ფუნქციონალური დაპროგრამება თანამედროვე ინფორმაციულ ტექნოლოგიებში

გიორგი დარჩიაშვილი, თალიკო ჟვანია
საქართველოს ტექნიკური უნივერსიტეტი

რეზიუმე

განხილულია ფუნქციონალური დაპროგრამების შესაძლებლობები და მისი გამოყენების ეფექტიანობა. მოყვანილია ფუნქციონალური დაპროგრამებისა და ობიექტზე ორიენტირებული დაპროგრამების შედარებითი ანალიზი. აღწერილია დაპროგრამების დეკლარატიული და იმპერატიული მიდგომები, უცვლელი მნიშვნელობები, პირველი კლასის მოქალაქე ფუნქციები, წმინდა ფუნქციები და ზარმაცი გამოთვლები. განხილულია თითოეული მათგანის უპირატესობები და ნაკლი.

საკვანძო სიტყვები: ფუნქციონალური დაპროგრამება. ობიექტზე ორიენტირებული დაპროგრამება.

1. შესავალი

ფუნქციონალური დაპროგრამება არის დაპროგრამების ერთ-ერთი პარადიგმა, რომლის რამდენიმე ძირითადი მახასიათებელია დეკლარატიულობა, უცვლელობა, პირველი კლასის მოქალაქე ფუნქციები, წმინდა ფუნქციები და სხვა.

დღევანდელი სტანდარტებით ფუნქციონალური დაპროგრამება სულაც არ იკავებს ძირითადი დანიშნულების ენის ადგილს. დღეს ეს ადგილი ობიექტზე ორიენტირებულ დაპროგრამებას ეკუთვნის. ფუნქციონალურს კი, ძირითადად, საკვლევ მონაცემთა (Data Science) პროექტებში იყენებენ. დღევანდელი სტანდარტებით ობიექტზე ორიენტირებული დაპროგრამება ყველაზე დახვეწილ პარადიგმად ითვლება ინფორმაციულ ტექნოლოგიებში, როგორც აკადემიურ, ისევე კორპორატიულ სფეროებში. ფუნქციონალური დაპროგრამება კი აღიქმება, როგორც უფრო ნაკლებად პოპულარული წინამორბედი თუ ალტერნატივა, რასთან შედარებასაც ობიექტზე ორიენტირებულის უპირატესობების წარმოსაჩენადაც იყენებენ.

საინტერესოა, თუ რას წარმოადგენს ფუნქციონალური დაპროგრამების რეალური მნიშვნელობა, როგორც აბსოლუტური, ასევე მის სხვა ალტერნატივებთან შედარებით და რამდენად შეესაბამება დღევანდელი პოზიციები სინამდვილეს.

2. ძირითადი ნაწილი

ფუნქცინალური დაპროგრამება თავისი მიდგომებით ბევრად გამოირჩევა სხვა პარადიგმებისგან. მისი ერთ-ერთი მთავარი, და, შესაძლოა, ითქვას, უმთავრესი მახასიათებელი, რომლიდანაც სხვა დანარჩენი გამომდინარეობს, ესაა მისი დეკლარატიულობა, ნაცვლად იმპერატიულობისა. პროგრამირებაში განსხვავდება დეკლარატიული და იმპერატიული მიდგომები.

იმპერატიული მიდგომა ესაა მიდგომა, სადაც ალგორითმის აღსაწერად გამოიყენება ინსტრუქციები. ენის ეს ელემენტები ატარებენ ბრძანებით ხასიათს, რაც პროცესორს

უბრძანებს, თუ რა გააკეთოს კონკრეტულად. მაგალითად, ეს შეიძლება იყოს "ამ ცვლადს მიანიჭე ეს მნიშვნელობა", "შეცვალე ამ ცვლადის მნიშვნელობა ამ მოცემულით", "გამოიტანე ეს ტექსტი ეკრანზე" და ა.შ... ეს ინსტრუქციები, როგორც წესი, არ არის დატვირთული ამოცანასთან დაკავშირებული სემანტიკით, იდეებით. პროგრამისტი კოდში იდეას კი არა, მხოლოდ მის რეალიზაციას გადმოსცემს, ანუ იმას, რასაც დაესმის კითხვა "როგორ?". იმპერატიულობა უფრო მეტად ობიექტზე ორიენტირებული და სხვა ენების მახასიათებელია. [1]

