2. მონაცემთა სტრუქტურები და ბაზური ოპერაციები

როგორც უკვე ავღნიშნეთ, პროგრამირების ფუნქციონალური პარადიგმის საფუძველს წარმოადგენს მათემატიკური აზროვნების განვითარების ისეთი მიმართულებები, როგორიცაა კომბინატორული ლოგიკა და λ–აღრიცხვა. ეს უკანასკნელი უფრო მჭიდროდ არის დაკავშირებული ფუნქციონალურ პროგრამირებასთან. სწორედ λ–აღრიცხვა არის ფუნქციონალური პროგრამირების თეორიული საფუძველი.

იმისათვის, რომ განვიხილოთ ფუნქციონალური პროგრამირების თეორიული საფუძვლები, პირველ რიგში აუცილებელია შემოვიტანოთ ზოგიერთი შეთანხმება, ავღწეროთ აღნიშვნები და ავაგოთ ფორმალური სისტემა.

ვთქვათ, მოცემულია რომელიღაც `A` პირველადი ტიპის ობიექტები. ამჟამად არ აქვს მნიშვნელობა, თუ კონკრეტულად რას წარმოადგენს გამოყოფილი ობიექტები. საზოგადოდ, ითვლება, რომ ამ ობიექტებზე განისაზღვრება ბაზისური ოპერაციების და პრედიკატების ერთობლიობა. ტრადიციულად, ეს მოდის მაკარტიდან (`Lisp`–ის ავტორისგან), რომ ობიექტებს უწოდებენ ატომებს. თეორიულად, არანაირი მნიშვნელობა არ აქვს ბაზისური ოპერაციებისა და პრედიკატების რეალიზების საშუალებებს, უბრალოდ ხდება მათი პოსტილურება. თუმცა, ყოველი ფუნქციონალური ენა თავისებურად რეალიზებს ბაზისურ ნაკრებს.

ტრადიციულად (და, უპირველეს ყოვლისა, ეს აიხსნება თეორიული აუცილებლობით) ბაზისურ ოპერაციად გამოიყოფა შემდეგი სამი ოპერაცია:

1. წყვილის შექმნის ოპერაცია – **prefix** `(x, y) = x : y = [x | y]`. მას ხშირად უწოდებენ კონსტრუქტორს ანუ შემდგენელს.

2. თავის გამოყოფის ოპერაცია – **head** `(x) = h (x)`. ეს არის პირველი სელექტორული ოპერაცია.

3. კუდის გამოყოფის ოპერაცია – **tail** `(x) = t (x)`. ეს არის მეორე სელექტორული ოპერაცია.

თავისა და კუდის გამოყოფის სელექტორულ ოპერაციებს ხშირად უწოდებენ უბრალოდ სელექტორებს. ეს ოპერაციები დაკავშირებულია ერთმანეთთან შემდეგი სამი აქსიომით:

1. **head** `(x : y) = x`
2. **tail** `(x : y) = y`
3. **prefix** (**head** `(x : y)`, **tail** `(x : y)) = (x : y)`

ყველა ობიექტის სიმრავლე, რომელიც შეიძლება კონსტრუირდეს პირველადი ტიპის ობიექტებისგან ბაზისური ოპერაციების გამოყენების შედეგად, უწოდებენ `S` გამოსახულებას (აღნიშვნა – `Sexpr (A)`). მაგალითად:

`a1 : (a2 : a3) in Sexpr`

შემდგომი კვლევისთვის შემოდის ცნება ფიქსირებული ატომი, რომელიც აგრეთვე ეკუთვნის პირველად `A` ტიპს. ამ ატომს შემდგომში ვუწოდებთ „ცარიელ სიას“ და ავღნიშნავთ სიმბოლოებით `[]` (თუმცა, ფუნქციონალური პროგრამირების სხვადასხვა ენაში შეიძლება გამოყენებული იყოს ცარიელი სიის სხვა აღნიშვნებიც). ეხლა ავღწეროთ ის, რაზეც ოპერირებს ფუნქციონალური პროგრამირება – `List (A) Sexpr (A)`, საკუთრივი ქვესიმრავლე, რომელსაც ეწოდება „სია `A`–ზე“.

