას $q_{\sim} x \to q_{r+2}$. q_{r+2} მდგომარეობას გააჩნია განსაკუთრებულობა: ნებისმიერი საწყის ასოს(შესაბამისად ნებისმიერი საწყის სიტყვას) ის გადაჰყავს ისევ q_{r+2} მდგომარეობაში(ანუ იგივე მდგომარეობაში). მდგომარეობას, რომელსაც ახასიათებს ასეთი თვისება ეწოდება ჩიხური მდგომარეობა.

მაგალითი 2. ნებისმიერი სასრული ენა სასრულ–ავტომატური ენაა.



მოსახერხებელია ინტერპრეტირებადი ავტომატის დიაგრამის სახით წარმოდგენა. საილუსტრაციო ენად ავიღოთ ¡ $= \{00, 01, 1, 111\}$ ენა. ავაგოთ სიტყვათა ამ სიმრავლისათვის სასრული ხე. ამ ხის ყოველი გამუქებული წვერო შეესაბამება ¡ ენის რომელიღაცა ერთ სიტყვას. ამის შემდეგ ეს ხე უნდა შევავსოთ დიაგრამამდე წიბოების დამატებით, ხის ფესვი გამოხატავს საწყისს მდგომარეობას.

დაუმტკიცებლად მოვიყვანოთ შემდეგი თეორემა.

თეორემა. არსებობს ალგორითმები,რომლებიც:

(*) მოცემული $\langle \P, q_0, Q \rangle$ სასრული ავტომატისათვის აგებს სასრულ ავტომატს, რომელიც წარმოადგენს $\neg\mu(\P, q_0, Q)$ ენას;

(**) მოცემული $\langle \P', q'_0, Q' \rangle$ და $\langle \P'', q''_0, Q'' \rangle$–თვის, რომლებსაც აქვს საერთო X ალფავიტი, აგებს სასრულ ავტომატს $\langle \P, q_0, Q \rangle$, რომელიც წარმოადგენს $\mu(\P', q'_0, Q') \cup \mu(\P'', q''_0, Q'')$ ენას;

(***) მოცემული $\langle \P, q_0, Q' \rangle$ სასრული ავტომატისათვის და მოცემული Y ალფავიტისათვის, აგებს სასრულ ავტომატს, რომელიც წარმოადგენს $\mu(\P, q_0, Q' \times Y)$ ენას.

2.2 სასრული ავტომატები

განმარტება 2.2.1 A სასრული ავტომატი ეწოდება შემდეგ ხუთეულს :

$$A = (Q,\ \Sigma,\ ¢,\ q_0,\ F)\ ,$$

სადაც

Q – სასრული ავტომატის არასტრუქტურირებულ მდგომარეობათა არაცარიელი სასრული სიმრავლეა;

$q_0 \in Q$ – სასრული ავტომატის საწყისი მდგომარეობა;

$F \in Q$ – სასრული ავტომატის საბოლოო მდგომარეობათა არაცარიელი სასრული სიმრავლე;

Σ – სასრული ავტომატის საწყის სიმბოლოთა არაცარიელი, სასრული სიმრავლე;

$¢$ – სასრული ავტომატის ე.წ. გადასვლების ფუნქცია, რომელიც განმარტებულია შემდეგნაირად:

$$¢ \ :\ Q \times \Sigma \to \Im(Q).$$

(ანუ $¢$ ფუნქცია მდგომარეობისა და საწყისი სიმბოლოს წყვილებს ასახავს Q -ს ყველა ქვესიმრავლეთა სიმრავლეზე.)

ზემოთ განმარტებული ავტომატი, გამოიყენება როგორც ამოცნობის მოწყობილება შემდეგნაირად: შეტანის ლენტაზე ჩაწერილია საწყისს სიმბოლოთა მოცემული ჯაჭვი(ანუ Σ^+ სიმრავლის ელემენტები); ავტომატის მიერ ლენტა

თანმიმდევრობით გადაიხედება(ანუ “იკითხება”) მარცხნიდან მარჯვნივ დაბრუნების გარეშე(ავტომატს არ გააჩნია გამოტანის მოწყობილობა).

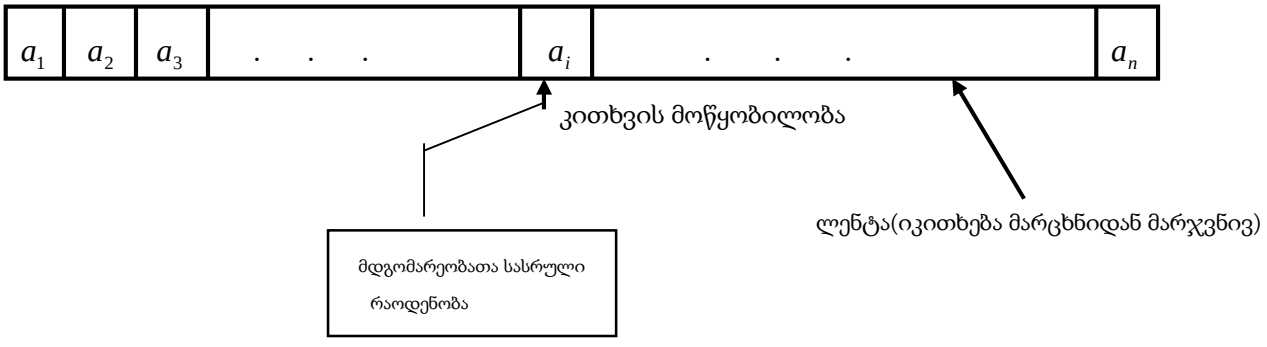
ამოცნობის დაწყებამდე A ავტომატი მდებარეობს საწყისს q_0 მდგომარეობაში და განიხილება პირველი საწყისი სიმბოლო(ვთქვათ a_1). დავუშვათ, რომ ზემოთ აღწერილი პროცესის მიხედვით, უკვე აღქმული(ამოცნობილი) იყო $a_1, a_2, ..., a_{i-1}$ $(i \geq 1)$ სიმბოლოები და ახლა განიხილება a_i-ური სიმბოლო, ამასთან A ავტომატი მდებარეობს q_{i-1} მდგომარეობაში($i = 1$-თვის არ არის ამოცნობილი არც ერთი სიმბოლო და განხილული იყო a_1).

a_i სიმბოლო ამოცნობადია A ავტომატის მიერ, თუ გადასვლათა $¢(q_{i-1}, a_i)$ ფუნქცია განმარტებულია და არაცარიელია. ამასთანავე, A ავტომატი გადადის q_{i-1} მდგომარეობიდან q_i მდგომარეობაში, სადაც $q_i \in ¢(q_{i-1}, a_i)$. ვინაიდან არ არსებობს წესი, რომელიც განმარტავდა $¢(q_{i-1}, a_i)$ სიმრავლის რომელი ელემენტი უნდა იყოს A ავტომატის შემდეგი მდგომარეობა, ამიტომ საქმე გვაქვს არადეტერმინირებულ ავტომატებთან.

თუ $¢(q_{i-1}, a_i)$ ფუნქცია ცარიელია, მაშინ a_i სიმბოლო ავტომატის მიერ ამოუცნობ სიმბოლოდ ითვლება და q_{i-1} მდგომარეობიდან არავითარი გადასვლა არ განხორციელდება. უფრო მეტიც A ავტომატი კარგავს უნარს ამოიცნოს დარჩენილი საწყისი სიმბოლოები.

განმარტება 2.2.2. საწყისს სიმბოლოთა $a_1, a_2, ..., a_n \in \Sigma^+$ $(n \geq 1)$ ჯაჭვი აღქმადია (ამოცნობადია) A ავტომატის მიერ მაშინ და მხოლოდ მაშინ, თუ არსებობს გადასვლათა მიმდევრობა $q_0, q_1, ..., q_n$ $(n \geq 1)$, ისეთი, რომ $q_{i+1} \in ¢(q_i, a_{i+1}), i = 0,1,...,n-1$ და $q_n \in F$.

სქემატურად, სასრული ავტომატი შეიძლება შემდეგნაირად იყოს გამოსახული:



შევნიშნოთ, რომ $a_1, a_2, ..., a_n \in \Sigma^+ (n \geq 1)$ სტრიქონის $a_i, a_{i+1}, ..., a_j$ $(i \geq 1, i \leq j \leq n)$ ქვესტრიქონი ამოცნობადია, თუ $q_{i-1} \in ¢(q_{i-2}, a_{i-1})$, $q_{i-1} = q_0$ და $q_j \in ¢(q_{j-1}, a_j)$, $q_j \in F$. ეს აიხსნება იმით, რომ ლენტის კითხვისას ავტომატი შეიძლება დაუბრუნდეს საწყისს მდგომარეობას, ასევე შეიძლება გადავიდეს ბოლო მდგომარეობაში, მანამ სანამ წაკითხული იქნება საწყისს სიმბოლოთა ბოლო a_n სიმბოლო.

განმარტება 2.2.3. ჯაჭვთა (კომბინაციათა) ერთობლიობას, რომელსაც აღიარებს სასრული ავტომატი A, ეწოდება რეგულარული სიმრავლე. იგი აღინიშნება ასე $T(A)$ და განიმარტება შემდეგნაირად:

$$T(A) = \{ X \mid X \in \Sigma^+ \wedge \hat{¢}(q_0, X) \cap F = \varnothing ,$$

სადაც ფუნქცია $\hat{¢}$ განისაზღვრება რეკურსიულად:

$$\hat{¢}(q, a_1 a_2 ... a_n) = \bigcup_p ¢(p, a_n), \quad n \geq 2 ,$$

$$P \in \hat{¢}(q, a_1 a_2 ... a_{n-1}) \text{ და } \hat{¢}(q, a_1) = ¢(q, a_1) .$$

2.3. კავშირი სასრულ ავტომატებსა, ფორმალურ გრამატიკებსა და ენებს შორის

[6]–ში დამტკიცებულია ორი თეორემა :

თეორემა 2.3.1. თუ $A = (Q, \Sigma, ¢, q_0, F)$ სასრული ავტომატია, მაშინ არსებობს წრფივი გრამატიკა $G = (N, \Sigma', P, S)$ ისეთი, რომ

$$L(G) = T(A).$$

დამტკიცება: თავდაპირველად განვსაზღვროთ N, Σ', P სიმრავლეები და S საწყისი სიმბოლო, ხოლო შემდეგ ვაჩვენოთ, რომ $L(G)$ ენა ემთხვევა იმ სიმრავლეს, რომელსაც აღიარებს A ავტომატი.

(1) $N = Q; \Sigma' = \Sigma; S = q_0 \in Q.$

შექმნის წესების სიმრავლე შემდეგია:

დავუშვათ, რომ $C \in ¢(B, b)$ ყველა $B, C \in Q$ და $b \in \Sigma$-თვის, მაშინ

- თუ $C \notin F$, ეს ნიშნავს, რომ $B \to bC \in P$,
- თუ $C \in F$, ეს ნიშნავს, რომ $B \to b \in P$,
- თუ $C \in F \wedge (\exists\, c \in \Sigma, D \in Q)\,[\, D \in ¢\,(C,c)]$, ეს ნიშნავს, რომ $B \to bC \in P$ და $B \to b \in P$.

ამრიგად, ჩვენ ნამდვილად მივიღეთ წრფივი გრამატიკა.

(2) ახლა საჭიროა დავამტკიცოთ, რომ $L(G) = T(A)$.

დავუშვათ, რომ $x \in T(A)$. 2.2.3 განმარტების თანახმად $x \neq £$, ამიტომაც $|x| \geq 1$. გარდა ამისა, ცნობილია, რომ $\hat{¢}(q_0, x) \cap F \neq \varnothing$ და არსებობს მდგომარეობათა $q_0, q_1, \ldots, q_n$ $(n \geq 1)$ მიმდევრობა, ისეთი, რომ x აღქმადია A ავტომატის მიერ (2.2.2 და 2.2.3 განმარტებების თანახმად). ეს მიმდევრობა ერთმნიშვნელოვნად განსაზღვრავს შექმნის წესების ერთობლიობას, რომლებიც თავის მხრივ განმარტავენ ერთ და მხოლოდ ერთ x გამოყვანას S საწყისი სიმბოლოდან დაწყებული. ამრიგად $x \in L(G)$.

დავუშვათ, რომ $x \in L(G)$. რადგან $x \neq £$, ამიტომ $|x| \geq 1$. მაშინ არსებობს შექმნის წესების არაცარიელი მიმდევრობა, ისეთი რომ $S \overset{*}{\underset{G}{\Rightarrow}} x$. თავის მხრივ, ეს მიმდევრობა განსაზღვრავს მოცემული A ავტომატის გადასვლათა მიმდევრობას, დაწყებულს $q_0 = S$ საწყისი მდგომარეობიდან. რადგან მითითებულ მიმდევრობაში უკანასკნელ შექმნის წესს აქვს სახე $B \to b$, ამიტომ ბოლო გადასვლას A ავტომატი გადაჰყავს საბოლოო მდგომარეობაში. ამრიგად $x \in T(A)$.

რ.დ.გ.

სამართლიანია შებრუნებული თეორემაც

თეორემა 2.3.2. თუ $G = (N, \Sigma, P, S)$ წრფივი გრამატიკაა, მაშინ არსებობს სასრული ავტომატი $A = (Q, \Sigma', ¢, q_0, F)$, ისეთი რომ

$$T(A) = L(G).$$

დამტკიცება: თავდაპირველად განვსაზღვროთ Q, Σ', F სიმრავლეები და q_0 საწყისი მდგომარეობა, ასევე ¢ ფუნქცია, ხოლო შემდეგ ვაჩვენოთ, რომ $T(A)$ სიმრავლე ემთხვევა G გრამატიკაზე განმარტებულ ენას.

(1) $Q = N \cup \{f\}$, სადაც $f \notin N$; $F = \{f\}$; $\Sigma' = \Sigma$; $q_0 = S \in N$.

ჩვენ მივიღეთ ავტომატის მდგომარეობათა სასრული მიმდევრობა და ასევე საწყისს სიმბოლოთა სასრული მიმდევრობა, და ასევე ერთადერთი საბოლოო მდგომარეობა.

$B \in N$ -თვის და $b \in \Sigma$ -თვის დავუშვათ, რომ $B \to bC_i \in P$ $(i = 0,1,2,\ldots,n \geq 0)$.

თუ $n = 0$ და $B \to b \in P$, მაშინ $¢(B, b) = \{f\}$.

თუ $n \neq 0$, მაშინ $¢(B,b) = \{C_1, C_2, \ldots, C_n\}$.

სხვა, ყველა დარჩენილ შემთხვევაში $¢(B, b) = \varnothing$.

(2) ახლა აუცილებელია დავამტკიცოთ, რომ $T(A) = L(G)$. ეს შეიძლება ისეთივე მეთოდით, როგორც ეს გავაკეთეთ 2.3.1 თეორემის დამტკიცების დროს.

რ.დ.გ.

დაუმტკიცებლად მოვიყვანოთ კიდევ ერთი თეორემა.

თეორემა 2.3.3. თუ A არადეტერმინირებული ავტომატია, მაშინ აუცილებლად არსებობს დეტერმინირებული ავტომატი A' ისეთი, რომ $T(A) = T(A')$.

თუ გადასვლათა $¢$ ფუნქცია შეიცავს მხოლოდ ერთ მდგომარეობას, მაშინ საქმე გვაქვს დეტერმინირებულ ავტომატთან. რეგულარული სიმრავლე შეიძლება განსაზღვრული იყოს როგორც სიმრავლე დაშვებული დეტერმინირებული ავტომატის მიერ. მაშინ ზემოთ მოყვანილი თეორემა ადგენს, რომ სიმრავლე, რომელიც დაშვებულია არადეტერმინირებული ავტომატის მიერ, ამავე აზრით წარმოადგენს რეგულარულ სიმრავლეს.

2.3.1 და 2.3.2 თეორემების შედეგებზე დაყრდნობით შესაძლებელი გახდა შემდეგი თეორემის დამტკიცება:

თეორემა 2.3.4. ნებისმიერი სინტაქსური გარჩევის ალგორითმს (რომელიმე კონკრეტული G გრამატიკის ფარგლებში) შეესაბამება სასრული ავტომატი.

დამტკიცება. ნებისმიერი სინტაქსური ანალიზის ალგორითმი რეალიზებას უკეთებს ფორმალური ენის (კონკრეტული G გრამატიკის ფარგლებში) სინტაქსისს, ანუ

$$P \in \Im = \{ (¶, \quad) \mid ¶ \in V^+ \times N \times V^* \wedge \quad \in V^+ \}.$$

სიმრავლეს. სადაც N არის G გრამატიკის ტერმინალური სიმბოლოების სასრული სიმრავლე, V კი არის G გრამატიკაზე აგებული ფორმალური ენის ლექსიკონი.

2.3.1 თეორემის თანახმად არსებობს $G = (N, \Sigma, P, S)$ გრამატიკის შესაბამისი სასრული ავტომატი $A = (Q, \Sigma', ¢, q_0, F)$ ისეთი, რომ

$$N = Q, \quad \Sigma = \Sigma', \quad S = q_0 \quad \text{და} \quad L(G) = T(A) \ .$$

(2), (3) და *(4)* დამოკიდებულებების თანახმად $L(G)$ ფორმალური ენის გამოყვა¬ნის წესები P სიმრავლეს ემთხვევა.

აქედან გამომდინარე,

$$P = T(A)$$

რ. დ. გ.

2.4 სასრული ავტომატების სისტემა

წინა თავში დაპროგრამების ენებს ჩვენ გავნიხილავთ, როგორც ფორმალურ ენებს, რომლებიც შეიძლება განმარტებული იყოს სასრული მეთოდით. კერძოდ ასეთი ენების ელემენტები, ანუ პროგრამები განიხილება როგორც მე–2 ტიპის გრამატიკები, რომლის შექმნის წესები შეიძლება მოცემული იყოს ბეკუს–ნაურის ფორმალიზმით. ამავე თავში დამტკიცებული იყო, რომ მე–3 ტიპის ნებისმიერ გრამატიკას შეესაბამება სასრული ავტომატი და პირიქით ნებისმიერ სასრულ ავტომატს შეესაბამება მე–3 ტიპის გრამატიკა. ამ პარაგრაფში ლაპარაკი იქნება სასრული ავტომატების სისტემაზე, იმდენად რამდენადაც მე–2 ტიპის გრამატიკას შეესაბამება გარკვეული თვისების მქონე სასრული ავტომატების სიმრავლე, ანუ სისტემა.