დეკლარატიული კოდი კი წარმოადგენს ამოცანის აღწერილობას. ენის მთავარი შემადგენელი ელემენტები მონაცემები და გამოსახულებებია. ისინი მნიშვნელოვნად არიან დატვირთული სემანტიკით და იდეებით. კოდის წერისას პროგრამისტი აღწერს ამოცანას და ამოხსნას, განსხვავებით ამოხსნის კონკრეტული ინსტრუქციებისა. კითხვა კი, რაც მას დაესმის, არის "რა?" [2, 3]

იმპერატიულთან შედარებით, დეკლარატიულ მიდგომას აქვს უამრავი უპირატესობა: ამოცანის ამონახსნი იწერება "პირდაპირ", ანუ კოდი ძალიან ახლოს არის ამოხსნის კონცეფციასთან, როცა იმპერატიულში ეს მანძილი ძალიან დიდია. ადვილია კოდის ცვლილება ამოცანის ცვლილების შესაბამისად, რადგან იდეურად ისინი ერთი და იგივეა, იმპერატიულში კი, ხანდახან საჭირო ხდება ძირეული ცვლილებები კოდში. დეკლარატიული კოდი ხშირად არის კომენტარის შემცველი და შემცვლელიც, რის გამოც ამ უკანასკნელის საჭიროება მცირდება. ასევე, ადვილია დეკლარატიული კოდის ანალიზი სპეციალური პროგრამული ბოტების გამოყენებით. ის ასევე იწერება გაცილებით მოკლედ, ლაკონურად და მარტივად, ვიდრე იმპერატიული. დეკლარატიული მიდგომის ნაკლი ისაა, რომ ის ხშირად უფრო ნელა და მძიმედ სრულდება, ვიდრე იმპერატიული. იმპერატიულ კოდს უფრო მეტი საერთო აქვს მანქანურ ენასთან, ამიტომაც ის უფრო ნაკლებ რესურსს მოიხმარს და უფრო სწრაფად სრულდება.

ფუნქციონალური დაპროგრამების საფირმო თვისება ისაა, რომ ის ფუნქციებს აღიქვამს ისევე, როგორც სხვა მონაცემებს (რიცხვებს, მასივებს, ობიექტებს და სხვას...). მოკლედ რომ ითქვას - ფუნქციებიც მონაცემებია. ეს გვაძლევს საშუალებას, რომ ფუნქციები შევადგინოთ ჰაერზე, როგორც სტატიკურად, ასევე დინამიურად, მათ შორის ლამბდა გამოსახულებებით თუ სხვადასხვა ფუნქციების კომბინირებით. ფუნქციონალური დაპროგრამების მოწინავე ენები აუცილებლად შეიცავენ ფუნქციებზე მანიპულაციის ისეთ საშუალებებს, როგორებიცაა არგუმენტების ნაწილობრივი გადაცემა (Partial Application), ფუნქციების კომბინაცია, მაღალი რიგის ფუნქციები (იგივე ფუნქციის პარამეტრად მიღება), გამოსახულების ფუნქციის არგუმენტად წინ გადაგდების ოპერატორი (Pipe forward, ანუ როდესაც ჯერ ფუნქციის არგუმენტი იწერება და შემდეგ ფუნქცია და სხვა... ფუნქციონალური ენები, როგორც წესი, იყენებენ მონაცემთა ძალიან მკაცრ ტიპიზაციას. იმდენად მკაცრს, რომ შესაძლებელი ხდება ტიპების ავტომატური დადგენა (Type inference) კოდში გამოყენებული გამოსახულებების მიხედვით. ხანდახან შეიძლება მთელი პროგრამის დაწერა ყველანაირი ტიპის ცხადად მითითების გარეშე, ან მხოლოდ რამდენიმე ადგილას მითითებით. ეს კოდს აცილებს დიდ ზედნადებს, რაც აადვილებს ფუნქციების ჰაერზე გამოცხადებას და დეკლარატიულ კოდს უფრო მკაფიოს, მსუბუქს და მოკლეს ხდის.

