მონაცემთა სტრუქტურა და ბაზისური ოპერაციები (გაგრძელება)

ტიპები ფუნქციონალურ ენებში

როგორც ცნობილია, ფუნქციის არგუმენტები შეიძლება იყოს არა მხოლოდ ბაზური ტიპის ცვლადები, არამედ სხვა ფუნქციებიც. ამ შემთხვევაში ჩნდება მაღალი რიგის ფუნქციის ცნება. შემოვიღოთ ფუნქციონალური ტიპის ცნება (ანუ ტიპის, რომელიც აბრუნებს ფუნქციას). ვთქვათ, რომელიღაც `f` – ფუნქცია არის ერთი ცვლადის ფუნქცია `A` სიმრავლიდან, რომელიც ღებულობს მნიშვნელობებს `B` სიმრავლიდან, მაშინ განსაზღვრების თანახმად:

`#(f) : A -> B`

აქ ნიშანი `#(f)` აღნიშნავს „ფუნქცია `f`-ის ტიპი“. ამრიგად, ტიპს, რომელსაც აქვს სიმბოლო ისარი `->`, ეწოდება ფუნქციონალური ტიპი. ზოგჯერ მისთვის გამოიყენება აღნიშვნა: B^A (შემდგომში გამოვიყენებთ მხოლოდ ისრიან ჩანაწერს, ვინაიდან ზოგიერთი ფუნქციის ტიპი ძალზე რთულად წარმოდგება ხარისხებით).

მაგალითად:

```
#(sin) : Real -> Real
#(Length) : List (A) -> Integer
```

მრავალარგუმენტიანი ფუნქციისთვის ტიპის განსაზღვრა შეიძლება გამოყვანილი იყოს ოპერაციით – დეკარტული ნამრავლით (მაგალითად, `#(add(x, y)) : Real x Real -> Real`). თუმცა ფუნქციონალურ პროგრამირებაში ასეთმა საშუალებამ გამოყენება ვერ ჰპოვა.

1924 წელს მ. შონფიკელმა მრავალარგუმენტიანი ფუნქცია წარმოადგინა როგორც ერთარგუმენტიანი ფუნქციების თანმიმდევრობა. ასეთ შემთხვევაში, ფუნქციის ტიპი, რომელიც შეკრებს ორ ნამდვილ რიცხვს, წარმოდგება ასე: `Real -> (Real -> Real)`. ანუ ასეთი ფუნქციის ტიპი მიიღება სიმბოლო ისრის `->` თანმიმდევრული გამოყენებით. განვსაზღვროთ ეს პროცესი შემდეგ მაგალითზე:

მაგალითი 8. ფუნქცია `add (x,y)`–ის ტიპი.

დავუშვათ, ფუნქცია `add`–ის თითოეული არგუმენტი უკვე აღნიშნულია, ვთქვათ `x = 5, y = 7`. ამ შემთხვევაში თუ ფუნქცია `add`–ს მოვაშორებთ პირველ არგუმენტს, მივიღებთ ახალ ფუნქციას – `add5`, რომელიც თავის ერთადერთ არგუმენტს უმატებს რიცხვს `5`–ს. ამ ფუნქციის ტიპი მიიღება ადვილად და

წარმოდგება ასე: `Real -> Real`. ეხლა, თუ უკან დავბრუნდებით, უკვე გვესმის, რატომ არის `add` ფუნქციის ტიპი `Real -> (Real -> Real)`.

