6.6 რეკომენდაციები რეკურსიის საკითხებზე

რეკურსიული ფუნქციების განსაზღვრა ველოსიპედის ტარებას გვაგონებს: გამოიყურება მარტივად, როცა ამას ვინმე სხვა აკეთებს, შეიძლება განუხორციელებელი მოგეჩვენოთ, თუ ამის გაკეთებას დამოუკიდებლად თავად მოინდომებთ პირველად, მაგრამ მარტივი და ბუნებრივი ხდება პრაქტიკის მიღების შემდეგ. ამ ნაწილში ჩვენ რეკომენდაციას ვიძლევით საერთოდ ფუნქციათა განსაზღვრის საკითხებზე და კერძოდ, რეკურსიულ ფუნქციათა აღწერის პრობლემებზე. ამისათვის მივმართავთ *ზუთეტაპიან პროცესს*, რომელსაც ავსახავთ *სამი კონკრეტული მაგალითის* საფუძველზე.

მაგალითი I - *product*

პირველი მარტივი მაგალითის სახით აღვწეროთ, როგორ ხდება ეტაპობრივად ამ თავში ადრე მოცემული *product* საბიბლიოთეკო ფუნქციის განსაზღვრა, რომელიც ანგარიშობს რიცხვთა სიის ნამრავლს.

ეტაპი 1: ტიპის აღწერა

დაფიქრება ტიპების თაობაზე შეიძლება ძალიან სასარგებლო აღმოჩნდეს ფუნქციათა განსაზღვრისას. ასე რომ ფუნქციის ტიპის განსაზღვრა საკუთრივ ფუნქციის განსაზღვრის დაწყებამდე – კარგი და მისასალმებელი ჩვევაა. განსახილველი მაგალითის შემთხვევაში ვიწყებთ ფუნქციის ფორმით:

product :: [*Int*] → *Int*

ამ სტრუქტურიდან ჩანს, რომ *product* ფუნქცია იღებს შესასვლელზე მთელი რიცხვების სიას და გამოაქვს გამოსასვლელზე განმხოლოებული მთელრიცხვა მნიშვნელობა. ამ მაგალითის მსგავსად, სადაც საუბარია *Int* ტიპზე, ხშირად სასარგებლოა განხილვის დაწყება სწორედ ასეთი მარტივი ტიპით, რომელიც მოგვიანებით შეიძლება დაზუსტდეს ან განზოგადდეს აუცილებლობის შემთხვევაში.

ეტაპი 2: შემთხვევათა ჩამოთვლა

არგუმენტთა ტიპების უმრავლესობისათვის არსებობს რიგი სტანდარტული შემთხვევა და მათი განხილვა აუცილებელია. სახელდობრ, სიებისათვის სტანდარტული შემთხვევებია ცარიელი სია და არაცარიელი სიები. ამიტომ შეგვიძლია ჩავწეროთ შემდეგი სქემატური განსაზღვრება შაბლონთან შედარების გამოყენებით:

product [] =
product (*n* : *ns*) =

არაუარყოფითი მთელი რიცხვებისათვის სტანდარტული შემთხვევებია 0 და *n* + 1, ლოგიკური მნიშვნელობებისათვის ასეთია *False* და *True* და ასე შემდეგ. მოგვიანებით, ტიპების მსგავსად, შეიძლება დაგვჭირდეს შემთხვევათა დაზუსტებაც, მაგრამ მათი განხილვა სასარგებლოა სტანდარტული შემთხვევებით დაიწყოს.

ეტაპი 3: მარტივ შემთხვევათა განსაზღვრა

მთელი რიცხვების ნამრავლი, როცა ამ რიცხვების რაოდენობა ნულს შეადგენს, იძლევა ერთს, ვინაიდან ერთიანი არ არღვევს გამრავლების ოპერაციაში გამოსახულების უცვლელობას. ამიტომ ცარიელი სიის შემთხვევაში ბუნებრივია განისაზღვროს, რომ:

product [] = 1
product (*n* : *ns*) =

ამ მაგალითის მსგავსად, მარტივი შემთხვევა ხშირად იძენს ძირითადი, საბაზო შემთხვევის მნიშვნელობას.