ფუნქციონალური დაპროგრამება რეკომენდაციას არ უწევს ცვლად მნიშვნელობებს. საქმე იმაშია, რომ სხვადასხვა დროს მისი მნიშვნელობა სხვადასხვა შეიძლება იყოს, რამაც ძალიან დიდი პრობლემა შეიძლება გამოიწვიოს, განსაკუთრებით მაშინ, როცა მასზე წვდომა ხდება სხვადასხვა კომპონენტების მიერ და განსაკუთრებით მაშინ, როცა ეს წვდომა ხდება პარალელურ რეჟიმში [1]. ერთი ცვლადით რამდენიმე კომპონენტი ხდება ერთმანეთში გადახლართული, რადგან ნებისმიერის მიერ მნიშვნელობის შეცვლამ, შეიძლება გავლენა მოახდინოს სხვა ნებისმიერის მუშაობაზე, რაც ხარისხიანი პროგრამული უზრუნველყოფის შექმნისთვის ფუნდამენტალური პრინციპის დარღვევაა. პრობლემა არ დადგებოდა, ეს ცვლადი, რომ ყოფილიყო მხოლოდ კითხვადი, რადგან კომპონენტები და პარალელური ნაკადები ერთმანეთზე გავლენას ვეღარ მოახდენდნენ. ცვლადების გამოყენება წარმოშობს კიდევ ერთ უდიდეს ნაკლს, რაც დეტერმინიზმის შესაძლო დარღვევაა. ფუნქციონალურ დაპროგრამებაში დიდი მნიშვნელობა ენიჭება დეტერმინიზმს, ანუ ფუნქციის თვისებას, მისი დაბრუნებული მნიშვნელობა იყოს ზუსტად დადგენადი მხოლოდ მისი არგუმენტების მნიშვნელობების მიხედ. ვითთუ ფუნქცია დამოკიდებულია რაიმე ცვლად მნიშვნელობაზე, რომელიც გარედანაც შეიძლება იცვლებოდეს, მაშინ ის ვეღარ იქნება დეტერმინირებული მხოლოდ არგუმენტების მნიშვნელობები მიხედვით [2, 3].

ცვლადების ნაცვლად აქტიურად გამოიყენება აღნიშვნები (let), რომელლებიც წარმოადგენენ შესრულების დროის კონსტანტებს და რომლებიც ფუნქციით იდენტური არიან readonly (C#-ში) და ფინალური (Java-ში) ცვლადებისა. ანუ ისეთი ცვლადების, რომლებსაც მნიშვნელობა ენიჭებათ მხოლოდ დასაწყისში და შემდეგ მათი შეცვლა შეუძლებელია. მეორეს მხრივ, ცვლადების ჩანაცვლება ხდება ფუნქციის პარამეტრებით და რეკურსიებით. ისეთ მოწინავე ენებში, როგორიცაა F#, ცვლადების გამოყენება დაშვებულია, თუმცა რეკომენდებული არაა. ისეთი ადგილებისთვის, სადაც ცვლადის გამოყენება ყველაზე კარგი გზაა, რეკომენდებულია ცალკე ფუნქციად ინკაფსულაცია და რეალიზაციის დეტალების მაქსიმალურად დამალვა. ძირითადად ასეთი ფუნქციები (სიის სორტირება...) უკვე გამზადებულია სტანდარტულ ბიბლიოთეკაში. ამ თვისებას უწოდებენ უცვლელობას - Immutability.

ბევრმა შეიძლება ვერ წარმოიდგინოს, თუ როგორ შეიძლება პროგრამირება ცვლადების გარეშე. თუმცა უცვლელობა გაცილებით მეტ პრობლემას აგვარებს, ვიდრე აჩენს. მათ შორისაა ძალიან დიდი სისტემური პრობლემები, თუნდაც პარალელური ნაკადების სინქრონიზაცია. ის ასევე გვაძლევს დეტერმინიზმის, ადვილი ტესტირების და დებაგირების საშუალებებს. ასევე მეხსიერების დაზოგვის საშუალებას, რადგან უცვლელობის შემთხვევაში ერთხელ შექმნილი ელემენტების ხელახლა გამოყენებას ხელს აღარაფერი უშლის. ეს ხდება იმავე პრინციპით, როგორც Java და C# ის String კლასში. ერთ-ერთი სიძნელე, რაც უცვლელობას შეუძლია ხანდახან გამოიწვიოს, ესაა რთული ობიექტების შექმნა, სადაც საფეხურობრივი ცვლილებებია საჭირო (Builder Pattern) [3].