განმარტება 2.4.1 სასრულ ავტომატთა სისტემა S წარმოადგენს სასარულ ავტომატთა არაცარიელ სიმრავლეს $\{A_1, A_2, \ldots, A_n\}$, რომელსაც ახასიათებს შემდეგი თვისებები:

- ზუსტად ერთი ავტომატი A_1 წარმოადგენს ე.წ. საწყისს ავტომატს;
- ნებისმიერი A_i ავტომატი, თავის მხრივ არის ხუთეული $\left(Q_i, \Sigma_i \cup Q`, ¢_i, q_{0i}, F_i\right)$, სადაც Q, Σ, q_0 და F აქვთ იგივე შინაარსი, რაც *2.2.1* განმარტებაში, ხოლო გადასვლების ფუნქცია განისაზღვრება შემდეგი სახით:

  $$¢_i : Q_i \times (\Sigma_i \cup Q`) \to \Im(Q_i)$$

  სადაც $Q` = \bigcup_{i=1}^{n} \{lab\ (q_{0i})\}$. ჩვენ აქ გამოვიყენებთ ე.წ. დაჭდევების ფუნქციას $lab(x) = (x = £ \to £,\ x \in \Sigma \to x,\ x = q_{0i} \to ' q_{oi} ')$, რომელიც ყოველ საწყის მდგომარეობას შეუსაბამებს შესაბამისს ჭდეს, რომელიც მხოლოდ და მხოლოდ კონკრეტულ საწყისს მდგომარეობას შეესაბამება, ანუ $lab^{-1}('q_{0i}') = q_{0i}$;

- თუ Q_i და Q_j – A_i და A_j ავტომატების მდგომარეობათა სიმრავლეა, მაშინ $Q_i \cap Q_j = \varnothing$;
- თუ Σ_i არის A_i ავტომატის საწყისს სიმბოლოთა სიმრავლე, $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$ და $Q = \bigcup_{i=1}^{n} Q_i$, მაშინ A_i ავტომატის გადასვლათა ფუნქციას აქვს სახე:

$$¢_i : Q \times (\Sigma \cup Q`) \to \Im(Q)$$

S სისტემიდან ნებისმიერი სასრული ავტომატი შეიძლება განხილული იყოს, როგორც ამომცნობი მოწყობილება, ზუსტად ისეთი, როგორიც აღწერილია წინა პარაგრაფში. თუმცა, რადგან $¢_i$ არგუმენტი შედგება წყვილისაგან, რომლის მეორე ელემენტი შეიძლება იყოს ამ სისტემის ავტომატის საწყისი მდგომარეობის ჭდე, ამიტომ შეიძლება ჩავთვალოთ, რომ ყოველ ავტომატს გააჩნია საკუთარი შეტანის ლენტა, რომელზეც მოთავსებულია სიმბოლოთა ჯაჭვი $(\Sigma_i \cup Q`)^+$ სიმრავლიდან. ან შესაძლებელია მთელი სისტემა განვიხილოთ როგორც ერთი მთლიანობა და ამასთან სისტემის შეტანის ლენტაზე მოთავსებულია სიმბოლოთა ჯაჭვი Σ^+ სიმრავლიდან, სადაც $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$.

პირველი ინტერპრეტაციისას ცალკე აღებული A_i ავტომატი 2.2.3 განმარტების თანახმად აღიარებს $T(A_i)$ ენას, რომელიც თავის მხრივ $(\Sigma_i \cup Q`)^+$ რეგულარული სიმრავლის ქვესიმრავლეა. მეორე ინტერპრეტაციისას შესაძლებელია საუბარი $T(S)$ ენაზე, რომელსაც თავის მხრივ უშვებს მთელი სისტემა.

ახლა განვმარტოთ ენა, რომელსაც აღიარებს სასრული ავტომატების სისტემა. ეს პროცესი წინა პარაგრაფში განხილული პროცესის ანალოგიური იქნება. ამასთან დავუშვათ, რომ $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$.

- დამოკიდებულება $\underset{G}{\Rightarrow}$, რომელიც მოცემულია V^* სიმრავლეზე განისაზღვრება შემდეგნაირად: $¥_1 \underset{G}{\Rightarrow} ¥_2$ (სადაც $¥_1, ¥_2 \in V^+$), თუ არსებობს $¢_1, ¢_2 \in V^*$ და შექმნის წესი $¶ \to \in P$, ისეთი რომ

$$¥_1 = ¢_1 ¶ ¢_2 \quad \wedge \quad ¥_2 = ¢_1 \quad ¢_2 \qquad (2)$$

ფაქტიურად, საქმე გვაქვს ჯაჭვების ე.წ. გარდაქმნასთან: ჯაჭვი $¥_1$ გარდაიქმნება $¥_2$ ჯაჭვში, თუ $¥_1$ ჯაჭვში შევცვლით ერთ ან რამდენიმე სიმბოლოს(ანუ ¶ ქვეჯაჭვს) ჯაჭვით. სიმბოლოები, რომლებიც ესაზღვრება ¶ ქვეჯაჭვს (ანუ $¢_1, ¢_2$) რჩება უცვლელი.

პირველი ნაბიჯი

- დამოკიდებულება $\underset{S}{\Rightarrow}$, რომელიც მოცემულია $(\Sigma \cup Q')^*$ სიმრავლეზე, განიმარტება ასე:

$¥_1 \underset{S}{\Rightarrow} ¥_2$, თუ არსებობს $¶_1, ¶_2 \in (\Sigma \cup Q')^*$, $'A' \in Q'$ და $\in T(B)$, ისეთი, რომ $¥_1 = ¶_1 'A' ¶_2 \wedge ¥_1 = ¶_1 \ ¶_2$ (სადაც B არის S სისტემის სასრული ავტომატი, რომლის საწყისი მდგომარეობაა A_1). ეს დამოკიდებულება არასიმეტრიული და არატრანზიტულია, ჩავთვალოთ, რომ იგი რეფლექსურია. აქ ჩვენ საქმე გვაქვს ჯაჭვების გარდაქმნასთან: $¥_1$ ჯაჭვი გარდაიქმნება $¥_2$ ჯაჭვად, თუ მასში მხოლოდ ერთ სიმბოლოს(რომელიც თავის მხრივ ეკუთვნის დაჭდევებულ საწყის მდგომარეობათა სიმრავლეს), შევცვლით ისეთი ჯაჭვით, რომელსაც აღიარებს ერთ–ერთი ავტომატი.

მეორე ნაბიჯი

- დამოკიდებულება $\overset{*}{\underset{S}{\Rightarrow}}$, რომელიც მოცემულია $(\Sigma \cup Q')^*$ სიმრავლეზე, განიმარტება ასე:

$$¥_0 \overset{*}{\underset{G}{\Rightarrow}} ¥_n, \quad \text{თუ } ¥_0, ¥_1, ..., ¥_n \in (\Sigma \cup Q')^* \ (n \geq 0) \text{ და } ¥_{i-1} \underset{G}{\Rightarrow} ¥_i \qquad (i = 1,2,...,n)$$

ამბობენ, რომ $¥_0, ¥_1, ..., ¥_n$ მიმდევრობის სიგრძე n–ის ტოლია. $\overset{*}{\underset{S}{\Rightarrow}}$ დამოკიდებულება რეფლექსურია, იგი წარმოადგენს ტრანზიტულ ჩაკეტვას $\overset{*}{\underset{S}{\Rightarrow}}(\Sigma \cup Q')^*$.

მესამე ნაბიჯი

განმარტება 2.4.2 $T(S)$ ენა, დაშვებული სასრული ავტომატების S სისტემის მიერ, განიმარტება ასე:

$$T(S) = \bigcup_{¥ \in T(A_1)} \left\{ x \mid ¥ \overset{*}{\underset{S}{\Rightarrow}} x \wedge \ x \in \Sigma^* \right\}.$$

S სისტემიდან ნებისმიერ სასრულ A_i ავტომატს შეესაბამება G_i წრფივი გრამატიკა, ისეთი, რომ $L(G_i) = T(A_i)$; G_i გრამატიკის არატერმინალები შეესაბამება A_i ავტომატის მდგომარეობებს, ხოლო ტერმინალური სიმბოლოები კი $\Sigma_i \cup Q`$ სიმრავლის ელემენტებს. $Q`$ სიმრავლის ტერმინალური სიმბოლოები შეესაბამება წრფივი გრამატიკის ერთადერთ საწყისს სიმბოლოს და ასევე მასთანაა დაკავშირებული რეგულარული სიმრავლეც.

თეორემა 2.4.1. სასრულ ავტომატთა ნებისმიერ S სისტემას შეესაბამება კონტექსტურად თავისუფალი გრამატიკა, ისეთი, რომ

$$L(G) = T(S).$$

დამტკიცება. დავუშვათ, რომ $A_i \in S$ $(i = 1,2,...,n)$ სასრულ ავტომატს შეესაბამება G_i წრფივი გრამატიკა, რომელიც მოიცავს შექმნის წესების P_i ნაკრებს, განვმარტოთ $G = (N, \Sigma, P, S`)$ შემდეგნაირად:

Σ იყოს 2.4.1 განმარტებაში მოყვანილი ცალკეული სასრული ავტომატის საწყის სიმბოლოთა სიმრავლეების გაერთიანება,

$N = Q$, სადაც Q არის ცალკეული ავტომატის მდგომარეობათა სიმრავლის გაერთიანება;

$P = \bigcup_{i=1}^{n} P_i`$, სადაც $P_i` = \{x \to lab^{-1}(y)\ z \mid x \to yz \in P_i\}$;

$S` = A$, სადაც $A \in Q_1$ და იგი არის საწყისი ავტომატის საწყისი მდგომარეობაა.

რადგანაც $L(G_1) = T(A_1)$, ამიტომ 2.4.2 განმარტების თანახმად $L(G) = T(S)$.

რ.დ.გ.

მაგალითი. განვიხილოთ სისტემა, რომელიც შედგება ორი სასრული ავტომატისაგან

$$S = \{A_1, A_2\}$$

$$A_1 = (\{I,1,2\}, \{'T','L'\}, ¢_1, I, \{1,2\})$$

$$A_2 = (\{L,3\}, \{a,b,c\}, ¢_2, L, \{3\})$$

სადაც $¢_1(1,'L') = \{1\}$ $¢_1(1,'T') = \{2\}$

$$¢_2(L,a) = ¢_2(L,b) = ¢_2(L,c) = \{3\}$$

ცხადია, რომ $T(A_1) = \{'L','L''I'\}$, $T(A_2) = \{a,b,c\}$ და $T(S) = \{a,b,c\}^+$.

წრფივი გრამატიკები, რომლებიც შეესაბამებიან A_1 და A_2 ავტომატებს, აქვთ შემდეგი სახე:

$$G_1 = (\{I,1\}, \{'L', 'I'\}, P_1, I)$$
$$G_2 = (\{L\}, \{a,b,c\}, P_2, L)$$

ხოლო

$$P_1: I \to 'L' \qquad P_2: L \to a$$
$$I \to 'L'1 \qquad L \to b$$
$$I \to 'I' \qquad L \to c$$

ასე, რომ

$$P_1^{`}: I \to L$$
$$I \to L1$$
$$I \to I$$

და $P_2^{`} = P_2$.

ეს ნიშნავს, რომ

$$G = (\{I, L, 1\}, \{a,b,c\}, P, I)$$

სადაც $P = P_1^{`} \cup P_2^{`}$ – კონტექსტურად– თავისუფალი გრამატიკაა, შექმნილი $\{a,b,c\}^+$–ზე.

თეორემა 2.4.2. ნებისმიერ კონტექსტურად თავისუფალ G გრამატიკას შეესაბამება სასრულ ავტომატთა S სისტემა ისეთი, რომ $T(S) = L(G)$.

დამტკიცება. დავუშვათ, რომ მოცემულ კონტექსტურად თავისუფალ გრამატიკაში $G = (N, \Sigma, P, S^{`})$ თითოეული არატერმინალი გვხვდება მარცხენა მხარეს m–ჯერ ($m \geq 1$), შემდეგი სქემის შესაბამისად:

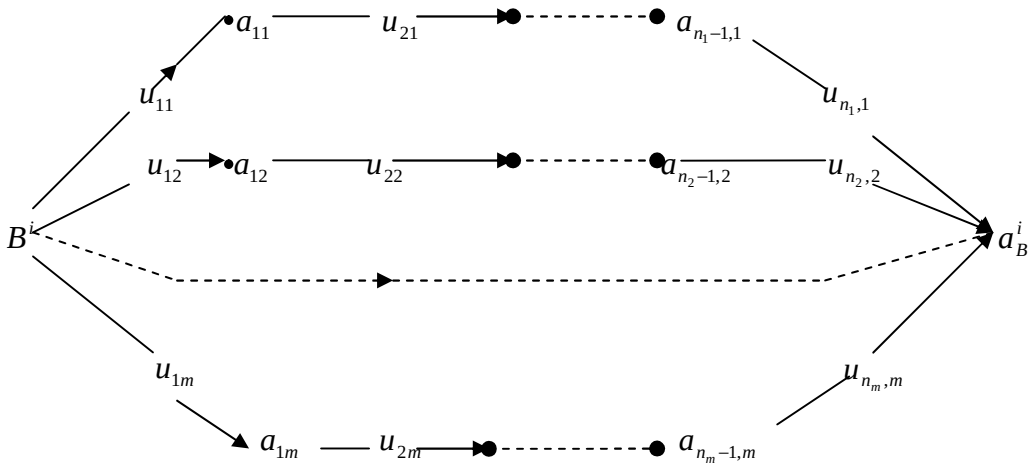
$$B^i \to u_{11}^i u_{21}^i \ldots u_{n_1 1}^i$$
$$B^i \to u_{12}^i u_{22}^i \ldots u_{n_2 2}^i$$
$$\ldots\ldots\ldots\ldots\ldots$$
$$B^i \to u_{1m}^i u_{2m}^1 \ldots u_{n_m m}^i$$

სადაც $u_{jk}^i \in N \cup \Sigma$. ავღნიშნოთ ყველა u^i–ების სიმრავლე U_i–ით, ყველა U_i–ის გაერთიანება U –ით. ზემოთ მოყვანილი წესები განსაზღვრავენ წესების შემდეგ P^i მატრიცას:

$$\begin{bmatrix} B & B & \dots & B \\ u_{11} & u_{12} & \dots & u_{1m} \\ \sim_{11} & \sim_{12} & \dots & \sim_{1m} \\ u_{21} & u_{22} & \dots & u_{2m} \\ \sim_{21} & \sim_{22} & \dots & \sim_{2m} \\ . & . & . & . \\ u_{n_1 1} & u_{n_2 12} & \dots & u_{n_m m} \\ \sim_{n_1 1} & \sim_{n_2 12} & \dots & \sim_{n_m m} \end{bmatrix}$$

რომელშიც ზედა ინდექსი i გამოტოვებულია. ამ მატრიცისათვის იგულისხმება, რომ $\sim^{i}_{jk} \notin U$ და ყველა $\sim$ წყვულ–წყვილად განსხვავებულია, მხოლოდ შემდეგი პირობაა გამონაკლისი $\sim^{i}_{n_1 1} = \sim^{i}_{n_2 2} = \dots = \sim^{i}_{n_m m} = \sim^{i}_{B}$. ავღნიშნოთ, ყველა $\sim$ –ების სიმრავლე, გაერთიანებული $\{B^i\}$ სიმრავლესთან Q^i–ით. j–ური სვეტის სიგრძე ზემოთ მოყვანილ მატრიცაში $2 \cdot n_j + 1$–ის ტოლია; იგულისხმება, რომ გვაქვს მატრიცა n, m განზომილებით, სადაც $n = 2 \cdot \max(n_j \mid 1 \le j \le m) + 1$, ხოლო ის სვეტები, რომელთა სიგრძე n–ზე ნაკლებია, შეივსება ნულებით.

წესების ზემოთ მოყვანილი მატრიცა, შეიძლება გამოვსახოთ მდგომარეობათა გრაფის დახმარებით



რომელიც თავის მხრივ, განსაზღვრავს შემდეგ სასრულ ავტომატს

$$A_i = \left(Q_i,\ \Sigma \cup Q`,\ ¢,\ B^i,\ \left\{\sim^{i}_{B}\right\}\right)$$

სადაც $Q` = \cup_i \left\{lab(B^i)\right\}$, როგორც ეს მოცემულია 2.4.1 განმარტებაში.

გადასვლების ფუნქციისათვის კი გვაქვს

$$¢\left(B^i, u^i_{1k}\right)=\left\{\sim^i_{1k_1}, \sim^i_{1k_2}, ..., \sim^i_{1k_p}\right\}$$

თუ
$$u^i_{1k}=u^i_{1k_1}=u^i_{1k_2}=...=u^i_{1k_p} \quad (1\le p\le m)$$
$$¢\left(\sim^i_{jk}, u^i_{j+1,k}\right)=\left\{\sim^i_{j+1,k}\right\} \quad j=1,2,...,n_k-1$$

ყველა $k=1,2,...,m\ge 1$ –თვის. (საზოგადოდ ეს არის არადეტერმინირებული ავტომატი, რადგანაც $u_{11}, u_{12}, ..., u_{1m}$–თვის არ არის აუცილებელი, რომ ისინი განსხვავებული წყვილები იყვნენ.)

G გრამატიკიდან ნებისმიერი არატერმინალი, ვთქვათ $S`=B^1, B^2, ..., B^p$ $(p\ge 1)$, განსაზღვრავს წესების მატრიცებს $P^1, P^2, ..., P^p$, როგორც ეს ზემოთ იყო გაკეთებული. ამ მატრიცებს თავის მხრივ შეესაბამება $A_1, A_2, ..., A_p$ სასრული ავტომატები; A_i ავტომატის მდგომარეობათა სიმრავლე თავის მხრივ არის Q_i. დავუშვათ, რომ $Q_1, Q_2, ..., Q_p$ სიმრავლეები წყვილ–წყვილად არ გადაიკვეთებიან, მაშინ $S=\{A_1, A_2, ..., A_p\}$ წარმოადგენს სასრული ავტომატების სიტემას, რომელიც აკმაყოფილებს 2.4.1 განმარტებას. ბუნებრივია A_1 საწყისი ავტომატია.

დარჩა ვაჩვენოთ, რომ $T(S)=L(G)$. ეს კი, თავის მხრივ, გამომდინარეობს იმ ანალოგიიდან რაც ცხადია 2.4.2 განმარტებასა და $T(S)$ სიმრავლეს შორის და ანალოგიიდან 2.1.4 განმარტებასა და $L(G)$ სიმრავლეს შორის. რ.დ.გ.

რადგან დაპროგრამების ენები კონტექსტურად თავისუფალი გრამატიკის საფუძველზე ბაზირებული ენებია, ამიტომ ბუნებრივია აქტუალურია შემდეგი თეორემა, რომელსაც *2.4.2* თეორემის საფუძველზე ჩამოვაყალიბებთ კიდევ ერთ ახალ

თეორემა 2.4.3. კონტექსტურად–თავისუფალი G გრამატიკის შესაბამისი სინტაქსური გარჩევის ალგორითმებს შეესაბამება სასრულ ავტომატთა $S=\{A_1, A_2, ..., A_p\}$ სისტემა.

შენიშვნა: ამ თეორემის დამტკიცება მოხდება ზუსტად იმავე წესით, როგორც 2.3.4 თეორემის დამტკიცება.