რათა ავიცილოთ `add5` ტიპის ფუნქციების დაწერა (როგორც წინა მაგალითში), მოგონებული იყო სპეციალური აპლიკაციური ჩაწერის ფორმა სახით „ოპერატორი – ოპერანდი". ამის წინაპირობა გახდა ახალი ხედვა ფუნქციისა ფუნქციონალურ პროგრამირებაში. ტრადიციულად ითვლებოდა, რომ გამოსახულება `f (5)` აღნიშნავს „`f` ფუნქციის გამოყენება არგუმენტის მნიშვნელობასთან, რომელიც ტოლია `5`–ის" (ანუ, გამოითვლება მხოლოდ არგუმენტი). ფუნქციონალურ პროგრამირებაში კი ითვლება, რომ ფუნქციის მნიშვნელობაც ასევე ითვლება. ასე, რომ, დავუბრუნდეთ მაგალით `8`–ს, ფუნქცია `add` შეიძლება ასე `(add (x)) y`, ხოლო როცა არგუმენტები იღებს კონკრეტულ მნიშვნელობებს (მაგალითად, `(add (5)) 7`), თავიდან ითვლება ყველა ფუნქცია, სანამ არ დარჩება ერთარგუმენტიანი ფუნქცია, რომელიც გამოიყენება უკანასკნელთან.

ამრიგად, თუ ფუნქცია `f`–ს აქვს ტიპი A_1 `->` (`(`A_2 `->` `( ... (`A_n `-> B) ... ))`, მაშინ, რათა სრულად გამოვთვალოთ მნიშვნელობა `f` (`(`a_1`,` a_2`, ...,` a_n`)`, აუცილებელია თანმიმდევრულად გამოვთვალოთ `( ... (f (`a_1`)` a_2`) ... )` a_n და გამოთვლის შედეგი იქნება `B` ტიპის ობიექტი.

შესაბამისად, გამოსახულება, რომელშიც ყველა ფუნქცია განიხილება როგორც ერთარგუმენტიანი ფუნქცია და ერთადერთ ოპერაციას წარმოადგენს აპლიკაცია (გამოყენება), ეწოდება გამოსახულება ფორმით „ოპერატორი–ოპერანდი". ასეთმა ფუნქციებმა მიიღეს სახელწოდება „კარირებული", ხოლო თვითონ ფუნქციის დაყვანის ზემოთ აღწერილმა პროცესმა – „კარირება" (კარი ჰასკელის სახელიდან გამომდინარე).

თუ გავიხსენებთ λ–აღრიცხვას, აღმოვაჩენთ, რომ მასში უკვე არის მათემატიკური აბსტრაქცია ჩანაწერების აპლიკაციური ფორმისთვის. მაგალითად:

```
f (x) = x2 + 5              <=>   λx.(x2 + 5)
f (x, y) = x + y            <=>   λy. λx.(x + y)
f (x, y, z) = x2 + y2 + z2  <=>   λz. λy. λx.(x2 + y2 + z2)
```

და ა.შ....

რამდენიმე სიტყვა აბსტრაქტული ენის ნოტაციის შესახებ

ნიმუშები და კლოზები

ავღნიშნოთ, რომ აბსტრაქტული ფუნქციონალური ენის ნოტაციაში, რომელსაც ვიყენებით ფუნქციის მაგალითების დაწერისას, შესაძლებელი იყო გამოგვეყენებინა

ისეთი კონსტრუქცია, როგორიცაა `if-then-else`. მაგალითად, ფუნქცია `Append`–ის აღწერისას (იხილეთ მაგალითი 7), მისი ტანი შეიძლება ჩაწერილიყო შემდეგნაირად:

```
Append (L1, L2) = if (L1 == []) then L2
                  else head (L1) : Append (tail (L1), L2)
```

თუმცა მოცემული ჩანაწერი ცუდად გასარჩევია, ამიტომაც მაგალითში 7 უკვე გამოვიყენეთ ნოტაცია, რომელიც მხარს უჭერს ე.წ. "ნიმუშებს".

განმარტება:

ნიმუში ეწოდება გამოსახულებას, რომელიც აგებულია მონაცემთა კონსტრუირების ოპერაციით და რომელიც გამოიყენება მონაცემებთან შესაბამისობისათვის. ცვლადები აღინიშნება დიდი ასოებით, კონსტანტები–პატარათი.

ნიმუშის მაგალითებია:

`5` – რიცხვითი კონსტანტა.

`X` –ცვლადი.

`X : (Y : Z)` – წყვილი.

`[X, Y]` – სია.