ერთ-ერთი უმთავრესი ცნება, რაზეც ფუნქციონალური დაპროგრამებაა აგებული არის წმინდა ფუნქციები (Pure Functions). წმინდას უწოდებენ ისეთ ფუნქციას, რომელიც არ იწვევს გვერდით ეფექტებს (უფრო ზუსტად, ბიზნეს-ლოგიკისთვის ხილვად გვერდით

ეფექტებს), მაგალითად ის, რომ არ ცვლიან გლობალურ მნიშვნელობებს თავიანთ გარეთ. მის მიერ დაბრუნებული მნიშვნელობა დამოკიდებულია მხოლოდ არგუმენტების მნიშვნელობებზე, ანუ დეტერმინირებულია. ასეთ ფუნქციებს ძალიან დიდი უპირატესობა აქვთ დაპროგრამების სხვადასხვა ასპექტებში. პირველ რიგში, ასეთი ფუნქციები აკმაყოფილებენ მხოლოდ ერთი პასუხისმგებლობის პრინციპს (Single Responsibility Principle) და მაქსიმალურად განცალკევებულნი არიან სხვა კომპონენტებისგან, რის ხარჯზეც, უადვილესია მათი ტესტირება იზოლაციაში, ანუ Unit Test-ები. ასევე ადვილია ასეთი ფუნქციების კომბინაციით აწყობა, დებაგირება და, რაც ძალიან მნიშვნელოვანია, პარალელური შესრულება. გარდა ამ ყველაფრისა, შესაძლებელია წმინდა ფუნქციის არგუმენტთა ყოველი კონფიგურაციისთვის თითოჯერ გამოანგარიშების შემდეგ ამ მნიშვნელობების დამახსოვრება, ხოლო ამ არგუმენტებით ყოველი შემდგომი გამოძახებისას მნიშვნელობის პირდაპირ, იმავე გამოთვლების ხელახალი შესრულების გარეშე დაბრუნება, ანუ მემორიზაცია. მემორიზაციის შესაძლებლობას იძლევა მისი დეტერმინიზმი და გვერდითი ეფექტების არარსებობა.

წმინდა ფუნქციების სისუსტე ისაა, რომ შეუძლებელია ისეთი ოპერაციების შესრულება, რაც იწვევს გვერდით ეფექტებს, მაგალითად მონაცემების შეტანა გამოტანის ოპერაციები (IO – Input Output), რადგან გატანის ოპერაცია ბუნებრივად იწვევს გვერდით ეფექტს, ხოლო წაკითხვის ოპერაცია კი დეტერმინირებული არ არის. ასეთ ოპერაციებს ასევე მიეკუთვნება შემთხვევითი რიცხვის გენერირება იმიტომ, რომ არც ესაა დეტერმინირებული. ამ სისუსტის გამო შეუძლებელია რეალური საწარმოო პროგრამის მთლიანად წმინდა ფუნქციებით აწყობა, თუმცა მაქსიმალური ეფექტის მისაღწევად პროგრამისტებს შეუძლიათ აპლიკაციის მთავარი და ცენტრალურ ნაწილი დაწერონ წმინდად, ხოლო აუცილებელი არაწმინდა ნაწილი კი რაც შეიძლება კიდეებში გარიყონ, ანუ განალაგონ ბოლოებში. ანალოგიის გამო, ამას ხშირად სენდვიჩირებას ეძახიან. [3]

პროცესის დასაყოვნებლად ფუნქცინალურ დაპროგრამებაში აქტიურად გამოიყენება ზარმაცი გამოთვლები (Lazy Evaluation). ის ასევე გვაძლევს ისეთ, თითქოს შეზღუდული რესურსების მქონე მანქანისთვის წარმოუდგენელ ცნებებთან მუშაობის საშუალებას, როგორებიცაა უსასრულო მიმდევრობები.

ფუნქციონალური კოდის ობიექტზე-ორიენტირებულთან შედარებისთვის შეგვიძლია მოვიყვანოთ შემდეგი მაგალითი ფიბონაჩის მიმდევრობის პირველი 10 ელემენტის ჯამის გამოთვლაზე.