2.5 სინტაქსური გარჩევის ერთი ალგორითმის შესახებ

<u>მაგალითი 1.</u> ავაგოთ მთელი დადებითი რიცხვის სინტაქსური გარჩევის ფუნქციის შესაბამისი სასრული ავტომატი.

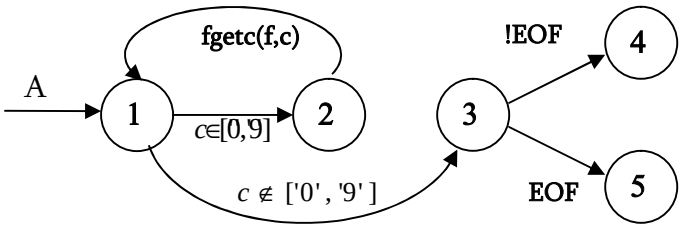
ბეკუს–ნაურის ფორმალიზმით:

<რიცხვი>::=<ციფრი>{<ციფრი>}.

სინტაქსური გარჩევის შესაბამისს ფუნქციას აქვს სახე:

```
bool number(char c)
{  while(c>='0'  &&  c<='9')
       fgetc(f,c);
   if(c!=EOF) return false;
            return true; }
```

ამ ფუნქციის შესაბამისი სასრული ავტომატის დიაგრამა შემდეგნაირია:



$¢(1,2) = c \in '0'..'9'$; $¢(2,1) = fgetc(f,c)$; $¢(1,3) = c \notin '0'..'9'$;

$¢(3,5) = c == EOF$; $¢(3,4) = c != EOF$

<u>მაგალითი 2.</u> ახლა განვიხილოთ უფრო რთული მაგალითი: არითმეტიკული გამოსახულების სინტაქსური გარჩევის ალგორითმი და ავაგოთ შესაბამისი ავტომატების სისტემა.

არითმეტიკული გამოსახულების სინტაქსური ფორმულა, ბეკუს – ნაურის[14] ფორმალიზმის ფარგლებში ჩაიწერება ასე :

< არითმეტიკული გამოსახულება > : : = [+ | -] < ტერმი > { + | - < ტერმი > }

< ტერმი > : : = < მამარავლი > { * | / | % < მამრავლი > } (5)

< მამარავლი > : : = < რიცხვი > | < იდენტიფიკატორი > | (< გამოსახულება >)

[14] ბეკუს-ნაურის ფორმალიზმი წარმოადეგნს ფორმალური ენის სინტაქსისის აღსაწერად გამოყენებულ ფორმალურ სისტემას.

(5) სინტაქსური ფორმულები, თავისთავად განმარტავენ შემდეგი პროგრამის სემანტიკას.

. . .

```
#define probel while(c==' ') c=getchar();
 void Expression();
 void Term();
 void mult();
 void number();
 void ident();

  main()
 {  char c;
    c=getchar();  probel
    Expression(); probel                                    (6)
 if(c!=EOF) {cout<<"Syntax Error";
              exit(0);}
        cout<<"Translation complete";
  }
```

. . .

```
void Expression()
{  if(c=='+' || c=='-') {c=getchar(); probel}
        Term(); probel                                      (6.1)
    while(c=='+' || c=='-')
      {Term(); probel}
}
```

. . .

```
void Term()
{   mult();
          while(c=='*' || c=='/')                           (6.2)
      {mult(); probel}
```

```
                    .   .   .
void mult()
{  if(c>='0' && c<='9')  number();
     else if(c>='a' && c<='z' ||  c>='A' && c<='Z' || c=='_')  ident();
          else if (c=='(')
               {Expression(); probel                                    (6.3)
                  if( c!=')' ){cout<<"Syntax Error";
                                  exit(0);}
                }
         c=getchar(); }
                    .   .   .
void number()
{  while((c=getchar())>='0'&&c<='9');                                   (6.4)
}
                    .   .   .
void ident()
{ while((c=getchar())>='a'&&c<='z'||c>='A'&&c<='Z'||c>='0'&&c<='9'||c=='_');(6.5)
}
```

შეიძლება ითქვას, რომ პროგრამა საკმაოდ ნათელია, მაგრამ აქვს ერთი მცირე სირთულე: ალგორითმი წარმოადგენს რეკურსიულ ალგორითმს, კერძოდ, საქმე გვაქვს ირიბ რეკურსიასთან.

ახლა შევეცადოთ ავაგოთ (6) -ის შესაბამისი სასრული ავტომატი. ყოველი წაკითხული სიმბოლო მოხვდება სასრულ ავტომატზე მხოლოდ ერთხელ და მისი დამუშავება მოითხოვს დროის სასრულ მონაკვეთს. აქედან გამომდინარე სასრული ავტომატის მუშაობის საერთო დრო არ აღემატება $\Theta(n)$ (სადაც n არის ალფავიტის სიგრძე – ანუ სიმბოლოთა რაოდენობა, რომლის საშუალებითაც ჩაიწერება არითმეტიკული გამოსახულება: ლათინური ალფავიტის 26 დიდ ასოითი სიმბოლო და 26 პატარა ასოითი სიმბოლო, 10 ციფრული სიმბოლო და 5 არითმეტიკული ოპერაციის ნიშნის შესაბამისი სიმბოლო).

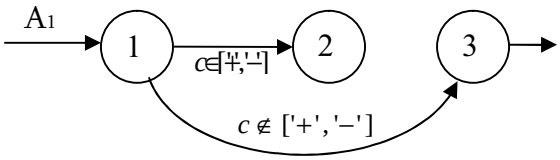
ყველაფერი ეს სამართლიანია, თუ ასეთი ავტომატი უკვე არსებობს – ამიტომ საჭიროა ავაგოთ არითმეტიკული გამოსახულების სინტაქსური ანალიზის შესაბამისი სასრული ავტომატი.

თავდაპირველად ავტომატი მდებარეობს საწყისს მდგომარეობაში - q_0. შემდეგ ის თანმიმდევრობით კითხულობს სიმბოლოებს ფაილიდან (არითმეტიკული გამოსახულება ჩაწერილია ფაილში). ავტომატი დგას რა რომელიმე q მდგომარეობაში, მორიგი C სიმბოლოს წაკითხვისასის გადადის ახალ $¢(q,C)$ მდგომარეობაში. თუ $¢(q,C) \in A$, მაშინ ცხადია რომ წაკითხული სიმბოლო ამოცნობილია ავტომატის მიერ. თუ $¢(q,C) \notin A$, ეს ნიშნავს რომ გადასვლა არ განხორციელდა და უკეთესს შემთხვევაში დაფიქსირდა სინტაქსური შეცდომა, ხოლო უარესს შემთხვევაში სინტაქსური გარჩევის პროგრამა არაკორექტულია.

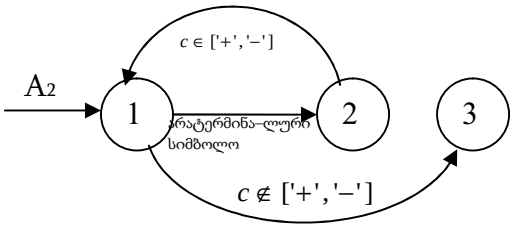
ამოცანა დასმულია, ახლა საჭიროა ავაგოთ სასრული ავტომატი.

იმის გამო, რომ არითმეტიკული გამოსახულების სინტაქსური გარჩევის ამოცანა შედარებით რთული ამოცანაა, ამიტომ შესაბამისი სასრული ავტომატი გრაფიკულად ცოტა არასტანდარტულად გამოიყურება.

შემოვიღოთ რამდენიმე სქემატური აღნიშვნა:



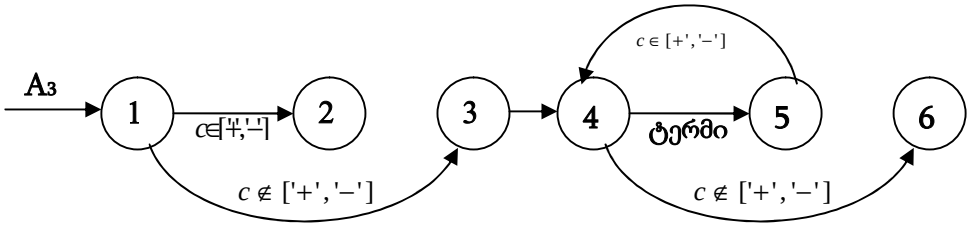
A_1 ავტომატი ამოიცნობს რა, რომ მორიგი წაკითხული სიმბოლო $c \in ['+','-']$, გადადის 1–ლი მდგომარეობიდან მე–2 მდგომარეობაში, ხოლო თუ მორიგი სიმბოლო $c \notin ['+','-']$, მაშინ 1–ლი მდგომარეობიდან გადადის მე–3 მდგომარეობაში.



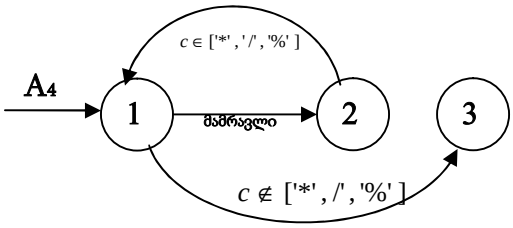
A_2 ავტომატი ნიშნავს განმეორებადობას.

ვინც იცნობს სინტაქსურ დიაგრამებს, მათთვის ასეთი ინტერპრეტაცია ბუნებრივია. სწორედ ასეთი ინტერპრეტაციების დახმარებით, ადგილი აქვს ე.წ. შემდეგ ფსევდოავტომატებს.

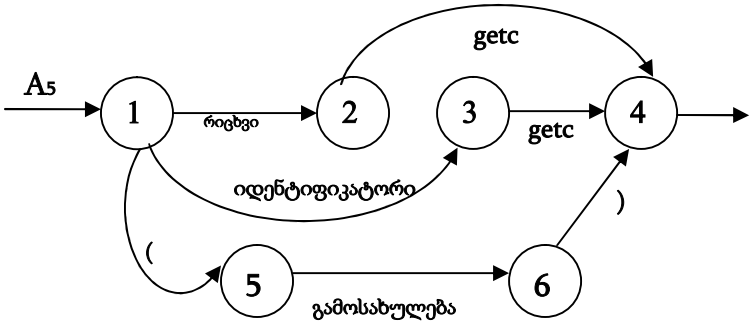
A_3 ფსევდოავტომატი, რომელიც შეესაბამება, არითმეტიკულ გამოსახულებას



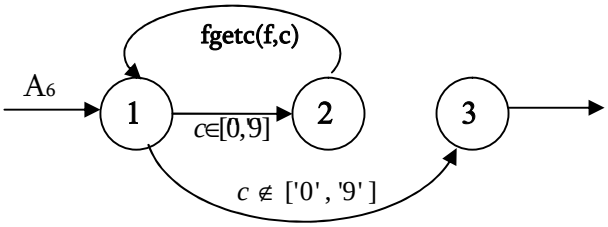
A_4 ფსევდოავტომატი, რომელიც შეესაბამება ტერმს:

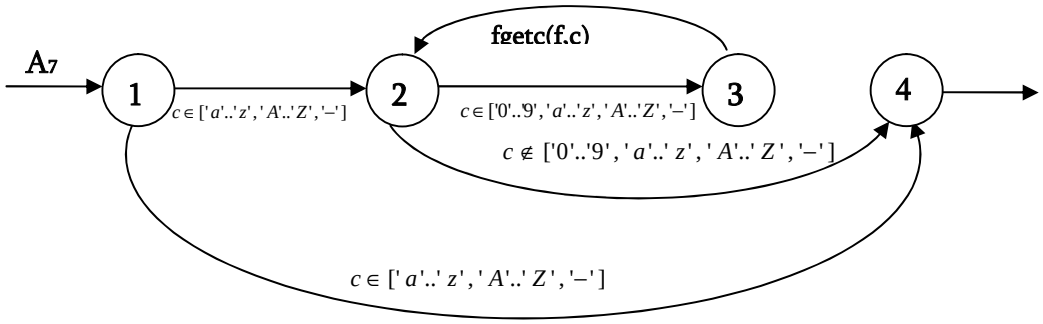


A_5 არის მამრავლის შესაბამისი ფსევდოავტომატი



A_6 და A_7 ფსევდოავტომატი, რომელიც შეესაბამება რიცხვს და იდენტიფიკატორს:





ამ ფსევდოავტომატების დახმარებით, იმ სასრული ავტომატის გრაფიკული წარმოდგენა, რომელიც შეესაბამება არითმეტიკული გამოსახულების სინტაქსური გარჩევის პროგრამას წარმოდგენილია 1-ელ ნახაზზე.

სადაც

$$\Sigma = \{'a'...'z',\ 'A'...'Z',\ '0'...'9',\ '+','-','/','\%','*','\_'\}$$

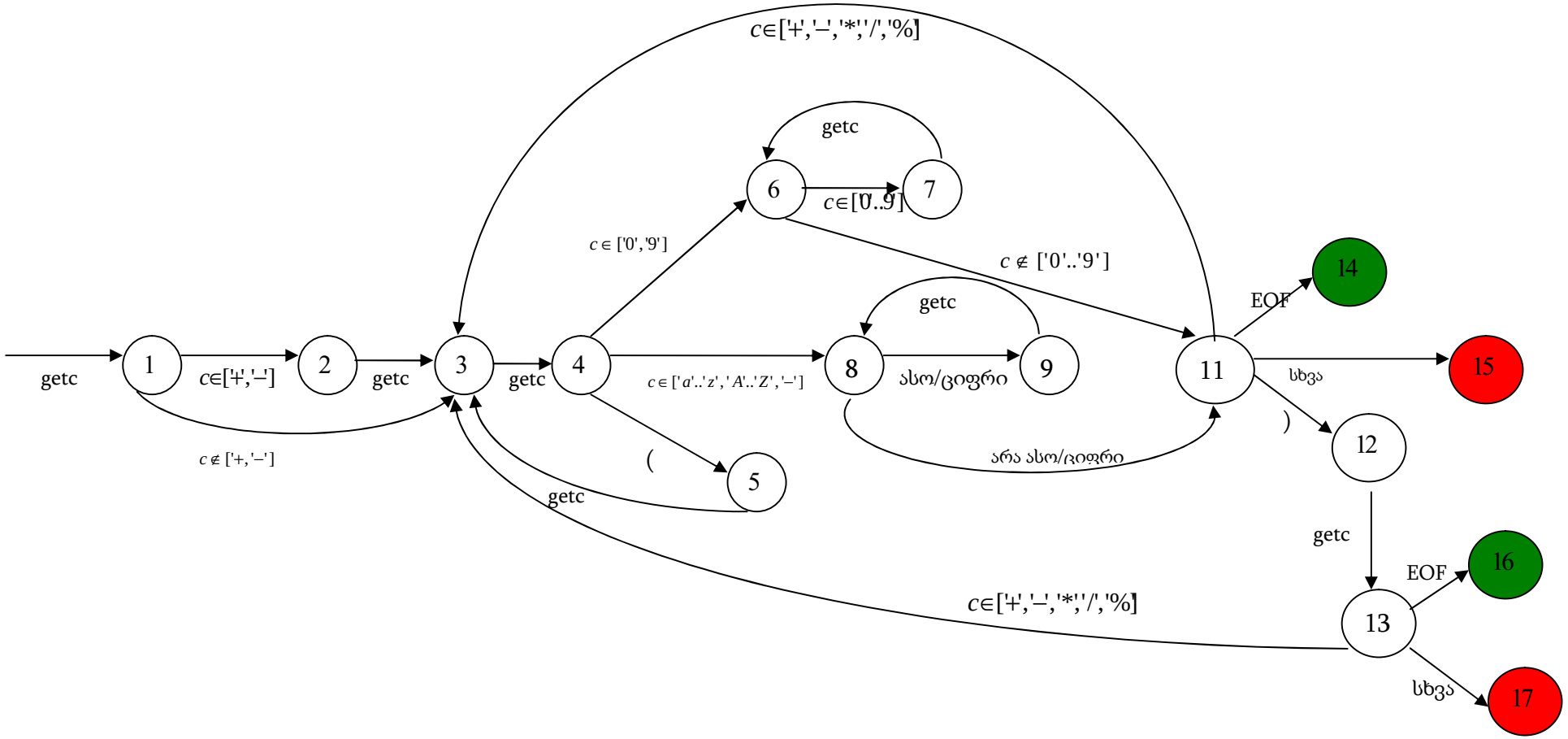
$$¢ = \{\text{Expression},\ \text{Term},\ \text{mult},\ \text{number},\ \text{ident}\}$$

$¢$ ფუნქციის როლში, რომელიც წარმოადგენს ასახვას $Q \times \Sigma$ სიმრავლიდან Q სიმრავლეში, დროის მოცემულ მომენტში შეიძლება იყოს ჩამოთვლილი ფუნქციებიდან ერთერთი.

Q სიმრავლე, ამოცანის სპეციფიკიდან გამომდინარე, უფრო წააგავს გრაფს, მაგრამ შესაძლო მდგომარეობათა თვალსაჩინოება მაინც შენარჩუნებულია.

თუ $q_{next} = ¢(q_{pre}, C) \notin A$, ეს ნიშნავს, რომ გადასვლა არ განხორციელდა არც ერთი მიმართულებით და სინტაქსური ანალიზი დასრულდა შეცდომით, ან შესაბამისი პროგრამა არაკორექტულია.

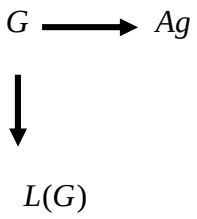
თუ ყველა $q_{next} = ¢(q_{pre}, C) \in A$ და A სასრული ავტომატის დახმარებით შევძელით q_{final} მდგომარეობამდე მისვლა, ეს ნიშნავს რომ სინტაქსური ანალიზი წარმატებით დასრულდა და შესაბამისი პროგრამაც კორექტულია.



