დაპროგრამების პარადიგმა და ინფორმაციის დამუშავების ასპექტები ფუნქციონალურ პარადიგმაში

ნათელა არჩვაძე*, მერაბ ფხოველიშვილი**

*ივ.ჯავახიშვილის თბილისის სახელმწიფო უნივერსიტეტი, თბილისი, 0128, ჭავჭავაძის გამზ. №1.
**ნიკო მუსხელიშვილის გამოთვლითი მათემატიკის ინსტიტუტი, თბილისი, 0193, აკურის ქ. №7.

ანოტაცია

სტატიაში განხილულია დაპროგრამების ძირითადი პარადიგმები. იმპერატიული, იგივე პროცედურული ენები შედარებულია ფუნქციონალურ ენებთან. განხილულია ფუნქციონალურ პარადიგმაში ინფორმაციის დამუშავების ასპექტები. ცოდნის წარმოდგენისთვის გამოყენებულია სიის სტრუქტურები, შემოთავაზებულია ინფორმაციის დამუშავებისა და ძებნის თანამედროვე, ეფექტური მეთოდები. ნაჩვენებია, თუ როგორ აიგება ფუნქციონალური ენა LISP-ზე სხვადასხვა ტიპის სისტემების პროტოტიპები, კერძოდ, ობიექტ-ორიენტირებული და პარალელური სისტემები.

I. დაპროგრამების პარადიგმა

ცნება "დაპროგრამების პარადიგმა" განმარტებულია შემდეგნაირად: "**დაპროგრამების პარადიგმა** - ეს არის მიდგომების, მეთოდების, სტრატეგიების, იდეებისა და ცნებების ერთობლიობა, რომელიც განსაზღვრავს პროგრამის დაწერის სტილს" [1]. თანამედროვე ინდუსტრიაში ძალზე ხშირად დაპროგრამება განისაზღვრება პროგრამისტის ინსტრუმენტების ნაკრებით - დაპროგრამების ენით და ოპერაციული სისტემით.

ტერმინის ისტორიის შესახებ

ტერმინი "პარადიგმა" შემოიღო **ტომას კუნმა**, რომელმაც პირველად ეს ტერმინი ახსენა თავის წიგნში "სამეცნიერო რევოლუციების სტრუქტურა". კუნი პარადიგმას უწოდებდა სამეცნიერო შეხედულებების არსებულ სისტემას, რომლის ჩარჩოშიც მიმდინარეობს კვლევა. კუნის მიხედვით სამეცნიერო დისციპლინის განვითარებისას შესაძლოა ერთი პარადიგმა შეიცვალოს მეორე პარადიგმით (მაგალითად, პტოლემეას ჰელეოცენტრული ციური მექანიკა შეიცვალა კოპერნიკის ჰელეოცენტრული სისტემით). ამასთან, შეიძლება გაკვეული დროის განმავლობაში ძველმა პარადიგმამ გააგრძელოს თავისი არსებობა და განვითარებასაც კი. ეს იმის გამო, რომ ზოგიერთი მისი მომხრე რაღაც მიზეზების გამო ვერ ახერხებს გადაერთოს სხვა პარადიგმაში.

ტერმინი "დაპროგრამების პარადიგმა" პირველად გამოიყენა ტიურინგის პრემიის ლაურიანტმა რობერტ ფლოიდმა (R.W.Floyd) 1979 წელს. ფლოიდი აღნიშნავდა, რომ დაპროგრამებაში შეიძლება დავაკვირდეთ მოვლენებს, რომლებიც მსგავსი არიან კუნის პარადიგმებისა, მათგან მხოლოდ იმით განსხვდებიან, რომ დაპროგრამების პარადიგმები არ არიან ურთიერთგამომრიცხავი. ფლოიდის აზრით: "მაშინ როცა პროგრესი დაპროგრამების ხელოვნებაში მუდმივად მოითხოვს პარადიგმების

ჩამოყალიბებასა და განვითარებას, ცალკეული პროგრამისტის დახელოვნებისთვის მოითხოვება, რომ მან გააფართოოს თავისი პარადიგმების რეპერტუარი" [1]. ამრიგად, რობერტ ფლოიდის აზრით, პროგრამისტმა საჭიროა შეუთავსოს ერთმანეთს პარადიგმები და ამით გაამდიდროს თავისი დაპროგრამების ინსტრუმენტარია, განსხვავებით კუნის მიერ მეცნიერებაში აღწერილი პარადიგმისგან.

მოვიყვანოთ ტერმინის "დაპროგრამების პარადიგმების" განმარტებები. ისინი ხშირად ძალზე განსხვავდებიან ერთმანეთისგან. მაგალითად: **დიომიდის სპინელისი** (D.D.Spinellis) იძლევა შემდეგ განმარტებას: "სიტყვა "პარადიგმა" გამოიყენება დაპროგრამებაში აღნიშვნების (ნოტაციის) განსაზღვრისთვის, რომელიც გამოყოფს ზოგად საშუალებას (მეთოდს) პროგრამის რეალიზაციისთვის" (ორიგინალში: "The word paradigm is used in computer science to talk about a family of notations, that share a common way for describing program implementations).

დენის ბობროვი (D.G.Bobrow) პარადიგმას განსაზღვრავს როგორც "დაპროგრამების სტილს, რომლითაც პროგრამისტის განზრახვები აღიწერება". **ბრიუს შრაივერის** (Bruce Shriver) აზრით დაპროგრამების პარადიგმა არის "პრობლემის ამოხსნის მოდელი ან მიდგომა", **ლინდა ფრიდმანი** (Linda Friedman) – პარადიგმა არის "დაპროგრამების პრობლემების გადაწყვეტის მიდგომა". **პამელა ზეივი** (Pamela Zave) _ "აზროვნების საშუალება კომპიუტერულ სისტემების შესახებ. (ორიგინალში: "way of thinking about computer systems".)

პიტერ ვეგნერი (Peter Wegner) გვთავაზობს ამ ტერმინის განმარტების სხვა მიდგომას. მასთან პარადიგმა განისაზღვრება, როგორც "დაპროგრამების ენების კლასიფიკაციის შესაძლებლობა იმ პირობების შესაბამისად, რომელთა შემოწმებაც შესაძლებელია". **ტიმიტი ბადდი** (T.A.Budd) გვთავაზობს ტერმინს "პარადიგმა" გავიგოთ, როგორც "საშუალება იმისა, თუ რას ნიშნავს გამოთვლა, და ამოცანები, რომლებიც კომპიუტერზე უნდა ამოიხსნან, როგორ უნდა იყვნენ სტრუქტურირებული და ორგანიზებული".

ასე, რომ დაპროგრამების პარადიგმა წარმოადგენს (და ამავე დროს განსაზღვრავს) იმას, თუ როგორ ხედავს პროგრამისტი პროგრამის შესრულებას. მაგალითად, ობიექტებზე ორიენტირებული დაპროგრამებისას პროგრამისტი პროგრამას განიხილავს როგორც ურთიერთდაკავშირებული ობიექტების ნაკრებს, მაშინ როცა ფუნქციონალური დაპროგრამება პროგრამას განიხილავს როგორც ფუნქციების გამოთვლის ჯაჭვს.

ყოველ პარადიგმას ჰყავს თავისი მიმდევრები და აქვს ამოცანათა კლასი, რომლისთვისაც ის საუკეთესოა. დაპროგრამების ხარისხის შეფასების მიზნით იყენებენ სხვადასხვა პრიორიტეტებს _ განასხვავებენ მუშაობის ინსტრუმენტებსა და მეთოდებს, და შესაბამისად, აზროვნების სტილსა და შემოქმედებით საშუალებებს. თუმცა, პარადიგმის არჩევაზე შეიძლება გავლენა მოახდინოს ინდივიდუალურმა გამოცდილებამ, გონების ყაიდამ, მოდამ ან სხვისმა რჩევამ.

ზოგჯერ კონკრეტული პიროვნება ერთ-ერთი პარადიგმის ისეთი ძლიერი მიმდევარია, რომ კამათი კომპიუტერთან დაკავშირებულ წრეებში სხვადასხვა პარადიგმის ნაკლსა და უპირატესობაზე ღებულობს ე.წ. Hჰოლივარის (*holy war - წმინდა, საღმრთო ომი)* ხასიათს. საზოგადოდ, ჰოლივარი - შეტყობინებების გაცვლაა ინტერნეტ ფორუმებსა და ჩეთებში, სადაც მიმდინარეობს ცხარე დისკუსია, რომლის მონაწილეები ცდილობენ დაუმტკიცონ ერთმანეთს ერთი ან რამდენიმე ალტერნატივის უპირატესობა.

დაპროგრამების ძირითადი პარადიგმები

არსებობს პროგრამირების შემდეგი პარადიგმები:

- **იმპერატიული,**
- **ფუნქციონალური,**
- **ლოგიკური,**
- **ობიექტზე ორიენტირებული.**

პროფესიონალების წრეში გამოყენებითი დაპროგრამების ძირითადმა პარადიგმამ - **იმპერატიულმა** მართვამ და **პროცედურულ-ოპერატორულმა** სტილმა უკვე 50 წელზე მეტია, რაც პოპულარობა მოიხვეჭა გამოთვლითი და ინფორმაციული პროცესების ორგანიზაციისას. უკანასკნელი ათწლეულის განმავლობაში ინფორმაციის გეოგრაფია ფართოდ გაიშალა და გახდა მასიური მოხმარებისა და გართობის საშუალებაც. ამან შეცვალა სისტემის შეფასების კრიტერიუმები და უპირატესობა მიენიჭა ინფორმაციის ამორჩევისა და დამუშავების მეთოდებს.

კომპიუტერული ერის დასაწყისიდანვე ჩამოყალიბდა დაპროგრამების ზოგადი პარადიგმები: **გამოყენებითი, თეორიული** და მათ შორის, **ფუნქციონალური პარადიგმები** და მათ დღემდე მყარი საფუძველი აქვთ.

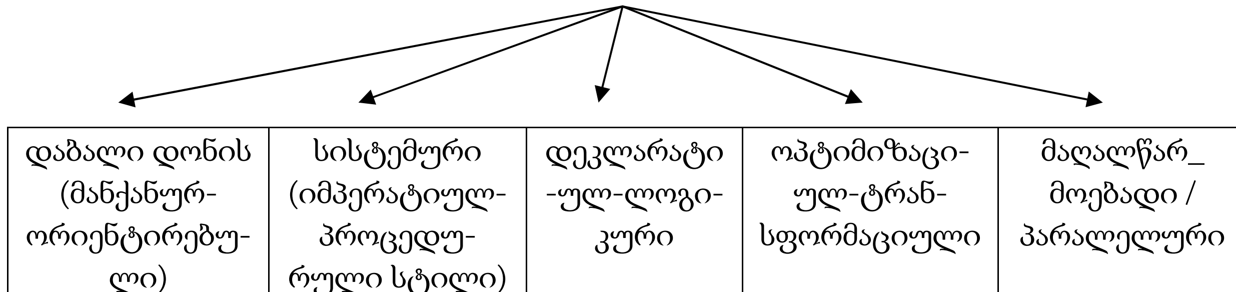
გამოყენებითი დაპროგრამება გულისხმობს ინფორმაციულ და გამოთვლების პროცესების კომპიუტერიზაციას - რიცხვით დამუშავებას. ეს კვლევები დაიწყო საკმაოდ ადრე, ჯერ კიდევ კომპიუტერების გაჩენამდე. ყველაზე ადრე სწორედ რიცხვით გამოთვლებში იქნა მიღებული პრაქტიკული შედეგები. ბუნებრივია, რომ აქ მოქმედებების წარმოსადგენად საკმარისია ოპერატორული სტილი. პრაქტიკაში მიღებულია **შაბლონებისა და ბიბლიოთეკურ პროცედურების** გამოყენება, ნაცვლად ექსპერიმენტებისა. დიდი ყურადღება ეთმობა მეცნიერული შედეგების სიზუსტეს და მდგრადობას. ენა Fortran გამოყენებითი დაპროგრამების ვეტერანია. მან მხოლოდ ბოლო ათეულ წლებში დაუთმო პოზიციები Pascal-ს და C-ს, ხოლო სუპერკომპიუტერებზე პარალელური და ასევე ფუნქციონალური დაპროგრამების ენას Sisal.

თეორიული დაპროგრამება არის ის მიმართულება, სადაც ხდება გამოქვეყნება მეცნიერული ექსპერიმენტების შედეგებისა დაპროგრამებისა და ინფორმატიკის დარგში. დაპროგრამება ცდილობს წარმოადგინოს თავისი ფორმალური მოდელები, აჩვენოს მათი მნიშვნელობა და ფუნდამენტურობა. ამ მოდელებში ძირითადი ცნებები მათემატიკიდან იქნა შემოტანილი და ისინი დამკვიდრდნენ როგორც ალგორითმული მიდგომა ინფორმატიკაში. განვითარდა სხვადასხვა მეთოდიკები პროგრამის სისწორის დამტკიცების, გამოთვლების ეფექტურობის, კორექტურობის, სემანტიკური სისწორის დამტკიცების მიზნით. მაგალითად, **წიგნიერი, იგივე ლიტერატურული დაპროგრამება** (Literate programming, Грамотное программирование)[2]. დონალდ კნუტმა (Donald Knuth) ეს ტერმინი შემოიღო 1981 წელს TeX -თან დაკავშირებით და ნიშნავდა დაპროგრამებას ფსევდოკოდის ფრაზებით, რომლებიც შემდეგ გადადიან ზუსტ მაკროში. ეს ხდება მარტივი უტილიტის საშუალებით ტექსტური ფაილიდან, რომელიც შეიცავს ტექსტურ კონცეფციას, თვით კოდს და ფსევდოკოდს.

სტანდარტული Pascal-ისა და Algol-ის ქვესიმრავლე იყო საცდელი მასალა თეორიული დაპროგრამებისთვის, ხოლო შემდეგ ისინი შეცვალა LISP -ის დიალექტებმა ML, Miranda, Sceme და სხვა. მათ შემდეგ მიემატათ C და Java.

ფუნქციონალური დაპროგრამებამ მათემატიკური კვლევა შეიტანა ხელოვნური ინტელექტის და ინფორმატიკის გამოკვლევებში. **ჯონ მაკარტიმ** LISP-ის [3,4,5,6] იდეების ჩამოყალიბებისას მიუთითა მისი დადებითი მხარეები: აბსტრაქტული მიდგომა ინფორმაციის წარმოდგენისას, ლაკონური, უნივერსალური სტილი ფუნქციის აგებისა, სხვადასხვა კატეგორიის ფუნქციის შესრულების ცხადად განსაზღვრული თანმიმდევრობა, თავისუფლება რეკურსიული წყობების აგებისას, წინასწარ მეხსიერების განაწილების უარყოფა, განსაზღვრების მოქმედების არის შეზღუდვებზე უარის თქმა. წინასწარ ჩამოყალიბებულობა და მეთოდური საფუძვლიანობა LISP-ის პირველი რეალიზებებისას იძლეოდა იმის საშუალებას, რომ დაგროვილიყო გამოცდილება ახალი ამოცანების ამოხსნისას, და მომზადებულიყო თეორიული და გამოყენებითი დაპროგრამებისთვის. ამჟამად არსებბოს ასობით ფუნქციონალური დაპროგრამების ენა, რომლებიც ორიენტირებული არიან სხვადასხვა კლასის ამოცანებზე და ტექნიკურ საშუალებებზე.

ამოსახსნელი **ამოცანების სირთულის გაზრდის** შესაბამისად მოხდა დაპროგრამების ძირითადი მეთოდებისა და საშუალებების ჩამოყალიბება. ინფორმაციის დამუშავების ტექნიკური დეტალების სიღრმისა და ზოგადობის შესაბამისად მოხდა დაპროგრამების პარადიგმების ფენებად დაყოფა. გამოყოფენ დაპროგრამების სხვადასხვა სტილს: **ქვედა დონის ანუ მანქანურ-ორიენტირებულს, სისტემური, დეკლარატიულ - ლოგიკური, ოპტიმიზაციურ - ტრანსფორმაციულ და მაღალწარმოებად ანუ პარალელურ დაპროგრამებას.**



დაბალი დონის დაპროგრამებას ახასიათებს მიმართულობა კომპიუტერის არქიტექტურაზე. ყურადღების ცენტრში არის მოწყობილობის კონფიგურაცია, მეხსიერების მდგომარეობა, ბრძანებები, მართვის გადაცემა, მოვლენათა რიგი, გამონაკლისი სიტუაციები, მოწყობილობის რეაქციის დრო და რეაგირების წარმატებულობა. ასემბლერმა გამომსახველობითი საშუალებებით დაუთმო ენა C–ის და pascal–ს, თუმცა, თუ მოხდება მისი ინტერფეისის გაუმჯობესება, იგი ისევ აღიდგენს თავის პოზიციებს.

