ISSN 1512-1232

უაკ (უდკ) 004.4

ქართული ენის ლექსიკონის წარმოდგენა დაპროგრამების ფუნქციონალური ენების საშუალებით და ძებნა „ტალღური მეთოდის“ გამოყენებით

არჩვაძე ნათელა
ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტი, ზუსტ და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი, 0179, თბილისი, ი.ჭავჭავაძის პრ. 1

მერაბ ფხოველიშვილი
საქართველოს ტექნიკური უნივერსიტეტი, ნიკო მუსხელიშვილის სახელობის გამოთვლითი მათემატიკის ინსტიტუტი, 0175, თბილისი, აკურის ქ.7

ანოტაცია

სტატიაში განხილულია ტექსტების მოხერხებული წარმოდგენის და სწრაფი დამუშავების პრობლემები. ყურადღება გამახვილებულია ქართული ენის ლექსიკონის ხის სტრუქტურულ წარმოდგენაზე და მის შემდგომ დამუშავებაზე ფუნქციონალური ენების საშუალებით. განხილულია ძებნის „ტალღური მეთოდი“, რომლის რეალიზება მრავალპროცესორულ კომპიუტერებზე იძლევა დროის დიდ მოგებას.

საკვანძო სიტყვები: *ფუნქციონალური პროგრამირება. სიის სტრუქტურა. ინფორმაციის ძებნის მეთოდი. ლექსიკონის წარმოდგენა.*

ინფორმაციული რესურსის სწრაფი ზრდა, უპირველეს ყოვლისა ინტერნეტში, თითოეული ჩვენგანისთვის აქტუალურს ხდის პრობლემას - სწრაფად და სწორედ მოძებნოს ინფორმაცია. თანამედროვე კომპიუტერული მეცნიერებების ერთ–ერთ მნიშვნელოვან ამოცანას წარმოადგენს სწორედ ინფორმაციის კომპიუტერული წარმოდგენის საშუალებების კვლევა ინფორმაციის ინტერნეტში შემდგომი განთავსებისა და ძებნის მიზნით. ეფექტური და ხარისხიანი ძებნისთვის აუცილებელია ინფორმაცია გარდაიქმნას ისეთ ფორმაში, რაც მოგვცემს საშუალებას ინფორმაციის ეფექტური ძებნის განხორციელებისათვის.

ჩვენი მიზანია გამოვიკვლიოთ ინფორმაციის კომპიუტერული წარმოდგენის საშუალებები კონკრეტული ამოცანისთვის – ქართული ენის ანბანის სიტყვებისთვის, რათა მოვახდინოთ ეფექტური ანალიზი ისეთი ტიპის ამოცანების გადასაწყვეტად, როგორიცაა, მართლწერის შემოწმება, ინფორმაციის ძებნა, კომპიუტერული თარგმანი, ქართული ენის ანალიზი და სხვა.

1. ინფორმაციის წარმოდგენა ხეების საშუალებით. რეალიზება Lisp-ზე

დავუშვათ, რომ ინფორმაცია კომპიუტერში წარმოდგენილია ხეების საშუალებით. ჩვენ განვიხილავთ ხეებზე წარმოდგენების რეალიზებას ფუნქციონალური ენების საშუალებებით, კერძოდ, გამოვიყენებთ Lisp-ის, Haskell–ის და F#–ის სიებს.

ISSN 1512-1232

ნებისმიერი ენის ლექსიკონის სიტყვების ერთობლიობის წარმოსადგენად შეიძლება გამოვიყენოთ შემდეგი სახის მონაცემების სტრუქტურა – ხე. ამ ხის თითოეულ წვეროში თავსდება მოცემული სიტყვის ცალკეული ასო სიტყვით განსაზღვრული თანმიმდევრობით. იმ წვეროს, რომელზეც მოცემული სიტყვა მთავრდება, შეიძლება მივაბათ ამ სიტყვის შესახებ მორფოლოგიური, სემანტიკური და სხვა სახის ინფორმაცია.

განვიხილოთ უფრო დეტალურად ქართული ენის ლექსიკონის ხის სტრუქტურით წარმოდგენის ალგორითმი. თავდაპირველად განვიხილოთ მაგალითები.

სიტყვები, რომლებიც ერთი და იგივე ასოთი იწყება, ერთიანდებიან ერთ ხეში. მაგალითად, სიტყვებში „აალდა“, „აალება“, „აალებადი“, „აალებები“ და „აალებული“ პირველი სამი ასო ერთიდაიგივეა. ამ ხუთი სიტყვის ერთობლიობას შეესაბამება ასეთი სახის ხე:

(1)

ხეზე შავად ნაჩვენებია წვერო, რომელზეც ერთი სიტყვა მთავრდება, ხოლო სხვა გრძელდება. ჩვენს მაგალითში შავად აღნიშნულ წვეროზე მთავრდება სიტყვა „აალება“ და გრძელდება სიტყვა „აალებადი“.