ობიექტზე-ორიენტირებული კოდი C# ენაზე:

```
// ცვლადების გამოცხადება ფიბონაჩის მიმდევრობის
// მომდევნო ორი ელემენტის აღსაწერად და მათი ინიციალიზაცია
// პირველი ელემენტის გამოსათვლელი მნიშვნელობებით
var p1 = 0L;
var p2 = 1L;

// ჯამის გამოსათვლელად განკუთვნილი აკუმულატორი ცვლადი
var sum = 0L;

// ციკლი ფიბონაჩის მიმდევრობის წევრების გამოსათვლელად
```

```
for (var i = 0 ; i < 10; i++)
{
    // შემდგომი ელემენტების გამოთვლა

    var c = p1 + p2;

    p2 = p1;
    p1 = c;

    // ჯამის აკუმულირება
    sum += c;
}

// შედეგის ეკრანზე გამოტანა
Console.WriteLine($"The sum of first ten Fibonacci numbers is: {sum}");
```

ფუნქციონალური კოდი F# ენაზე:

```
// ფუნქციის აღწერა, რომელიც ითვლის ფიბონაჩის მიმდევრობის
// შემდგომ ელემენტებს წინა ორი მომდევნო ელემენტის წყვილის მიხედვით
let nextFibonacci (p1, p2) = let c = p1 + p2 in c, (c, p1)

// ფიბონაჩის მიმდევრობის აღწერა გამოანგარიშების ფუნქციის და
// პირველი ელემენტისთვის საჭირო საწყისი მნიშვნელობების მიხედვით
let fibonacciSequence = Seq.unfold (Some << nextFibonacci) (0L, 1L)

// აღწერილ მიმდევრობაზე მუშაობა
// ამ მიმდევრობიდან პირველი 10 წევრის ჯამის დაბეჭდვა ეკრანზე
fibonacciSequence
|> Seq.take 10
|> Seq.sum
|> printfn "The sum of first ten Fibonacci numbers is: %d"
```

მაგალითიდან ჩანს ფუნქციონალური კოდის ზემოთ ჩამოთვლილი მახასიათებლები:

- დეკლარატიულობა - მოცემული ფუნქციონალური კოდი ატარებს უფრო აღწერით ხასიათს და უფრო ლაკონურია, განსხვავებით მოცემულ ობიექტზე ორიენტირებული კოდისა, რომელიც უფრო ბრძანებით ხასიათს ატარებს და სიტყვამრავალია.
- პირველი კლასის მოქალაქე ფუნქციები - მოცემულ ფუნქცინალურ კოდში ფუნქცია nextFibonacci ისევე ცხადდება, როგორც სხვა მონაცემები, პირდაპირ, ჰაერზე და ხდება ფინალურ ცვლადზე მინიჭება, ჩვეულებრივი მინიჭების ოპერატორის გამოყენებით.
- მკაცრი ტიპიზაცია - მოცემულ ფუნქციონალურ კოდში საერთოდ არ გვხვდება მონაცემთა ტიპების მითითებები, რადგან მკაცრი ტიპიზაციის საშუალებით, ამ შემთხვევაში, კომპილატორი ყველა ტიპის თავად დადგენას ახერხებს.
- უცვლელობა - მოცემულ ფუნქციონალურ კოდში საერთოდ არ გამოიყენება ცვლადები. მის ნაცვლად გვხვდება მხოლოდ აღნიშვნები (ფინალური ცვლადები).

- წმინდა ფუნქციები - კონკრეტულ შემთხვევაში ფუნქცია - nextFibonacci - არის წმინდა ფუნქცია. მისი დაბრუნებული მნიშვნელობა დამოკიდებულია მხოლოდ თავის არგუმენტებზე და არც გვერდით მოვნელებს იწვევს.
- ზარმაცი გამოთვლები - მიმდევრობის - fibonacciSequence - გამოცხადება არის ზარმაცი, ანუ გამოთვლების შესრულება საჭიროების დადგომამდე არ ხდება. ეს მიმდევრობა უსასრულო მიმდევრობაა. მის გამოყენებაში მითითებული აღწერა, რომელიც მის მხოლოდ პირველი 10 ელემენტს ღებულობს (Seq.take 10), იძლევა უსასრულო მიმდევრობის სასრულ ნაწილთან მუშაობის საშუალებას.
- გარდა ზემოთჩამოთვლი მახასიათებლებისა, ფუნქციონალურ კოდში დეკლარატიული გზით გამოცხადებული მიმდევრობა - fibonacciSequence - უნივერსალურია და შეიძლება მისი ხელახლა გამოყენება ამ ცნებასთან დაკავშირებულ სხვა ამოცანებშიც.