სისტემური დაპროგრამება დიდი დროის განმავლობაში ვითარდებოდა სერვისული და შეკვეთილი სამუშაოების პრესის ქვეშ. ასეთი ტიპის სამუშაოებისთვის დამახასიათებელია სტაბილურობა მრავალჯერადი გამოყენებისთვის და არა შემოქმედებითი მიდგომა. ასეთ პროგრამებს აქვთ დამუშავების კომპილატორული სქემა, თვისებების სტატიკური ანალიზი, ოპტიმიზაცია და კონტროლი. ამ არისთვის დომინირებს დაპროგრამების იმპერატიულ_პროცედურული სტილი და წარმოადგენს

გამოყენებითი დაპროგრამების გაფართოებას. ის უშვებს სტანდარტიზაციას და მოდულურ დაპროგრამებას.

მაღალწარმოებადი დაპროგრამება მიმართულია განსაკუთრებული ამოცანების ამოსახსნელად. ბუნებრივი რეზერვი კომპიუტერების წარმოებადობის გასაზრდელად არის პარალელური პროცესები. მათი ორგანიზაცია ითხოვს დროითი დამოკიდებულებების ზუსტ ანგარიშს და მოქმედებათა მართვის არაიმპერიულ სტილს. სუპერკომპიუტერები, რომლებიც მხარს უჭერენ მაღალწარმოებად გამოთვლებს, ითხოვენ სისტემური დაპროგრამების განსაკუთრებილ ტექნიკას. პარალელური არქიტექტურის სისტემებისა და პროცესების Gგრაფულ-ქსელური მიდგომამ გამოხატულება ნახა პარალელური დაპროგრამების სპეციალურ ენებსა და სუპერკომპიუტერებში.

დეკლარატიული (ლოგიკური) დაპროგრამება გაჩნდა როგორც გამარტივებული ფუნქციონალური დაპროგრამება მათემატიკოსებისა და ლინგვისტებისთვის, რათა ამოეხსნათ სიმბოლური დამუშავების ამოცანები. განსაკუთრებულ მომხიბლავი აღმოჩნდა არადეტერმინიზმი, რომელიც ანთავისუფლებდა წინასწარი დალაგებისგან ფორმულილების დამუშავების დაპროგრამებისას. პროცესების გაჩენის პროდუქციული სტილი დაბრუნებებით საკმარისად ბუნებრივია ექსპერტების ცოდნის ფორმალიზებული წარმოდგენის ლინგვისტური მიდგომისთვის.

ტრანსფორმაციულმა დაპროგრამებამ მეთოდოლოგიურად გააერთიანა დაპროგრამების ოპტიმიზაციის, მაკროგენერაციისა და ნაწილობრივი გამოთვლების ტექნიკა. ამ დარგის ცენტრალური ცნება არის ინფორმაციის ექვივალენტობა. შერეული გამოთვლები, გადატანილი მოქმედებები, "ზარმაცი" (ленивое программирование) დაპროგრამება, შეჩერებული პროცესები და ა.შ. გამოიყენება როგორც ინფორმაციის დამუშავების ეფექტურობის გაზრდის მეთოდები.

"**ზარმაცი დაპროგრამება" იგივე "გადატანილი გამოთვლები" (ინგ.** lazy evaluation) გულისხმობს, რომ გამოთვლები გადაიტანება მანამ, სანამ არ გახდება საჭირო ამ გამოთვლების შედეგები. ეს იძლევა საშუალებას შემცირდეს გამოთვლების მოცულობა იმ გამოთვლების ხარჯზე, რომელთა შედეგები არ იქნება გამოყენებული. პროგრამისტს შეუძლია აღწეროს დამოკიდებულებები ფუნქციებს შორის და არ ადევნოს თვალი, ხდება თუ არა "ზედმეტი გამოთვლები". ზარმაცი დაპროგრამება ბუნებრივად ჩაჯდა ფუნქციონალური პარადიგმაში, ვინაიდან ფუნქციონალურმა დაპროგრამების ენებმა, რომლებშიც ის რეალიზებულია, თავი დაიმკვიდრეს როგორც ისეთმა ინსტრუმენტების მქონემ, რომლებიც მოსახერხებელია პროტოტიპირებისთვის (Software Prototyping), მათემატიკური უზრუნველყოფის სწრაფი დამუშავებისთვის და ასევე, ელექტრო-გამოთვლითი მანქანების დაპროექტებისთვის.

დაპროგრამების პარადიგმების შემდგომი განვითარება დაკავშირებულია გამოყენებით დაინტერესებულობასთან. ხდება გამოთვლითი საშუალებების გადასვლა ტექნიკური ინსტრუმენტარიის კლასიდან საყოფაცხოვრებო მოწყობილობებამდე. გაჩნდა იმის ნიადაგი, რომ განახლდეს დაპროგრამების ისეთი მიდგომები, რომლებიც ნაკლებად განვითადენენ კომპიუტერების დაბალი ტექნოლოგიურობისა და წარმადობის გამო. საინტერესო გახდა განვითარება კვლევითი, ევოლუციური, კოგნიტიური და ადაპტირებული მიდგომებისა.

ევოლუციური მიდგომა კარგად ჩანს **ობიექტურ-ორიენტირებული დაპროგრამების** კონცეფციაში, რომელიც თანდათან გადაიზარდა **სუბიექტურ-ორიენტირებულ** და **ეპო-ორიენტირებულ** დაპროგრამებაში. სუბიექტურ-ორიენტირებული დაპროგრამება გულისხმობს სისტემის დაყოფას სუბიექტებად და წესების დაწერას მათი სწორი

კომპოზიციისთვის. სუბიექტურ-ორიენტირებული დაპროგრამება ავსებს ობიექტურ-ორიენტირებულ დაპროგრამებას, ხსნის რა ისეთ პრობლემებს, რომლებიც დიდ სისტემებთან არის დაკავშირებული - ესაა ინტეგრაციისა და გადატანადობის ამოცანები. სუბიექტურ-ორიენტირებულ დაპროგრამებაში სუბიექტი არის კლასების კოლექცია, რომლებიც ქმნიან თავიანთ (სუბიექტურ) იერარქიას. სუბიექტი შეიძლება იყოს თვითონ გამოყენებითი პროგრამა ან მისი ნაწილი, რომელიც სხვა სუბიექტთან ერთად ქმნის მთლიანად გამოყენებით პროგრამას.

პროგრამის ხარისხის შეფასების კრიტერიუმები

არსებობს ძალზე განსხვავებული კრიტერიუმები პროგრამების ხარისხის შესაფასებლად. მათი როლი დამოკიდებულია ამოცანათა კლასზე და პროგრამის გამოყენების პირობებზე. სხვადასხვა ლიტერატურაში ჩამოთვლილია შემდეგი კრიტერიუმები:

- შედეგიანობა;
- საიმედოობა;
- სიმყარე;
- ავტომატიზირებულობა;
- რესურსების გამოყენების ეფექტურობა (დრო, მეხსიერება, მოწყობილობა, ინფორმაცია, ადამიანები);
- დამუშავებებისა და გამოყენების მოსახერხებულობა;
- პროგრამის ტექსტის თვალსაჩინოება;
- პროგრამის მუშაობაზე თვალყურის დევნება;
- მომხდარის დიაგნოსტიკა.

ეს კრიტერიუმები ხშირად განიცდიან ცვლილებებს პროგრამის გამოყენების არის განვითარების, მომხმარებლის კვალიფიკაციის ზრდის, მოწყობილობების, საინფორმაციიული ტექნოლოგიებისა და დაპროგრამების ტექნიკის შესაბამისად. ეს კი საინფორმაციო სისტემების დაპროგრამების სტილს დამატებით მოთხოვნებს უყენებს:

- მოქნილობა;
- მოდიფიკაციის საშუალება;
- გაუმჯობესება.

დაპროგრამება როგორც მეცნიერება, ხელოვნება და ტექნოლოგია იკვლევს და შემოქმედებითად ანვითარებს პროგრამების შექმნისა და გამოყენების პროცესებს.

დაპროგრამების ენების კატეგორიები

არსებობს სირთულე დაპროგრამების ენების კლასიფიკაციაში და იმაში, თუ მოცემული ენა რომელ პარადიგმას ეკუთვნის. მონაცემთა დამუშავების, მონაცემთა შენახვისა და მონაცემთა დამუშავების მართვის მიხედვით გამოიყოფა პარადიგმის სამი კატეგორია:

- დაბალი დონის დაპროგრამება
- მაღალ დონეზე დაპროგრამება
- პროგრამის მომზადება ზემაღალი დაპროგრამების ენისთვის.

ზემაღალ ენებს მიეკუთვნება: `АПЛ`, `СЕТЛ`, `Python`, `Плэнер` და სხვა.

`Generic programming`–**უნივერსალური დაპროგრამება** ეს არის დაპროგრამების სტილი, რომლის დროსაც ალგორითმები დაწერილია *to-be-specified-later (განისაზღვროს მოგვიანებით)* საფუძველზე. ტიპები განისაზღვრება როგორც პარამეტრები და ხდება მათი დაზუსტება მოგვიანებით, მაშინ, როცა ეს საჭიროა. ამ

სტილის პიონერია ენა Ada, რომელიც 1983 წელს განისაზღვრა. უნივერსალური დაპროგრამება გამოიყენება ენებში Eiffel, Java, C #, Visual Basic.NET და Haskell, შაბლონები C++ და ასევე, პროექტის პარამეტრიზირებული შაბლონები (1994 წ.)

გამონაკლისი სიტუაციების დამუშავება

გამონაკლისი სიტუაციების დამუშავება ანუ მოკლედ გამონაკლისები დაპროგრამების ენა C++ -ის ერთ-ერთი ბოლო დამატებაა. C++ -ის გაფართოების შესაძლებლობა ზრდის შეცდომების რაოდენობასა და მრავალფეროვნებას, ვინაიდან თითოეული ახალი კლასი უმატებს თავის შესაძლო შეცდომებს. გამონაკლისი სიტუაციების დამუშავება იძლევა საშუალებას ვწეროთ უფრო გასაგები, "უღალატო" პროგრამები.

გამანაკლისი სიტუაციების რამდენიმე გავრცელებული მაგალითია: მეხსიერების ნაკლებობა, მასივის ინდექსის საზღვრებს გარეთ გასვლა, არითმეტიკული გადავსება, ნულზე გაყოფა, ფუნქციის დაუშვებელი პარამეტრები და სხვა.

C++ -ის გამონაკლისი სიტუაციების დამუშავების ახალ შესაძლებლობებს შეცდომების დამუშავება გააქვს პროგრამის შესრულების ძირითადი ხაზის გარეთ. ეს საშუალებას იძლევა გამარტივდეს პროგრამის წაკითხვა და ადვილად მოხდეს მასში ცვლილებები.

ამ მიდგომის შემდეგი უპირატესობა არის ის, რომ შესაძლებელი ხდება გამოჭერილ იქნას ყველა სახის გამონაკლისი სიტუაციები, ან განსაზღვრული ტიპის გამონაკლისი სიტუაციები ანდა ურთიერთდაკავშირებული გამონაკლისი სიტუაციები. ეს პროგრამას ხდის შეცდომებისადმი უფრო მედგარს და ამცირებს იმის ალბათობას, რომ რომელიღაცა შეცდომა პროგრამის მიერ არ იქნება გამოჭერილი. გამონაკლისი სიტუაცია გენერირდება C++ -ში ოპერატორი throw -ის საშუალებით, ხოლო მუშავდება კონსტრუქციით try…catch.

სტანდარტული ბიბლიოთეკა

არსებობს ოპერაციები, რომელთა რეალიზაციაც აუცილებელია თითქმის ყველა პროექტში. მათ მიეკუთვნება: მეხსიერების გამოყოფა, სიების ორგანიზება, დახარისხება და ცალკეული ელემენტების ძებნა. ასეთი პრობლემების გადასაწყვეტად C++-ის სტანდარტში გათვალისწინებულია სტანდარტული ბიბლიოთეკის რეალიზაცია, რომელიც საშუალებას იძლევა ამოიხსნას გავრცელებული ამოცანები მსგავსი საშუალებებით.

სტანდარტული ბიბლიოთეკის ყველაზე ძლიერი ნაწილია STL (Standart Template Library), რომელიც წარმოადგენს კლასების შაბლონების ბიბლიოთეკას, რომელიც საშუალებას იძლევა დავამუშავოთ მონაცემთა ტიპიური მასივები. აუცილებელია ბიბლიოთეკა STL რეალიზებული იყოს ნებისმიერ კომპილატორში.

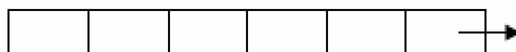
რა ტიპის კონტეინერების მხარდაჭერაა განსაზღვრული STL -ში?

STL კლასის კონტეინერები მართავენ ელემენტების კოლექციას. არსებობს **მიმდევრობითი** და **ასოციური** კონტეინერები. მიმდევრობით კონტეინერებში ელემენტის პოზიცია დამოკიდებულია ჩასმის რიგზე ანუ ელემენტები იმ რიგით არიან, როგორც მოთავსდნენ კონტეინერში. ასოცირებულ კონტეინერებში კი ელემენტები ინახება სორტირებული (დალაგებული) სახით. ელემენტის მდგომარეობა

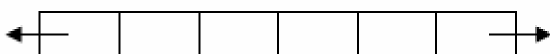
განისაზღვრება არა ჩასმის რიგით, არამედ სორტირების კრიტერიუმით (რომელიც შეიძლება დანიშნულ იქნას).