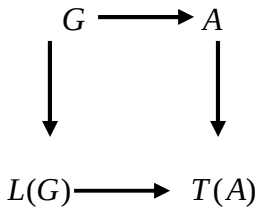
¢(1,2) = $c \in$ ['+','–']; ¢(1,3) = $c \notin$ ['+','–']; ¢(1,2) = $c \in$ ['+','–']; ¢(4,6) = $c \in$ ['0'..'9']; ¢(4,6) = $c \in$ ['+','–']; ¢(6,7) = $c \in$ ['0'..'9']; ¢(6,11) = $c \notin$ ['0'..'9']; ¢(4,8) = $c \in$ ['a'..'z','A'..'Z','_']; ¢(8,9) = $c \in$ ['a'..'z','A'..'Z','_','0'..'9']; ¢(8,11) = $c \notin$ ['a'..'z','A'..'Z','_','0'..'9']; ¢(11,3) = $c \in$ ['+','–','*','/','%']; ¢(4,5) = c ==('; ¢(13,3) = $c \in$ ['+','–','*','/','%']; ¢(11,12) = c ==')'; ¢(11,14) = c == *EOF*; ¢(13,16) = c == *EOF*; ¢(11,15) = *ERROR*; ¢(13,17) = *ERROR*;

2.6 სინტაქსური ანალიზის ალგორითმების კორექტულობის პრინციპი

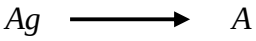
ავღნიშნოთ Ag სინტაქსური ანალიზის ალგორითმი, რომელიც შეესაბამება G გრამატიკას. თავის მხრივ G გრამატიკაზე განმარტებულია $L(G)$ ენა. სქემატურად ეს შეიძლება შემდეგი დიაგრამის სახით გამოისახოს:



სქემატურად ანალოგიურად შეიძლება გამოვსახოთ თეორემები 2.3.1, 2.3.2 და 2.3.4

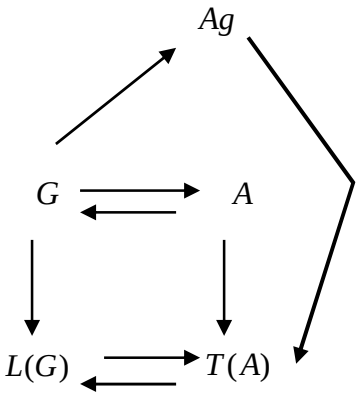


(თეორემები 2.3.1 და 2.3.2)



(თეორემა 2.3.4)

ზემოთ მოყვანილი ყველა სქემატური გამოსახულებების გაერთიანებით მიიღება შემდეგი სქემა



მიღებული დიაგრამა საშუალებას იძლევა ჩამოვაყალიბოთ სინტაქსური ანალიზის ალგორითმების კორექტულობის პრინციპი.

სინტაქსური ანალიზის ალგორითმების კორექტულობის პრინციპი მდგომარეობს შემდეგში:

***G* გრამატიკის შესაბამისი სინტაქსური ანალიზის *Ag* ალგორითმი კორექტულია მაშინ და მხოლოდ მაშინ, თუ ის ჩაწერილია რეგულარულ ხდომილებათა ენაზე.**

ჩამოყალიბებული პრინციპის ფორმულირება შესაძლებელია სხვაგვარად: *G* გრამატიკის შესაბამისი სინტაქსური ანალიზის *Ag* ალგორითმი კორექტულია მაშინ და მხოლოდ მაშინ, თუ შესაძლებელია აიგოს ისეთი სასრული ავტომატი, რომელიც შეესაბამება *Ag* ალგორითმი შესაბამისს პროგრამას.

თავი 3. პროგრამული უზრუნველყოფის ვერიფიკაციის ამოცანა კრიპტოგრაფიული სისტემებისათვის

კრიპტოლოგია[15] ერთ-ერთი უძველესი მეცნიერებაა, მისი ისტორია ითვლის რამდენიმე ათასს წელს, ის არის მეცნიერება ინფორმაციის შიფრაციისა და დეშიფრაციის მეთოდების შესახებ. კრიპტოლოგია შედგება ორი ნაწილისაგან – კრიპტოგრაფია[16] და კრიპტოანალიზი[17] . კრიპტოგრაფია გულისხმობს მონაცემების შიფრაციის მეთოდების შექმნას, მაშინ როცა კრიპტოანალიზი აფასებს და სწავლობს რა შიფრაციის მეთოდების სუსტ წერტილებს, გულისხმობს ისეთი მეთოდების შემუშავებას, რომელიც საშუალებას იძლევა შესაძლებელი იყოს კრიპტო სისტემების გატეხვა – დეშიფრაცია.

კრიპტოგრაფია არის მეცნიერება ისეთი მეთოდების შესახებ, რომელიც უზრუნველყოფს ინფორმაციის კონფედენციალობას და მთლიანობას. ტრადიციული კრიპტოგრაფია აღიარებს ე.წ. სიმეტრიულ კრიპტო სისტემებს, რომელშიც ინფორმაციის შიფრაცია/დეშიფრაცია ხდება ერთი და იგივე საიდუმლო გასაღების საშუალებით. პარალელურად, თანამედროვე კრიპტოგრაფია მოიცავს ასიმეტრიულ კრიპტოსისტემებს, ელექტრონული ციფრული ხელმოწერის სისტემებს, ხეშ-ფუნქციებს, კვანტურ კრიპტოგრაფიას და ა.შ.

თანამედროვე კრიპტოგრაფიისათვის დამახასიათებელია შიფრაციის ღია ალგორითმების გამოყენება, რაც თავის მხრივ გულისხმობს გამოთვლითი სისტემების გამოყენებას. ცნობილია შიფრაციის ათობით შემოწმებული ალგორითმები, რომლებიც საკმაოდ გრძელი გასაღებისათვის და შესაბამისი ალგორითმის კორექტული რეალიზაციის შემთხვევაში (მიაქციეთ ყურადღება: ალგორითმის კორექტული რეალიზაციის შემთხვევაში) კრიპტოლოგიურად საკმაოდ მდგრადია. ყველაზე გავრცელებული ალგორითმებია:

- სიმეტრიული *DES, AES, ГОСТ 28147-89, Camellia, Twofish, Blowfish, IDEA, RC4* და სხვა.
- ასიმეტრიული *RSA* და *Elgamal.*

---

[15] წარმოებულია ბერძნული სიტყვისაგან – κρυ ός(დაფარული), λόγος(სიტყვა).
[16] წარმოებულია ბერძნული სიტყვისაგან - κρυ ός(დაფარული), γρά ω(ვწერ).
[17] წარმოებულია ბერძნული სიტყვისაგან – κρυ ός და ანალიზი.

- ხეშ – ფუნქციები *MD4, MD5, MD6, SHA-1, SHA-2, ГОСТ Р 34.11-94.*

კრიპტოანალიზი არის მეცნიერება დაშიფრული ინფორმაციის დეშიფრაციის მეთოდების შესახებ. არაფორმალურად, კრიპტოანალიზს – შიფრის(კოდის) გატეხვასაც უწოდებენ.

ხშირ შემთხვევაში კრიპტოანალიზის უკან მოიაზრება დაშიფრული ინფორმაციის გასაღების აღმოჩენა, მაგრამ გარდა ამისა კრიპტოანალიზი გულისხმობს კრიპტოლოგიური ალგორითმებისა და ოქმების(ე.წ. "პროტოკოლების") სუსტი წერტილების გამოვლენას.

თავდაპირველად, კრიპტოანალიზის მეთოდები დაფუძნებული იყო ბუნებრივი ენის ლინგვისტურ კანონზომიერებაზე და მისი რეალიზება ხდებოდა მხოლოდ ფურცლისა და ფანქრის გამოყენებით. დროთა განმავლობაში კრიპტოანალიზში მკვიდრდებოდა მათემატიკური მეთოდები, რომელთა რელიზაციისათვის გამოიყენება სპეციალური კრიპტოანალიზური კომპიუტერები.

კრიპტოანალიზის მეთოდების გამოყენებით რაიმე კონკრეტული შიფრის გახსნის ცდას ამ შიფრზე კრიპტოლოგიური შეტევა ჰქვია. ისეთ კრიპტოლოგიურ შეტევას, რომლის შედეგადაც მოხერხდა შიფრის გახსნა, ეწოდება შიფრის გატეხვა.
კრიპტოანალიზის თანამედროვე მეთოდებია:

- შეტევა შიფროტექსტის საფუძველზე.
- შეტევა ღია ტექსტის და შესაბამისს შიფროტექსტის საფუძველზე.
- შეტევა არჩეული ღია ტექსტის საფუძველზე.
- შეტევა ადაპტიურად არჩეული ტექსტის საფუძველზე.

3.1 ბლოკური დაშიფვრის სიმეტრიული კრიპტოგრაფიული სისტემა. დაშიფვრა.

ბლოკური შიფრი ეს არის კრიპტოგრაფიული სისტემა, რომელიც ე.წ. ღია ტექსტს ყოფს ერთი და იგივე ზომის ცალკეულ ბლოკებად და როგორც წესი თითოეულ მათგანთან ოპერირება ხდება დამოუკიდებლად, შედეგად მიიღება დაშიფრული ტექსტის ბლოკების მიმდევრობა. ტექსტის დაშიფვრა და გაშიფვრა ხდება ერთი და იგივე გასაღებით, ამიტომაც მსგავსს სისტემებს ბლოკური სიმეტრიული კრიპტოგრაფიული სისტემა ეწოდება.

ბლოკური დაშიფვრის სიმეტრიული კრიპტოგრაფიული სისტემა მიეკუთვნება *DES* სტანდარტს, ეს სისტემა გამოკვლეულია და დამუშავებულია ნ. მუსხელიშვილის სახელობის გამოთვლითი მათემატიკის ინსტიტუტის წამყვანი მეცნიერ–თანამშრომლების მიერ ბატონ ზურაბ ყიფშიძის ხელმძღვანელობით, ხოლო შესაბამისი ალგორითმების პროგრამული რეალიზაცია გაკეთებულია ჩემს მიერ. ამდენად ეს ამოცანა ჩემთვის საინტერესოა კორექტულობის თვალსაზრისით.

ნახაზზე მოცემულია ბლოკური დაშიფვრის სიმეტრიული კრიპტოგრაფიული სისტემის ზოგადი სქემა, იგულისხმება რომ f ფუნქცია არაწრფივი ფუნქციაა. თუ ფუნქცია წრფივი ფუნქციაა, ამ შემთხვევაში სისტემის მდგრადობა სერიოზულ რისკებთან არის დაკავშირებული და ამიტომ რაც უფრო რთულია f ფუნქციის ანალიზური სახე, მით უფრო დიდია სისტემის მდგრადობის ალბათობა.

დასაშიფრი ინფორმაცია იყოფა 64 ბიტი სიგრძის ბლოკებად. თითოეული ბლოკი ე.წ. *IP* საწყისი გადანაცვლების შემდეგ იყოფა მარცხენა და მარჯვენა ნახევრებად: L_0 და R_0, თითოეული მათგანის სიგრძე 32 ბიტია. უნდა აღინიშნოს, რომ *IP* გადანაცვლება არ იმართება გასაღებით. ამის შემდეგ ალგორითმი აწარმოებს შემდეგი სახის n ციკლს:

$$L_i = R_{i-1},$$
$$R_i = L_i \oplus f(R_{i-1}, K_i), \quad i = 1, 2, \ldots, n$$

K_i- 32 ბიტი სიგრძის ნაწარმოები გასაღებია, რომელიც აირჩევა $32m$ სიგრძის ძირითადი გასაღებებიდან (m–ის მნიშვნელობა და ციკლების რაოდენობა მოყვანილია ცხრილში). $f(R_{i-1}, K_i)$ ფუნქცია განისაზღვრება , როგორც ორი ასახვის კომპოზიცია $f = Pf_i$, სადაც P არის 32 ბიტი სიგრძის მიმდევრობების გადანაცვლებაა, რომელიც არ იმართება გასაღებით, ხოლო

$$f_i(R_{i-1}, K_i) = a_i * x_{i-1} \oplus b_i * y_{i-1}$$

სადაც a_i და b_i შესაბამისად K_i გასაღების მარცხენა და მარჯვენა ნახევარია და ბუნებრივია, თითოეული მათგანი 16 ბიტის სიგრძისაა. x_{i-1} და y_{i-1} ასევე R_{i-1}-ის მარცხენა და მარჯვენა 16 ბიტიანი ნახევრებია.

უკანასკნელი ციკლის შემდეგ L_n და R_n გაერთიანდება 64 ბიტი სიგრძის (L_n, R_n) მიმდევრობად, რომელიც ექვემდებარება შემდგომ IP^{-1} გადანაცვლებას,

რომელიც საწყისი გადანაცვლების შებრუნებულია და სისტემის გამოსავალზე ფორმირდება როგორც კრიპტოგრამა

$$E = IP^{-1}(L_n, R_n).$$

აღწერილი პროცესის სქემა მოცემულია 3.1 ნახაზზე.

გაშიფვრა.

გაშიფვრის სქემა განსხვავდება დაშიფვრის სქემისაგან, მხოლოდ იმით, რომ მასში ნაწარმოები გასაღებები გამოიყენება შებრუნებული მიმდევრობით. ე.ი. პირველ ციკლში გამოყენებულია K_n გასაღები, მეორეში K_{n-1} და ა.შ. K_1. მართლაც, მივიღებთ რა გაშიფვრის სქემის შესავალზე E კრიპტოგრამას, მასზე პირველად იმოქმედებს საწყისი IP გადანაცვლება, მივიღებთ

$$IP(E) = IP(IP^{-1}(L_n, R_n)) = L_n R_n$$

გვექნება რა L_n , R_n მივიღებთ:

$$R_{n-1} = L_n$$

$$L_{n-1} = R_n \oplus f(L_n, K_n) = R_n \oplus f(R_{n-1}, K_n)$$

საზოგადოდ კი i -ურ ციკლში

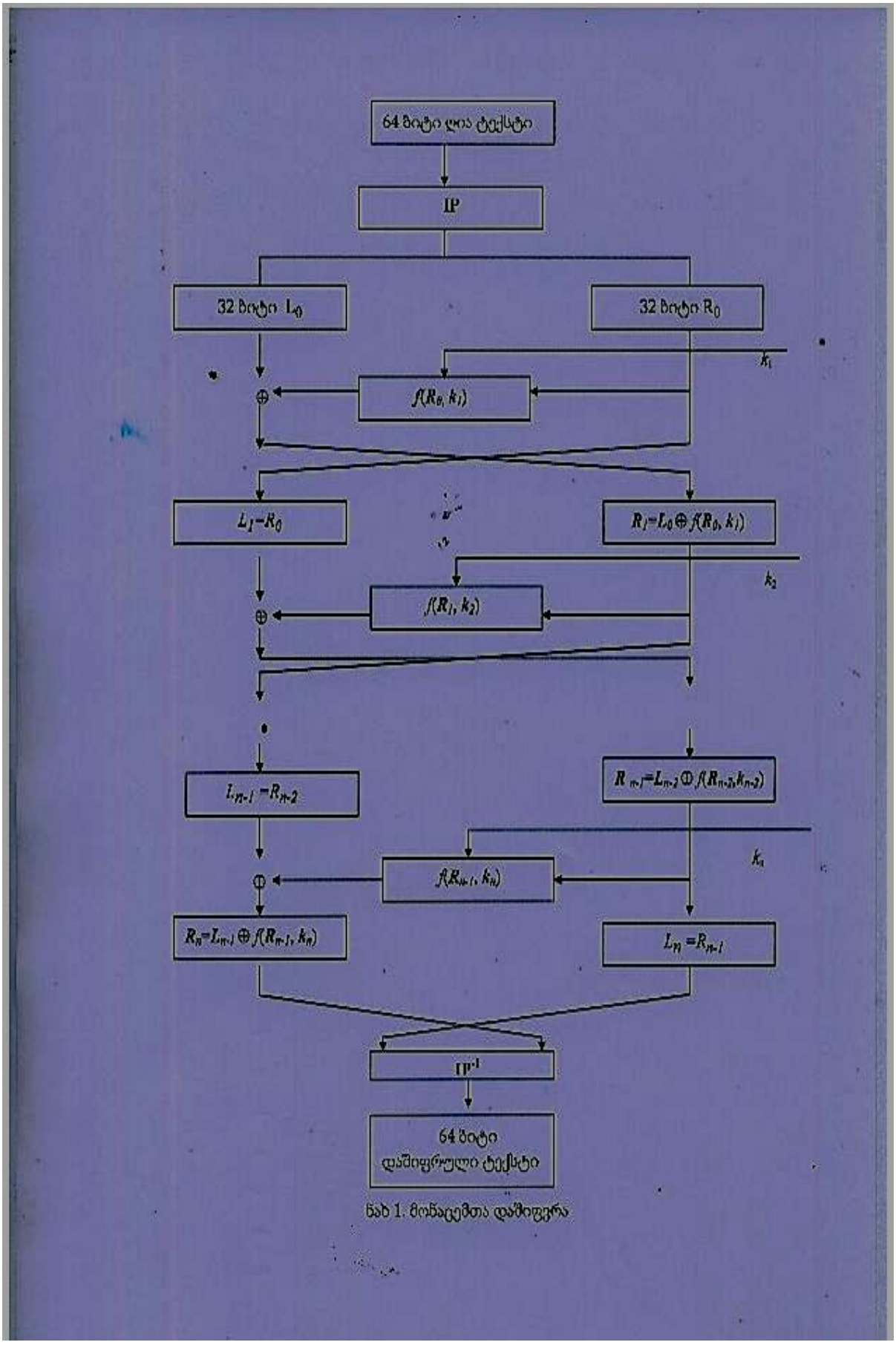
$$R_{i-1} = L_i$$

$$L_{i-1} = R_i \oplus f(L_i, K_i) = R_i \oplus f(R_{i-1}, K_i)$$

K_i წარმოებული გასაღების არჩევა K ძირითადი გასაღებიდან წარმოებს მე–2 ნახაზზე მოცემული სქემის მიხედვით. წარმოებული გასაღების არჩევის წესში გათვალისწინებულია ის ფაქტი, რომ საუკეთესო გასაიდუმლოება მიიღწევა იმ შემთხვევაში, როცა ძირითადი გასაღების ყველა ბიტი გამოყენებული იქნება თანაბარი სიხშირით

$$\neg = \frac{32n}{K} = \frac{32n}{32m} = \frac{n}{m}.$$

ამ მიზნის მიღწევის ერთ–ერთ ხერხს წარმოადგენს წარმოებული გასაღების ფორმირების შემდეგი პროცედურა.