3. დასკვნა

ამგვარად, ფუნქციონალური დაპროგრამება მდიდარია ისეთი მიდგომებით, ცნებებით და ინსტრუმენტებით, რომ მისი გამოყენება ქმნის საკმაოდ დიდ ფუნდამენტს დღევანდელი პროგრამირების, როგორც საუკეთესო რეკომენდაციების დასაცავად, ასევე ყველაზე გავრცელებული პრობლემების აღმოსაფხვრელად.

არის შემთხვევა, სადაც ფუნქციონალური დაპროგრამება გახდა საიდუმლო იარაღი, რომლის საშუალებითაც დიდმა ორგანიზაციამ ტექნოლოგიური უპირატესობა მოიპოვა მის კონკურენტებზე.

პასუხი იმაზე, თუ რატომ არ ხდება ამ ენის ისევე ფართოდ გამოყენება, როგორიც, მაგალითად, ობიექტზე ორიენტირებულის, მიუხედავად გაკეთებული დასკვნისა, შეიძლება იყოს ის, რომ პირველი კომპიუტერების რესურსები ძალზე შეზღუდული იყო, ამიტომაც, იმ დროისთვის, რესურსების დაზოგვა დიდ საჭიროებას წარმოადგენდა. ამ საქმისთვის კი იმპერატიულ მიდგომას კონკურენტი არ ჰყავს. დღეს კომპიუტერული რესურსები - ოპერატიული მეხსიერება, პროცესორის სიხშირე და ბირთვების რაოდენობა - აღარ წარმოადგენს მწირ რესურსს, რაც ყველანაირად ამაღლებს ფუნქციონალური დაპროგრამების ეფექტურობას, თუმცა იმ საწყისი პერიოდიდან მოყოლებულმა ტენდენციამ იმპერატიული მიდგომებისკენ, შექმნა უფრო დიდი ბაზა - საზოგადოება, ბიბლიოთეკები, წიგნები, სასწავლო პროგრამები, ტექსტური რედაქტორები და ინტეგრირებული დაპროექტების გარემოები (IDE - Integrated Development Environment) - ამ უკანასკნელის სასარგებლოდ, რამაც გამოიწვია თოვლის გუნდის ეფექტი. ფუნქციონალური დაპროგრამების ძირითადი დანიშნულებით გამოყენება კი, ჯერჯერობით, ნელი ტემპით იზრდება.

ლიტერატურა - References - Литература:

1. Chris Smith. (2009). Programming F#: A comprehensive guide for writing simple code to solve complex problems (Animal Guide). Sebastopol, O'Reilly Media.
2. Scott Wlaschin. (2018). Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#. Raleigh, Pragmatic Bookshelf.
3. F# for fun and profit. https://fsharpforfunandprofit.com/

FUNCTIONAL PROGRAMMING IN THE MODERN INFORMATION TECHNOLOGY

Darchiashvili Giorgi, Zhvania Taliko
Georgian Technical University

Summary

The work contains a review of functional programming capabilities and efficiency of its use. The comparative analysis of functional programming and object oriented programming is given. Declarative and imperative approaches to programming, immutable values, functions as a first-class citizens, pure functions and lazy evaluation are described. The advantages and disadvantages of each one are discussed.

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ В СОВРЕМЕННОЙ ИНФОРМАЦИОННОЙ ТЕХНОЛОГИИ

Дарчиашвили Г., Жвания Т.
Грузинский Технический Университет

Резюме

Рассматриваются возможности функционального программирования и эффективность его применения. Приведен сравнительный анализ функционального и объектно-ориентированного программирования. Описаны декларативный и императивный подходы программирования, неизменяемые значения, функции первого класса, чистые функции и ленивые вычисления. Рассмотрены преимущества и недостатки каждого из них.