განმარტება:

`1°. ცარიელი სია [] in List (A)`

`2°. x in A & y in List (A) => x : y in List (A)`

სიის მთავარი თვისება არის: `x in List (A) & x =/= [] =>` **`head`** `(x) in A;` **`tail`** `(x) in List (A).`

`n` ელემენტიანი სიის აღსანიშნავად შეიძლება გამოყენებული იყოს სხვადასხვა ნოტაცია, თუმცა ჩვენ გამოვიყენებთ მხოლოდ ასეთს: `[a1, a2, ..., an].` სიასთან ოპერაციების `head`–ის და `tail`–ის მეშვეობით შეიძლება სიის თითოეულ ელემენტთან წვდომა, რადგანაც:

`head` `([a1, a2, ..., an]) = a1`

`tail` `([a1, a2, ..., an]) = [a2, ..., an] (როცა n > 0).`

სიების გარდა შემოდის მონაცემების კიდევ ერთი ტიპი, რომელსაც ეწოდება „სიური სტრუქტურა `A`–ზე“ (აღნიშვნა – `List_str (A)`), ამასთან, შესაძლოა აგებული იყოს დამოკიდებულების შემდეგი სტრუქტურა: `List (A) List_str (A) Sexpr (A).` სიური სტრუქტურის განმარტებას აქვს შემდეგი სახე:

განმარტება:

`1°. a in A => a in List_str (A)`

`2°. List (List_str (A)) in List_str (A)`

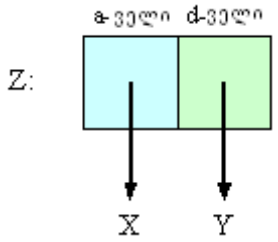
ანუ, ჩანს, რომ სიური სტრუქტურა – ეს არის სია, რომლის ელემენტები შეიძლება იყოს როგორც ატომები, ასევე სხვა სიური სტრუქტურები, მათ შორის ჩვეულებრივი სიები. სიური სტრუქტურის მაგალითი, რომელიც ამავე დროს არ არის მარტივი სია, არის შემდეგი გამოსახულება: `[a1, [a2, a3, [a4]], a5].` სიური სტრუქტურისთვის შემოდის ისეთი ცნება, როგორიცაა ჩადგმის დონე.

რამდენიმე სიტყვა პროგრამულ რეალიზაციაზე

განვიხილოთ სიებისა და სიური სტრუქტურების პროგრამული რეალიზაციები. ეს საჭიროა იმისთვის, რომ გავიგოთ, რა ხდება ფუნქციონალური პროგრამის

მუშაობისას როგორც რომელიმე კონკრეტულ ფუნქციონალურ ენაზე, ასევე აბსტრაქტულ ენაზე.

თითოეული ობიექტი მანქანის მეხსიერებაში იკავებს რაღაც ადგილს. ატომები წარმოადგენს მიმთითებლებს (მისამართებს) უჯრედებზე, რომლებშიც ობიექტი ინახება. ასეთ შემთხვევაში წყვილი `z = x : y` გრაფიკულად შეიძლება წარმოვადგინოთ, როგორც ნაჩვენებია შემდეგ სურათი 1-ზე:

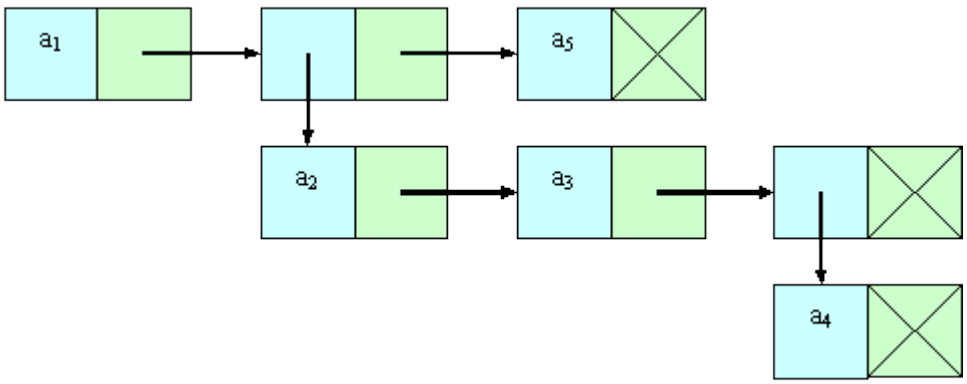


სურათი 1. წყვილის წარმოდგენა კომპიუტერის მეხსიერებაში

უჯრედის მისამართი, რომელიც შეიცავს მიმთითებლებს `x`–ზე და `y`–ზე, არის ობიექტი `z`. როგორც ნახატიდან ჩანს, წყვილი წარმოდგენილია ორი მისამართით – მიმთითებლით თავზე და კუდზე. ტრადიციულად პირველ მიმთითებელს უწოდებენ `a`-ველს, მეორე მიმთითებელს – `d`-ველს.

ობიექტები, რომელზეც `a`-ველი და `d`-ველი მიუთითებს, რომ უფრო მოსახერხებლად წარმოვადგინოთ, შემდგომში მათ ჩავწერთ უშუალოდ ველებში. ცარიელ სიას ავღნიშნავთ გადახაზული კვადრატით.

ამრიგად, სიური სტრუქტურა `[a1, [a2, a3, [a4]], a5]` შეიძლება წარმოდგეს შემდეგნაირად (სურათი 2):



სურათი 2. `[a1, [a2, a3, [a4]], a5]` გრაფიკული წარმოდგენა სიური სტრუქტურის

სურათზე კარგად ჩანს ჩადგმის დონეები – `a1` და `a5` ატომებს აქვთ ჩადგმის დონე 1, ხოლო ატომებს `a2` და `a3` – 2, ხოლო ატომს `a4` – 3 შესაბამისად.

ავღნიშნოთ, რომ ოპერაცია prefix ითხოვს მეხსიერებას, ვინაიდან წყვილის კონსტუირების დროს გამოიყოფა მეხსიერება მიმთითებლებისთვის. მეორეს მხრივ, ოპერაციებს head და tail არ სჭირდება მეხსიერება, ისინი უბრალოდ აბრუნებენ მისამართებს, რომლებიც შეიცავს შესაბამისად a-ველს და d-ველს.

მაგალითები

მაგალითი 5. ოპერაცია prefix.

თავდაპირველად უფრო დატალურად განვიხილოთ ოპერაცია prefix–ის მუშაობა. ოპერატორის მუშაობა განვიხილოთ სამ ზოგად მაგალითზე:

1°. prefix (a1, a2) = a1 : a2 (ამასთან, შედეგი არ არის List_str (A)–ის ელემენტი).

2°. prefix (a1, [b1, b2]) = [a1, b1, b2]

3°. prefix ([a1, a2], [b1, b2]) = [[a1, a2], b1, b2]

მაგალითი 6. სიის სიგრძის განსაზღვრის ფუნქცია Length.

მანამ, სანამ დავიწყებდებდეთ უშუალოდ ფუნქცია Length–ის რეალიზებას, ვნახოთ, თუ რას აბრუნებს იგი. ფუნქცია Length–ის შედეგი არის ელემენტების რაოდენობა იმ სიაში, რომელიც გადაეცემა მას პარამეტრად. აქ ორი შემთხვევაა– ფუნქციას გადაეცა ცარიელი სია და ფუნქციას გადაეცა არაცარიელი სია. პირველ შემთხვევაში ცხადია შედეგი უნდა იყოს 0. მეორე შემთხვევაში ამოცანა ორ ქვეამოცანად იყოფა, სია იყოფა თავად და კუდად ოპერაციების head–ის და tail–ის საშუალებით.