ეტაპი 4: სხვა შემთხვევათა განსაზღვრა

როგორ შეიძლება გამოვიანგარიშოთ მთელი რიცხვების არაცარიელი სიის ნამრავლი? ამ ეტაპზე სასარგებლო იქნება ჯერ განვიხილოთ შემადგენელი ნაწილები, რომლებიც შეიძლება იქნეს გამოყენებული. ასეთია, მაგალითად, საკუთრივ ფუნქცია (*product*), არგუმენტები (*n* და *ns*) და შესაბამისი ტიპის (+, −, ∗ და ა.შ.) საბიბლიოთეკო ფუნქციები. ამ შემთხვევაში ჩვენ მხოლოდ ვამრავლებთ პირველ მთელ რიცხვს და მთელი რიცხვების დარჩენილი სიის ნამრავლს:

product [] = 1
product (*n* : *ns*) = *n* ∗ *product ns*

ამ მაგალითის მსგავსად, სხვა შემთხვევები ხშირად იძენს რეკურსიულ ხასიათს.

ეტაპი 5: განზოგადება და გამარტივება

მას შემდეგ, რაც ფუნქცია განისაზღვრება ზემოთ აღწერილი პროცესის გამოყენებით, ხშირად ნათელი ხდება, რომ იგი შეიძლება განზოგადდეს ან გამარტივდეს. მაგალითად, *product* ფუნქცია დამოკიდებული არ არის იმ რიცხვთა ზუსტ სახეზე, რომელთა მიმართ იგი გამოიყენება. ამიტომ მისი ტიპი შეიძლება განზოგადდეს და მთელი რიცხვებიდან სავსებით დასაშვებია ნებისმიერ რიცხვით ტიპზე გადასვლა:

product :: *Num a* ⇒ [a] → *a*

რაც შეეხება გამარტივებას, მე-7 თავში ვნახავთ, რომ *product* ფუნქციაში გამოყენებული რეკურსიის შაბლონი ინკაფსულირებულია (ე.ი. კიდევ რაღაცას შეიცავს საკუთარ თავში) *foldr* დასახელების საბიბლიოთეკო ფუნქციით, რომელსაც *product* ფუნქცია იყენებს. ამიტომ *product* ფუნქცია შეიძლება ხელახლა განისაზღვროს ერთი განტოლებით

product = *foldr* (∗) 1

ამრიგად, საბოლოო განსაზღვრებას *product* ფუნქციისათვის შემდეგი სახე აქვს:

product :: *Num a* ⇒ [*a*] → *a*
product = *foldr* (∗) 1

ეს არის *product* ფუნქციის ზუსტი განმარტება A დანართში მოცემული სტანდარტული prelude ფაილიდან, იმის გამოკლებით, რომ ეფექტურობის მოსაზრებით *foldr* ფუნქციის გამოყენება იქ ჩანაცვლებულია მონათესავე *foldl* საბიბლიოთეკო ფუნქციით, რომელიც ასევე მე-7 თავშია განხილული.

მაგალითი II - *drop*

ახლა უფრო არსებით მაგალითზე ვუჩვენოთ, როგორ შეიძლება ხუთეტაპიანი პროცესის გამოყენებით ავაგოთ საბიბლიოთეკო *drop* ფუნქციისათვის ადრე მოცემული განსაზღვრება. როგორც ვიცით, ეს ფუნქცია ანადგურებს სიაში ელემენტების მოცემულ რაოდენობას (ამ სიის დასაწყისიდან).

ეტაპი 1: ტიპის აღწერა

დავიწყოთ ტიპით, რომელიც გვეუბნება, რომ *drop* ფუნქცია იღებს შესასვლელზე მთელ რიცხვსა და გარკვეული *a* ტიპის მნიშვნელობათა სიას, ხოლო გამოსასვლელზე ამავე ტიპის სიდიდეთა ახალი სიის ფორმირებას ახორციელებს:

drop :: $Int \to [a] \to [a]$

ყურადღება მიაქციეთ, რომ ამ ტიპის განსაზღვრებას ოთხი თავისებურება ახასიათებს, სახელდობრ: 1. პირველ არგუმენტად გამოყენებულია მთელი რიცხვი და არა უფრო ზოგადი რიცხვითი ტიპი, რაც სიმარტივის უზრუნველსაყოფად კეთდება; 2. ნაცვლად იმისა, რომ ფუნქცია შესასვლელზე იღებდეს თავის ორ არგუმენტს წყვილის სახით, მეტი მოქნილობისათვის შექმნილია კარინგის გამოყენების შესაძლებლობა (იხ. პუნქტი 3.6); 3. მთელრიცხვა არგუმენტი განთავსებულია მეორე არგუმენტამდე, რომელსაც ელემენტთა სიის სახე აქვს, რაც წაკითხვის მოხერხებულობას ემსახურება (*drop n xs* გამოსახულება შეიძლება ასე იკითხებოდეს, მაგალითად: «*n* ელემენტის ამოგდება *xs* სიიდან»); 4. იქმნება ფუნქცია, რომელიც პოლიმორფულია ელემენტთა სიის ტიპით (გამოყენების უნივერსალობის უზრუნველსაყოფად).