ნახ 1. მონაცემთა დაშიფრა

პირველად K ძირითადი გასაღების $32m$ ბიტი დაიყოფა 32 ბიტიან m ცალ ბლოკად, თითოეული ბლოკი ექვემდებარება ციკლურ ძვრას d ბიტზე. ყოველი ცალკე იტერაციის დროს n–დან ისე, რომ უკანასკნელ იტერაციაზე აღდგენილი იყოს საწყისი მდგომერეობა. ამისთვის საჭიროა, რომ დაცული იყოს შემდეგი პირობა:

$$nd = 32$$

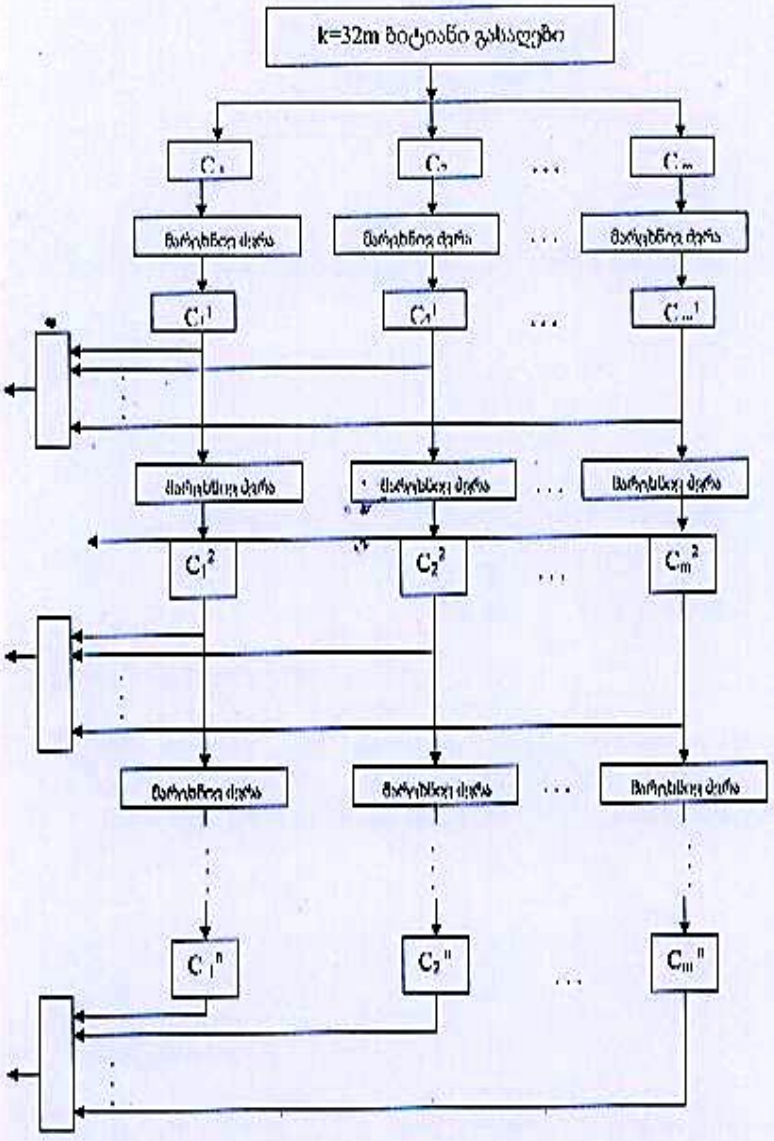
K_i ნაწარმოები გასაღები წარმოიქმნება i–ური ციკლური ძვრის შემდეგ ყოველი ბლოკის პირველი $\neg d$ ელემენტის გაერთიანებით.

ამრიგად, წარმოებული გასაღების ფორმირებისათვის მოითხოვება ორი პარამეტრი n და m, ასევე m/n და $n/32$. ძირითადი K გასაღების რეკომენდირებული სიგრძე და $m, n, \neg, d$ პარამეტრების მნიშვნელობები მოყვანილია შემდეგ ცხრილში

M	n	d	K	v
2	3	4	64	4
2	16	2	64	8
2	32	1	64	16
4	8	4	128	2
4	16	2	128	4
4	32	1	128	8

იმისათვის, რომ სისტემის იმედიანობა იყოს მაღალი, არაა რეკომენდირებული, რომ $n \le 4$. m–ის დიდი მნიშვნელობა კი ართულებს გასაღების საიდუმლოდ შენახვის პროცედურას მისი გადაცემის შემთხვევაში.

ბოლო ციკლის დასრულების შემდეგ მივიღებთ L_0, R_0 , რომელზეც საწყისი IP გადანაცვლების მოქმედების შემდეგ მიიღება 64 ბიტიანი საწყისი ტექსტი.



ნახ 2. გასაღების ფორმირების სქემა

3.2 ბლოკური დაშიფვრის სიმეტრიული კრიპტოგრაფიული სისტემის ალგორითმი.

კავშირის არხში ინფორმაციის დაცვის ალგორითმის ბლოკ–სქემა შედგება მონაცემთა დაშიფვრის საერთო ბლოკ–სქემისაგან და შემდეგი დეტალიზირებული კრიპტოგრაფიული პროცედურებისაგან:

- საწყისი გადანაცვლების ქვესისტემა;
- საშუალედო გადანაცვლებები;
- წარმოებული გასაღების ფორმირება;
- სპეციალური კრიპტოგრაფიული ფუნქციის გამოთვლა;
- საბოლოო გადანაცვლება;

გაშიფვრის პროცედურები იდენტურია დაშიფვრის პროცედურების და იმართება შებრუნებული რიგით შემოსული წარმოებული გასაღებების მიმდევრობით.

შემდეგ პარაგრაფებში განხილული დაწვრილებითი ალგორითმების შესაბამისი პროგრამის ამონაბეჭდები თან ერთვის სადისერტაციო ნაშრომს დანართის სახით.

3.2.1 მონაცემთა დაშიფვრის ალგორითმის ზოგიერთი განმარტებები.

M - 64 ბიტიანი ღია ტექსტი;

$K_1, K_2, \ldots, K_{16}$ – 32 ბიტიანი წარმოებული გასაღებები;

IP – საწყისი გადანაცვლება;

L_i – კონკრეტული ბლოკის მარცხენა ნახევარი;

R_i – კონკრეტული ბლოკის მარჯვენა ნახევარი;

x_i – L_i–ის შესაბამისი რიცხვითი მნიშვნელობა;

y_i – R_i–ის შესაბამისი რიცხვითი მნიშვნელობა;

a_{i+1} – გასაღების პირველი ნახევრის შესაბამისი რიცხვითი მნიშვნელობა;

b_{i+1} – გასაღების მეორე შესაბამისი ნახევრის რიცხვითი მნიშვნელობა;

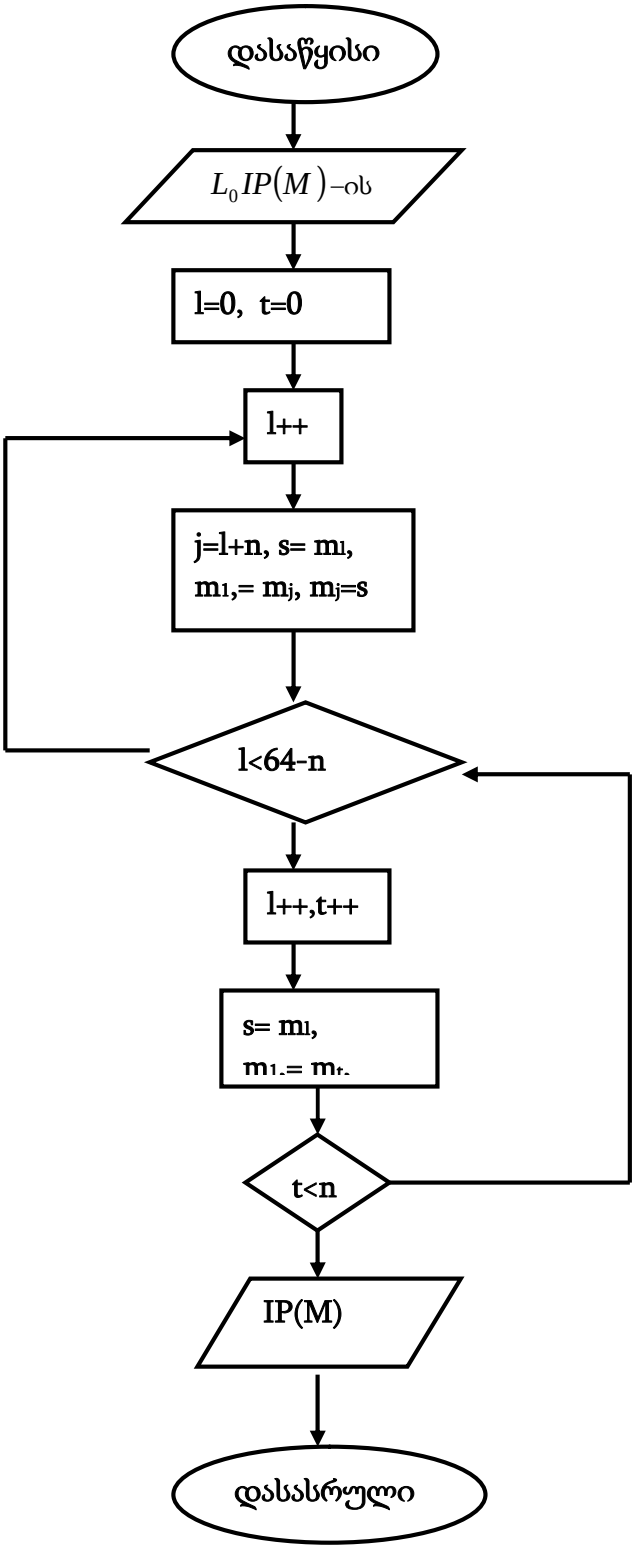
$C_i = a_{i+1} x_i \oplus b_{i+1} y_i$

P – ფიქსირებული გადანაცვლება;

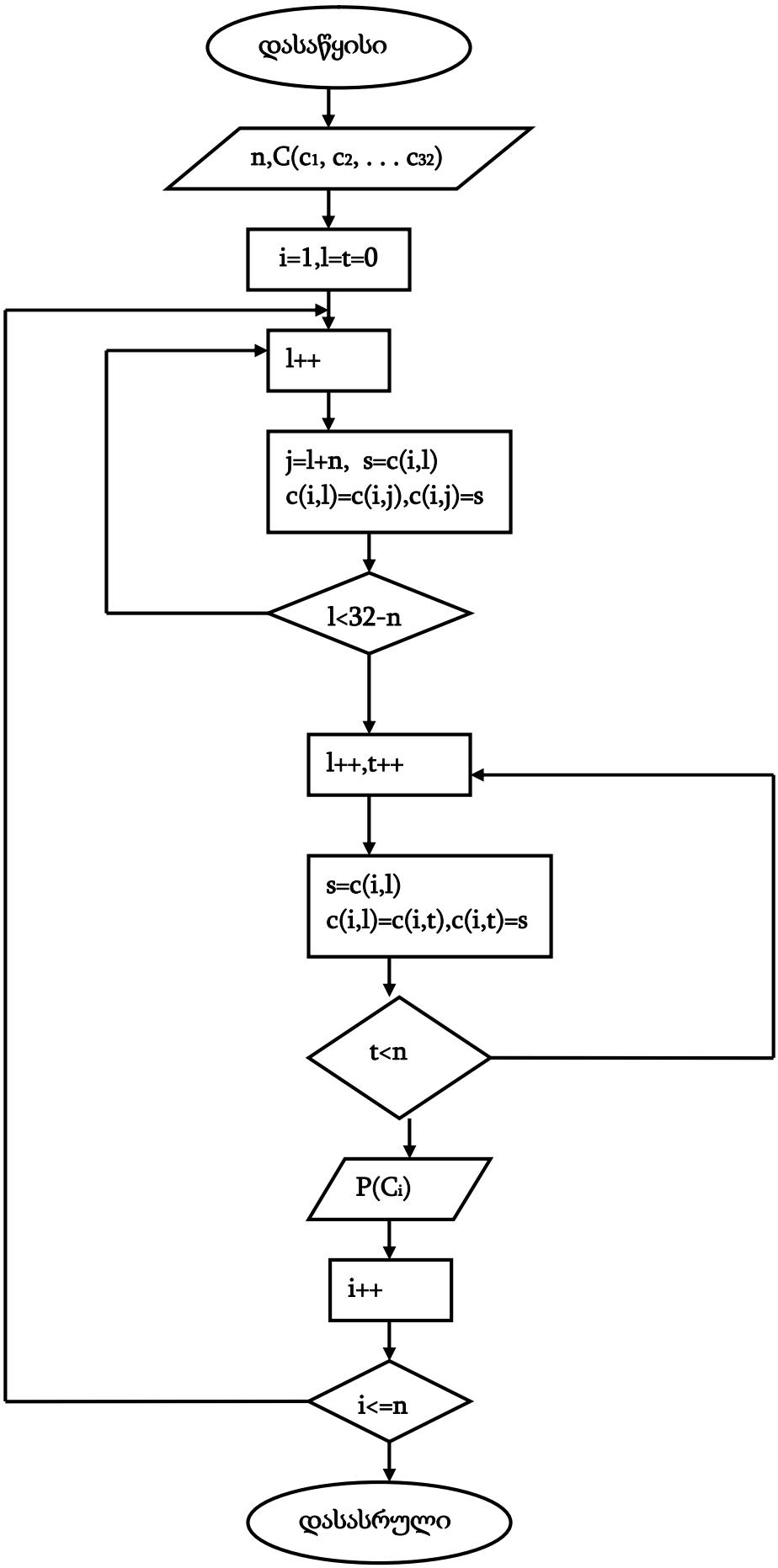
B – 64 ბიტიანი თანმიმდევრობა, რომელიც მიიღება R_{16} და L_{16} ბლოკების გაერთიანებით;

IP^{-1} – საწყისი IP გადანაცვლების შებრუნებული გადანაცვლება;

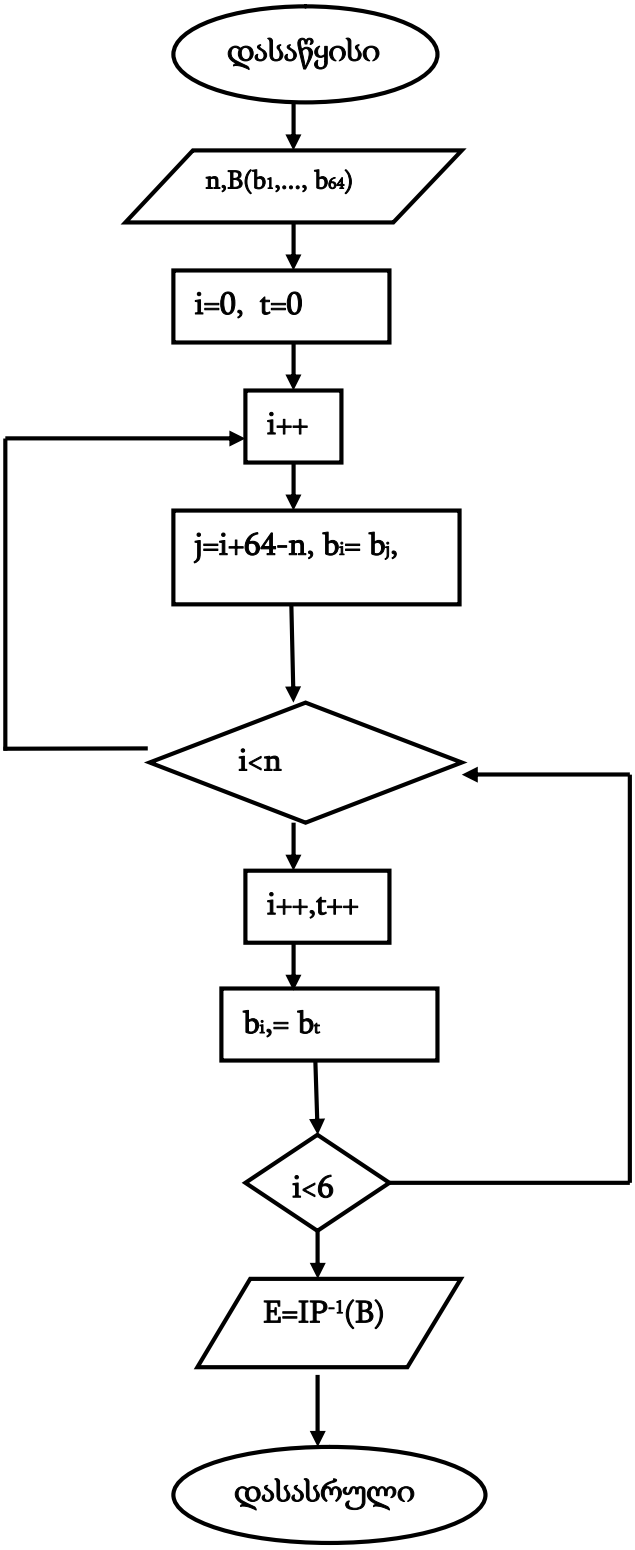
E – გამოსავალი კრიპტოგრამა $E = IP^{-1}(B)$.



IP გადანაცვლების ბლოკ–სქემა



P გადანაცვლების ბლოკ–სქემა



IP-1 გადანაცვლების ბლოკ–სქემა

3.2.2 $K_1, K_2, ..., K_{16}$ წარმოებული გასაღებების გამოთვლის ალგორითმის ზოგიერთი განმარტებები.

K – 64 ბიტიანი მიმდევრობა,

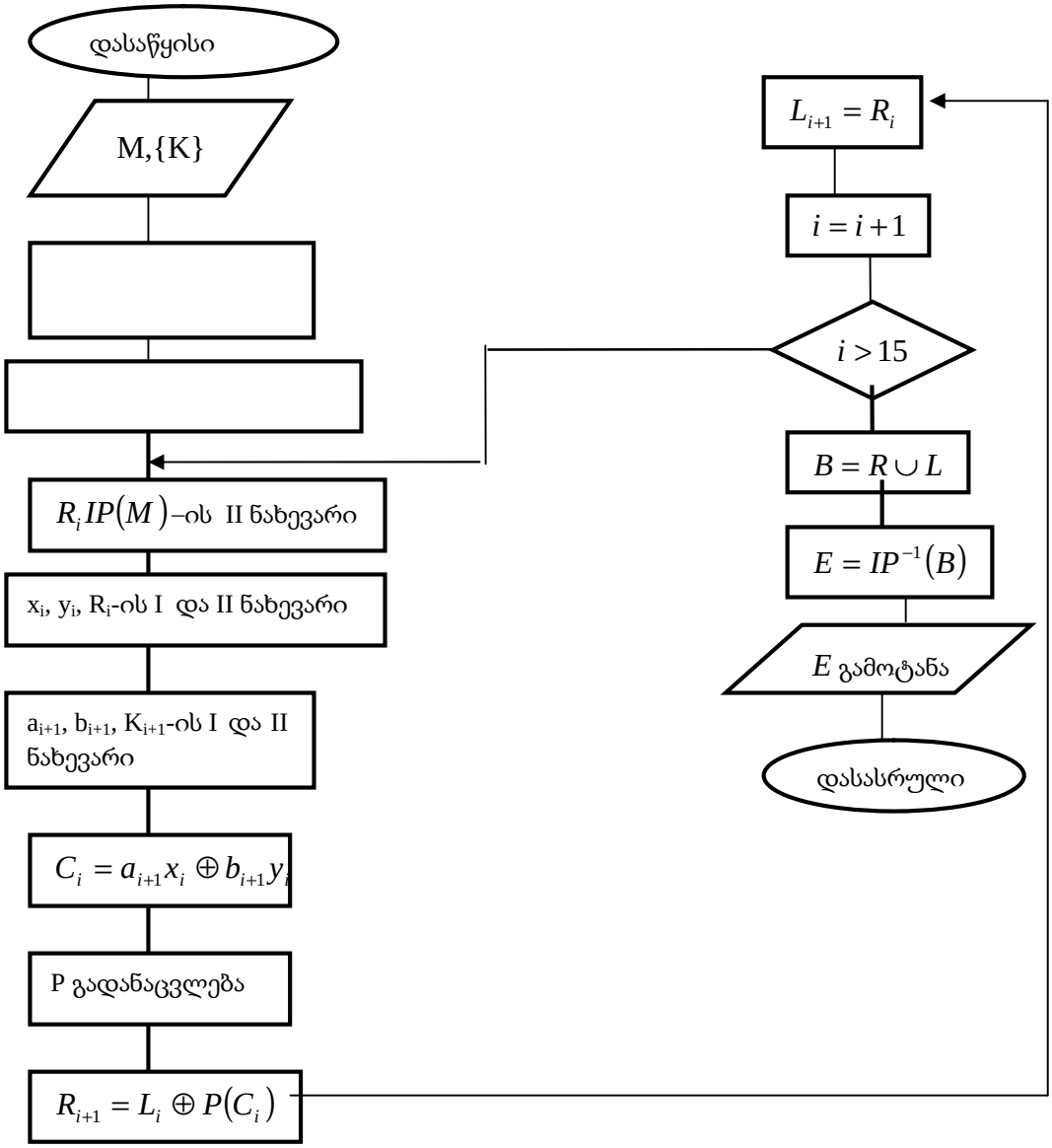
B_i^1 – K მიმდევრობის პირველი 32 ბიტი $B_i^1 = (b_1, b_2, ..., b_{32})$,

Q_i^1 – K მიმდევრობის მორიგი 32 ბიტი $Q_i^1 = (q_1, q_2, ..., q_{32})$,

B_i^2 – B_i მიმდევრობის პირველი 16 ბიტი, რომელიც მიიღება ორი ბიტით ციკლური დაძვრის შედეგად,

Q_i^2 – Q_i მიმდევრობის პირველი 16 ბიტი, რომელიც მიიღება ორი ბიტით ციკლური დაძვრის შედეგად,

K_i – წარმოებული 32–ბიტიანი გასაღები, რომელიც მიიღება B_i^2 და Q_i^2 მიმდევრობების გაერთიანებით.