ნიმუშმა აუცილებლად უნდა დააკმაყოფილოს ერთი მოთხოვნა, წინააღმდეგ შემთხვევაში მასთან შედარება არასწორად შესრულდება. ეს მოთხოვნა ასე ჟღერს: ნიმუშთან მონაცემების შედარებისას ცვლადისთვის მნიშვნელობის მინიჭება უნდა მოხდეს მხოლოდ ერთადერთი გზით, ანუ, მაგალითად, გამოსახულება `(1 + X ==> 5)` შეიძლება გამოვიყენოთ როგორც ნიმუში, რადგანაც `X` ცვლადის აღნიშვნა ხდება ერთადერთი გზით `(X = 4)`, ხოლო შემდეგი გამოსახულების `(X + Y ==> 5)` გამოყენება ნიმუშად არ შეიძლება, ვინაიდან `X` და `Y` ცვლადების აღინიშნება სხვადასხვანაირად (არაცალსახად).

ფუნქციონალურ პროგრამირებაში ნიმუშის გარდა შემოდის ისეთი ცნება, როგორიცაა „კლოზი" (ინგლისურიდან „`clause`"). განმარტებით, კლოზი ეს არის:

`def` `f` p_1`, ..., `p_n **`=`** `expr`

სადაც:

`def` და **`=`** – აბსტრაქტული ენის კონსტანტებია.

`f` – განსაზღვრულიი ფუნქციის სახელია.

p_i – ნიმუშების თანმიმდევრობაა (ამასთან, `>= 0`).

`expr` – გამოსახულებაა.

ამრიგად, ფუნქციონალურ პროგრამირებაში ფუნქციების განსაზღვრება არის უბრალოდ კლოზების თანმიმდევრობა (შესაძლოა, მხოლოდ ერთი ელემენტისგან შემდგარი). რათა გავამარტივოთ ფუნქციის განსაზღვრების ჩაწერა, შემდგომში სიტყვა `def`–ს გამოვტოვებთ.

მაგალითი 9. ნიმუშები და კლოზები ფუნქციაში `Length`.

```
Length ([]) = 0
Length (H:T) = 1 + Length (T)
```

ვთქვათ ფუნქცია `Length`–ის გამოძახება ხდება პარამეტრით `[a, b, c]`. ამ დროს მუშაობას იწყებს ნიმუშთან შედარების მექანიზმი. სათითაოდ გადაისინჯება ყველა კლოზი და ხდება შედარებების მცდელობები. ამ შემთხვევაში წარმატებით მოხდება მხოლოდ მეორე კლოზთან შედარება (რადგანაც სია `[a, b, c]` არ არის ცარიელი).

ფუნქციის გამოძახების ინტერპრეტაცია მდგომარეობს შემდეგში: ხდება შედარება ზემოდან ქვემოთ ნიმუშებში და რიგით პირველი ნიშუშის პოვნა, რომელიც წარმატებით შედარდა ფაქტიურ პარამეტრებს. ნიმუშის ცვლადების მნიშვნელობები, რომლებიც მათ მიენიჭათ შედარების შედეგად, ჩაისმის კლოზის მარჯვენა მხარეს (გამოსახულებაში `expr`), რომლის მნიშვნელობის გამოთვლაც მოცემულ კონტექსტში წარმოადგენს ფუნქციის გამოძახების მნიშვნელობას.

დაცვა

აბსტრაქტულ ნოტაციაში ფუნქციის დაწერისას დაშვებულია ე.წ. დაცვის გამოყენება, ანუ ნიმუშის ცვლადებზე შეზღუდვების გამოყენება. მაგალითად, დაცვის გამოყენებით ფუნქცია `Length`–ის განსაზღვრა შეიძლება იყოს შემდეგი:

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

განხილულ კოდში სიტყვა `when` (მაშინ) და `otherwise` (წინააღმდეგ შემთხვევაში) წარმოადგენს ენის დარეზერვირებულ სიტყვებს. თუმცა, ამ სიტყვების გამოყენება არ არის დაცვის გამოყენებისთვის აუცილებელი პირობა. დაცვა შეიძლება განვახორციელოთ სხვადასხვა საშუალებით, მათ შორის λ–აღრიცხვით:

```
Append = λ[].(λL.L)
Append = λ(H:T).(λL.H : Append (T, L))
```

წარმოდგენილი ჩანაწერი ცუდი წასაკითხია, ამიტომ მას მხოლოდ უკიდურეს შემთხვევებში გამოვიყენებთ.