მიმდევრობით კონტეინერებს მიეკუთვნება:

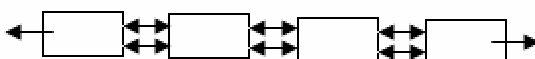
ვექტორი (vector) - კონტეინერის ეს ტიპი ძალიან ჰგავს დინამიურ მასივს. მასში ნებადართულია ნებისმიერ ელემენტს მივმართოთ ინდექსის საშუალებით (კოლექციის ყოველ ელემენტს აქვს თავისი პოზიცია). სწრაფად სრულდება ელემენტის ჩასმისა და ამოღების ოპერაცია კოლექციის ბოლოდან. სქემატურად ეს კონტეინერი ასე შეიძლება გამოვსახოთ:



დეკა (deque) - ორმხრივი რიგი (double-ended queue) ასევე წარმოადგენს დინამიურ მასივს, სადაც შესაძლებელია კოლექციის ნებისმიერ ელემენტს მივმართოთ მისი ინდექსით. კოლექციის ეს ტიპი შეიძლება ასე წარმოდგეს:

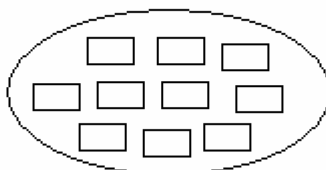


სია (list) - ორმხრივად დაკავშირებული სია, რომელშიც ელემენტებს არ აქვთ მკვეთრად განსაზღვრული პოზიცია, არამედ აქვთ მიმთითებლები მის წინა და მომდევნო ელემენტებზე, რომლებიც ქმნიან ჯაჭვს. ნებისმიერი ადგილას ელემენტის ჩასმა არის მარტივი (ვექტორისა და დეკისკან განსხვავებით, რომლებშიც ასეთი ოპერაციები ძალზე ძვირია, რადგანც ხდება ძველი კოლექციის წაშლა და ახლის შექმნა). ახალი მნიშვნელობის ჩასმისას საჭიროა მხოლოდ გასწორდეს მეზობელი ელემენტების მიმთითებლები. თუმცა ვექტორისა და დეკასგან განსხვავებით კოლექციის ელემენტთან მიმართვა ინდექსით არ არის გათვალისწინებული. მაგალითად, მეათე ელემენტთან მიმართვისთვის საჭიროა პირველი ელემენტიდან ათჯერ გადახვიდეთ მიმთითებლების ჯაჭვზე. სქემატურად ეს კონტეინერი ასე შეიძლება გამოისახოს:



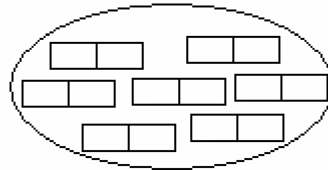
ასოციატიურ კონტეინერებს მიეკუთვნება:

სიმრავლე (set) - კოლექცია, რომელიც ინახავს უნიკალურს (არაგანმეორებად) ელემენტებს დალაგებული სახით (სორტირების კრიტერიუმი შეიძლება შეიცვალოს). ბიბლიოთეკა STL შეიცავს "მულტისიმრავლეს" (multiset), რომელშიც შეიძლება განმეორებადი ელემენტების შენახვაც. კონტეინერის ეს ტიპი, მსგავსად სიისა, არ იძლევა ინდექსით მიმართვის საშუალებას (ვინაიდან ელემენტის ადგილი განისაზღვრება სორტირების კრიტერიუმით). კონტეინერის ეს ტიპი მოსახერხებელია ძებნისთვის, რადგანაც ასეთი კონტეინერი ორობითი ხის სახით არის რეალიზებული. კონტეინერის ეს სახე სქემატურად ასე წარმოდგება:



ასახვა (map) - კოლექცია უნიკალური წყვილების შესანახად, რომლებიც შეესაბამება "გასაღები-მნიშვნელობას". როგორც სიმრავლეში, ასახვაშიც ელემენტები დალაგებული სახითაა. აქაც შეიძლება იყოს "მულტიასახვა" (multimap)

განმეორებადი ელემენტების კოლექციის შესანახად. სქემატურად ასახვა ასე გამოიხატება:



II. იმპერატიული ანუ პროცედურული ენებისა და ფუნქციონალური ენების ურთიერთშედარება

ტრადიციულ დაპროგრამების ენებზე დაწერილი პროგრამები მუშაობის პროცესში ცვლიან ცვლადების მნიშვნელობების ერთობლიობას, რომლებსაც ***მდგომარეობა*** ეწოდება. ჩავთვალოთ, რომ ვუგულვებელყოფთ შეტანა-გამოტანის ოპერაციებსა და იმას, რომ შესაძლოა პროგრამა მუდმივად მუშაობდეს (ასეთია, მაგალითად, საწარმოს მართვის სისტემა), მაშინ შეგვიძლია პროგრამის მუშაობის პროცესი შემდეგნაირად განვაზოგადოთ:

პროგრამის საწყისი მონაცემები ქმნიან თავდაპირველ მდგომარეობას. დავუშვათ, მას აქვს მნიშვნელობა σ. პროგრამის შესრულების შემდეგ მდგომარეობას ექნება ახალი მნიშვნელობა σ', რომელიც შედეგით წარმოიდგინება. თითოეული ოპერატორის შესრულება ნიშნავს, რომ იცვლება მდგომარეობა და რომ მდგომარეობა თანმიმდევრულად გადადის ერთი მნიშვნელობიდან მეორეზე (განსაზღვრულ, სასრულ რაოდენობაჯერ):

$$\sigma = \sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow . \ . \ . \rightarrow \sigma_n = \sigma'$$

მაგალითად, სორტირების პროგრამაში თავდაპირველი მდგომარეობა შეიცავს მნიშვნელობების მასივს, ხოლო მას შემდეგ, როცა პროგრამა დაამთავრებს მუშაობას, მდგომარეობა იცვლება იმგვარად, რომ ეს მნიშვნელობები ხდება დალაგებული. ამ დროს შუალედური მდგომარეობები წარმოადგენენ ცალკეულ სვლებს ამ მიზნის მისაღწევად.

მდგომარეობები, საზოგადოდ იცვლებიან მინიჭების ოპერატორის საშუალებით, რომელიც საზოგადოდ ჩაიწერება ასე: `v = E` ან `v := E`, სადაც `v` არის ცვლადი, ხოლო `E`–გამოსახულება. პროგრამის ტექსტში მინიჭების ოპერატორები თანმიმდევრულად განთავსდებიან ერთმანეთის მიყოლებით. ხშირად მათ გამყოფად გამოიყენება წერტილ-მძიმე (;). შედგენილი ოპერატორების მეშვეობით, ისეთების, როგორიცაა `if` და `while`, შესაძლოა ოპერატორები შესრულდეს რაიმე პირობის მიხედვით ან ციკლურად (განმეორებით) მიმდინარე მდგომარეობის შესაბამისად. ამრიგად, შეგვიძლია პროგრამა წარმოვადგინოთ როგორც მდგომარეობების შეცვლის ინსტრუქციების ნაკრები. ამიტომაც დაპროგრამების ასეთ სტილს უწოდებენ ***იმპერატიულს ანუ პროცედურულს.*** შესაბამისად, ტრადიციულ დაპროგრამების ენებს, რომლებიც მხარს უჭერენ ამ სტილს, უწოდებენ ***იმპერატიულს ანუ პროცედურულ დაპროგრამების ენას.***

ფუნქციონალური დაპროგრამება

ფუნქციონალური დაპროგრამება მკვეთრად განსხვავდება ზემოთმოყვანილი მოდელისაგან. არსებითად, ფუნქციონალური პროგრამა წარმოადგენს გამოსახულებას, ხოლო პროგრამის შესრულება კი, ამ გამოსახულების გამოთვლის პროცესია.

ფუნქციონალურ დაპროგრამებას ხშირად უწოდებენ ***აპლიკატიურ დაპროგრამებას,*** ვინაიდან ამ მექანიზმის საფუძველია აპლიკაცია (გამოყენება) ფუნქციისა მის არგუმენტებზე.

საზოგადოდ, **იმპერატიული პროგრამა დეტერმინირებულია,** ანუ შედეგი სრულად განისაზღვრება შესავლით. ჩვენ შეიძლება ვთქვათ, რომ საბოლოო მდგომარეობა არის საწყისი მდგომარეობის ფუნქცია, მაგალითად, $\sigma' = f(\sigma)$. მსგავსად წერდა ნაური, რომ შესაძლოა ჩაიწეროს ნებისმიერი პროგრამა შემდეგი სახით: `Output = Program (Input)`. ფუნქციონალურ დაპროგრამებაში ამ შეხედულებას განსაკუთრებული მნიშვნელობა აქვს: პროგრამა არის გამოსახულება, რომელიც შეესაბამება მათემატიკურ ფუნქციას `f`. ფუნქციონალური ენები ასეთ გამოსახულებებს მხარს უჭერენ იმის გამო, რომ ამ ენებში ძლიერი ფუნქციონალური კონსტრუქციების გამოყენებაა შესაძლებელი.

ფუნქციონალური დაპროგრამება შეიძლება დაუპირისპირდეს იმპერატიულს როგორც კარგი მხრით, ისე ცუდი მხრივ. ფუნქციონალური პროგრამების უარყოფით მხარედ შეიძლება დავასახელოთ ის, რომ არ იყენებს ცვლადებს-ანუ **მას არ აქვს მდგომარეობა**. შესაბამისად, ფუნქციონალურ ენაში არ არსებობს მინიჭება, რადგანაც ვერავის მიენიჭება. ასევე, ოპერატორების თანმიმდევრული შესრულების იდეა ასევე უაზროა, ვინაიდან ერთ ოპერატორს არ შეუძლია იმოქმედოს მეორეზე, რადგანაც არ არის მდგომარეობა, რომელსაც ერთმანეთს გადასცემენ. ფუნქციონალური დაპროგრამების ღირსებად შეიძლება ჩაითვალოს ის, რომ ფუნქციას იყენებს განსაკუთრებულად. ფუნქცია შეიძლება ისევე იქნას განხილული, როგორც სხვა, უფრო მარტივი ობიექტები, მაგალითად, რიცხვები: ფუნქცია შეიძლება გადაეცეს სხვა ფუნქციებს როგორც არგუმენტები, ფუნქცია დაბრუნდეს როგორც სხვა ფუნქციის შედეგი, ასევე შესაძლოა ფუნქცია გამოყენებული იყოს გამოსახულებებში. ოპერატორების თანმიმდევრული შესრულებისა და განმეორებითი ოპერატორების (ციკლების) ნაცვლად, ფუნქციონალური დაპროგრამების ენები გვთავაზობენ რეკურსიულ ფუნქციებს, ანუ ფუნქციებს, რომლებიც თავისივე ტერმინებში განისაზღვრებიან. ტრადიციული ენების უმრავლესობა რეკურსიის საკმაოდ მწირ საშუალებებს იძლევიან. ენა `C`-ის აქვს ფუნქციებთან მუშაობის გარკვეული, შეზღუდული საშუალებები მიმთითებლების გამოყენებით, თუმცა არ იძლევა ფუნქციების დინამიურად შექმნის საშუალებას. ენა `FORTRAN` საერთოდ არ უჭერს მხარს რეკურსიას.

ვაჩვენოთ განსხვავება იმპერატიულსა და ფუნქციონალურ დაპროგრამებას შორის ფაქტორიალის გამოთვლის მაგალითზე. იგი შეიძლება ჩაიწეროს იმპერატიულ ენა C -ზე შემდეგნაირად:

```
int  fact (int n)
     { int x = 1;
       while ( n > 0 )
                  { x = x * n;
                    n = n - 1;
                   }
       return x;
      }
```

ფუნქციონალურ ენა `LISP`–ზე ეს ფუნქცია შეიძლება ასე ჩაიწეროს:

```
( define fact ( lamnda ( n ) ( cond (( eq n 0) 1)
                                    ( t ( fact ( n -1 ))))) )
```

უნდა აღინიშნოს, რომ ასეთი განსაზღვრება საკმაოდ ადვილად რეალიზდება ენა `C`-ზეც, მაგრამ უფრო რთულ შემთხვევებში ფუნქციონალური ენები შეუცვლელები არიან.

ფუნქციონალური ენების ღირსებები

ერთი შეხედვით, დაპროგრამების ენა, რომელშიც არ არის ცვლადები და არ არის განსაზღვრული ინსტრუქციების თანმიმდევრულად შესრულება, ძალზე არაპრაქტიკულ ენად ჩანს, თუმცა ფუნქციონალური სტილის მეშვეობით შესაძლებელია სხვადასხვაგვარი ამოცანების ამოხსნა.

დაპროგრამების იმპერატიული სტილი, რა თქმა უნდა, არ არის დაურღვეველი დოგმა. იმპერატიული ენების მრავალი თვისება განვითარდა მანქანური კოდიდან ასემბლერამდე, შემდეგ მაკროასამბლერამდე, ენა FORTRAN -მდე და ა.შ. არ არსებობს საფუძველი იმის მტკიცებისთვის, რომ ეს ენები ადამიანისა და მანქანის ურთიერთქმედებისათვის საუკეთესო საშუალებებს წარმოადგენენ. კომპიუტერების არქიტექტურის განვითარებაში ბოლო სიტყვა ჯერ არ თქმულა. რა თქმა უნდა, კომპიუტერები უნდა ემსახურებოდნენ ჩვენს საჭიროებებს და არა პირიქით. ალბათ სწორი იქნებოდა არა აპარატურიდან დაწყება, არამედ პირიქით, **აგვეღო საფუძვლად** დაპროგრამების ენა, როგორც ალგორითმების ასახვის საშუალებები და **გვემოძრავა ქვევით, აპარატურისკენ.** ასეთი სტრატეგია შეიძლება აღმოვაჩინოთ ტრადიციულ დაპროგრამების ენებში. ენა FORTRAN -იც კი იძლევა საშუალებას ჩვეულებრივად ჩაიწეროს არითმეტიკული გამოსახულება ისე, რომ პროგრამისტმა არ იზრუნოს გამოთვლების თანმიმდევრობაზე და შუალედური შედეგების შესანახად მეხსიერებიის გამოყოფაზე.

ამ მოსაზრებებისგან შეიძლება დავასკვნათ, რომ იდეა ისეთი დაპროგრამების ენის დამუშავებაზე, რომელიც ძლიერ განსხვავდება ტრადიციული, იმპერატიული ენებისგან, ნამდვილად კანონიერია. თუმცა, იმის საჩვენებლად, რომ ცვლილება არ ხდება მხოლოდ ცვლილებისთვის, საჭიროა ნაჩვენები იქნას ის, თუ რატომ მიაჩნიათ ფუნქციონალური ენები იმპერატიულ ენებთან შედარებით უპირატესად.

შესაძლოა, მთავარი მიზეზი არის ის, რომ პროგრამა ფუნქციონალურ ენაზე უფრო ზუსტად შეესაბამება მათემატიკურ ობიექტებს და მისი თვისებები ადვილად დამტკიცდება. იმისთვის, რომ ვაჩვენოთ, თუ რას აღნიშნავს პროგრამა, ჩვენ შეიძლება დავუკავშიროთ პროგრამას ან ოპერატორს აბსტრაქტული მათემატიკური აზრი. ეს არის სწორედ **დენოტაციური სემანტიკის** მიზანი. იმპერატიულ ენებში ამის გაკეთება უფრო მეტად გვერდითი მოვლენით თუ მოხდება იმის გამო, რომ მდგომარეობაზე დამოკიდებულება არაცხადია. მარტივი იმპერატიული ენებისთვის ოპერატორი შეიძლება განვსაზღვროთ როგორც ფუნქცია $\Sigma \rightarrow \Sigma$, ($\Sigma$ დასაშვებ მდგომარეობათა სიმრავლეა), ანუ ოპერატორი ღებულობს ერთ მდგომარეობას და წარმოშობს მეორეს. თუმცა ყველა ოპერატორი როდი ამთავრებს თავის მუშაობას (მაგალითად, while TRUE do x:=x), ასე, რომ ეს ფუნქცია, საზოგადოდ, ნაწილობრივ განსაზღვრულია. ზოგჯერ უფრო მისაღები სემანტიკის ფორმალიზების ალტერნატიული საშუალებებია, მაგალითად, დეიქსტრას პრედიკატების გარდამქმნელები. თუმცა თუ ენას დაემატება შესაძლებლობები, რომლებსაც შეუძლიათ ოპერატორების თანმიმდევრობა შეცვალონ, მაგალითად goto, ან C ენის კონსტრუქციები break და continue, მაშინ ეს ამოხსნები აღარ იმუშავებს, ვინაიდან ერთმა ოპერატორმა შეიძლება მოახდინოს ტექსტში მისი მომდევნო რამდენიმე ოპერატორის გამოტოვება. ამის ნაცვლად ხშირად გამოიყენება უფრო რთული სემენტიკები, რომლებიც დაფუძნებულია **გაგრძელებებზე** (continuations).