(1) ხე, თავის მხრივ, წარმოვადგინოთ ფუნქციონალური დაპროგრამების ენა Lisp-ის სიების საშუალებით:

```
(ა (ა (ლ (დ (ა ))
          (ე (ბ (ა* (დ (ი )))))
                (ე (ბ (ი )))
                (უ (ლ (ი ))) )) )))
```

სიით წარმოდგენაში გამოყენებულია სიმბოლო * იმ ასოს შემდეგ, სადაც ერთი სიტყვა მთავრდება, მაგრამ მეორე გრძელდება.

ქართული ანბანის თითოეული ასოსათვის არის შესაძლებელი შედგეს ასეთი ტიპის მონაცემების სტრუქტურები, რომლებიც ხეებს შეესაბამება. პირველი ხე გააერთიანებს ასო „ა“–ზე დაწყებულ სიტყვებს, მეორე ხე „ბ“–ზე დაწყებულ სიტყვებს და ა.შ. სულ იარსებებს 33 ხე, რომლებზეც ძებნა პარალელური მეთოდებით მოხდება.

შემდეგი მაგალითი ასახავს ქართული ენის ლექსიკონიდან იმ სიტყვების ჯგუფის წარმოდგენას სიის სტრუქტურის მიხედვით, რომელიც იწყება „აა“ (ყველა ქართული სიტყვა) და „აბ“ ასოებით (მხოლოდ სამი სიტყვა). ეს სიტყვებია: „ააგო“, „აადამიანებს“, „აადგილებს“, “ აავსო“, „აალდა“, „აალება“, „აალებადი“, „აალებები“, „აალებული“, „აამაღლა“, „აამოქმედა“, „აამღვრია“, „აანაზღაურა“, „აანთო“, „აარიდა“,

ISSN 1512-1232
„აარსებს“, „აარჰუსი“, „აასფალტებს“, „ააქაფა“, „ააქტივებს“, „ააღელვა“, „ააღორძინა“, „ააშენა“, „ ააშკარავებს“, „ააჩარა“, „ააცდინა“, „ააცილა ‘, „ააწყო“, „ააჭრელა“, „აახენი“, „აახლებს“, „აბა“, „აბაზანა“, „აბაკი“. სიტყვები ანბანურად არის დალაგებული, თუმცა ეს არ არის ალგორითმის მოთხოვნა. სიტყვები ნებისმიერი თანმიმდევრობით შეიძლება გადაეცეს, მათი დალაგება ანბანურად მოხდება სიური წარმოდგენის აგებისას.

პროგრამირების ენა Lisp-ის სიების საშუალებით მოცემული სიტყვების წარმოდგენას აქვს შემდეგი სახე [1]:

```
(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    (ლ    (ლ    (ა)   )))
                  (ო    (ქ    (მ    (ე    (დ    (ა)   )))   ))
                  (ლ    (ვ    (რ    (ი    (ა)   )))   ))
            (ნ    (ა    (ზ    (ლ    (ა    (უ    (რ    (ა)   )))   ))))
                  (თ    (ო    )))
            (რ    (ი    (დ    (ა    )))
                  (ს    (ე    (ბ    (ს    )))   )
                  (ჰ    (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    ))    )           ))
```

ხის წარმოდგენისას სიმბოლო * გამოყენებულია იმ წვეროსთვის–ასოსთვის, რომელზეც ერთი სიტყვა მთავრდება, ხოლო სხვა გრძელდება. ასეთებია: „აალება“ და „აალებადი“, „აბა“ და „აბაზანა“.

სიის შედგენის ალგორითმი ასეთია: სიტყვა დაიშლება ასოებად, პირველი ასო განსაზღვრავს იმ ხეს, რომელსაც მოცემული სიტყვა დაემატება, თუ ასეთი ხე უკვე არსებობს, წინააღმდეგ შემთხვევაში თითოეული ასოს წინ დაისმება ფრჩხილი,

ISSN 1512-1232
რომელიც სიტყვის ბოლოს იხურება. იმ შემთხვევაში, თუ სიტყვის პირველი ასოს შესაბამისი ხე უკვე არსებობს, სიტყვის თითოეული მომდევნო ასო თანმიმდევრულად შედარდება ხის წვეროებს. თუ ასეთი წვერო არსებობს, ხდება შემდეგი ასოსთან შედარება, თუ არ არსებობს, დაემატება ეს წვერო და მისი მომდევნო წვეროებიც სიტყვის დარჩენილი ასოებისთვის. მაგალითად, სიტყვა „აავსო“ წარმოდგენილია სიით: (ა (ა (ვ (ს (ო))))). საჭიროა სიტყვა „აალდა“–ს წარმოდგენა. ვინაიდან მოცემულ სიტყვებს პირველი ორი ასო ერთნაირი აქვთ, „ლ“ ასო დაემატება მესამე დონეზე და ამ მიმართულებით ხე გაიზრდება სიტყვის დარჩენილი „დ“ და „ა“ ასოსთვის. შედეგად მიიღება შემდეგი სია:

(ა (ა (ვ (ს (ო))

(ლ (დ (ა))))).