ლოკალური ცვლადები

როგორც უკვე ავღნიშნეთ, ლოკალური ცვლადების ცვლადების გამოყენება იწვევს გვერდით ეფექტს, ამიტომ იგი დაუშვებელია ფუნქციონალურ ენებში. თუმცა, ზოგიერთ შემთხვევაში ლოკალური ცვლადების გამოყენება არის ოპტიმალური, რაც იძლევა გამოთვლების დროის და რესურსების ეკონომიის საშუალებას.

დავუშვათ, `f` და `h` ფუნქციებია და აუცილებელია გამოითვალოს გამოსახულება `h (f(X), f(X))`. თუ ენაში არ არის ჩადებული ოპტიმიზაციის მეთოდები, მაშინ ხდება `f (X)` გამოსახულების ორჯერ გამოთვლა. ეს რომ არ მოხდეს, შეიძლება მივმართოთ ასეთ საშუალებას: `(λv.h (v, v))(f (X))`. ბუნებრივია, რომ ამ შემთხვევაში გამოსახულება `f (X)` გამოითვლება პირველად და ერთხელ. იმისათვის, რომ λ–აღრიცვის გამოყენება მინიმალურად მოხდეს, შემდგომში შემდეგი სახის ჩანაწერს გამოვიყენებთ:

```
let v = f (X) in h (v, v)
```

(სიტყვები **let, =** და **in** – ენაში დარეზერვირებულია). ამ შემთხვევაში `v` ვუწოდებთ ლოკალურ ცვლადს.

პროგრამირების ელემენტები

პარამეტრების დაგროვება – აკუმულატორი

ფუნქციის შესრულებისას შეიძლება დადგეს მეხსიერების ხარჯვის სერიოზული პრობლემა. ავხსნათ ეს პრობლემა ფუნქციის მაგალითზე, რომელიც ითვლის რიცხვის ფაქტორიალს:

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

თუ მოვიყვანთ ამ ფუნქციის გამოთვლის მაგალითს არგუმენტზე 3, მაშინ დავინახავთ შემდეგ თანმიმდევრობას:

```
Factorial (3)
3 * Factorial (2)
3 * 2 * Factorial (1)
3 * 2 * 1 * Factorial (0)
3 * 2 * 1 * 1
3 * 2 * 1
3 * 2
```

```
6
```

ამ გამოთვლის მაგალითზე ვხედავთ, რომ ფუნქციის რეკურსიული გამოყენება ძალიან ძლიერად იყენებს მეხსიერებას. ამ შემთხვევაში მეხსიერება არგუმენტის მნიშვნელობის პროპორციულია, მაგრამ არგუმენტებიც შეიძლება იყოს დიდი. ჩნდება კითხვა, შესაძლებელია თუ არა ისე დაიწეროს ფაქტორიალის გამოთვლის ფუნქცია (და მისი მსგავსი ფუნქციები), რომ მეხსიერება მინიმალურად იქნას გამოყენებული?

ამ შეკითვაზე დადებითი პასუხისთვის აუცილებელია განვიხილოთ აკუმულატორის (დამგროვებლის) ცნება. ამისთვის განვიხილოთ შემდეგი მაგალითი:

მაგალითი 10. ფაქტორიალის გამოთვლის ფუნქცია აკუმულატორის გამოყენებით.

```
Factorial_A (N) = F (N, 1)


F (0, A) = A
F (N, A) = F ((N - 1), (N * A))
```

ამ მაგალითში ფუნქცია F–ის მეორე პარამეტრი ასრულებს აკუმულატორი ცვლადის როლს. სწორედ იგი შეიცავს შედეგს, რომელიც ბრუნდება რეკურსიის დამთავრებისას. თვითონ რეკურსიას კი, ამ დროს აქვს „კუდური“ რეკურსიის სახე, ამასთან, მეხსიერება გამოიყენება მხოლოდ ფუნქციის მნიშვნელობების დასაბრუნებელი მისამართების შენახვისათვის.