ზემოთ თქმულის საწინააღმდეგოდ, ფუნქციონალური პროგრამები, ხენსონის (Martin C. Henson) განმარტებით "თვითონ ატარებენ თავის სემანტიკას". ეს

შეიძლება გაჩვენოთ ფუნქციონალური დაპროგრამების ენა ML-ის მაგალითზე. ძირითადი ტიპები შეიძლება განვიხილოთ როგორც მათემატიკური ობიექტები. თუ გამოვიყენებთ სტანდარტულ ჩანაწერს [[X]] "X-ის სემანტიკის" აღსანიშნავად, მაშინ შეიძლება ვთქვათ, რომ [[int]]=Z. მაგალითად, ენა ML-ის ფუნქცია fact, განისაზღვრება გამოსახულებით:

```
let rec fact n =
      if n = 0 then 1
   else n * fact ( n - 1 )
```

fact ფუნქციას აქვს ერთი მთელი ტიპის არგუმენტი და აბრუნებს ერთ მთელი ტიპის მნიშვნელობას. ასე, რომ იგი დაკავშირებულია ნაწილობრივ განსაზღვრულ ფუნქციასთან Z → Z:

$$[[fact]](n) = \begin{cases} n!, n > 0 \\ \perp \text{ წინააღმდეგ შემთხვევაში} \end{cases}$$

აქ ნიშანი ⊥ აღნიშნავს განუსაზღვრელობას, რადგანაც უარყოფითი არგუმენტებისთვის პროგრამა არ დამთავრდება. თუმცა მარტივი იტერაციის ეს საშუალება ვერ მუშაობს არაფუნქციონალური ენებისთვის, ვინაიდან ე.წ. “ფუნქციები” შეიძლება არ იყვნენე ფუნქციები მათემატიკური აზრით. მაგალითად, ენა C-ის სტანდარტულ ბიბლიოთეკაში არის ფუნქცია rand(), რომელიც თანმიმდევრული გამოძახებებისას აბრუნებს სხვადასხვა ფსევდო შემთხვევით მნიშვნელობებს. წინა შემთხვევითი მნიშვნელობის შესანახად ლოკალური სტატიკური ცვლადის გამოყენებაა შესაძლებელი, მაგალითად, ასე:

```
int  rand ( void )
                  { static int n=0;
                    return n=2147001325 * n + 715136305;}
```

ამრიგად, ჩვენ შეიძლება განვიხილოთ ცვლადებსა და მინიჭების ოპერატორზე უარი, როგორც goto გადასვლის ოპერატორის უარყოფის შემდეგი ნაბიჯი, ვინაიდან თითოეული ნაბიჯი სემანტიკის უფრო ნათლად წარმოდგენის საშუალებას იძლევა. უფრო მარტივი სემანტიკისას კი პროგრამის თვისებების დამტკიცება უფრო ნათლად არის შესაძლებელი. ეს, თავის მხრივ, უფრო დიდ შესაძლებლობას იძლევა დამტკიცდეს პროგრამის კორექტულობა და მოიძებნოს გარდაქმნები პროგრამის ოპტიმიზაციის მიზნით.

ფუნქციონალურ ენებს აქვთ კიდევ ერთი პოტენციური უპირატესობა. ვინაიდან გამოსახულების გამოთვლას არ შეიძლება ჰქონდეს გვერდითი ეფექტები ნებისმიერი მდგომარეობისთვის, ამიტომ ცალკეული ქვეგამოსახულებები შეიძლება გამოთვლილი იქნას ნებისმიერი რიგით ისე, რომ ვერ შეძლებენ ერთი მეორეზე გავლენის მოხდენას. ეს კი ნიშნავს, რომ ფუნქციონალური დაპროგრამება კარგად ექვემდებარება გაპარალელებას, ანუ კომპიუტერს შეეძლება ცალკეული ქვეგამოსახულება გამოითვალოს ცალკეულ პროცესორებზე. ავღნიშნოთ, რომ იმპერატიული პროგრამებში ხშირად მკაცრად არის განსაზღვრული გამოთვლების თანმიმდევრობა.

ფუნქციონალური დაპროგრამების ენები, მაგალითად, LISP, ML არ წარმოადგენენ წმინდა ფუნქციონალურ ენებს, რადგანაც საჭიროების შემთხვევაში შეგიძლიათ გამოიყენოთ ცვლადები და მინიჭების ოპერაცია. მაშინაც კი, როცა თქვენ სარგებლობთ მინიჭებით და ამის გამო ზემოთ ჩამოთვლილი ღირსებებიდან კარგავთ ზოგიერთს, ძირითადი რჩება უცვლელი: ფუნქციებთან მუშაობის დიდი მოქნილობა. პროგრამები შეიძლება ძალზე მოკლედ და ელეგანტურად გამოისახოს მაღალი რიგის

ფუნქციებით, რომელთაც ფუნქციონალები ეწოდებათ. ფუნქციონალები - ეს ფუნქციებია, რომელთა არგუმენტები ფუნქციებია. ასეთი კოდი შეიძლება იყოს უფრო ზოგადი, რადგანაც კოდი შეიძლება შეიძლება პარამეტრიზებული იყოს სხვა ფუნქციებით. მაგალითად, პროგრამა, რომელიც რიცხვების სიას შეკრებს და პროგრამა, რომელიც ამრავლებს რიცხვების სიას, შეიძლება განვიხილოთ როგორც ერთიდაიგივე პროგრამის ეგზემპლარები, რომელიც პარამეტრიზდება არითმეტიკული ოპერაციით და ერთი ელემენტით. პირველ შემთხვევაში ეს იქნება + და 0, მეორე შემთხვევაში - * და 1. D

ამავე დროს ფუნქციონალური პროგრამები არ არიან განთავისუფლე-ბულნი საკუთარი პრობლემებისგან. რადგანაც ფუნქციონალური პროგრამები ნაკლებად ორიენტირებულია აპარატურაზე, გამოყენებული რესურსების, როგორიცაა დრო და მეხსიერება, ზუსტი გამოთვლა შეუძლებელია.

ზოგადად, ფუნქციონალური ჯგუფის ენების გამოყენების საწინააღმდეგო არგუმენტად ასახელებენ მათი ეფექტური რეალიზაციის განხორციელების შეუძლებლობას. თუმცა, ეს შეიძლება სწორად ჩაითვალოს მხოლოდ პირველი თაობის ფუნქციონალური ენებისთვის, რომლებიც დაფუძნებულია ინტერპრეტაციასა და ტიპების დინამიკურ კონტროლზე. თანამედროვე ფუნქციონალური ენებისთვის ეს არგუმენტი მიუღებელია.

თანამედროვე ფუნქციონალური დაპროგრამების ენების ძირითადი მახასიათებლებია [7]:

- პარამეტრული პოლიმორფიზმი, რომელიც არაა დამოკიდებული ობიექტთა იერარქიულ მემკვიდრეობასთან;
- მონაცემთა აბსტრაქტულ ტიპებზე მხარდაჭერა სრული მოცულობით;
- სტატიკური ტიპიზაცია;
- დაყოფადი ტრანსლაციის მხარდაჭერა;
- დინამიკური მეხსიერების ავტომატური მართვა (ე.წ. დანაგვიანებული მეხსიერების გასუფთავება);
- მხარდაჭერა ჩადგმული და მაღალი დონის ფუნქციებისა (ე.წ. ფუნქციონალები), რომლებიც ფუნქციის პარამეტრად იღებენ ფუნქციას და მნიშვნელობად აბრუნებენ ახალ ფუნქციას.

უკანასკნელი თაობის ფუნქციონალური დაპროგრამების ენა, რომელსაც ზემოთ ჩამოთვლილი თვისებები ახასიათებს, არის ენა Objective Caml [8], რომელიც შექმნილია INRIA-ში (ინფორმატიზაციისა და ავტომატიზაციის ინსტიტუტი, საფრანგეთი).

III. ინფორმაციის დამუშავების ასპექტები ფუნქციონალურ პარადიგმაში

რეალური პროექტების რეალიზაციისას ტექნიკური ხასიათის მრავალი პრობლემა ამ დარგის ექსპერტების შეფასებით არის ის, რომ დაპროგრამების ენებს, რომლებიც გამოიყენებიან კომერციული პროდუქციის შესაქმნელად, არ გააჩნიათ მთელი რიგი საშუალებები. მათ შორის ყველაზე ხშირად სახელდება აბსტრაქციის არასაკმარისი დონე, რომელიც საჭიროა უნივერსალური მონაცემებისა და მათი დამუშავების პროცედურების აღსაწერად.

ასევე ცნობილია, რომ პროგრამული უზრუნველყოფის შექმნის ცხოვრებისეულ ციკლში კოდის დაწერასთან ერთად ძალზე დიდი წილი მოდის მის თანხლებაზე, ამიტომ დიდი პროექტების დაწყებამდე საჭიროა კარგად იქნას გააზრებული, მოხერხდება თუ არა მოდულებს შორის ფუნქციონალური დაყოფის შენარჩუნება, ანდა, საკმარისი იქნება თუ არა ფუნქციის თავდაპირველად დაგეგმილი პარამეტრიზაცია და

ხომ არ იქნება პროექტში მონაცემთა ერთი და იგივე სტრუქტურის სხვადასხვა რეალიზაციები.

ამ თვალსაზრისით, ძირითადი მოთხოვნა იმ დაპროგრამების ენაზე, რომელიც გვინდა ავირჩიოთ რეალიზაციისთვის, არის მონაცემთა აბსტრაქტული ტიპებისა და პოლიმორფიზმის სრული მხარდაჭერა. ამასთან პრაქტიკაში ფართოდ გამოყენებული ენები C++ და java გვაძლევენ ამ საშუალებებს, მხოლოდ არასაკმარისად სრულად - ობიექტების პრიზმიდან. ამავე დროს ფუნქციონალურმა ენებმა თავის განვითარებაში მიაღწიეს შედეგებს, რომლებიც შეიძლება ჩაითვალოს ტრადიციული კომერციული ენების ალტერნატივად.

ცოტა რამ ისტორიიდან. თეორიული "წმინდა" LISP განსაზღვრავდა ფუნქციათა მცირე სიმრავლეს, რომლებიც შესაძლებელს ხდიდა შეგვექმნა და შეგვეცვალა ნებისმიერ სია, რომელიც შედგებოდა ელემენტებისგან ან ქვესიებისგან, დაგვემატებინა ახალი ელემენტი სიის თავში, ბოლოში, ან თავი მოგვეცილებინა. "წმინდა" LISP -ში იყო პირობითი ფუნქციაც, რაც პრინციპში უკვე ქმნიდა ენას სრულყოფილ დაპროგრამების ენად. თუმცაღა ლისპმა დიდი გავრცელება მოიპოვა მდიდარი ბიბლიოთეკური ფუნქციების გამო, რომელიც შეიცავდა უკვე არაელემენტარულ ფუნქციებს. ამიტომაც LISP-ის დაარსებიდან (იგი შექმნა ჯომ მაკარტიმ 1962 წელს) ამ ენის ვერსიების, ინტერპრეტატორების რაოდენობა წარმოუდგებლად დიდია. მე-20-ე საუკუნის 70-80 -იანი წლების დაპროგრამების ენების ცნობილი სპეციალისტი ჯონ სამიტი წერდა, რომ ყველა დაპროგრამების ენა შესაძლოა უხეშად გაიყოს ორ ნაწილად: ერთშია LISP, მეორეში-ყველა დანარჩენი დაპროგრამების ენა.

LISP-ში, მაგალითად, პროცედურები შეიძლება იყოს მონაცემები-ანუ არგუმენტებად შეიძლება იყოს გამოყენებული. ასევე პროცედურა შეიძლება დაბრუნდეს ფუნქციის მნიშვნელობად, რომელიც ისევ არგუმენტად შეიძლება იყოს გამოყენებული.

LISP-ზე რეალიზებულია პროგრამების უზარმაზარი მოცულობა, შექმნილი ხელოვნურ ინტელექტსა და ექსპერტულ სისტემებში, ეგმ ოჯახი VAX, საინჟინრო პროექტირების სისტემა აუტოცად. LISP მანქანების აგებამ შესაძლებელი გახადა LISP-ის გამოყენება გამოთვლითი ამოცანებისთვის.

თავდაპირველად LISP ჩაფიქრებული იყო როგორც თეორიული საშუალება რეკურსიების ასაგებად, დღეს კი ის გადაიქცა მძლავრ საშუალებად, რომელიც პროგრამისტს საშუალებას აძლევს ააგოს საკმაოდ სერიოზული, სხვადასხვა ტიპის სისტემების პროტოტიპები. სწორედ ამ თემას შევეხებით.

მონაცემთა წარმოდგენა სიის სტრუქტურებით

სამეცნიერო-ტექნიკური პროგრესის განვითარებამ ინფორმაციული აფეთქება გამოიწვია, რაც იმით გამოიხატება, რომ მთელს მსოფლიოში ახალი ინფორმაციის მოცულობა დროის ერთეულში ექსპონენციალურად იზრდება. ასეთ პირობებში ბუნებრივი ენის ავტომატიზებული დამუშავება განსაკუთრებულად აქტუალური ხდება. თუმცა, უნდა აღინიშნოს, რომ გამოთვლით მანქანებზე ბუნებრივენოვანი ტექსტის დამუშავების იდეა გაჩნდა პირველი გამოთვლითი ტექნიკის გამოჩენისთანავე.

ტექსტის, წიგნების, სხვადასხვა დოკუმენტის თავმოყრა ჯერ კიდევ არ წარმოადგენს ცოდნას, თუნდაც ისინი ერთ სივრცეში იყოს თავმოყრილი. მხოლოდ სტრუქტურირებული ინფორმაცია შეიძლება წარმოვიდგინოთ როგორც ცოდნა, მაგრამ ჩნდება პრობლემა, როგორ უნდა წარმოიქმნას ცოდნის სტრუქტურები და როგორ უნდა იყოს ისინი გამოყენებული. ცოდნის წარმოდგენის ცნობილი სტრუქტურები (მინსკის

ფრეიმები, ვუდსის სემანტიკური ქსელები ფართოდ გამოიყენება ხელოვნური ინტელექტის სისტემებში, თუმცა არც თუ წარმატებულად. ასევე სუსტი შედეგები აჩვენა ცოდნის წარმოდგენის პროცედურულმა საშუალებებმა (პროლოგმა და სხვა ლოგიკური დაპროგრამების ენებმა). როგორც წესი, შეუძლებელია აიგოს წინასწარგანსაზღვრული სტრუქტურა – ცოდნის სტრუქტურის ჩონჩხი, რადგანაც წარმოუდგენელია ინფორმაციულ ნაკადში წინასწარ განისაზღვროს როგორც ცოდნა, ასევე მისი სტრუქტურა.

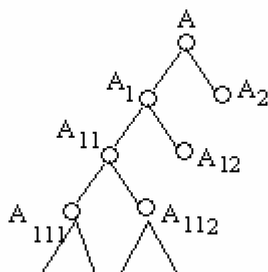
კომპიუტერში ინფორმაციის წარმოდგენა, ჩვენი აზრით, ძალზე მოხერხებულია სიების საშუალებით. სიის სტრუქტურა ბუნებრივია და ახლოა ადამიანის აზროვნებაში ინფორმაციის წარმოდგენასთან. იგი არ არის დამოკიდებული მანქანის მეხსიერების სტრუქტურაზე, მეხსიერებაში ინფორმაციის ჩაწერის, ძებნისა და წაკითხვის საშუალებებზე.

დაპროგრამების ენა, რომელიც სიებთან სამუშაოდ გამოვიყენეთ, არის ფუნქციონალური დაპროგრამების ენა `LISP`.

სია არის ფრჩხილებში ჩასმული ჩვეულებრივი სიტყვები (`LISP`-ში მათ უწოდებენ ატომებს) ან ჩვეულებრივი სიტყვების ერთობლიობა. სია ანალოგიურია წინადადების და ისევე, როგორც ბუნებრივ ენაშია რთული და შედგენილი წინადადებები, ასევე შესაძლებელია სიაში იყოს ჩადგმული სიები.

თავდაპირველად განვიხილოთ ცოდნის წარმოდგენა ხეების (შემდგომში ქსელების) საშუალებით და მათი რეალიზაცია `LISP`-ის სიებით [9].