3.3 კრიპტოლოგიური ამოცანის კორექტულობის პრინციპი

3.1 და 3.2 ქვეთავში განხილული ალგორითმები საინტერესოა კორექტულობის თვალსაზრისით. ცნობილია შიფრაციის/დეშიფრაციის ათობით შემოწმებული ალგორითმები, რომლებიც საკმაოდ გრძელი გასაღებისათვის და შესაბამისი ალგორითმის კორექტული რეალიზაციის შემთხვევაში კრიპტოლოგიურად საკმაოდ მდგრადია.

მიაქციეთ ყურადღება: ალგორითმის კორექტული რეალიზაციის შემთხვევაში. ბუნებრივია, იმისათვის რომ კრიპტოანალიზის და ასევე კრიპტოგრაფიის პრობლემა სწორად იყოს გადაჭრილი, საჭიროა შესაბამისი ალგორითმების კორექტულად რეალიზება. ამიტომაც კორექტულობის პრობლემა, ისევე როგორც ნებისმიერი სხვა ამოცანის რეალიზაციასა, აქაც დღის წესრიგში დგას.

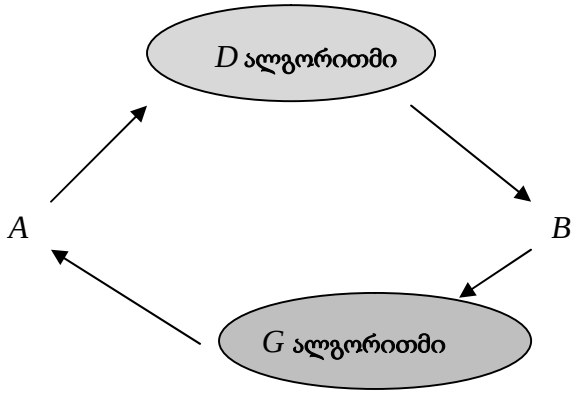
თავისთავად, საკუთრივ, შიფრაციისა და დეშიფრაციის ალგორითმების შესაბამისი პროგრამების კორექტულობის დამტკიცება შესაძლებელია სასრულ–ავტომატური მეთოდით, ანუ თუ თითოეული პროგრამისათვის შესაძლებელი გახდება სასრული ავტომატის აგება, მაშინ ალგორითმები ბუნებრივია კორექტულია. მაგრამ არსებობს ერთი უხერხულობა: დავუშვათ იმუშავა შიფრაციის შესაბამისმა პროგრამამ და მიიღო შიფროგრამა, თუ დეშიფრაციის პროგრამის მუშაობის შემდეგ, რომელიც აკეთებს შიფრის დეშიფრაციას, მიღებული ტექსტი არ ემთხვევა შიფრაციის პროგრამის საწყისს ტექსტს ბუნებრივია აშკარაა, რომ კრიპტოლოგიური სისტემის შესაბამისი პროგრამები არასწორია.

დავუშვათ A არის დაშიფვრის ამოცანის საწყის მონაცემთა ნაკრები. D იყოს დაშიფვრის ამოცანის ალგორითმის შესაბამისი "ფუნქცია". მაშინ $B = D(A)$ იქნება კრიპტოგრამა, რომელსაც მიიღებს დაშიფვრის ამოცანა.

გაშიფვრის ამოცანის საწყის მონაცემთა ნაკრებად გვევლინება $B = D(A)$. დავუშვათ გაშიფვრის ამოცანის ალგორითმის შესაბამისი "ფუნქციაა" G , მაშინ ცხადია რომ თუ G ფუნქციას მოვდებთ B მონაცემთა ნაკრებს, უნდა მივიღოთ:

$$G(B) = A$$

სქემატურად ეს შეიძლება ასე გამოვსახოთ:



ნახ.1

ზემოთ თქმულის ფორმულირება გავაკეთოთ ცოტა სხვაგვარად:

ვთქვათ, არსებობს დაშიფვრის შესაბამისი პროგრამა, რომლის საწყისი მონაცემია დასაშიფრი A ტექსტი. პროგრამის მუშაობის შემდეგ მიიღება B კრიპტოგრამა, რომელიც შესაბამისი არხებით გადაეცემა და მიიღებს ე.წ. დეშიფრატორის პროგრამა, როგორც საწყისს მონაცემს. დეშიფრატორის პროგრამა შესაბამისი კრიპტოგრამის დამუშავების შემდეგ გამოსავალზე ღებულობს საშედეგო მონაცემს, რომელიც უნდა ემთხვეოდეს A-ს.

ზემოთ თქმულიდან გამომდინარე შეიძლება ჩამოყალიბდეს კრიპტოლოგიის ამოცანის შესაბამისი პროგრამების ჭეშმარიტების კრიტერიუმი.

კრიტერიუმი: ვთქვათ D არის დაშიფვრის ალგორითმის შესაბამისი პროგრამა და A არის საწყის მონაცემთა ნაკრები. მაშინ თუ

$$B = D(A)$$

იმისათვის, რომ D პროგრამა ჭეშმარიტი იყოს აუცილებელი და საკმარისია, რომ არსებობდეს დეშიფრაციის პროგრამა G ისეთი, რომ

$$G(B) = A$$

და პირიქით.

შენიშვნა: 3.1 და 3.2 პარაგრაფებში მოყვანილი ალგორითმების შესაბამისი პროგრამული კოდები თან ერთვის სადისერტაციო ნაშრომს დანართის სახით(დანართი 2).

დასკვნა

სადისერტაციო ნაშრომში მიღწეულია თემით განსაზღვრული მიზნები - დასმულია და გადაწყვეტილია სპეციალური კლასის ალგორითმულ ენებზე დაწერილი კომპიუტერული პროგრამების ვერიფიკაციის(კორექტულობის) ამოცანა.

ალგორითმული ენებისა და პროგრამების ვერიფიკაციის ზემოთ აღნიშნულ ამოცანასთან დაკავშირებით დისერტაციაში განვითარებულია ახალი მიდგომა, რომელიც არსებითად ეფუძნება სასრული ავტომატების კლასიკურ თეორიას. ამ მიდგომის საფუძველს წარმოადგენს ფორმალური მოდელი - სასრული ავტომატი, რომლის ფარგლებში მოხერხებული აღმოჩნდა არა მარტო ალგორითმული ენების(სინტაქსი და სემანტიკა) აღწერა და ანალიზი, არამედ ამ ენებზე დაწერილი პროგრამების ვერიფიკაციის მათემატიკურად კორექტული ამოცანის როგორც დასმა, ისე გადაწყვეტა. განსაკუთრებული მნიშვნელობა ენიჭება პროგრამების მიმართ წაყენებულ არაფორმალურ მოთხოვნების მათემატიკურად კორექტული ვერსიის ფორმულირებას. ასეთი თეორიულ-ავტომატური ფორმალიზაცია საშუალებას იძლევა მკაცრად დამტკიცდეს კონკრეტული პროგრამისა და სპეციფიკაციის შესაბამისობა.

პროგრამების დამუშავებისა და კორექტულობის(სისწორის) ანალიზის ფორმალურ მეთოდებს, რომლებიც დისერტაციაშია დამუშავებული და წარმოდგენილი, თეორიულ მნიშვნელობასთან ერთად გააჩნიათ პრაქტიკული მნიშვნელობაც, ვინაიდან მათი საშუალებით შეიძლება აიგოს პროგრამული სისტემა, რომელიც განკუთვნილია კონტექსტურად-თავისუფალი ალგორითმული ენების სინტაქსური გარჩევის ამოცანისათვის. ამ სისტემის პრაქტიკაში გამოყენება მოხდა ბლოკური დაშიფვრის სიმეტრიული კრიპტოლოგიური სისტემის შესაბამისი ალგორითმისთვის. დანართში მოყვანილია ამ ალგორითმის შესაბამისი პროგრამის პროგრამული კოდები.

ლიტერატურა

[1]. У. Дал, Э. Дейкстра, К. Хоор. Структурное программирование. Издательство «МИР». МОСКВА 1975

[2]. Э. Дейкстра. Дисциплина программирования. Издательство «МИР». МОСКВА 1978.

[3.1]. Кнут Д. Искусство программирования для ЭВМ, том 1. Издательство «МИР». МОСКВА 1976. (Основные алгоритмы) – 736ст.

[3.2]. Кнут Д. Искусство программирования для ЭВМ, том 2. Издательство «МИР». МОСКВА 1977. (Получисленные алгоритмы) – 724ст.

[3.3]. Кнут Д. Искусство программирования для ЭВМ, том 3. Издательство «МИР». МОСКВА 1978. (Сортировка и поиск) – 848ст.

[4]. Н. Вирт. Алгоритми + Структуры Данных = Программы. Издательство «МИР». МОСКВА 1985.

[5]. Кларк Э. М., Грамберг О., Пелед Д. Верификация моделей программ. Model Checking. М.: МЦНМО. 2002.

[6] А. Оллонгрен. Определение языков программирования интерпретирующими автоматами. Издательство «МИР». МОСКВА 1977

[7]. Ю.Г. Карпов. Model Checking. Верификация параллельных и распределенных программных систем. – СПБ.:БХВ – Петербург, 2010. – 560ст.

[8]. А. Ахо, Дж. Ульман. Теория синтаксического анализа, перевода и компилации. Т 1,2. Издательство «МИР». МОСКВА 1978.

[9]. Р. Андерсон. Доказательство правильности программ. Издательство «МИР». МОСКВА 1982.

[10]. С. Алагич, М. Арбиб. Проектирование корректных структурированных программ. МОСКВА «Радио и связь» 1984.

[11]. Б. А. Трахтенброт, Я. М. Барздинь. Конечние Автоматы (Поведение и Синтез). Издательство «НАУКА». Главная редакция физико – математической литературы. МОСКВА 1970.

[12]. Томас Кормен,Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайню Алгоритмы. Анализ и постороние. Второе издание. Москва – Санк-Петербург - Киев 2005.

[13]. Э. Дейкстра. Взаимодействие параллельных процессов. Издательство «МИР». МОСКВА 1972.

[14]. Katz S., Manna Z. Logical analysis of algorithms. 1975.

[15]. Marmier E. Automatic Verification of Pascal Programs. Ph. D. Dissertation. Swiss Federal Institute of Technology, Zurich. 1975.

[16]. Aho A., Hopcroft J. E., Ullman J.D. The Design and Analysis of Compiuter Algorithms. Addison- Wesley. 1974.

[17]. Alagic S. Principles of Programming. (In Serbo-Croatian). Svjetlost-Sarajevo. Yugoslavia. 1976.

[18]. Alagic S., Arbib M. A., Taylor R. W. Algebraic Models of Syntax, Semantics and Data Structures. COINS Technical Report 74B-1: Compiuter and Information Sciens Department, University of Massachusetts at Amherst. 1974.

[19]. Dijkstra E. W. Goto statement considered harmful. Communications of the ACM 11. 1968.

[20]. Floyd R. W. Assigning meanings to programs. In: Proc. Sym. in Applied Math., Vol. 19, Mathematical Aspects of Compiuter Sciens. 1967.
[21]. Foley M. Hoare C. A. R. Proof of recursive program: Quicksort. Compiuter Journal 14, 391-395. 1971.
[22]. Gries D. Compiler Construction for Digital Compiuters. 1971.
[23]. Hoare C. A. R. Towards a theory of parallel programming. In: Operating Systems Techniques. Academic Press 61-71. 1972.
[24]. Hoare C. A. R. Lauer P. E. Consistent and complementary formal theories of the semantics of programming languages. Acta Informatica 3, 135-153. 1973.
[25]. Owicki S., Gries D. Verifying Properties of parallel programs. Acta Informatica 6, 319-340. 1976.
[26]. Н. Бенидзе. Задача оценки верификации (корректности) алгоритмов и компьютерных программ. Международний научно-технический журнал «OPTOELECTRONIC INFORMATION-POWER TECHNOLOGIES» №2(20), 2010, ст. 80.
[27]. Н. Бенидзе. Об устанавлении правильности компьютерных программ. “Internet Education Science” IES-2010. New Informational and Computer Technologies in Education and Science. Ukraine, Vinnytsia VNTU. september 28 - octomber 3, 2010 ,volume 1, section E, page 221-*224*
[28]. Н. Бенидзе. К вопросу о корректности программ. “Internet Education Science” IES-2008. New Informational and Computer Technologies in Education and Science. Ukraine, Vinnytsia VNTU. Octomber 7-11, 2008 Volume 2, Section H, page 545-550 .
[29]. საქართველოს ტექნიკური უნივერსიტეტის საერთაშორისო სამეცნიერო კონფერენციის „საინფორმაციო და კომპიუტერული ტექნოლოგიები, მოდელირება, მართვა“ (2010) თეზისები.
[30]. ნიკო მუსხელიშვილის გამოთვლითი მათემატიკის ინსტიტუტისა და საქართველოს საპატრიარქოს წმიდა ანდრია პირველწოდებულის სახელობის ქართული უნივერსიტეტის ერთობლივ საერთაშორისო კონფერენციის „ინფორმაციული და გამოთვლითი ტექნოლოგიები“ (2010) თეზისები.
[31]. Morris J. H., Verification Oriented Language Design, University of California, Berkley, Compiuter Science Technical Report 7, 1972.
[32]. Naur P. Programming by Action Clusters, B. I. T., 9(3), pp 250-258, 1969.
[33]. Mills H. D. How to Write Correct Programs and Know It, Proceedings of the International Conference on Reliable Software, Los Angeles, April 21-23, pp. 351-354.
[34]. Manna Z. The Correctness of Non-Deterministic Programs, Articl Intelligence, An International Journal, 1(1), pp 1-26, 1970.
[35]. London R. L. Bibliography on Proving the Correctness of Compiuter Programs, Machine Intelligence, 5, Meltzer B., Michie D. (Eds.), American Elsevier Publ. Co., New York, pp. 569-580, 1970.
[36]. Keller R. M. Formal Verification of Parallel Programs, Comm. ACM, 19(7), pp. 371-384, July 1976.
[37]. King J. C. A Program Verifier, Ph. D. thesis, Carnegie- Mellon University, 1969.
[38]. Elspas B. The Semiautomatic Generation of Inductive Assertions for Proving Program Correctness, Stanford Research Institute Project 2686 Interim Report, July 1974.
[39]. Good D. I. Toward a Man-Machine System for Proving Program Correctness, Ph. D. thesis, University of Wisconsin, June 1970.

[40]. А. Аграновский, В. Зайцев, Б. Телеснин, Р. Хади. Верицикация программ с помощью моделей. – Открытые системы, N12, 2003.
[41]. Ю. Г. Карпов, А. В. Толстяков. Аналитический метод верификации протоколов. – Автоматика и вычислительная техника, N1, 1985.
[42]. Ю. Г. Карпов. О корректности параллельных алгоритмовю – Программирование, N4, 1996.
[43]. Ю. Г. Карпов. Анализ корректности параллельной программы разделения множеств. - Программирование, N6, 1996.

დანართი 1

ალგორითმული ენების(C,C++,Java-ს მაგალითზე) ზოგადი სინტაქსი ბეკუს–ნაურის ფორმალიზმის დახმარებით.