კუდური რეკურსია წარმოადგენს რეკურსიის სპეციალურ სახეს, რომლის დროსაც მხოლოდ ერთხელ ხდება რეკურსიული ფუნქციის გამოძახება და ეს გამოძახებაც სრულდება ყველა გამოთვლის შემდეგ.

კუდური რეკურსიის რეალიზაცია შეიძლება შესრულდეს იტერაციის პროცესის საშუალებით. პრაქტიკაში ეს ნიშნავს, რომ ფუნქციონალური ენის „კარგ“ ტრანსლიატორს უნდა „შეეძლოს“ აღმოაჩინოს კუდური რეკურსია და მოახდინოს მისი რეალიზება ციკლის სახით. თავის მხრივ, პარამეტრის დაგროვების მეთოდს ყოველთვის არ მოვყავართ კუდურ რეკურსიამდე, თუმცა იგი ცალსახად გვეხმარება საერთო მეხსიერების მოცულობის შემცირებაში.

დამგროვებელი პარამეტრებით განსაზღვრებების აგების პრინციპები:

1. შემოდის ახალი ფუნქცია დამატებითი არგუმენტით (აკუმულატორით), რომელშიც გროვდება გამოთვლების შედეგები.

2. აკუმულატორი არგუმენტის საწყისი მნიშვნელობა მოიცემა ტოლობით, რომელიც აკავშირებს ძველ და ახალ ფუნქციებს.

3. საწყისი ფუნქციის ის ტოლობა, რომელიც შეესაბამება რეკურსიიდან გამოსავალს, იცვლება აკუმულატორით დაბრუნებით.

4. ტოლობა, რომელიც შეესაბამება რეკურსიულ განსაზღვრებას, გამოიხატება როგორც ახალ ფუნქციაზე მიმართვა, რომელშიც აკუმულატორი იღებს იმ მნიშვნელობას, რომელიც ბრუნდება საწყისი ფუნქციით.

ისმის შეკითვა: ნებისმიერი ფუნქცია შეიძლება გარდავქმნათ აკუმულატორის გამოთვლის ფორმით? ბუნებრივია, რომ ამ შეკითხვის პასუხი არის უარყოფითი. დამგროვებელი პარამეტრიანი ფუნქციის აგების ხერხი არ არის უნივერსალური და იგი არ არის კუდური რეკურსიის აგების გარანტია. მეორეს მხრივ, განსაზღვრების აგება დამგროვებელი პარამეტრით არის შემოქმედებითი საქმე. ამ პროცესში აუცილებელია ზოგიერთი ევრისტიკის გამოყენება.

განსაზღვრება:

რეკურსიული განსაზღვრების ზოგად სახეს, რომელიც საშუალებას იძლევა ტრანსლიაციისას უზრუნველყოს გამოთვლები მეხსიერების მუდმივ მოცულობაში იტერაციის საშუალებით, უწოდებენ იტერაციული სახის ტოლობებს.

$f_i\ (p_{ij}) = e_{ij}$

ამასთან, გამოსახულება e_{ij}-ს ედება შემდეგი შეზღუდვები:

1. e_{ij} — „მარტივი“ გამოსახუილებაა, ანუ ის შეიცავს მხოლოდ მონაცემებზე ოპერაციებს და არ შეიცავს რეკურიულ გამოძახებებს.

2. e_{ij}-ს აქვს სახე $f_k\ (v_k)$, ამასთან v_k — მარტივი გამოსახულებების თანმიმდევრობაა. ეს არის კუდური რეკურსია.