ვიცით, რომ head აბრუნებს სიის პირველ ელემენტს, ხოლო ოპერაცია tail აბრუნებს დანარჩენი ელემენტების სიას. თუ გვეცოდინება რისი ტოლია tail ოპერაციით მიღებული სიის სიგრძე, მაშინ თავდაპირველი სიის სიგრძე იქნება ეს სიგრძე ერთით გადიდებული. ეხლა უკვე ადვილად შეგვიძლია დავწეროთ ფუნქცია Length–ის განმარტება:

```
Length ([]) = 0
Length (L) = 1 + Length (tail (L))
```

მაგალითი7. ორი სიის შერწყმის ფუნქცია Append.

ორი სიის შერწყმა (ანუ გაერთიანება) შეიძლება რამდენიმე საშუალებით. პირველი –დესტრუქციული მინიჭება, ანუ შევცვალოთ [] სიაზე მინიჭება მეორე სიის თავზე მიმთითებლით და ამით მივიღებთ შედეგს პირველ სიაში. მაგრამ ამ დროს

იცვლება პირველი სია. ასეთი მიდგომები ფუნქციონალურ პროგრამირებაში დაუშვებელია (თუმცა ზოგიერთ ენაში ეს დასაშვებია).

მეორე მიდგომა მდგომარეობს შემდეგში: მოვახდინოთ პირველი სიის კოპირება ზედა დონეზე და მოვათავსოთ კოპიოს ბოლო მიმთითებლის ნაცვლად მიმთითებელი მეორე სიის პირველ ელემენტზე. ეს მიდგომა კარგია იმით, რომ არ ასრულებს დესტრუქციულ მოქმედებას და არ აქვს გვერდითი ეფექტები, თუმცა მოითხოვს დამატებით მეხსიერებასა და დროს.

```
Append ([], L2) = L2
Append (L1, L2) = prefix (head (L1), Append (tail (L1), L2))
```

ბოლო მაგალითი გვიჩვენებს, თუ როგორ შეიძლება თანდათანობითი კონსტრუირებით აიგოს ახალი სია, რომელიც იქნება ორი მოცემულის შერწყმა.

სავარჯიშოები

1. ააგეთ ფუნქცია, რომელიც გამოითვლის შემდეგი მიმდევრობების n-ურ წევრს:

a. $a_n = x_n$

b. $a_n = \text{Summ } i, \ (i = 1,n)$

c. $a_n = \text{Summ } (\text{Summ } i), \ (j = 1,n \ i = 1,j)$

d. $a_n = \text{Summ } n^{-i}, \ (i = 1,p)$

e. $a_n = e^n = \text{Summ } (n^i / i!), \ (i = 0,\text{infinity})$

2. ახსენით prefix ოპერაციის შედეგი, რომელიც მოყვანილია მაგალითად 5-ში. ახსნისას შეგიძლიათ გამოიყენოთ გრაფიკული მეთოდი.

3. ახსენით ფუნქცია Append-ის შედეგი (მაგალითი 7). ახსენით, რატომ არ არის ფუნქცია დესტრუქციული.

4. ააგეთ ფუნქცია, რომელიც მუშაობს სიებთან:

a. GetN — ფუნქცია, რომელიც მოცემული სიიდან n-ურ ელემენტს გამოყოფს.

b. ListSumm — ორი სიის ელემენტების შეკრება. აბრუნებს სიას, რომელიც არის პარამეტრი სიების ელემენტების ჯამი. გაითვალისწინეთ, რომ სიების სიგრძე შეიძლება იყოს სხვადასხვა.

c. OddEven — ფუნქცია უცვლის ადგილებს მეზობელ ლუწ და კენტ ელემენტებს მოცემულ სიაში.

d. Reverse — ფუნქცია, რომელიც აბრუნებს სიას (სიის პირველი ელემენტი ხდება ბოლო, მეორე–ბოლოდან მეორე და ა.შ. ბოლო ელემენტამდე).

e. Map — ფუნქცია, რომელიც იყენებს მეორე ფუნქციას (რომელიც პარამეტრად გადაეცემა თავდაპირველ ფუნქციას) მოცემული სიის ყველა ელემენტთან.