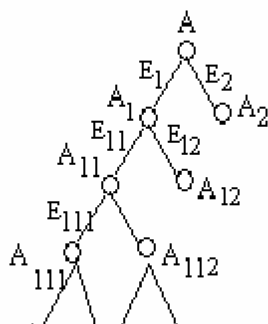
ყველაზე მარტივი ხე, რომლითაც შეიძლება ინფორმაცია იყოს წარმოდგენილი, არის შემდეგი სტრუქტურის:



ასეთი ხის შესაბამისი `LISP`-ის სიის სტრუქტურას იქნება შემდეგი სახე:

$(A\ (A_1(A_{11}(...)A_{12}(...)...A_{1m}(...))$
$(A_2(A_{21}(...)A_{22}(...)...A_{2m}(...))$
...
$(A_k(A_{k1}(...)A_{k2}(...)...A_{km}(...)))$

განვიხილოთ ხე, რომლის წვეროებს შორის წახნაგებზე შეიძლება მოთავსებული იყოს მნიშვნელობა: ამ წვეროებს შორის კავშირის მახასიათებელი:

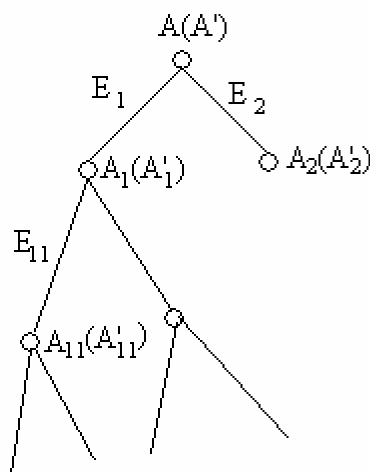


ასეთ ხეს ვუწოდოთ AE ტიპის ხე. თუ კარგად დავაკვირდებით, შევნიშნავთ, რომ წვეროების აღნიშვნებში ინდექსების რაოდენობა მიუთითებს წვეროდან დაშორების დონეს, ხოლო კავშირების აღნიშვნებში ინდექსი გვიჩვენებს, თუ რომელ წვეროს უკავშირდება. იმის მითითება, თუ რომელი წვეროდან გამოდის კავშირი, საჭირო არ არის, ვინაიდან ყოველი წვეროს პირველი ინდექსი მის მშობელ წვეროზე მიუთითებს.

AE ტიპის ხეს შეესაბამება LISP-ის შემდეგი სტრუქტურა:

$$(A(E_1(A_1(E_{11}(A_{11}(...))(E_{12}(A_{12}(...))...(E_{1m}(A_{1m}(...)))))$$
$$(E_2(A_2(E_{21}(A_{21}(...))(E_{22}(A_{22}(...))...(E_{2m}(A_{2m}(...)))))$$
$$...$$
$$(E_k(A_k(E_{k1}(A_{k1}(...))(E_{k2}(A_{k2}(...))...(E_{km}(A_{km}(...))))))$$

მესამე ხე, რომელიც გამოიყენება ინფორმაციის წარმოსადგენად, არის ე.წ. AES ტიპის. ასეთ ხეებში მნიშვნელობები, წახნაგების გარდა, შეიძლება მითითებულ იქნას წვეროებშიც. ასეთი წვერო აღვნიშვნოთ $(S\ A'_{km})$-ით. საჭიროა იმის სიმბოლური მითითება (მაგალითში გამოყენებული გვაქვს სიმბოლო S), რომ წვეროში მოთავსებულია მნიშვნელობა. AES ხეს აქვს შემდეგი სახე:



AEV ხეს LISP-ზე აქვს შემდეგი წარმოდგენა:

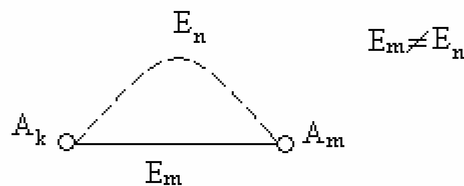
$$(\ A(V\ A')(E_1(A_1(V\ A'_1)(E_{11}(A_{11}...))(E_{12}(A_{12}(V\ A'_{12})...))$$
$$...\ (E'_m(A_{1m}...))\ ))$$
$$(E_2(A_2...))$$
$$...$$
$$(E_k(A_k))\ )$$

ასეთი ტიპის ხეებზე შეიძლება ისეთი ამოცანის გადაჭრა, რომელიც მოითხოვს მსგავსი სიტყვების ან წინადადებების პოვნას. მსგავსი შეიძლება იყოს სიტყვები, რომლებიც ერთმანეთისგან განსხვავდება ერთი ასოთი (იგულისხმება გამოტოვებული ან ზედმეტი ასო).

ხშირად შეუძლებელია ან არაეფექტურია ერთი ქსელის წვეროებით წარმოვადგინოთ ყველა სიტუაცია, ამიტომ უნდა გვქონდეს პროცედურები, რომლებიც საჭიროა განსხვავებების აღმოსაჩენად, ქსელის ბოლომდე გასასვლელად და შედეგში განსხვავებების დასაფიქსირებლად.

განვიხილოთ ზოგიერთი ტიპიური განსხვავება:

1. წახნაგი, რომელიც ორ მოცემულ წვეროს აერთებს, არის განსხვავებული ქსელზე არსებული წახნაგისაგან:



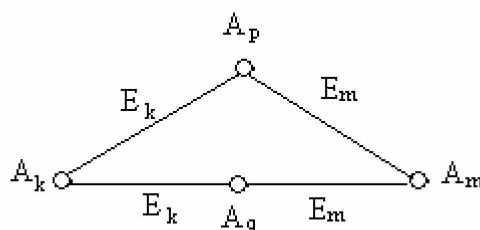
ასეთი შემთხვევა საჭიროა შეცდომების აღმოსაჩენად და სინონიმების დასაფიქსირებლად.

მაგალითად, ასეთი შემთხვევაა შემდეგი ორი სიის ქსელური წარმოდგენისას:

(ბორჯომი (მანქანა (ბაკურიანი)))

(ბორჯომი (ჯიპი (ბაკურიანი)))

2. ემთხვევა ქსელის წვეროები და ფერდები, მაგრამ განსხვავდება მათ შორის წვეროები:

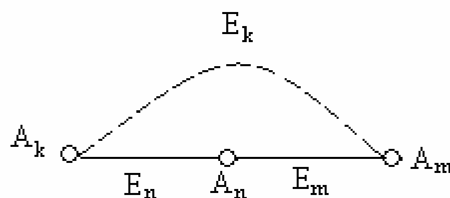


მაგალითად, ასეთი შემთხვევებია შემდეგი სიების ქსელური წარმოდგენისას:

(გიორგი (ივანეს-ძე (გიორგობიანი))) და (გიორგი (პეტრეს-ძე (გიორგობიანი)));

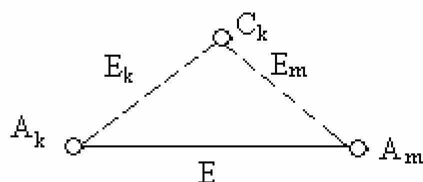
(დავითი (ექიმი (კვერნაძე))) და (დავითი (პავლეს-ძე (კვერნაძე)))

3. გამოტოვებულია ქსელის წვერო:

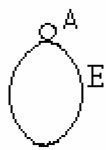


მაგალითად, (გიორგი (ივანეს-ძე (გიორგობიანი))) და (გიორგი (გიორგობიანი)).

4. საჭიროა ზედმეტი წვეროს გავლა:



5. როცა საჭიროა წვეროზე ერთხელ მაინც ზედმეტად გავლა:



ასეთი ქსელით წარმოდგება, მაგალითად, ინფორმაცია პიროვნებაზე, რომელიც ორ გვარს ატარებს.

აღნიშნულ ნახაზებზე გამოყენებულია შემდეგი აღნიშვნები: $A, A_k, A_p, A_n, A_m, C_k$ წვეროებია, E, E_k, E_m, E_n – წახნაგები, ხოლო წყვეტილი ხაზები გვიჩვენებს განსხვავებებს ქსელებს შორის.

ინფორმაციის შინაარსის ან ნებისმიერი წვეროს ქვექსელის მოსაძებნად გამოიყენება LISP-ის ფუნქციათა კომბინაციები და ფუნქციონალები. მათი საშუალებით რეალიზებულია შემდეგი ოპერაციები სიებზე: წვეროს ამოშლა, გადაადგილება და ახალი წვეროს დამატება, წახნაგის შეცვლა, ინფორმაციის შინაარსის შეცვლა, დამატება და ამოშლა.

ზემოთ მოყვანილი AE და AEV ტიპის ქსელების საშუალებით შეიძლება წარმოდგენილი და დამუშავებული იყოს ინფორმაცია სხვადასხვა საგნობრივი არიდან. განვიხილოთ რამდენიმე მაგალითი.

მონაცემთა ბაზა, რომელიც წარმოდგენილია შემდეგი ობიექტებისაგან: *დასახლებული-პუნქტი, ქალაქი, სოფელი, თბილისი, ბათუმი, დიღომი* და *ბაკურიანი* ასე წარმოდგება:

(დასახლებული-პუნქტი (S ადგილმდებარეობა ( ) მოსახლეობა ( ))
*(სოფელი (S ადგილმდებარეობა (* რაიონი) მოსახლეობა (: 5 5000))*
(დიღომი (S მცხეთა 800)))
*(ქალაქი (S ადგილმდებარეობა (* ქვეყანა) მოსახლეობა (: 10000 *))*
(თბილისი (S საქართველო 1000000))
(ბათუმი (S საქართველო 300000))
(დასახლება (S ადგილმდებარეობა (რაიონი) მოსახლეობა (: 1000 10000) ღირშესანიშნაობა ( ))*
(ბაკურიანი (S ბორჯომი კოხტაგორა))

ქართული ენის ლექსიკონის ქსელური წარმოდგენა

AE ტიპის ქსელების გამოყენება შეიძლება მოცემული ლექსიკონის სიტყვების ერთობლიობის წარმოსადგენად. ამ ქსელის თითოეული წვერო იქნება სიტყვის შემდეგი ასო. თუ მოცემული სიტყვა დამთავრდა ამ ასოთი, მაშინ ამ წვეროზე შეიძლება მივაბათ ინფორმაცია (მორფოლოგიური, სემანტიკური და ა.შ.) ამ სიტყვის შესახებ.

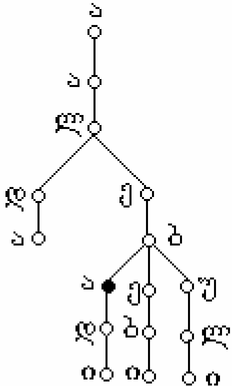
შემდეგი მაგალითი (ფრაგმენტი) ასახავს ქართული ენის ლექსიკონიდან სიტყვების ჯგუფის წარმოდგენას სიის სტრუქტურის მიხედვით, რომლებიც იწყება აა და აბ ასოებით (მხოლოდ ბოლო ორი სიტყვა). ქართული ანბანის თითოეული ასოსათვის შესაძლებელია შედგეს ასეთი ტიპის სიის სტრუქტურები, რომლებიც ქსელებს შეესაბამება, ე.ი. 33 ქსელი იქნება.

```
(ა   (ა   (გ   (ო   ))
          (დ   (ა   (მ   (ი   (ა   (ნ   (ე   (ბ   (ს   )))   )))   )))
               (გ   (ი   (ლ   (ე   (ბ   (ს   )))   )))   )
          (ვ   (ს   (ო   )))
          (ლ   (დ   (ა   ))
               (ე   (ბ   (ა*  (დ   ი))
                         (ე   (ბ   (ი)  ))
```

```
                    (უ   (ლ   (ი)  ))   )))
     (მ   (ა   (ღ   (ლ   (ა)  )))
          (ო   (ქ   (მ   (ე   (დ   (ა)  )))  ))
          (ღ   (ვ   (რ   (ი   (ა)  )))  ))
     (ნ   (ა   (ზ   (ღ   (ა   (უ   (რ   (ა)  )))  )))
          (თ   (ო   )))
     (რ   (ი   (დ   (ა   )))
          (ს   (ე   (ბ   (ს   )))  )
          (პ   (უ   (ს   (ი   )))  ))
     (ს   (ფ   (ა   (ლ   (ტ   (ე   (ბ   (ს   )))  )))  ))
     (ქ   (ა   (ფ   (ა   )))
          (ტ   (ი   (ვ   (ე   (ბ   (ს   )))  )))  )
     (ღ   (ე   (ლ   (ვ   (ა   )))  )
          (ო   (რ   (ძ   (ი   (ნ   (ა   )))  )))
     (შ   (ე   (ნ   (ა   )))
          (კ   (ა   (რ   (ა   (ვ   (ე   (ბ   (ს   )))  )))  )))
     (ჩ   (ქ   (ა   (რ   (ა   )))  ))
     (ც   (დ   (ი   (ნ   (ა   )))  )
          (ი   (ლ   (ა   )))  )
     (წ   (ყ   (ო   )))
     (ჭ   (რ   (ე   (ლ   (ა   )))  ))
     (ხ   (ე   (ნ   (ი   )))
          (ლ   (ე   (ბ   (ს   )))  ))   )
(ბ   (ა*  (ზ   (ა   (ნ   (ა   )))  ))
     (ა   (კ   (ი   ))   )         ))
```

აქ გამოყენებული გვაქვს სიტყვები კომპიუტერული ლექსიკონიდან (English Georgian Dictionary), რომელიც შეიცავს 50000 ქართულ სიტყვას.

თუ დავაკვირდებით, შევამჩნევთ, რომ სიტყვები, რომლებიც ერთი და იმავე ასოებით იწყება, გაერთიანებულია ერთ სიაში. ასეთია, მაგალითად: აალდა, აალება, აალებადი, აალებები, აალებული. ამ სიტყვების ერთობლიობის შესაბამის ქსელურ წარმოდგენას ექნება ასეთი სახე:



სიით წარმოდგენაში *-ით, ხოლო ქსელურ წარმოდგენაში შავად ნაჩვენებია წვერო, რომელზეც ერთი სიტყვა მთავრდება, ხოლო სხვა გრძელდება; მაგალითად, აალება და აალებადი, აბა და აბაზანა.

ასეთი წარმოდგენის უპირატესობა ისაა, რომ შესაძლებელია აღმოვაჩინოთ შეცდომით მოცემული სიტყვები იმ მიზნით, რომ მომხმარებელს შევატყობინოთ შეცდომა (საუკეთესო შემთხვევაში, ავტომატურად გავასწოროთ სიტყვა). ვგულისხმობთ, მაგალითად, ასეთ სიტუაციას _ ინტერნეტ-საძიებო სისტემაში მომხმარებელმა შეცდომით აკრიფა სიტყვა **აალებუბი.** ამ სიტყვის გარჩევისას აღმოჩნდება, რომ ლექსიკონში სიტყვის 6 ასოს დამთხვევის შემდეგ არსებობს ერთადერთი ალტერნატივა-სიტყვა **აალებული,** რომელიც ერთი ასოთი განსხვავდება მოცემული სიტყვისგან. მომხმარებელს ეკრანზე შესაბამისი შეტყობინება გამოუვა, სადაც ის გააკეთებს არჩევანს: მიიღოს შემოთავაზებული სიტყვა, თუ მოითხოვოს მსგავსი სიტყვების მოძებნის პროცესის გაგრძელება. ამ უკანასკნელ შემთხვევაში სიტყვების შედარების პროცესი ერთი წვეროთი ზემოთ ამოვა (ჩაითვლება, რომ სწორია სიტყვის პირველი 5 ასო: აალებ. სისტემა შემდეგ სიტყვებს: **აალება**, **აალებადი**, **აალებები**, **აალებული** ჩათვლის **აალებუბი**-ს ალტერნატიულ სიტყვებად და შესთავაზებს მათ მომხმარებელს.

ზოგადად, ბუნებრივი ენის ლექსიკონის ზემოთ მოყვანილი ჩადგმული ტიპის სიის წარმოდგენა საშუალებას იძლევა ერთი სტრუქტურით "გადაითარგმნოს" სიტყვის მნიშვნელობა რამდენიმე სხვა ენაზე. ამ მიზნით საჭიროა სიტყვის დამთავრების შემდეგ მოთავსდეს (ისევ რთული სიის სახით) ჩამონათვალი ამ სიტყვის შესაბამისი მნიშვნელობებისა სხვადასხვა ენაზე. ჩადგმული სია კი საჭიროა იმისათვის, რომ მიეთითოს მოცემული სიტყვის მნიშვნელობათა სინონიმები.

ამრიგად, ჩვენ ვაჩვენეთ, თუ როგორ შეიძლება ცოდნის წარმოდგენის სიის სტრუქტურების გამოყენება რაც შეიძლება ბუნებრივად, ისე, რომ ადვილი იყოს როგორც მათი იდენტიფიცირება, ასევე დამუშავების, ინფორმაციის ძებნის თანამედროვე, ეფექტური მეთოდების გამოყენება. სიის სტრუქტურებს დღემდე არ დაუკარგავთ ფასი და მოსალოდნელია მათი ეფექტური გამოყენება ინფორმაციის მოსაძებნად ინტერნეტ – საძიებო სისტემებში.