შენიშვნა: გამოყენებული მეტასიმბოლოები:

მეტასიმბოლო ::= ნიშნავს განმარტების თანახმად;

მეტასიმბოლო <არატერმინალური სიმბოლო> ანუ ცნება, რომელიც განმარტებას საჭიროებს;

მეტასიმბოლო { } ნიშნავს განმეორებადობას 0–ჯერ და მეტად;

მეტასიმბოლო / ნიშნავს ალტერნატივას, ან–ან;

მეტასიმბოლო [ ] ნიშნავს შეიძლებს იყოს, შეიძლება არა;

<პროგრამა>::={<წინაპროცესორის დირექტივა>} {<ფუნქციის პროტოტიპის აღწერა>;} {<კლასების აღწერა>;} <ძირითადი main ფუნქციის აღწერა> / {<jar ფაილების იმპორტირება>;} {<კლასების აღწერა>;} <ძირითადი Main კლასის აღწერა> / package <პაკეტის სახელი>;
<კლასის აღწერა> ;{<კლასის აღწერა>;}

<წინაპროცესორის დირექტივა>::=#include “<”ბიბლიოთეკის_სახელი.h”>” / #define <მაკროსის სახელი> <მაკრო_განმარტება>

<ფუნქციის პროტოტიპის აღწერა>::=<დასაბრუნებელი ტიპი> <ფუნქციის სახელი>([<ფორმალური პარამეტრების სია>]);

<კლასის აღწერა>::= class „{“ {[<წვდომის სპეციფიკატორი>:] <ველების აღწერა>; }
{[<წვდომის სპეციფიკატორი>:]<მეთოდების აღწერა>;}
[public: <კონსტრუქტორების აღწერა>;
<დესტრუქტორის აღწერა>;] „}“ ; /
[<წვდომის სპეციფიკატორი>] [static] class
„{“ {[<წვდომის სპეციფიკატორი>] <ველის აღწერა>; }
{[<წვდომის სპეციფიკატორი>]<მეთოდის აღწერა>;}
[public <კონსტრუქტორების აღწერა>; „}“ ;

< main ფუნქციის აღწერა>::= main([<პარამეტრის აღწერა>])
„{“ {<ცვლადის აღწერა>;}

< ფუნქციის ტანი>;

return <გამოსახულება>; „}“

<ცვლადის აღწერა>::= <ტიპი> <ცვლადის იდენტიფიკატორი>{ ,<ცვლადის იდენტიფიკატორი> } ;

<ტიპი>::=<სტანდარტული ტიპი> / <რთული ტიპი> / <კლასის სახელი> / <სტრუქტურის სახელი>

<იდენტიფიკატორი>::= <ასო>{<ასო> / <ციფრი>}

<ასო>::=<პატარა ლათინური ასოები> / <დიდი ლათინური ასოები> / „_“

<Main კლასის აღწერა>::=public class <კლასის სახელი>

„{“ {<წვდომის სპ.> <ველის აღწერა>;}

{<წვდომის სპ.> <მეთოდის აღწერა>;}

{public <კონსტრუქტორების აღწერა>;}

public static void main(String <იდენტიფიკატორი>“[]“)

„{“ <main მეთოდის ტანი>: „}“

„}“ ;

<jar ფაილების იმპორტირება>::= import <ბიბლიოთეკის სახელი>;

<მეთოდების აღწერა>::= <მეთოდის სათაური> <მეთოდის ტანი>

<მეთოდის სათაური>::= <დასაბრუნებელი ტიპი> მეთოდის სახელი([<ფორმალური პარამეტრების სია>])

<დასაბრუნებელი ტიპი>::=<სტანდარტული ტიპი> / void / String

<ფორმალური პარამეტრების სია>::={ <ტიპი> <ფორმალური პარამეტრის სახელი> / <ტიპი> &<ფორმალური პარამეტრის სახელი> /<ტიპი> *<ფორმალური პარამეტრის სახელი> }

<მეთოდის ტანი>::= „{“ <ლოკალური ცვლადების აღწერა>;

{<ოპერატორი>;}

[return <გამოსახულებს>;] „}“

<მაკრო_განმარტება>::=<გამოსახულება> / <ოპერატორი>; { \ <ოპერატორი>;}

<სტანდარტული ტიპები>::=int / long int / short int / unsigned int / byte / float / double / char / unsigned char / bool / boolean

<რთული ტიპები>::= <მასივი> / <სტრუქტურა> / <ფაილი> / <კლასი>

<მასივის აღწერა>::= <ტიპი> <მასივის სახელი>“[“<განზომილება> „]“ / <ტიპი> <მასივის სახელი>“[“ „]“ / <ტიპი> „[“ „]“<მასივის სახელი>

<განზომილება>::=<მთელი დადებითი რიცხვი>

<რიცხვი>::=<მთელი რიცხვი> / <ნამდვილი რიცხვი> / <თექვსმეტობითი რიცვი>

<მთელი რიცხვი>::=[+ / –] <ციფრი>{<ციფრი>}

<ნამდვილი რიცხვი>::=[+/–]<ციფრი>{<ციფრი>}.<ციფრი>{<ციფრი>}/. <ციფრი>{<ციფრი>}

<თექვსმეტობითი რიცხვი>::=0x<თექვსმეტობითი ციფრი>

<თექვსმეტობითი ციფრი>::=<ათობითი ციფრი> /a/b/c/d/e/f/A/B/C/D/E/F

<სტრუქტურის აღწერა>::= struct <სტრუქტურის სახელი>

„{“ <ველების აღწერა>; „}“;

<ფაილის აღწერა>::= FILE <ფაილის სახელი>

<ოპერაციები>::=<არითმეტიკული ოპერაციები> / <თანადობის ოპერაციები> / <ლოგიკური ოპერაციები> / <ბიტური/თანრიგობრივი ოპერაციები>

<არითმეტიკული ოპერაციები>::= „+“ / „–“ / „*“ / „/“ / „%“

<თანადობის ოპერაციები>::= > / < / == / >= / <= / !=

<ლოგიკური ოპერაციები>::= && / || / !

<ბიტური/თანრიგობრივი ოპერაციები>::= & / | / ~ / ^ / << / >>

<გამოსახულება>::= <არითმეტიკული გამოსახულება> / <ლოგიკური გამოსახულება> / <ბიტური/თანრიგობრივი გამოსახულება>

< არითმეტიკული გამოსახულება > : : = [+ | -] < ტერმი > { + | - < ტერმი > }

< ტერმი > : : = < მამარავლი > { * | / | % < მამრავლი > }

< მამარავლი > : : = < რიცხვი > | < იდენტიფიკატორი > | (< გამოსახულება >)

<ლოგიკური გამოსახულება>::= [!] <ლოგიკური ტერმი>{ || <ტერმი>}

<ლოგიკური ტერმი>::=<ლოგიკური მამრავლი>{&&<ლოგიკური მამრავლი>}

<ლოგიკური მამრავლი>::=<თანადობა>“(“<ლოგიკური გამოსახულება“)“>

<თანადობა>::=<ოპერანდი1> <თანადობის ნიშანი> <ოპერანდი2>

<ოპერანდი>::=<ცვლადის იდენტიფიკატორი> / <რიცხვი>

<ოპერატორები>::=<მინიჭების ოპერატორი> / <ინკრიმენტი> / <დეკრიმენტი> / <პირობითი გადასვლის ოპერატორი> / <ციკლის ოპერატორი> / <ამორჩევის ოპერატორი> / <დაბრუნების ოპერატორი> / <ფუნქციის გამოძახების ოპერატორი>

<მინიჭების ოპერატორი>::= <კლასიკური მინიჭება> / <მინიჭება არითმეტიკული ოპერაციით> / <მინიჭება ბიტური/თანრიგობრივი ოპერაციით> / <მინიჭება ლოგიკური ოპერაციით>

<კლასიკური მინიჭება>::= <ცვლადის იდენტიფიკატორი> = <გამოსახულება>

<მინიჭება არითმეტიკული ოპერაციით>::= <ცვლადის იდენტიფიკატორი> <არითმეტიკული ოპერაცია> = <გამოსახულება>

<მინიჭება ბიტური/თანრიგობრივი ოპერაციით>::= <ცვლადის იდენტიფიკატორი> <ბიტური ოპერაცია> = <გამოსახულება>

<მინიჭება ლოგიკური ოპერაციით>::= <ცვლადის იდენტიფიკატორი> <ლოგიკური ოპერაცია> = <გამოსახულება>

<ინკრიმენტი>::=<pre ინკრიმენტი> / <post ინკრიმენტი>

<pre ინკრიმენტი>:=+ + <ცვლადის იდენტიფიკატორი>

<post ინკრიმენტი>:=<ცვლადის იდენტიფიკატორი>+ +

<დეკრიმენტი>::=<pre დეკრიმენტი> / <post დეკრიმენტი>

<pre დეკრიმენტი>:=– – <ცვლადის იდენტიფიკატორი>

<post დეკრიმენტი>:=<ცვლადის იდენტიფიკატორი>– –

<პირობითი გადასვლის ოპერატორი>::=if (<ლოგიკური გამოსახულება>) <შედგენილი ოპერატორი1>; [else <შედგენილი ოპერატორი2>;] /

if (<გამოსახულება>) <შედგენილი ოპერატორი1>; [else <შედგენილი ოპერატორი2>;] /

<შედგენილი ოპერატორი>::="{„ <ოპერატორი>; {<ოპერატორი>;} „}"

<ციკლის ოპერატორი>::=<ციკლი წინა პირობით> / <ციკლი შემდეგი პირობით> / <ციკლი პარამეტრით>

<ციკლი წინა პირობით>::= while (<გამოსახულება>) <ციკლის ტანი> /

while (<ლოგიკური გამოსახულება>) <ციკლის ტანი>

<ციკლი შემდეგი პირობით>::=do {<ციკლის ტანი> } while (<გამოსახულება>); /

do {<ციკლის ტანი> } while (<ლოგიკური გამოსახულება>);

<ციკლი პარამეტრით>::=for([<ციკლის პარამეტრის ინიციალიზაცია>]; [<ციკლის გაგრძელების პირობა>]; [<ციკლის პარამეტრის ცვლილების წესი>])

<ციკლის ტანი>;

<ციკლის ტანი>::=<ოპერატორი> / <შედგენილი ოპერატორი>

<ციკლის პარამეტრის ინიციალიზაცია>::=<ციკლის პარამეტრი>=<გამოსახულება>

<ციკლის პარამეტრი>::=<ცვლადის იდენტიფიკატორი>

<ციკლის გაგრძელების პირობა>::=<ლოგიკური გამოსახულება> / <არითმეტიკული გამოსახულება>

<ციკლის პარამეტრის ცვლილების წესი>::=<ციკლის პარამეტრი>=<გამოსახულება>

<ამორჩევის ოპერატორი>::=switch(<ცვლადის იდენტიფიკატორი>)

<ცვლადის მნიშვნელობა1>: <ოპერატორი>; break;

{<ცვლადის მნიშვნელობა1>: <ოპერატორი>; break;}

<ფუნქციის გამოძახების ოპერატორი>::= <ცვლადის იდენტიფიკატორი>=<ფუნქციის სახელი>([<ფაქტიური პარამეტრების სია>]); /

<ფუნქციის სახელი>([<ფაქტიური პარამეტრების სია>]);

<ფაქტიური პარამეტრების სია>::=<ცვლადის იდენტიფიკატორი>{<ცვლადის იდენტ.>}

<დაბრუნების ოპერატორი>::= return <გამოსახულება>;

```
/*****************************************************************************************************/
/*        ბლოკური სიმეტრიული დაშიფვრის ალგორითმის შესაბამისი პროგრამული   */
/*                                          კოდი                                                     */
/*****************************************************************************************************/
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
unsigned int Left_T(unsigned char*);
unsigned int Left_K(unsigned char*);
unsigned int Right_T(unsigned char*);
unsigned int Right_K(unsigned char*);
unsigned int F(unsigned int,unsigned int);
void IP(unsigned char*,int );
void IP1(unsigned char*,int );
unsigned int P(unsigned int,int);
unsigned int key_formiration();
void String_formiration(unsigned int,unsigned int,unsigned char*);
void dashifvra(unsigned char*);
void gashifvra(unsigned char*);
void key_array();
unsigned int  key_r,key_l;
unsigned int key[16];
int n;
unsigned char *String,*Key,*Kriptograma,*String1;
char s[9],name[20]="File Name";
```

```
/*******                    ძირითადი ფუნქცია                    ****************/

main()
{ int k;
  FILE *x,*y;
  union {int x[2];
         unsigned char y[8];
         }  Z;
 time_t t;     //   drois faqtori
 String=new unsigned char[8];
 Key=new unsigned char[8];
 Kriptograma=new unsigned char[8];
 String1=new unsigned char[8];
 cout<<"Input Key\n";
   for(int i=0;i<8;i++)
   cin>>Key[i];
   cout<<"n=";
   cin>>n;
key_array();
cout<<"Input Source File Name\n";
cin>>name;
  x=fopen(name,"r");
  y=fopen("kriptograma.txt", "w");
  k=x!=NULL;
  if(k==0){cout<<"Cannot open input file";
           getch();
           exit(0);}
      time(&t);
      printf("Today's date and time: %s\n", ctime(&t));
         while(k)
         {
```

```
    k=(fgets(s,9,x)!=NULL);
                        for(int i=0;i<8;i++)
                        String[i]=s[i];
                        dashifvra(String);
                        for(int i=0;i<8;i++)
                        Z.y[i]=Kriptograma[i];
                        fprintf(y," %d %d",Z.x[0],Z.x[1]);
                                            }
     time(&t);
     printf("Today's date and time: %s\n", ctime(&t));
 fclose(x);
 fclose(y);
cout<<"Press any key";
    getch();
    }

/ ***********                    დაშიფვრის შესაბამისი ფუნქცია          ***************/
    void dashifvra(unsigned char* String)
  {unsigned int S,Str_l,Str_r;
   unsigned int C;
      IP(String,n);
  Str_l=Left_T(String);
  Str_r=Right_T(String);
  for(int i=0;i<=15;i++)
  { C=F(Str_r,key[i]);
  C=P(C,n);          }
        S=Str_l;
        Str_l=Str_r;
        Str_r=S;
    String_formiration(Str_l,Str_r,Kriptograma);
     IP1(Kriptograma,n);     }
```

```
/ **********                გასაღებების მასივის ფორმირება                **************/
  void key_array()
 {key_l=Left_K(Key);
    key_r=Right_K(Key);
  for(int i=0;i<=15;i++)
  key[i]=key_formiration();
 }


  //   ***********************************************************************
  String_formiration(RN_1,LN,String1);
     IP1(String1,n);
}

/ ******************************          IP გადანაცვლება                ***********************/
   void IP(unsigned char* S,int n)
   {int l,t,j,m;
   unsigned char x,s;
   unsigned char M[64];
   for(int i=0;i<8;i++)
   { t=7;
    for(int k=i*8;k<i*8+8;k++)
    {
      M[k]=(S[i]>>t)&1;
       t--;     } }
   for(l=0;l<=63-n;l++)
   {s=M[l];
    M[l]=M[l+n];
    M[l+n]=s;     }
    for(int k=0,l=63-n+1;l<=63;l++,k++)
    {s=M[l];
```

```
    M[l]=M[k];
    M[k]=s;}
    for(int i=0;i<8;i++)
    {int p=7;
    S[i]=0;
      for(int k=i*8;k<i*8+8;k++)
        {M[k]<<=p;
         S[i]|=M[k];
         p--; } } }

// ***************************************************************************
void IP1(unsigned char* S,int n)
{int i,j;
unsigned char B[64],s,x;
 for(int l=0;l<8;l++)
  {
   int t=7;
   for(int k=l*8;k<l*8+8;k++)
    {B[k]=(S[l]>>t)&1;
     t--;
     }
     }
 int l;
for(i=n-1,l=63;i>=0;i--,l--)
    {s=B[i];
     B[i]=B[l];
     B[l]=s;   }
for(j=63;j>=n;j--)
  {s=B[j];
   B[j]=B[j-n];
   B[j-n]=s;}
```

```
 for(int i=0;i<8;i++)
     {int p=7;
      S[i]=0;
       for(int k=i*8;k<i*8+8;k++)
         {B[k]<<=p;
          S[i]|=B[k];
          p--; }
     }
}
// ***********************************************************************
unsigned int Left_K(unsigned char* Key)
 { union {unsigned int x;
         unsigned char y[4]; }   Z;
         for(int i=0;i<4;i++)
           Z.y[i]=Key[3-i];
           return Z.x;}

// ***********************************************************************
unsigned int Right_K(unsigned char* Key)
 { union {unsigned int x;
         unsigned char y[4]; } Z;
         for(int i=0;i<4;i++)
           Z.y[i]=Key[7-i];
           //cout<<"key_r="<<Z.x<<endl;
           return Z.x;}

// ***********************************************************************
unsigned int Left_T(unsigned char* T)
 { union {unsigned int x;
         unsigned char y[4]; } Z;
         for(int i=0;i<4;i++)
```

```
        {Z.y[i]=T[3-i]; }
          return Z.x;}
// ***********************************************************************
unsigned int Right_T(unsigned char* T)
 { union {unsigned int x;
        unsigned char y[4]; } Z;
        for(int i=0;i<4;i++)
          Z.y[i]=T[7-i];
          return Z.x;}

/******************        გასაღების ფორმირების ფუნქცია                  *****************/
unsigned int key_formiration()
{unsigned int M1,M2,M,key=0;
M1=key_r>>31;  //M1=key_r & 0x80000000
M2=(key_r>>30)&1;   //  M2=key_r & 0x40000000
key_r=(key_r<<2)|(M1<<1)|M2;
M1=key_l>>31;
M2=(key_l>>30)&1;
      key_l=(key_l<<2)|(M1<<1)|M2;
 M=0xFFFF0000;
 key=key_l&M;
  key|=(key_r&M)>>16;
  return key;}

// ***********************************************************************
void String_formiration(unsigned int l,unsigned int r,unsigned char* s)
{  union {unsigned char String[8];
       unsigned int S[2];  }   Z;
       Z.S[0]=l;
       Z.S[1]=r;
       for(int i=0;i<8;i++)
```

```
        s[i]=Z.String[i];
          for(int i=0;i<2;i++)
             {int k=s[i];
             s[i]=s[3-i];
             s[3-i]=k;}
          for(int i=4;i<6;i++)
             {int k=s[i];
               s[i]=s[11-i];
               s[11-i]=k;};
       }
 //  ************************************************************************
unsigned int F(unsigned int R,unsigned int K)
{union {unsigned int U1;
        unsigned short int U2[2]; }  Z;
unsigned short int x,y,a,b;
unsigned int C;
Z.U1=R;
x=Z.U2[0];
y=Z.U2[1];
Z.U1=K;
a=Z.U2[0];
b=Z.U2[1];
C=a*x^b*y;
return C;
}
 //
 ***************************************************************************************************************************
unsigned int P(unsigned int C,int n)
{unsigned char c[32],s;

for(int i=31;i>=0;i--)
```

```
{ c[i]=C&1;
  C>>=1;}
  for(int l=0;l<=31-n;l++)
        {s=c[l];
        c[l]=c[l+n];
        c[l+n]=s;}
  for(int t=0,l=31-n+1;l<=31;t++,l++)
  {s=c[l];
  c[l]=c[t];
  c[t]=s;
      }
  C=0;
for(int i=31;i>=0;i--)
C|=(c[i]<<(31-i));
return C;}
```

```
/*****************************************************************************************************/
/*      ბლოკური სიმეტრიული დაშიფრაციის ალგორითმის შესაბამისი პროგრამული   */
/*                                  კოდი                                     */
/*****************************************************************************************************/
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
unsigned int Left_T(unsigned char*);
unsigned int Left_K(unsigned char*);
unsigned int Right_T(unsigned char*);
unsigned int Right_K(unsigned char*);
unsigned int F(unsigned int,unsigned int);
void IP(unsigned char*,int );
void IP1(unsigned char*,int );
unsigned int P(unsigned int,int);
unsigned int key_formiration();
void String_formiration(unsigned int,unsigned int,unsigned char*);
void gashifvra(unsigned char*);
void key_array();
unsigned int  key_r,key_l;
unsigned int key[16];
int n;
unsigned char *String,*Key,*Kriptograma,*String1;
char s[10],name[20];

/*********************        ძირითადი ფუნქცია        ************************************/
main()
{ int k;
  FILE *x,*y;
```

```
 union {int x[2];
       unsigned char y[8];
       }   Z;
String=new unsigned char[8];
Key=new unsigned char[8];
Kriptograma=new unsigned char[8];
String1=new unsigned char[8];
cout<<"Input Key\n";
  for(int i=0;i<8;i++)
  cin>>Key[i];
  cout<<"n=";
  cin>>n;
key_array();
x=fopen("Kriptograma.txt","r");
y=fopen("gashifruli.txt","w");
 k=x!=NULL;
       while(k)
      { k=fscanf(x,"%d",&Z.x[0])!=EOF;
        k=fscanf(x,"%d",&Z.x[1])!=EOF;
                    for(int i=0;i<8;i++)
                      Kriptograma[i]=Z.y[i];
                       gashifvra(Kriptograma);
                       for(int i=0;i<8;i++)
                       s[i]=String1[i];
                       s[8]='\0';
                       fputs(s,y);
                       }
 fclose(x);
 fclose(y);
  cout<<"Press any key";
   getch();}
```

```
   // ********************************************************************
 void key_array()
 {key_l=Left_K(Key);
    key_r=Right_K(Key);
  for(int i=0;i<=15;i++)
  key[i]=key_formiration();
}

  /************************* დეშიფრაციის ფუნქცია ************************************/
 void gashifvra(unsigned char* Kriptograma)
 {unsigned int S,LN,RN,RN_1,L[15],R[15],LN_1;
   unsigned int C;
   int i;
     IP(Kriptograma,n);
LN=Left_T(Kriptograma);
RN=Right_T(Kriptograma);
    C=F(RN,key[15]);
    C=P(C,n);
  RN=LN;
   LN=LN^C;
   for(i=14;i>=0;i--)
   {C=F(LN,key[i]);
    C=P(C,n);
    S=LN;
    LN=RN_1^C;
    }
S=LN;
LN=RN_1;
RN_1=S;
  String_formiration(RN_1,LN,String1);
      IP1(String1,n);}
```

```
// ********************************************************************
   void IP(unsigned char* S,int n)
   {int l,t,j,m;
   unsigned char x,s;
   unsigned char M[64];
   for(int i=0;i<8;i++)
   {t=7;
    for(int k=i*8;k<i*8+8;k++)
    {M[k]=(S[i]>>t)&1;
      t--;     }
    }
   for(l=0;l<=63-n;l++)
   {s=M[l];
    M[l]=M[l+n];
    M[l+n]=s;
    }
    for(int k=0,l=63-n+1;l<=63;l++,k++)
    {s=M[l];
     M[l]=M[k];
     M[k]=s;}
     for(int i=0;i<8;i++)
     {int p=7;
     S[i]=0;
       for(int k=i*8;k<i*8+8;k++)
         {M[k]<<=p;
          S[i]|=M[k];
          p--; } } }

// ********************************************************************
void IP1(unsigned char* S,int n)
{int i,j;
```

```
unsigned char B[64],s,x;
 for(int l=0;l<8;l++)
  {
   int t=7;
   for(int k=l*8;k<l*8+8;k++)
    {B[k]=(S[l]>>t)&1;
     t--; } }
 int l;
for(i=n-1,l=63;i>=0;i--,l--)
    {s=B[i];
     B[i]=B[l];
     B[l]=s;}
for(j=63;j>=n;j--)
   {s=B[j];
    B[j]=B[j-n];
    B[j-n]=s;}
 for(int i=0;i<8;i++)
     {int p=7;
      S[i]=0;
       for(int k=i*8;k<i*8+8;k++)
         {B[k]<<=p;
          S[i]|=B[k];
          p--; } }  }

// ************************************************************************

unsigned int Left_K(unsigned char* Key)
 { union {unsigned int x;
         unsigned char y[4]; }   Z;
         for(int i=0;i<4;i++)
           Z.y[i]=Key[3-i];
```

```
        return Z.x;}

// ***********************************************************************
unsigned int Right_K(unsigned char* Key)
 { union {unsigned int x;
        unsigned char y[4]; } Z;
        for(int i=0;i<4;i++)
          Z.y[i]=Key[7-i];
          //cout<<"key_r="<<Z.x<<endl;
          return Z.x;}

// ***********************************************************************
unsigned int Left_T(unsigned char* T)
 { union {unsigned int x;
        unsigned char y[4]; } Z;
        for(int i=0;i<4;i++)
        {Z.y[i]=T[3-i];
 }
          return Z.x;}

// ***********************************************************************
unsigned int Right_T(unsigned char* T)
 { union {unsigned int x;
        unsigned char y[4]; } Z;
        for(int i=0;i<4;i++)
          Z.y[i]=T[7-i];
          return Z.x;}

// ***********************************************************************
unsigned int key_formiration()
{unsigned int M1,M2,M,key=0;
```

```
M1=key_r>>31;  //M1=key_r & 0x80000000
M2=(key_r>>30)&1;   //  M2=key_r & 0x40000000
key_r=(key_r<<2)|(M1<<1)|M2;
M1=key_l>>31;
M2=(key_l>>30)&1;
      key_l=(key_l<<2)|(M1<<1)|M2;
 M=0xFFFF0000;
 key=key_l&M;
   key|=(key_r&M)>>16;
   return key;}

// ********************************************************************
void String_formiration(unsigned int l,unsigned int r,unsigned char* s)
{  union {unsigned char String[8];
       unsigned int S[2];  }   Z;
       Z.S[0]=l;
       Z.S[1]=r;
       for(int i=0;i<8;i++)
       s[i]=Z.String[i];
         for(int i=0;i<2;i++)
            {int k=s[i];
            s[i]=s[3-i];
            s[3-i]=k;}
         for(int i=4;i<6;i++)
            {int k=s[i];
              s[i]=s[11-i];
              s[11-i]=k;};}
```

```
// ********************************************************************
unsigned int F(unsigned int R,unsigned int K)
{union {unsigned int U1;
       unsigned short int U2[2]; }  Z;
unsigned short int x,y,a,b;
unsigned int C;
Z.U1=R;
x=Z.U2[0];
y=Z.U2[1];
Z.U1=K;
a=Z.U2[0];
b=Z.U2[1];
C=a*x^b*y;
return C;
}

// **********************shecdomaa***************************************************
unsigned int P(unsigned int C,int n)
{unsigned char c[32],s;
for(int i=31;i>=0;i--)
{ c[i]=C&1;
  C>>=1;}
  for(int l=0;l<=31-n;l++)
           {s=c[l];
           c[l]=c[l+n];
           c[l+n]=s;}
 for(int t=0,l=31-n+1;l<=31;t++,l++)
 {s=c[l];
 c[l]=c[t];
 c[t]=s;
       }
```

```
C=0;
for(int i=31;i>=0;i--)
C|=(c[i]<<(31-i));
return C;}
```