3. e_{ij} — პირობითი გამოსახულებაა პირობაში მარტივი გამოსახულებით, რომლის შტოები განისაზღვრება ამავე სამი პირობით.

სავარჯიშოები

1. ააგეთ ფუნქცია, რომლებიც მუშაობს სიებთან. საჭიროების შემთხვევაში გამოიყენეთ დამატებითი ფუნქციები და ზემოთ განსაზღვრული ფუნქციები.

a. Reverse_all — ფუნქცია, რომელიც შესასვლელზე იღებს სიურ სტრუქტურას და აბრუნებს მის ყველა ელემენტს და ასევე მას.

b. Position — ფუნქცია, რომელიც აბრუნებს მოცემული ატომის სიაში პირველად შესვლის ნომერს.

c. Set — ფუნქცია, რომელიც აბრუნებს სიას, რომელშიც მოცემული სიის ყველა ატომი მხოლოდ ერთხელ შედის.

d. Frequency — ფუნქცია, რომელიც აბრუნებს წყვილების სიას (სიმბოლო, სიხშირე). თითოეული წყვილი განისაზღვრება მოცემული სიის ატომით და ამ სიაში მისი შესვლის სიხშირით.

2. დაწერეთ ფუნქციები დამგროვებელი პარამეტრებით (თუ ეს შესაძლებელია) სავარჯიშო 1-ში მოყვანილი ფუნქციებისთვის.

პასუხები თვითშემოწმებისთვის

1. შემდეგი აღწერები რეალიზებს მოთხოვნილ ფუნქციებს. ზოგიერთ პუნქტებში რეალიზებულია დამატებითი ფუნქციები, რომლებზეც გვერდის ავლა ძნელად წარმოგვიდგენია. a. Reverse_all:

```
    Reverse_all (L) = L when atom (L)
    Reverse_all ([]) = []
    Reverse_all (H:T) = Append (Reverse_all (T), Reverse_all (H))
otherwise
```

b. Position:

```
    Position (A, L) = Position_N (A, L, 0)

    Position_N (A, (A:L), N) = N + 1
    Position_N (A, (B:L), N) = Position_N (A, L, N + 1)
    Position_N (A, [], N) = 0
```

c. Set:

```
    Set ([]) = []
    Set (H:T) = Include (H, Set (T))

    Include (A, []) = [A]
    Include (A, A:L) = A:L
    Include (A, B:L) = prefix (B, Include (A, L))
```

d. Frequency:

```
    Frequency L = F ([], L)
```

```
F (L, []) = L
F (L, H:T) = F (Corrector (H, L), T)

Corrector (A, []) = [A:1]
Corrector (A, (A:N):T) = prefix ((A:N + 1), T)
Corrector (A, H:T) = prefix (H, Corrector (A, T))
```

2. სავარჯიშო 1–ის ყველა ფუნქციისთვის შეიძლება ავაგოთ განმარტებები დამგროვებელი პარამეტრებით. მეორეს მხრივ, შესაძლოა ზოგიერთი ახლადგანმარტებული ფუნქცია არ იყოს ოპტიმიზირებული.

a. Power_A:

```
Power_A (X, N) = P (X, N, 1)

P (X, 0, A) = A
P (X, N, A) = P (X, N - 1, X * A)
```

b. Summ_T_A:

```
Summ_T_A (N) = ST (N, 0)

ST (0, A) = A
ST (N, A) = ST (N - 1, N + A)
```

c. Summ_P_A:

```
Summ_P_A (N) = SP (N, 0)

SP (0, A) = A
SP (N, A) = SP (N - 1, Summ_T_A (N) + A)
```

d. Summ_Power_A:

```
Summ_Power_A (N, P) = SPA (N, P, 0)

SPA (N, 0, A) = A
SPA (N, P, A) = SPA (N, P - 1, (1 / Power_A (N, P)) + A)
```

e. Exponent_A:

```
Exponent_A (N, P) = E (N, P, 0)

E (N, 0, A) = A
E (N, P - 1, (Power_A (N, P) / Factorial_A (P)) + A)
```