მონაცემთა ეფექტური ძიების ხერხები სიური სტრუქტურებისთვის

ინფორმაციის დამუშავების სხვადასხვა პროცესში ობიექტები შეიძლება წარმოდგნენ ან სახელით (შეფუთული, კონვერტირებული წარმოდგენა) ან როგორც თვისებათა მნიშვნელობების ერთობლიობა (გაშლილი, გახსნილი წარმოდგენა). მეხსიერების სტრუქტურამ უნდა უზრუნველყოს ერთი წარმოდგენიდან მეორეზე მარტივად გარდაქმნა. სტრუქტურიზაციის ძირითადი პროცესი ხდება კლასების მოდელების-ცნებების მნიშვნელობებს შორის ასოციური კავშირების ფორმირების მეშვეობით, ასევე იერარქიული დალაგებით, კლასიფიკაციით, ლოგიკური მნიშვნელობების განზოგადოებით.

მეხსიერების ორგანიზაციისას გავითვალისწინოთ ის, რომ მონაცემები წარმოდგენილი გვაქვს ქსელური სტრუქტურით, რომელიც ასახავს იერარქიულობას. საჭიროა ქსელმა ასახოს რეალური გარემოს იერარქიულობა და ამასთან იყოს მოსახერხებელი შედგენილი ობიექტების ნათესაური კავშირებისა და სტრუქტურის წარმოსადგენად. ქსელს უნდა ჰქონდეს განვითარებული ასოციატიური თვისებები, რათა შესაძლებელი იყოს მრავალმხრივი საძიებო ოპერაციების ჩატარება. ასევე, საჭიროა ქსელს ჰქონდეს ისეთი თვისებები, რაც თანამედროვე, ეფექტური ძიების მექანიზმების გამოყენების საშუალებას მოგვცემს.

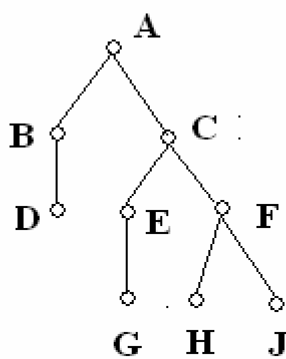
თავდაპირველად დავუშვათ, რომ ინფორმაცია წარმოდგენილია ხეების საშუალე-ბით (ქსელებს შემდგომში განვიხილავთ), რომლებიც წარმოდგებიან LISP-ის სიური სტრუქტურებით.

სემანტიკური ხეებით წარმოდგენილი ინფორმაცია ტრადიციულად მუშავდებოდა არსებული ტექნიკური შესაძლებებიდან გამომდინარე. კერძოდ, არსებობს ხეებზე ინფორმაციის მოძებნის შემდეგი საშუალებები [10]:

- **გავლათა პირდაპირი მიმდევრობისას** ჯერ მოდის წვერო, ხოლო შემდეგ მარცხენა ქვეხე. ალგორითმი ასეთია: მოხვდი წვეროში, გაიარე მარცხენა ქვეხე, გაიარე მარჯვენა ქვეხე.
- **გავლათა უკუ მიმდევრობისას** ალგორითმი ასეთია: გაიარე მარცხენა ქვეხე, მოხვდი წვეროში, გაიარე მარჯვენა ქვეხე.
- **ბოლო რიგით მიმდევრობისას** ალგორითმი ასეთია: გაიარე მარცხენა ქვეხე, გაიარე მარჯვენა ქვეხე, მოხვდი წვეროში.

დასახელება "პირდაპირი რიგი" ასახავს იმ ფაქტს, რომ თითოეული წვერო (ფესვი) გაივლის მანამ, სანამ მისი მარცხენა ქვეხე; "უკუ მიმართულებით" თითოეული წვერო (ფესვი) გაივლის თავისი მარცხენა ქვეხის შემდეგ.

განვიხილოთ, მაგალითად, შემდეგი ხე:



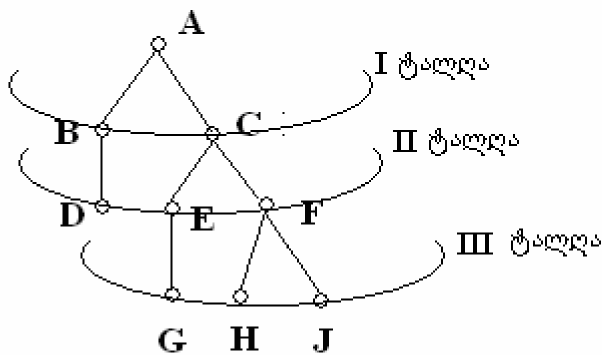
გავლათა პირდაპირი მიმდევრობისას განიხილება წვეროები შემდეგი მიმდევრობით:
A B D C E G F H J

გავლათა უკუ მიმდევრობისას განიხილება წვეროები შემდეგი მიმდევრობით:
D B A G E C H F J

ხოლო ბოლო რიგით მიმდევრობისას:
D B G E H J F C A

იმის გათვალისწინებით, რომ ფართოდ გამოიყენება მრავალპროცესორიანი კომპიუტერები, აქტუალურ მნიშვნელობას იძენს ძებნის თანამედროვე მეთოდები. მათ შორის ყურადღებას გავამახვილებთ "ტალღური ძებნის" მეთოდის პრინციპზე.

ეს პრინციპი შემდგომშია: ძებნა იწყება ხის წვეროდან. თუ წვერო მოიძებნა, მაშინ წარმატებულად დამთავრდება საძიებო პროცესი, თუ არა და შედარება გაგრძელდება წვეროდან გამომავალ "პირველ ტალღაზე", ანუ წვეროს შვილობილ წვეროებზე. საჭირო წვეროს მოძებნის შემთხვევაში პროცესი მთავრდება, თუ არა და შედარების პროცესი გრძელდება შვილობილი წვეროების შვილობილ წვეროებზე ("მეორე ტალღაზე") ხდება და ა.შ. ეს წარმოდგენილია შემდეგ ნახაზზე:



“ტალღური” ძებნისას წვეროები განიხილება შემდეგი მიმდევრობით:
A - წვერო,
B C - პირველი ტალღის წვეროები,
D E F – მეორე ტალღის წვეროები,
G H J - მესამე ტალღის წვეროები.

სემანტიკური ხის შემთხვევაში აღწერილი, წარმოდგენილი პროცესი მთავრდება, როცა იქნება მოძებნილი საჭირო წვერო ან როცა მიიღწევა ბოლო წვეროები (ფოთლები).

სემანტიკური ქსელის შემთხვევაში ძებნა იწყება სპეციალურად მონიშნული წვეროდან (სათავიდან) და ძებნის პროცესისთვის საჭირო ხდება დამატებით პირობების შემოტანა:

- მივუთითოთ მაქსიმალურად რამდენი დონის “ტალღაზე” გასვლაა დასაშვები;
- უკვე გავლილი წვეროების “მონიშვნა” შემდგომი გავლების ასაკრძალავად.

ამ მეთოდის უპირატესობა მდგომარეობს იმაში, რომ მისი რეალიზება შესაძლებელია მრავალპროცესორიან კომპიუტერზე. [11]-ში განხილულია ასეთი ხეების წარმოდგენა ფუნქციონალური დაპროგრამების ენა LISP-ის სიების საშუალებით, ხოლო ძებნის ალგორითმი რეალიზებულია ე.წ. Map-ფუნქციონალების საშუალებით საზოგადოდ ფუნქციონალი mapcar უზრუნველყოფს ფუნქციონალური არგუმენტის რეალიზაციას სიის ყველა ელემენტზე და შედეგს აერთიანებს სიაში. მაგალითად, (mapcar 'list '(a b c)) გვაძლევს შედეგს '((a)(b)(c)).

ძებნის პროცესის განსახირციელებლად ფუნქციონალზე მიმართვას ექნება სახე:

(mapcar <ძებნის ფუნქცია> <ძირითადი ქსელი>)

ჩვენს შემთხვევაში mapcar ასე მუშაობს: ძებნის ფუნქცია გამოიყენება ხის წვეროსთვის, იგივე ფუნქცია თითოეულ ქვეხეზე და ა.შ. პროცესი მთავრდება როცა იქნება მოძებნილი საჭირო წვერო ან როცა მიიღწევა ბოლო წვეროები (ფოთლები).

LISP-ის გაფართოებისას პარალელური დაპროგრამების შესაძლებლობებით, როდესაც რეალიზებულია Map-ფუნქციონალები მრავალპროცესორიანი კომპიუტერისთვის, შესაძლებელია ძებნა თითოეული წვეროსთვის მოხდეს დამოუკიდებლად სხვა ძებნებისა. ეს, ცხადია, “ტალღური ძებნის” მეთოდს წარმატებულს ხდის.

ცოდნის სემანტიკური ქსელებით წარმოდგენისთვის ვუდსის [12,13] მიერ დამატებით შემოტანილი იყო შემდეგი საშუალება - ქსელის ნებისმიერი წვეროდან შესაძლებელი იყო სხვა ქსელზე გადასვლა. თუ ის დამატებითი ქსელი ძირითადი ქსელის ტიპის იქნებოდა, მაშინ ამას განსაკუთრებული აზრი არ ექნებოდა, ვინაიდან ამ წვეროზე იგივე ქსელითაც იქნებოდა შესაძლებელი გაგრძელება. ამ შემთხვევაში არ ირღვეოდა არც ქსელის სტრუქტურა, არც იცვლებოდა ძებნის ხერხები. ვუდსის დამსახურება ის იყო, რომ წვეროზე მითითებული ახალი ქსელი იყო განსხვავებული

სტრუქტურის, მისი დამუშავება ხდებოდა ძირითადი ქსელისგან განსხვავებით და ხდებოდა მიღებული მნიშვნელობის ძირითად ქსელზე დააბრუნება.

ამრიგად, ქვექსელის დანიშნულება არის სხვა ტიპის ქსელზე გადასვლა. მიზანი-ქვექსელი დავამუშავო და შედეგი გადავცე უკან.

მრავალპროცესორიანი კომპიუტერებზე დაპროგრამებისას, ვუდსის ეს იდეა ახალ ელფერს იძენს, ვინაიდან ქვექსელზე დამუშავება ცალკეულ პროცესორებზე შეიძლება, ვინაიდან ეს პროცესი დამოუკიდებელია ძირითადი ქსელიდან.

მაგალითად, დავუშვათ აღწერილი გვაქვს საგნობრივი არე ლოგიკო-ლინგვისტიკური მეთოდებით. ძირითად ქსელში აღწერილი ცოდნა-წინადადებები სემანტიკური ქსელებითაა წარმოდგენილი. ასეთ შემთხვევაში ქვექსელებით იქნება რეალიზებული სიტყვების მორფოლოგიური ანალიზი. ცხადია, რომ ქსელს და ქვექსელს განსხვავებული სტრუქტურები აქვთ.

მეორე მაგალითი, ძირითად ქსელური სტრუქტურით აღწერილი გვაქვს ბუნებრივი ენის ანბანი, ქვექსელში უცხო ენაზე (ენებზე) გადათარგმნილი სიტყვის სრული, განმარტებითი ინფორმაცია. ამ შემთხვევაშიც ქვექსელის სტრუქტურა განსხვავებულია ძირითადი ქსელის სტრუქტურისაგან.

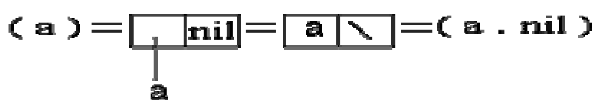
განვიხილოთ მეხსიერების სტრუქტურა “ტალღური ძიების” განსახორციელებლად.

მანქანის მეხსიერებაში სიები ინახება არა როგორც სიმბოლოების თანმიმდევრობა, არამედ სტრუქტურული ფორმების სახით, რომლებიც აგებულია მანქანური სიტყვებისაგან, როგორც ხეების ნაწილი. მისამართები ასეთ ჩანაწერებში მითითებულია ტეგებით-მონაცემთა სპეციალური ტიპით, რომელიც მიმთითებლით არის მოცემული. სიის სტრუქტურის სქემატურად წარმოსადგენად მანქანური სიტყვის დიაგრამის სახით ხატავენ როგორც მართკუთხედს, რომელიც ორ ნაწილად არის გაყოფილი: მისამართი და დეკრემენტი.

სიების სტრუქტურით S-გამოსახულებების მეხსიერებაში ჩაწერას მთელი რიგი უპირატესობა აქვს:

- გამოსახულების ზომისა და რაოდენობის, რომელთანაც პროგრამას აქვს საქმე, შეიძლება წინასწარ არ განისაზღვროს. ამასთან, ნებისმიერი გამოსახულების ფიქსირებული სიგრძის მეხსიერების ბლოკებში ჩაწერა გამორიცხულია;
- მეხსიერების უჯრედები შეიძლება გადატანილ იქნას თავისუფალ მეხსიერებაში იმ წუთშივე, როცა მათი გამოყენების აუცილებლობა ქრება.
- გამოსახულება, რომელიც რამდენიმე გამოსახულების გაგრძელებას წარმოადგენს, შეიძლება შენახული იყოს მხოლოდ ერთ ეგზემპლიარად.

მაგალითად, ერთი ატომისგან შემდგარი სია წარმოდგება ასე:



nil მიუთითებს სიის ბოლოს

nil-ის ნაცვლად წერენ - \ -ს.

სია მიიღება როგორც ოპერაცია (cons 'a nil).

ორი ელემენტისგან შემდგარი სია (b a)

ნებისმიერი სია შეიძლება ჩავწეროთ წერტილოვანი ნოტაციით.

```
(a) <=> (a.nil)
```

```
(a b c) <=> (a.(b.(c.nil)))
```

“ტალღური ძიების” განხორციელება სასურველია Lisp-მანქანების [14] საშუალებით. ასეთი ტიპის მანქანები მოხერხებულია იმით, რომ არ არის შეზღუდვა მეხსიერების უჯრედის სიგრძეზე და მთელი მეხსიერება გამოიყოფა, როგორც სიური სტრუქტურა. ჩანაწერი, ატომი შეიძლება იყოს ნებისმიერი, წარმოუდგენლად დიდადგილიანი (ნაცვლად 16 ან 32 ბიტისა).

უპირატესობა ასეთი ტიპის მეხსიერების გამოყენებისას ისაა, რომ ყველა სტრუქტურა მარტივად თავსდება და ძებნა სწრაფად ხდება.

Lisp-მანქანების პროექტირების მიზანი იყო შექმნილიყო პერსონალური გამოთვლითი მანქანა (მკვეთრად განსხვავებული არქიტექტურით, ვიდრე IBM PC), რომლის გამოყენებაც შესაძლებელი იქნებოდა როგორც ხელოვნური ინტელექტის გამოკვლევებისთვის, ასევე სხვადასხვა საწარმოო და კომერციული პროექტებისთვის. მის გავრცელებას ხელი შეუშალა დიდი მოცულობის პროგრამული უზრუნველყოფის გადატანის საჭიროებამ. Lisp-მანქანებს შემდეგი უპირატესობა აქვთ:

- ძირითადი მეხსიერების დიდი მოცულობა;
- აპარატული საშუალებების ორიენტაცია Lisp-ზე გამოთვლებისთვის;
- ტიპების შემოწმება აპარატურის დონეზე (tagged architecture);
- Lisp-მანქანაზე ინტეგრირებული გარემოს გამოყენების შესაძლებლობა.

ამრიგად, როდესაც ცოდნა წარმოდგენილი გვაქვს სემენტიკური ქსელებით, მაშინ მიგვაჩნია, რომ ძებნის ალგორითმი უნდა იყოს “ტალღური” და მისი რეალიზება მოხდეს ან პარალელური დაპროგრამების პრინციპებით გაფართოებულიL LISP-ის გამოყენებით მრავალპროცესორიან კომპიუტერებზე ან lisp-მანქანებზე.

პარალელური დაპროგრამება ფუნქციონალური დაპროგრამების მეშვეობით

კომპიუტერების წარმოებადობის გაზრდის ერთ-ერთი საშუალებაა პარალელური პროცესების მართვა, რომლის ორგანიზება ითხოვს დროითი კავშირების გათვალისწინებას და მოქმედებების მართვის არაიმპერატიულ სტილს.