დანართი 2.3

გაშიფვრის პროგრამის საწყისი ტექსტი. text.rtf ფაილის შიგთავსი.

ბილეთი 1

1. მოცემული გაქვთ მთელი ტიპის მასივი. იპოვეთ ამ მასივში მინიმალური რიცხვი და გაარკვიეთ იგი მარტივი რიცხვია თუ არა.
2. მოცემული გაქვთ ნამდვილი ტიპის მასივი. გაარკვიეთ ეს მასივი ზრდად მიმდევრობას წარმოადგენს თუ არა.
3. შეგაქვთ მთელი რიცხვი. გაარკვიეთ ამ რიცხვის ათობით ჩანაწერში გხვდებათ თუ არა ციფრი 1?

ბილეთი 2

1. მოცემული გაქვთ მთელი ტიპის მასივი. იპოვეთ ამ მასივში მაქსიმალური ელემენტი და გაარკვიეთ მისი ინდექსი ფიბონაჩის რიცხვს წარმოადგენს თუ არა.
2. მოც. x : array[1..30] of real;

დაითვალეთ შემდეგი ჯამი:

y=x[1]+x[1]*x[2]+ x[1]*x[2]*x[3]+ x[1]*x[2]*x[3]*x[4]+...+ x[1]*x[2]*x[3]*x[4]*...*x[30];

3. შეგაქვთ მთელი რიცხვი. გაარკვიეთ ამ რიცხვის ათობით ჩანაწერში

ზუსტად 2-ჯერ გხვდებათ თუ არა ციფრი 7?

ბილეთი 3

1. მოცემული გაქვთ მთელი ტიპის მასივი. იპოვეთ ამ მასივში მინიმალური და მაქსიმალური ელემენტების ჯამი.
2. მოცემული გაქვთ ნამდვილი ტიპის მასივი. იპოვეთ ამ მასივში მხოლოდ უარყოფითი რიცხვების საშუალო არითმეტიკული.
3. შეგაქვთ მთელი რიცხვი. გაარკვიეთ რამდენ თანრიგაა ეს რიცხვი.

დანართი 2.4

გაშიფვრის პროგრამის მუშაობის შედეგი – შიფროგრამა. kriptograma.txt ფაილის შიგთავსი.

```
-1412240438 -1499980092 -943550610 1315146646 -55976188 -1796104486 -959209538 -
397947659 741342585 1993045473 -1211379772 1888149841 180243550 637797942 -
210263507 381222293 -1305176516 -675119218 -325772951 -1246843223 -1805524615
165591097 1914706173 -1565853727 -1966836349 -202727527 -1370311353 1736707996
1162885980 -1415345307 1911436639 513120521 -787022145 66050906 -1262859197
2133711697 -290999837 -1639019744 810119007 -1394466259 -488040860 636159614
1812634239 1425569573 742997954 1876670827 1410596653 1230736174 -829574342 -
1315205172 -1126755666 304405730 -1126835694 -1200151984 1405766211 -47189139 -
535065872 113840088 -1359613064 -1221887565 961553555 150270697 975771990
1459057246 -2040415788 11021846 1333503430 773136965 492264132 -1021974651 -
1527927097 1437744945 -1243436157 -19412157 302256965 2047060998 371524470
1603198357 201300913 -605038857 1911436639 513120521 413961845 213994664 -
1382583623 1672985682 1974441808 -767774122 -2086388710 -1564034069 -805198576
775954509 -829574342 -1315205172 621758006 228828028 748572044 -1343392345 -
1809383947 -863561795 742997954 1876670827 54391827 1965678395 -2139195563
1202756062 1333503430 773136965 -1533425023 360262404 -1235329823 -1816753654 -
1243436157 -19412157 302256965 2047060998 -1845090018 -195086546 -501338063
1460496967 1333503430 773136965 -1824223050 1654277618 1612994332 1340069081 -
1243436157 -19412157 302256965 2047060998 388037290 1485417374 555074148 -
1251123373 -589921797 -1620799379 -1801451439 -1490779262 -2060267101 1734074259
-769659899 -501572255 -1390665044 -193974580 1627647903 -1688579258 2016963778
869060664 -383749066 -571422120 1911436639 513120521 471461494 686227751 -
1382583623 1672985682 1974441808 -767774122 -1437332752 1482020497 1590927140 -
1628116632 789784178 2104361114 1911436639 513120521 -767691771 1731983638 -
1382583623 1672985682 1974441808 -767774122 -1478120533 -1363624053 -849247615
952614316 1224764071 -85867075 108758 -868040452 1333503430 773136965 492264132
-1021974651 -1527927097 1437744945 337800537 1936796935 -1228749203 -1090066196
-1177861173 6698562 1911436639 513120521 413961845 213994664 -925063416
1818178006 -681188245 -1026957685 781006820 -2011590711 -829574342 -1315205172
621758006 228828028 748572044 -1343392345 -126019034 -1851147931 54391827
1965678395 -1965897601 -1202506763 1333503430 773136965 -1533425023 360262404 -
1235329823 -1816753654 337800537 1936796935 -324249865 209396096 1365470009 -
644320182 -1835746002 70227625 604203086 -1493936879 136223751 -817989997
1688145350 288749098 1480415551 -1784432279 8346900 -999122118 1186155713
442682923 13283672 -95588215 -1071459102 -1344210788 1898024761 1122788398 -
151236081 -1728109169 2095269701 -1904063382 -418940633 -805113018 -1818549760
422713041 -1569988225 886159032 -652269412 -199046724 -798461765 1301391315 -
611617105 601701535 749340685 -2021732625 1849862311 1928534617 1523921369
644434780 1071808926 -593491564 -1334638020 -685373640 -1331496690 979875595 -
839448283 -1613410657 -1828602414 -322944782 1891785345 1119350608 -154050493
1480897928 2095269701 -1904063382 13283672 -95588215 -405184257 1121956924
482435600 1752393466 686199740 -544515783 -530981896 2020415944 2003749036
99059531 -700426599 -659339189 -1503929655 320358232 748464065 -683843332
1816609484 -1694997623 -530981896 2020415944 -1046363058 -1985344464 749340685 -
2021732625 382258395 1757302891 -1986993560 -858951018 -145933103 -334182992
531443491 1033285254 -520274622 408037034 459045903 39841441 2028208990 -
2032244712 -1785810820 -732380915 -242395590 601200427 906451933 1085331896
129418073 -326820025 -189301559 1226630063 -1067087239 1537462456 1872037745
520336059 -1972235349 -1753032794 125903380 276630312 -779860673 -207504705
1754277991 -597675985 -428819193 1208220293 1531584860 207230032 1092630586 -
1843816428 2113087115 272490835 406278757 768546590 -1876082009 -463079565 -
1631218302 -1019882026 1290098780 651181470 -1092343444 1579804892 -1068718761 -
2130083029 -357707356 1618688295 754621212 1875611377 113313009 960494088 -
416930713 -1278980313 955792714 -1885335994 -2092672010 98316576 1202821999
1727756524 -928929625 744968634 1479192222 -274687380 -695579392 626305493
532579541 -1774908649 -1451405088 -1283031908 -986037609 -1415198564 -1106126912
-930053994 180333185 -580658866 1050205371 -157251976 -188915442 1217945250 -
```

633673561 561705346 -1795933523 1834340260 1050205371 -157251976 -209276417
677503866 1645363234 680978590 -300859867 383552645 167680303 307110342
2095553212 -1188105569 -1265717086 650279590 -131318800 -943586181 -131318800 -
943586181 955811335 -1985685278 820381550 -724781091 1620763332 -1254754990
751219688 -1030006175 31425707 418832289 -684527988 -1976437805 43319156 -
232729093 -1226324390 -1192061793 474892626 977746526 388028286 2005973058
235052432 -1730202757 -1301097503 -780593630 -1034255869 791652315 -1666278703
1289131878 -2102989911 640655207 -55529851 1865622887 -1533378821 -542631825
1043729615 1271991388 -1488496530 1277087323 992334287 -827504136 618883635
2140861671 1243987925 -683531768 -658130949 -1056996842 203534637 1999473099
265413988 -1426546304 -928929625 744968634 -434949453 1831724698 -2057041935
461477779 -232028081 -88030042 1690992633 -1573494121 1642856975 -1160861362 -
223019907 -1437365509 2074449020 -969449913 -1370712339 -1711124209 -643467441 -
384900058 -1776423050 2035744170 1581903147 576866200 -1272973068 -1833291316 -
691061247 100482185 155056274 -1708292273 -711807320 1338777832 -1415111942 -
1658981726 -65499552 -1795311815 -273988321 569465523 -1638106524 -711523389 -
944664022 1625439324 2024519826 380661905 438587879 -1696652186 582925751
647897394 629852370 80184043 -2041457401 1318862701 41886035 1868233656
1014748700 842593043 -1371040514 1702462333 320552429 1204874547 1101660849
1376666016 -593627993 -1913019492 -1525025465 -740785194 1276759038 -1407741982
1592053042 1637065080 324515222 -2057247222 1464455710 88160968 184593453
414790742 1179062 -1241485755 -206547566 307557423 -938769556 1155102964
74111219 -1273951161 1191405720 -902522745 779328141 1673461556 215061705
1846814639 -1062440050 -2042037335 438587879 -1696652186 582925751 647897394
629852370 80184043 -1035891005 -586485817 454018005 879304226 124563218 -
1730915651 1935925679 -1091576963 1818922610 -1704110919 1189415953 487103776
1469590602 -684574793 -492042217 -1629424364 -623663617 -1725966875 -1649212328
1086035297 -1495783158 44392095 -1970555261 -1614404608 1648310283 1138189672 -
15280978 -451090287 -770616227 -1764923715 -206547566 307557423 -688741850
2056815184 -478385083 1699283496 1832277748 -2129244490 730342269 -1811294457 -
2107444942 -1752915022 -244077069 2039943905 274494364 1928420176 510819873 -
1240893669 41886035 1868233656 1014748700 842593043 -796627312 580306642 -
1240128199 832447589 508194095 1350725561 1424887702 751970957 -1863886337 -
886679761 1749924778 -137531497 559988566 -207660529 980644283 380548762 -
1036640584 1278876507 1191528465 -809878895 155056274 -1708292273 -711807320
1338777832 -1415111942 -1658981726 -65499552 -1795311815 -273988321 569465523 -
1638106524 -711523389 -944664022 1625439324 2024519826 380661905 438587879 -
1696652186 582925751 647897394 629852370 80184043 -1035891005 -586485817
454018005 879304226 124563218 -1730915651 1935925679 -1091576963 -1133840777
1223948421 1189415953 487103776 1469590602 -684574793 -492042217 -1629424364 -
726743033 1424439376 -353245554 1480571432 383818464 1423618470 1916673439 -
1494438576 926289142 -1489146684 842657368 -89930741 2056865384 -603235042
1748639350 -1845112854 833060441 -1335936266 217000199 -585352941 849065277 -
1975657625 581681124 -4049067 -1123660395 -1960416181 1889106050 1755240375 -
1110538181 -73597443 -551658640 -360033226 -1165582463 -435042251 -2022987492 -
2047850136 -1906977410 539585921 -2051211981 -967804249 -1139483185 -592743584
889871810 1921037692 -1017984432 -2032477103 1115227458 50450555 -1948389541
1830694691 -1812923905 815887470 -1482974261 -1441094486 1314239587 -770966607
1511943920 -1573733414 1748639350 -1845112854 -1739546957 1169411309 2145520772
1399252326 -729829310 633948068 -273988321 569465523 -688741850 2056815184
1822240493 -997537792 -1424639511 -761924740 -1893688356 440500686 -1282585497
1067655372 2103946310 268468177 -1806858508 1279037494 -1393181261 -332329226 -
688354711 292924793 -1021855364 182660688 658806539 124689884 -143123666
987070857 -1896591128 -1154008702 874441434 -508102357 -633996200 463327753 -
1971932096 1739853033 444137178 -727677302 -1694945781 1783866758 -1799971874
744473100 1382068843 -2011771155 -323669889 1077776691 -1638106524 -711523389
283131116 -431762023 -1093778504 1482755229 -1244217630 472506458 -982849627
600908752 646769149 800111520 99392268 1199078001 -72745963 -262818547 -23190173
1758025077 738677548 -681041452 2020367935 22935428 720293514 225547897
784421148 -1185696005 -582654220 2009637124 892728195 -392292062 -2064394542 -
644384691 -1128845303 -1730849807 -353245554 1480571432 -495154721 1401557532
2130026706 1891495334 926289142 -1489146684 842657368 -89930741 2056865384 -
603235042 1748639350 -1845112854 833060441 -1335936266 217000199 -585352941
849065277 -1975657625 581681124 -4049067 -1123660395 -1960416181 18891060