ეტაპი 2: შემთხვევათა ჩამოთვლა

ვინაიდან მთელრიცხვა არგუმენტისათვის ორი (0 და $n + 1$) სტანდარტული შემთხვევა არსებობს, ხოლო არგუმენტების სიისათვის ასევე ორ ([] და $x : xs$) განსაკუთრებულ მნიშვნელობასთან გვაქვს საქმე, ქვემოთ მოცემული სქემატური ჩანაწერი განსაზღვრებისათვის საერთო ჯამში ოთხ გამოსახულებას მოითხოვს:

drop 0 [] =
drop 0 (x : xs) =
drop (n + 1) [] =
drop (n + 1) (x : xs) =

ეტაპი 3: მარტივ შემთხვევათა განსაზღვრა

ლოგიკურად, ნებისმიერი სიის დასაწყისიდან მისი ნული ელემენტის ამოგდება იმავე სიას იძლევა. ასე რომ პირველი ორი შემთხვევის განსაზღვრა უშუალოდ ხდება:

drop 0 [] = []
drop 0 (x : xs) = x : xs
drop (n + 1) [] =
drop (n + 1) (x : xs) =

ცარიელი სიიდან ერთი ან რამდენიმე ელემენტის ამოგდების მცდელობა დაუშვებელია. ამიტომ მესამე შემთხვევა უნდა იქნეს გამოტოვებული. მაგრამ ეს გამოიწვევს შეცდომის გაჩენას, თუ ასეთი სიტუაცია წარმოიქმნება. მაგრამ პრაქტიკაში მიიღება გადაწყვეტილება შეცდომის გაჩენის ასაცილებლად, რისთვისაც ამ შემთხვევაში ცარიელი სიის დაბრუნება ხდება:

drop 0 [] = []
drop 0 (x : xs) = x : xs
drop (n + 1) [] = []
drop (n + 1) (x : xs) =

ეტაპი 4: სხვა შემთხვევათა განსაზღვრა

როგორ ამოვაგდოთ ერთი ან რამდენიმე ელემენტი არაცარიელი სიიდან? ერთით ნაკლები რაოდენობის ელემენტის მარტივი ამოღებით სიის კუდიდან:

```
drop 0 []          =    []
drop 0 (x : xs)    =    x : xs
drop (n + 1) []    =    []
drop (n + 1) (x : xs)    =    drop n xs
```

ეტაპი 5: განზოგადება და გამარტივება

drop ფუნქცია დამოკიდებული არ არის იმ მთელი რიცხვის ზუსტ სახეზე, რომელსაც იგი პირველ არგუმენტად იღებს. ამიტომ ხსენებული რიცხვის ტიპი შეიძლება განზოგადდეს და ნებისმიერ *Integral* მთელრიცხვა ტიპად გამოცხადდეს, რომლის სტანდარტულ ეგზემპლარებს[1] *Int* და *Integer* ტიპები წარმოადგენს:

```
drop    ::    Integral b ⇒ b → [a] → [a]
```

მაგრამ ეფექტურობის მოსაზრებებით ასეთი განზოგადება ფაქტობრივად არ ხდება სტანდარტულ prelude ფაილში, როგორც ეს აღინიშნებოდა პუნქტში 3.9. გამარტივების თვალსაზრისით კი პირველი ორი განტოლება *drop* ფუნქციისათვის შეიძლება ერთ განტოლებად ჩაიწეროს, რომელიც გვეუბნება, რომ ნებისმიერი სიიდან ნულოვანი რაოდენობის ელემენტთა ამოგდება იმავე სიას იძლევა:

```
drop 0 xs          =    xs
drop (n + 1) []    =    []
drop (n + 1) (x : xs)    =    drop n xs
```

გარდა ამისა, *x* ცვლადი უკანასკნელ განტოლებაში შეიძლება შევცვალოთ ჩასმის შაბლონით, ვინაიდან ეს ცვლადი განტოლების ტანში არ გამოიყენება:

```
drop 0 xs          =    xs
drop (n + 1) []    =    []
drop (n + 1) ( _ : xs)    =    drop n xs
```