დაპროგრამების ენა LISP თავისი არსებობის ნახევარსაუკუნოვანი ისტორიის მანძილზე და დღემდე კვლავაც რჩება პრაქტიკულად შეუზღუდავი შესაძლებლობის სიმბოლური დაპროგრამების სისტემად. როგორც ცნობილია, LISP და მისი დიალექტები გახდა საფუძველი მთელი რიგი გამოყენებითი ხასიათის კვლევებისა, რომლებმაც დიდი როლი შეასრულეს საინფორმაციო ტექნოლოგიების გავრცელებაში.

როგორც ცნობილია, LISP არის დაპროგრამებადი დაპროგრამების ენა (LISP is a programmable programming language). ჩვენს მიერ შესწავლილი იყო LISP-ის ბაზაზე უნივერსალური დაპროგრამების საკითხები [11]. ვაჩვენოთ, როგორ შეიძლება იგი გახდეს პარალელური დაპროგრამების ენა, ანუ როგორ შეიძლება მოხდეს პარალელური გამოთვლების ორგანიზება მრავალპროცესორიანი პერსონალური კომპიუტერისთვის.

პარალელური გამოთვლების შესაძლებლობა – ეს ისაა, რაც LISP-ს თავიდანვე ჩადებული აქვს ე.წ. ფუნქციონალების სახით. ფუნქციონალების შესრულება LISP-ის პირველ ვერსიებსაც შეეძლოთ. ასე, რომ დაპროგრამების ფუნქციონალური სტილიდან პარალელურ დაპროგრამებაზე გადასვლა, მიგვაჩნია ბუნებრივად. ამასთან, გვინდა ავღნიშნოთ, რომ ამჟამად, როცა გამოთვლითი ტექნიკის განვითარება მრავალპროცესორიანი პერსონალური კომპიუტერების შექმნის მიმართულებით ხდება,

ძალზე აქტუალურია ის, რომ თანამედროვე დაპროგრამების ენებში ჩადებული იყოს გამოთვლების პარალელურად შესრულების შესაძლებლობა.

ფუნქციონალურ ენებში განსაზღვრულია მაღალი რიგის ფუნქციები, რომელთაც ფუნქციონალები ეწოდებათ. ფუნქციონალები – ეს ფუნქციებია, რომლებიც არგუმენტად იყენებენ და/ან ფუნქციის შედეგად განსაზღვრავენ სხვა ფუნქციებს. ფუნქციონალის განსაზღვრებისას ცვლადების როლს ასრულებენ ფუნქციების სახელები, რომელთა განსაზღვრებებიც მოცემულია გარე ფორმულებით, რომელთაც ფუნქციონალები იყენებენ.

საჭიროა აღინიშნოს, რომ მონაცემები და პროგრამები LISP-ში წარმოდგებიან ერთნაირად, ამიტომ განსხვავება ცნებებს შორის "მონაცემი" და "ფუნქცია" განისაზღვრება არა მათი სტრუქტურით, არამედ მათი გამოყენებით. თუ არგუმენტი ფუნქციაში გამოიყენება როგორც ობიექტი, რომელიც მხოლოდ გამოთვლებში მონაწილეობს, მაშინ იგი არის ჩვეულებრივი არგუმენტი-მონაცემი, ხოლო თუ იგი გამოიყენება როგორც საშუალება, რომელიც განსაზღვრავს გამოთვლებს, მაგალითად, გამოდის lambda გამოსახულების როლში, მაშინ იგი არის ფუნქცია.

მაგალითად:

```
(car '(lambda (x)(list x))) → lambda     ;მონაცემი
((lambda (x)(list x)) car)→(car)          ;ფუნქცია
```

LISP-ში განსაზღვრულია ამსახველი ფუნქციონალები ანუ map-ფუნქციონალები როგორც ფუნქციები, რომლებიც სიას (თანმიმდევრობას) რაღაც საშუალებით ასახავენ (map) ახალ სიაში, ან მეორადი მოქმედებით წარმოქმნიან ამ თანმიმდევრობას. map-ფუნქციონალების სახელები იწყება სიტყვა map-ით, მათ გამოძახებას აქვს სახე:

(MAPx f_n l_1 l_2 ... l_N), სადაც l_1 ... l_N-სიებია, ხოლო f_n- N არგუმენტის ფუნქცია.

როგორც წესი, map-ფუნქციები გამოიყენება ერთ არგუმენტზე-სიაზე, ანუ f_n-ფუნქცია ერთ არგუმენტიანია: (MAPx fn სია). აღნიშვნა MAPx გამოყენებული გვაქვს როგორც ერთიანი სახელი ფუნქციებისთვის: mapcar, mapcan, maplist, mapcon და ა.შ.

ამსახველი ფუნქციონალები იყოფა ორ ჯგუფად. პირველი, რომლებიც უზრუნველყოფენ მოცემული სიის თითოეული ელემენტის დამუშავებას (mapcar, mapcan) და მეორე, რომლებიც უზრუნველყოფენ საწყისი სიისა და მისი ყოველი კუდის დამუშავებას (maplist, mapcon). სიის კუდი არის სია, საიდანაც პირველი ელემენტია ამოგდებული.

ფუნქციონალი mapcar უზრუნველყოფს ფუნქციონალური არგუმენტის რეალიზაციას სიის ყველა ელემენტზე და შედეგს აერთიანებს სიაში. მაგალითად, (mapcar 'list '(a b c)) გვაძლევს შედეგს '((a) (b) (c)). ფუნქციონალი mapcan ანალოგიურია mapcar იმ განსხვავებით, რომ შედეგის გამოსატანად გამოიყენება ფუნქცია ncons. მაგალითად, (mapcan 'list '(a b c)) გვაძლევს შედეგს '(a b c).

ფუნქციონალი maplist უზრუნველყოფს ფუნქციონალური არგუმენტის რეალიზაციას სიაზე და მის ყველა კუდზე. მაგალითად, (maplist 'list '(a b c)) გვაძლევს შედეგს '(((a b c))((b c))((c))). ფუნქციონალი mapcon ანალოგიურია maplist-ის იმ განსხვავებით, რომ შედეგის გამოსატანად გამოიყენება

ფუნქცია ncons. მაგალითად, (mapcon 'list '(a b c)) გვაძლევს შედეგს '((a b c)(b c)(c)).

მაგალითი:

(mapc 'list '(a b c)) გვაძლევს შედეგს '(a b c).

(mapl 'list '(a b c)) გვაძლევს შედეგს '(a b c).

ენა Clisp-ში ფუნქციონალებს მიეკუთვნება შემდეგი ფუნქციები: Mapc, Mapcan, Mapcar, Mapcon, Mapl, Maplist. LISP-ის ერთ-ერთ თანამედროვე ვერსიაში Objective CAML შემოღებულია ნაკადების ცნება და არის პარალელური ალგორითმის ჩაწერის საშუალება. Objective CAML-ს აქვს ბიბლიოთეკა "მსუბუქი" პროცესების ნაკადებისთვის, რომლებიც ორგანიზებულია თვით პროცესის და არა ოპერაციული სისტემის მიერ. ასეთი ნაკადები იყენებენ მათი წარმომქმნელი პროცესის მისამართების არეს და ამიტომ ითხოვენ ნაკლებ რესურსებს. პრინციპული განსხვავება ნაკადსა და პროცესს შორის არის ის, ერთდროულად იყენებენ თუ არა მეხსიერებას მონაცემებისთვის ერთი და იგივე პროგრამის შვილობილი პროცესებისთვის. ნაკადების გამოყენება არის საშუალება მოხდეს პარალელური ალგორითმების შესრულება ენის ფარგლებში.

ფუნქციონალების გამოყენება დაპროგრამების ენებში იძლევა მთელ რიგ უპირატესობებს:

- მათი მეშვეობით შესაძლებელია აიგოს პროგრამები უფრო მსხვილი მოქმედებებისაგან (ვიდრე ფუნქციების მეშვეობით);
- უზრუნველყოფენ ასახვის მოქნილობას;
- ფუნქციის განსაზღვრება შეიძლება არ იყოს დამოკიდებული ფუნქციის სახელთან (ფუნქციის განსაზღვრება lambda გამოსახულებით);
- ფუნქციონალების საშუალებით შეიძლება საშედეგო ფორმების მართვა;
- ფუნქციონალის პარამეტრი შეიძლება იყოს ნებისმიერი ფუნქცია, რომელიც გარდაქმნის ფუნქციის ელემენტებს;
- ფუნქციონალები საშუალებას იძლევიან მოხდეს ფუნქციათა სერიის ფორმირება საერთო მონაცემებიდან;
- ურთიერთდაკავშირებული ფუნქციების ნებისმიერი სისტემა შეიძლება გარდაიქმნას ერთ ფუნქციად, უსახელო ფუნქციების გამოძახებებით.

როგორც ვნახეთ, map-ფუნქციონალები ისე ახდენენ გამოთვლებს, რომ პირველი არგუმენტით მოცემული ფუნქცია გამოითვლება მეორე არგუმენტით მოცემულ სიის თითოეულ წევრზე (პირველი ჯგუფის ფუნქციონალებისას), ან მეორე არგუმენტით მოცემულ სიასა და მის კუდებზე (მეორე ჯგუფის ფუნქციონალებისას). ამიტომ შეგვიძლია ვთქვათ, რომ map-ფუნქციონალები თავისი ბუნებით არიან "პარალელურები". საჭიროა მრავალპროცესორიანი კომპიუტერისთვის ენის კომპილატორი შეიქმნას ისე, რომ შესაძლებელი იყოს ფუნქციის გამოთვლა ყოველ არგუმენტზე ჩატარდეს სხვადასხვა პროცესორზე. თითოეული გამოთვლა მოხდეს დამოუკიდებლად პროცესორზე, რომელიც დაუბრუნებს შედეგს map-ფუნქციონალს იმ რიგის მითითებით, რომლითაც მოხდა მისი გამოძახება. უფრო ზუსტად, პირველი ჯგუფის ფუნქციონალების დროს პროცესორს გადაეცემა ფუნქციის სახელი და ის სია, რომელზეც მოქმედებას ასრულებს, ხოლო მეორე ჯგუფის ფუნქციონალების დროს _ ფუნქციის სახელები და გამოთვლილი კუდები. პროცესორი გამოთვლების შედეგს ამ გამოთვლის ნომრის მითითებით დაუბრუნებს map-ფუნქციონალს, და განთავისუფლდება თუ არა, შეასრულებს შემდეგ გამოთვლებს. ცხადია, ეს პროცესი არ იქნება დამოკიდებული კომპიუტერში პროცესორების რაოდენობაზე.

მიზანშეწონილად მიგვაჩნია, რომ თანამედროვე პროცედურული და ობიექტზე ორიენტირებული ენები გაფართოვდნენ map-ფუნქციონალებით და სწორედ ეს ფუნქციონალები უნდა გახდნენ ამ ენების "გაპარალელების" საშუალებები. ამასთან, უნდა მოხდეს თითოეული ტიპის map–ფუნქციონალის გადატვირთვა ისე, რომ მეორე არგუმენტად შეიძლება გამოყენებული იყოს სია, მასივი (მთელი და ნამდვილი რიცხვების), სტრიქონი, ფაილი და მომხმარებლის მიერ განსაზღვრული ტიპიც. map–ფუნქციონალების გამოყენებით ნებისმიერი პარალელური ალგორითმის ჩაწერაა შესაძლებელი.

პარალელური გამოთვლების შესრულება მრავალპროცესორიან კომპიუტერებზე მოგვცემს მთელ რიგ უპირატესობას ინფორმაციის ძიების ამოცანებში. ინფორმაცია, წარმოდგენილი ქსელებით, ადვილად წარმოდგება LISP-ის სიების საშუალებით. მათზე სტანდარტული ძიების ალგორითმების ნაცვლად, როგორებიცაა პირდაპირი გზით გავლა, უკუ მიმართულებით გავლა და გავლა ბოლო რიგით, შეიძლება რეალიზებული იყოს ძიება ტალღური პრინციპით. ტალღური ძიების მეთოდით შესაძლებელია ძიება მოხდეს წვეროდან პირველ დონეზე. თუ მოიძებნა შედეგი, პროცესი დამთავრდება, თუ არა და, ძიება გაგრძელდება ყველა ტოტის მეორე დონეზე. თითოეული ტოტისთვის ცალკეული პროცესორია გამოყოფილი (თუ, რა თქმა უნდა, ტექნიკურად ამის შესაძლებლობა არსებობს). ასე, რომ ვინაიდან გვაქვს პარალელური სტრუქტურები, ერთდროულად, პარალელურად ხდება ძიება.

ასეთი ტიპის ამოცანა არის, მაგალითად, ბუნებრივი ენის ლექსიკონში სიტყვების ძებნა, როცა ენის სემანტიკური ქსელი წარმოდგენილია ცალკეულ ხეებად. ძებნა შეიძლება მოხდეს ცალკეული პროცესორების მიერ ცალკეულ ხეებში, რითაც შესაძლებელია მკვეთრად შემცირდეს ძებნის პროცესისთვის საჭირო დრო.

ნაჩვენებია, როგორ შეიძლება LISP-ზე, რომელიც ფუნქციონალური დაპროგრამების ენაა, მოხდეს პარალელური გამოთვლების ორგანიზება. პარალელური გამოთვლების შესაძლებლობა განპირობებულია მაღალი რიგის ფუნქციების-ფუნქციონალების არსებობით. საჭიროა არსებული სინტაქსისა და სემანტიკის ფარგლებში შეიცვალოს map–ფუნქციონალების ინტერპრეტაცია ისე, რომ შესაძლებელი იყოს გამოთვლების თითოეული ნაწილის ცალკეულ პროცესორზე შესრულება (იგულისხმება, რომ კომპიუტერი მრავალპროცესორიანია). თანამედროვე პროცედურული და ობიექტზე ორიენტირებული ენების map–ფუნქციონალების ტიპის ფუნქციონალებით გაფართოება საშუალებას მისცემს მომხმარებლებს ჩაწერონ პარალელური ალგორითმები.

ობიექტებზე ორიენტირებული დაპროგრამება ფუნქციონალური დაპროგრამების ბაზაზე

იმპერატიულ დაპროგრამებაში მოქმედებები ხორციელდება ოპერატორების ცხადად განსაზღვრული თანმიმდევრობით. ფუნქციონალურ დაპროგრამებაში მოქმედებათა შესრულების რიგს აუცილებელი გამოთვლები გვიჩვენებს. რაც შეეხება ობიექტებზე ორიენტირებულ დაპროგრამებას, აქ გამოთვლები იმართება მონაცემებით. ობიექტების გამოყენებას მივყავართ პროგრამების ახალ ორგანიზაციამდე. განისაზღვრება კლასები და მათი ობიექტები. კლასი აერთიანებს მონაცემებს და ოპერატორებს _ მეთოდებს, რომლებიც აღწერენ კლასის შესაძლო მოქმედებებს.

LISP არის დაპროგრამებადი დაპროგრამების ენა (Lisp is a programmable programming language), რაც ნიშნავს, რომ თვით ამ ენაზევე შეიძლება დაიწეროს

მისივე კომპილატორი. ჩვენს მიერ ნაჩვენები იყო, თუ როგორ შეიძლება გამოიყოს LISP–ში ბირთვი, რომელზეც შესაძლებელია უნივერსალური ენის აგება.

ვაჩვენოთ, როგორ შეიძლება ობიექტებზე ორიენტირებული დაპროგრამების სქემების რეალიზაცია ისე, რომ დავრჩეთ ფუნქციონალური დაპროგრამების ჩარჩოში. ამისთვის ავაგოთ ობიექტებზე ორიენტირებული ენის მოდელი, რომელიც ჩადგმული იქნება LISP-ში და რომლის საშუალებით ავსახავთ ობიექტებზე ორიენტირებული დაპროგრამების ტიპიურ ცნებებს ფუნქციონალური დაპროგრამებისთვის დამახასიათებელ ფუნდამენტურ აბსტრაქციებში.