პასუხები თვითშემოწმებისთვის

პასუხები ხშირად წარმოდგენილი გვაქვს რამდენიმე შესაძლო ვარიანტიდან ერთ–ერთი.

1. ფუნქციები, რომლებიც ითვლის მიმდევრობის N–ურ წევრს:

a. Power:

```
Power (X, 0) = 1
Power (X, N) = X * Power (X, N - 1)
```

b. Summ_T:

```
Summ_T (1) = 1
Summ_T (N) = N + Summ_T (N - 1)
```

c. Summ_P:

```
Summ_P (1) = 1
Summ_P (N) = Summ_T (N) + Summ_P (N - 1)
```

d. Summ_Power:

```
Summ_Power (N, 0) = 1
Summ_Power (N, P) = (1 / Power (N, P)) + Summ_Power (N, P - 1)
```

e. Exponent:

```
Exponent (N, 0) = 1
Exponent (N, P) = (Power (N, P) / Factorial (P)) + Exponent (N, P - 1)

Factorial (0) = 1
Factorial (N) = N * Factorial (N - 1)
```

2. ოპერაცია prefix–ის მუშაობის მაგალითი შეიძლება წარმოვადგინოთ სამი მიდგომით (ისევე, როგორც ეს განხილულია მაგალითში). შესაძლებელია ოპერაცია prefix–ის წარმოდგენა ინფიქსური ჩანაწერის, სიმბოლო :–ის გამოყენებით.

a. ოპერაციის მუშაობის პირველი მაგალითია თვით ოპერაციის განმარტება. მის განხილვას არ აქვს აზრი, რადგან prefix სწორედ ასე განიმარტება.

b. prefix (a1, [b1, b2]) = prefix (a1, b1 : (b2 : [])) = a1 : (b1 : (b2 : [])) = [a1, b1, b2]

(ეს გარდაქმნები მოყვანილია სიის განმარტების მიხედვით).

c. prefix ([a1, a2], [b1, b2]) = prefix ([a1, a2], b1 : (b2 : [])) = ([a1, a2]) : (b1 : (b2 : [])) = [[a1, a2], b1, b2].

3. Append ფუნქციის მუშაობის მაგალითად განვიხილოთ ორი სიის შერწყმის მაგალითი, რომლიდანაც თითოეული შედგება ორი ელემენტისგან: [a, b] და [c, d]. რომ არ გადავტვირთოთ ახსნით, ოპერაცია prefix–სთვის გამოვიყენოთ ინფიქსური ფორმის ჩანაწერი:

Append ([a, b], [c, d]) = a : Append ([b], [c, d]) = a : (b : Append ([], [c, d])) = a : (b : ([c, d])) = a : (b : (c : (d : []))) = [a, b, c, d].

4. ფუნქციები, რომლებიც მუშაობენ სიებთან:

a. GetN:

```
GetN (N, []) = _
GetN (1, H:T) = H
GetN (N, H:T) = GetN (N - 1, T)
```

b. ListSumm:

```
ListSumm ([], L) = L
ListSumm (L, []) = L
ListSumm (H1:T1, H2:T2) = prefix ((H1 + H2), ListSumm (T1, T2))
```

c. OddEven:

```
OddEven ([]) = []
OddEven ([X]) = [X]
OddEven (H1:[H2:T]) = prefix (prefix (H2, H1), OddEven (T))
```

d. Reverse:

```
Reverse ([]) = []
```

```
Reverse (H:T) = Append (Reverse (T), [H])
```

e. Map:

```
Map (F, []) = []
Map (F, H:T) = prefix (F (H), Map (F, T))
```