ამის მსგავსად, ჩნდება მოსაზრება, რომ მეორე განტოლებაში *n* სიდიდე უნდა იყოს ჩანაცვლებული (_) ჩანაწერით, მაგრამ ეს დაუშვებელ და მცდარ სახეს აძლევს განსაზღვრებას, ვინაიდან *n + k* ფორმის შაბლონები მოითხოვს, რომ *n* სიდიდე წარმოადგენდეს ცვლადს. ამ შეზღუდვის თავიდან აცილება შესაძლებელი იქნება, თუ მეორე განტოლებაში მთლიანად ჩავანაცვლებთ *n* + 1 შაბლონს (_) ჩანაწერით. მაგრამ ეს ფუნქცის ქცევასაც შეცვლის. მაგალითად, *drop* (−1) [] გამოსახულების გამოთვლა ამ დროს ცარიელი სიის შემთხვევაშიც მოხდება, მაშინ როცა ახლა ეს შეცდომას იწვევს, ვინაიდან (_) ჩანაწერთან შესაბამისობაში ნებისმიერი მთელი რიცხვის მოყვანა შეიძლება, ხოლო (*n* + 1) ჩანაწერს მხოლოდ ისეთი მთელი რიცხვები შეესაბამება, რომლებიც ერთზე ნაკლები არ არის (≥ 1).

დასასრულ, საბოლოო განსაზღვრება *drop* ფუნქციისათვის სწორედ იმ სახეს იღებს, რომელიც მას A დანართის prelude სტანდარტულ ფაილში აქვს:

---

[1] კლასის ეგზემპლარი (ინგლ. instance) – კონკრეტული ობიექტი მოცემული კლასის ობიექტების სიმრავლიდან. სისტემაში, ჩვეულებრივ, სხვადასხვა კლასის მრავალი ეგზემპლარი ფუნქციონირებს. ერთი კლასის ყველა ეგზემპლარს ოპერაციათა ერთნაირი ნაკრები აქვს.

```
drop                    ::      Int → [a] → [a]
drop 0 xs               =       xs
drop (n + 1) []         =       []
drop (n + 1) ( _ : xs)  =       drop n xs
```

მაგალითი III - *init*

დასკვნითი მაგალითის სახით ვნახოთ განსაზღვრების აგება ხუთეტაპიან პროცესში *init* საბიბლიოთეკო ფუნქციისათვის, რომელიც არაცარიელი სიის უკანასკნელ ელემენტს ანადგურებს.

ეტაპი 1: ტიპის აღწერა

დავიწყოთ ტიპის აღწერით, რომელიც გვეუბნება, რომ *init* ფუნქცია იღებს შესასვლელზე გარკვეული ტიპის მნიშვნელობათა სიას და აგებს ასეთივე მნიშვნელობების სხვა სიას:

```
init    ::      [a] → [a]
```

ეტაპი 2: შემთხვევათა ჩამოთვლა

ვინაიდან ცარიელი სია არ არის დასაშვები არგუმენტი *init* ფუნქციისათვის, განსაზღვრების ქვემოთ ნაჩვენები სქემატური ჩანაწერი შაბლონთან შედარების გამოყენებით მხოლოდ ერთი შემთხვევის მითითებას მოითხოვს:

```
init (x : xs)       =
```

ეტაპი 3: მარტივ შემთხვევათა განსაზღვრა

მაშინ, როცა ორ წინა მაგალითში ეს ეტაპი სავსებით ცხადი იყო, *init* ფუნქციის შემთხვევაში ოდნავ მეტი დაფიქრება დაგვჭირდება. ლოგიკურად, უკანასკნელი ელემენტის ამოღება ერთელემენტიანი სიიდან ცარიელ სიას იძლევა. ამიტომ შეგვიძლია მცველის შემოტანა ამ მარტივი შემთხვევისათვის:

```
init (x : xs) | null xs       =       []
              | otherwise     =
```

გავიხსენოთ, რომ საბიბლიოთეკო *null* ფუნქცია ადგენს ცარიელია სია, თუ არა.

ეტაპი 4: სხვა შემთხვევათა განსაზღვრა

როგორ შეიძლება უკანასკნელი ელემენტის ამოღება სიიდან, რომელშიც სულ ცოტა ორი ელემენტი მაინც არის? თავის მარტივი შენარჩუნებით და უკანასკნელი ელემენტის ამოღებით კუდიდან:

```
init (x : xs) | null xs       =       []
              | otherwise     =       x : init xs
```

ეტაპი 5: განზოგადება და გამარტივება

init ფუნქციისათვის ტიპი უკვე იმდენად ზოგადია, რამდენადაც ეს შესაძლებელია, მაგრამ მიუხედავად ამისა, თავად განსაზღვრება შეიძლება გამარტივდეს შაბლონთან შედარებით მცველების ნაცვლად და პირველ განტოლებაში ჩასმის შაბლონის გამოყენებით და არა ცვლადის:

```
init            ::      [a] → [a]
init [ _ ]      =       []
init (x : xs)   =       x : init xs
```

ამ შემთხვევაშიც სწორედ ასეთია *init* ფუნქციის განსაზღვრება prelude სტანდარტულ ფაილში.