მემკვიდრეობითობის ორგანიზებისთვის საჭიროა განვსაზღვროთ განსხვავებები განზოგადოებული ფუნქციის მოდელებსა და შეტყობინებების გაცვლას შორის. ობიექტებს აქვთ თვისებები, ობიექტები აგზავნიან შეტყობინებებს, ისინი მემკვიდრეობით იღებენ თვისებებსა და მეთოდებს.

საზოგადოდ, ობიექტებზე ორიენტირებული დაპროგრამება არის პროგრამის ორგანიზაცია მეთოდების, კლასების, ეგზემპლიარებისა და მემკვიდრეობითობის ტერმინებში. ასეთი ორგანიზაციის უპირატესობაა ის, რომ პროგრამის შეცვლა ადვილადაა შესაძლებელი. მაგალითად, თუ გვინდა შევცვალოთ რომელიღაც კლასის ნებისმიერ ობიექტზე მანიპულაციის საშუალება, მაშინ ჩვენ ვცვლით მხოლოდ ამ კლასის მეთოდს. ჩვენ თუ გვსურს მოცემული ობიექტის მსგავსი ობიექტის შექმნა, რომელიც განსხვავდება მისგან ცალკეული თვისებებით, ჩვენ შეგვიძლია შევქმნათ ამ კლასის ქვეკლასი. სწორედ მას შევუცვალოთ ცალკეული თვისებები. ასეთი ცვლილებები არ იწვევს საწყისი კოდის ცვლილებებს.

LISP-ში არის სხვადასხვა საშუალებები ობიექტებზე ორიენტირებული პროგრამის წარმოდგენისთვის. ერთ-ერთია – ობიექტების წარმოდგენა ე.წ. ჰეშ-ცხრილების საშუალებით [15]. ასეთი რეალიზაციის დროს ობიექტები არ არის დაყოფილი კლასებად და ეგზემპლიარებად. ეგზემპლიარი განიხილება მხოლოდ, როგორც კლასი ერთადერთი მშობლით.

[15]-ში ობიექტებზე ორიენტირებული პროგრამის ვექტორული რეალიზაცია არის გამოყენებულია. ამ დროს კლასებად და ეგზემპლიარებად დაყოფა რეალური ხდება, თუმცა ეგზემპლიარს თვისებების უბრალო ცვლილებით კლასად ვერ გადავაქცევთ.

თუ გამოვიყენებთ LISP-ის თვისებათა სიას, შესაძლებელია აიგოს ობიექტებზე ორიენტირებული დაპროგრამების შემდეგი მოდელი, რაც გვიჩვენებს ფუნქციონალური და ობიექტებზე ორიენტირებული დაპროგრამებების ნათესაობას:

```
(DEFUN CLASSES (CL)
   (COND (CL (CONS (CDAR CL) (CLASSES (CDR CL)))) ))

; კლასის არგუმენტების ფორმულის გამოყვანა მეთოდის პარამეტრების
; Nil — ნებისმიერი კლასი

(DEFUN ARGUM (CL) (COND
   (CL (CONS (CAAR CL) (ARGUM (CDR CL)))) ))

; არგუმენტის სახელების გამოყვანა
; მეთოდის პარამეტრის განსაზღვრებიდან

(DEFUN DEFMET (FMN C-AS EXPR)
  (SETF (GET FMN 'CATEGORY) 'METHOD)
  (SETQ ML (CONS(CONS(CONS FMN (CLASSES C-AS))
     (LIST 'LAMBDA (ARGUM C-AS) EXPR) ) ML))
                                  FMN )
```

```
    ; მეთოდის გამოცხადება და მისი განსაზღვრების განცალკევება
  ; კლასის არგუმენტებთან შეთანადების მიზნით
          (DEFUN DEFCL (NCL SCL FCL ))
    ; სახელი, სუპერკლასი და კლასის ველები/სლოტები
       (SETQ ALLCL (CONS NCL ALLCL))
       (SET NCL (APPEND FCL SCL)) )

    ;  კლასის სახელი არის მისი ველების სია, შესაძლოა მნიშვნელობებით
     (DEFUN EV-CL (VARGS) (COND
    ;  ფაქტიური არგუმენტების ფორმატის გამოყვანა
  ; მათი დამუშავების მეთოდის შესარჩევად
        (VARGS (CONS (COND
             ((MEMBER (CAAR VARGS) ALLCL) (CAAR VARGS)) )
                                 (EV-CL (CDR VARGS)))) ))
    ; NIL  იმ შემთხვევაში, თუ კლასი არაა
     (DEFUN M-ASSOC (PM MELI) (COND (MELI
           (COND ((EQUAL (CAAR MELI) PM)(CDAR MELI))
                 (T(M-ASSOC PM (CDR MELI)))))))
    ;  შესაბამისი მეთოდის მოძებნა, რომელიც შეესაბამება
    ;  კლასის მონაცემების
    (DEFUN METHOD (MN ARGS &OPTIONAL C)
       (APPLY (M-ASSOC (CONS MN (EV-CL ARGS)) ML)
                                     ARGS C))
    ;  იმ შემთხვევაში, თუ მეთოდი პროგრამაში არ მოიძებნა,
  ;  საჭიროა მოხდეს პარამეტრების დაყვანა საჭირო კლასზე
     (DEFUN INSTANCE (CLASS &OPTIONAL CP) (COND
     ;  მსგავსად  LET  -კონტექსტის უსახელო კოპიო
   (CLASS (COND ((ATOM (CAR CLASS))(INSTANCE
                                       (CDR CLASS) CP))
               ((ASSOC (CAAR CLASS) CP) (INSTANCE
                                       (CDR CLASS) CP))
               (T(INSTANCE (CDR CLASS) (CONS (CAR CLASS)
                                        CP)))
                                          )) ) CP)
     (DEFUN SLOT (OBJ FLD) (ASSOC FLD OBJ))
     ;  ობიექტის ველის მნიშვნელობა
```

საჭირო ხდება Lisp-ის ინტერპრეტატორში ჩაემატოს ახალი განშტოებები. ამ შემთხვევაში კლასებს და ობიექტების ეგზემპლიარებს ასე ავღწერთ:

```
(defclass ob () (f1 f2 ...))
```

ეს იმაზე მიუთითებს, რომ თითოეულ ობიექტს ექნება ველები-სლოტი f1 f2 ... (სლოტი არის ჩანაწერის ველი ან თვისებათა სია). კლასის წარმოდგენისთვის უნდა გამოძახებული იყოს ზოგადი ფუნქცია:

```
(setf c (make-instance 'ob))
```

ველის მნიშვნელობის განსაზღვრისთვის საჭიროა სპეციალური ფუნქცია იყოს გამოყენებული:

```
(setf (slot-value c) 1223).
```

სლოტის აღწერისთვის საჭიროა მას მივცეთ სახელი და თვისებების ველი. სლოტის თვისებები სპეციფირდება, როგორც ფუნქციის გასაღები პარამეტრები. ეს საშუალებას იძლევა განისაზღვროს საწყისი მნიშვნელობები. ასეთი სლოტის ცვლილება იქნება ხელმისაწვდომი კლასის ყველა ეგზემპლიარისთვის.

როგორც ვხედავთ, კონცეპტუალურად ობიექტებზე ორიენტირებული დაპროგრამება მეტი არაფერია, თუ არა Lisp-ის იდეების პერეფრაზირება. ობიექტებზე ორიენტირებული დაპროგრამება ეს ისაა, რისი გაკეთებაც ამ ენას თავიდანვე შეეძლო. ფუნქციონალური სტილიდან ობიექტებზე ორიენტირებული დაპროგრამების სტილზე გადასვლაში არაფერია მოულოდნელი. ხდება მხოლოდ ფუნქციონალური ობიექტების განშტოებების გადარჩევის მექანიზმების მცირე დაკონკრეტება.

როგორც ცნობილია, პირველი თაობის ფუნქციონალური ენების გამოყენების საწინააღმდეგო არგუმენტად ასახელებდნენ მათი ეფექტური რეალიზაციის განხორციელების შეუძლებლობას, ვინაიდან ისინი დაფუძნებულები იყვნენ ინტერპრეტაციასა და ტიპების დინამიურ კონტროლზე. თანამედროვე ფუნქციონალური ენებისთვის ეს არგუმენტი მიუღებელია.

ფუნქციონალური დაპროგრამების ბოლო თაობის ენა არის Objective Calm, რომელიც დამუშავდა INRIA–ში (ინფორმატიკისა და ავტომატიზაციის ინსტიტუტი, საფრანგეთი). Objective Calm-ს აქვს აქვს ტრადიციული იმპერატიული ენების მთელი რიგი კარგაც ნაცნობი მახასიათებლები და ენა C–ის კონსტრუქციები. როგორც სახელიდან ჩანს, იგი მხარს უჭერს ობიექტზე ორიენტირებულ დაპროგრამებას, კერძოდ მას ახასიათებს პარამეტრიზირებული პოლიმორფიზმი. განვიხილოთ იგი.

ფუნქციონალების არსებობა საშუალებას იძლევა გამოითვალოს მთელი რიგი კონსტრუქციები. მაგალითად, ფუნქცია:

```
let o f g = fun x -> f ( g x )
```

აბრუნებს ფუნქციას, რომელიც არის პარამეტრების კომპოზიცია. რა ტიპი აქვს ამ შემთხვევაში პარამეტრებს და ფუნქცია o-ს შედეგს? ამ სიტუაციის გარკვევაში გვეხმარება Objective CAML-ის პოლიმორფიზმის ცნება.

ასე, რომ ფუნქცია o-ს პარამეტრებისა და შედეგის ტიპების ზუსტი განსაზღვრა შეუძლებელია, თუმცა, სინტაქსიდან გამომდინარე შესაძლებელია შემდეგი დასკვნების გაკეთება:

- ფუნქცია o -ს გამოყენების შედეგად დასაბრუნებელი მნიშვნელობა არის ფუნქცია (ვინაიდან მის ტანში არის კონსტრუქცია fun;
- ფუნქცია o -ს არგუმენტები ფუნქციებია (ვინაიდან ტანში ხდება მათი გამოძახება);
- g ფუნქციის შედეგი არის f ტიპის არგუმენტი (ენაში არაა განსაზღვრული ტიპებზე დაყვანის ოპერაცია);
- ფუნქცია o -ს შედეგის ფუნქციის შედეგის ტიპი იგივეა, რაც f–ის შედეგის ტიპი;

- ფუნქცია o -ს შედეგის ფუნქციის არგუმენტის ტიპი იგივეა, რაც g არგუმენტის ტიპი..

ამ მაგალითში ფუნქციის აღწერა შედარდა მის ტიპს, რომელიც განსაზღვრული იყო მოცემული ტიპიური გამოსახულებით. ტიპიური გამოსახულები არიან Objective CAML-ის სინტაქსური კონსტრუქციები. თუ ტიპიურ გამოსახულებაში არის ერთი ცვლადი მაინც, მაშინ ამ ტიპს ეწოდება პოლიმორფიზმი. სწორედ ამ ტიპის პოლიმორფიზმს უწოდებენ პარამეტრულს-მნიშვნელობის ტიპი შეიძლება პარამეტრიზებული იყოს მისი ქვემნიშვნელობებით. პოლომორფიზმი კი ობიექტზე ორიენტირებული დაპროგრამების ერთ-ერთი Pძირითადი მახასიათებელია.

პოლიმორფიზმი და ნიმუშთან შედარება საშუალებას იძლევა მკვთრად გაიზარდოს ენის შესაძლებლობები მონაცემთა სტრუქტურებითა და მათი დამუშავების პროცედურებით.

საზოგადოდ, ძნელია ამტკიცო, რომ ფუნქციონალური ენები ნებისმიერ სიტუაციაში საუკეთესო არჩევანია. მაგალითად, მათ რეკომენდაციას არ გაეწევა დრაივერების დასაწერად. საზოგადოდ ითვლება, რომ ფუნქციონალური ენები ნაკლებად ეფექტურია, ვიდრე C ან Fortran, მაგრამ ამ დროს არ ითვალისწინებენ, რომ გამომხატველობით ფუნქციონალური ენები აჭარბებს ობიექტურ-ორიენტირებულ ენებს, ხოლო პოლიმორფიზმს განზოგადოების ისეთივე ხარისხი აქვს, როგორც C++-ის შაბლონებს. იმპერატიულსა და ფუნქციონალურ ენებს შორის განსხვავებად კვლავ რჩება ცვლადების არ არსებობა, თუმცა უნდა აღინიშნოს, რომ Objective CAML არ წარმოადგენს სუფთა ფუნქციონალურ ენას. იმპერატიული თვისებები (ისეთები, როგორიცაა მინიჭება და ცვლადები) ენის სისრულეს ბევრს არაფერს მატებს, სამაგიეროდ ხშირ შემთხვევებში აადვილებს პროგრამის დაწერას.

და ბოლოს, ავღნიშნოთ, რომ მემკვიდრეობის მეშვეობით პოლიმორფიზმის ორგანიზაცია იწვევს დიდ დანახარჯებს. ჯერ ერთი, საჭიროა დაიწეროს დიდი მოცულობის პროგრამული კოდი: აღიწეროს კლასები, მემკვიდრეობითი კლასები, ვირტუალური ფუნქციები-წევრები. მეორეც, შედეგი მხოლოდ ვირტუალური ფუნქციებით მიიღწევა. სხვა სიტყვებით რომ ვთქვათ, სასარგებლოა ფუნქციონალურ ენაში მოვახდინოთ პოლიმორფიზმის მოდელირება.

მიგვაჩნია, რომ სწორედ ამ მიზეზების გამო თანამედროვე ფუნქციონალური ენები ვერ გვთავაზობს ბიბლიოთეკებისა და მზა ფუნქციების ისეთ რაოდენობას, რამდენსაც იმპერატიული და ობიექტზე ორიენტირებული. ამიტომაც, სამწუხაროდ, ფუნქციონალური ენები ვერ გახდნენ ძირითადი ინსტრუმენტი კომერციული პროგრამული უზრუნველყოფის შესაქმნელად.

ლიტერატურა

1. პარადიგმა. მასალა ენციკლოპედია ვიკიპედიიდან: http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F
2. http://www.literateprogramming.com/
3. П.Хендерсон. Функциональное программирование. Применение и реализация. Москва: “Мир”, 1983.
4. Хьювенен Э., Сеппанен Й. Мир Лиспа., т.1,2, Москва: “Мир”, 1990.
5. Лавров С., Силагадзе Г. Автоматическая обработка данных – Язык Лисп и его реализация. Наука, Москва,1978.
6. ნ. არჩვაძე, ლ. შეწირული. დაპროგრამების ენა LISP. გამ. “უნივერსალი”, 143 გვ., თბილისი, 2008 წ.
7. Д. Булычев. Снова о программировании. Открытые системы. http://www.osp.ru/os/2002/05/181454/_p1.html
8. დაპროგრამების ენა Objective Caml-is აღწერა: http://caml.inria.fr/pub/docs/oreilly-book/html/index.html
9. ნ. არჩვაძე, მ. ფხოველიშვილი, ლ. შეწირული. მონაცემების წარმოდგენა სიის სტრუქტურებით. მეცნიერება და ტექნოლოგიები, №7-9, 2008, გვ.18-24.
10. Кнут. Искусство программирования. Том 1.
11. Archvadze N., Pkhovelishvili M. The issue of universal programming. ”Science and Technologies”. №7-9, 2003, 49-52.
12. W.A.Woods and J.G. Schmolze. The KL-ONE Family. In: Semantic Networks in Artificial Intelligence. Ed: Fritz Lehmann, Special Issue of International Journal Computers & Mathematics with Applications. V. 23, N 2-5, January- March, 1992. Part 1.,p.133-178.
13. Вудс В.А. Сетевые грамматики для анализа естественных языков. // Кибернетический сборник. Новая Серия. Вып. 13. М.: Мир, 1978. С. 120-158.
14. Лисп-машина. материал из Википедии-свободной энциклопедии. http://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D1%81%D0%BF-%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0
15. Интернет университет информационных технологий. http://www.intuit.ru. Курсы: Введение в программирование на Лиспе, Основы функционального программирования.

სტატია მიღებულია: **2009-04-26**