სიას მესამე დონეზე ორი ელემენტი აქვს. დონეს, რომელიც სიტყვების რაოდენობას ემთხვევა, განსაზღვრავს ის, თუ ერთი სიტყვის რამდენი ასო ემთხვევა მეორე სიტყვის ასოებს.

აქ გამოყენებული გვაქვს სიტყვები კომპიუტერული ლექსიკონიდან (English Georgian Dictionary), რომელიც შეიცავს 50000 ქართულ სიტყვას. სიური წარმოდგენა განხორციელებულია ფუნქციონალური დაპროგრამების ენა Lisp-ის სიებით. მსგავსი წარმოდგენები შეიძლება რეალიზებული იყოს ფუნქციონალური ენების Haskell-ისა და F#–ის სტრუქტურების საშუალებით.

1.1 ხით წარმოდგენის უპირატესობები

ლექსიკონის ხის სტრუქტურით წარმოდგენის უპირატესობა არის ის, რომ მისი საშუალებით შეიძლება რამდენიმე ამოცანის გადაჭრა. პირველი: 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 გამოვა, 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.

მეორე ამოცანა: მთარგმნელი სისტემების აგება. 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 ა) მოთავსდეს (ისევ რთული სიის სახით) ჩამონათვალი ამ სიტყვის შესაბამისი მნიშვნელობებისა სხვადასხვა ენაზე. ჩადგმული სია კი საჭიროა იმისათვის, რომ მიეთითოს მოცემული სიტყვის მნიშვნელობათა სინონიმები; ბ) მოხდეს გადასვლა დამატებით ხეზე–სადაც შეძლება ნებისმიერი სახის ინფორმაციის შენახვა (შემდგომ განვიხილავთ).

ISSN 1512-1232

1.2. ხის სტრუქტურის განსაზღვრა. ხის აგების ალგორითმი და მისი რეალიზაცია ენა Haskell–ში

განვსაზღვროთ ფუნქციონალური ენის Haskell-ზე ტიპი Tree(A), რომელიც წარმოადგენს ხეს მონიშნული წვეროებით და ტიპი Forest(A), რომელიც წარმოადგენს ტყეს (ხეების სიმრავლეს) [2, 3]. მოცემული ტიპები შეიძლება გამოვიყენოთ მონაცემთა ისეთი სტრუქტურისთვის, რომელთათვისაც არ არის მნიშვნელოვანი კავშირი მშობლიურ და შვილობრივ იერარქიულ ელემენტებს შორის. ასეთი მონაცემებისთვის მნიშვნელოვანია მხოლოდ ინფორმაცია წვეროს შესახებ, რომელიც არის ბაზური A ტიპის.

```
Tree(A) = A x Forest(A);
Forest(A) = List(Tree(A));
node = constructor Tree( A);
root, children = selectors Tree(A).
```

როგორც ვხედავთ, ტიპი `Forest(A)` არის `Tree(A)` ტიპის სია.

განვმარტოთ თითოეული ტიპი შემდეგნაირად:

```
data Tree a = Node (a, [Tree a])
root (Node (a, _)) = a
children (Node (_, c)) = c
```

ტიპი `Tree a` იყენებს კონტრუქტორს `Node`, რომელიც გამოიყენება ერთი წვეროს ასაგებად თავისი შთამომავლებით. ფუნქციები `root` და `children`, რომლებიც გადაეცემა `Tree a`–ს არგუმენტად, წარმოადგენს ამ ტიპის სელექტორებს.

ხშირად დგას ისეთი ამოცანები, სადაც მნიშვნელოვანია კავშირის ტიპი მშობლიურ და შვილობრივ წვეროებს შორის. მაგალითად, ქართული ანბანის წარმოდგენის მაგალითში ეს შეიძლება იყოს რაიმე დამატებითი ინფორმაცია: მორფოლოგიური, სემანტიკური და სხვა სახის.

ამ მიზნისთვის შესაძლოა გამოყენებული იყოს ხეები მონიშნული წვეროებითა და რკალებით. წვეროზე ჭდე ინახავს ინფორმაციას ამ წვეროს შესახებ. ვთქვათ, მას აქვს ბაზური ტიპი `A`. ჭდე რკალზე განისაზღვრება ამ რკალის ტიპით, რომელიც დაკავშირებულია მოცემული ამოცანის შინაარსზე. ვთქვათ, მას ავს ბაზური ტიპი `B`. ტიპი `MTree(A, B)` შეიძლება განისაზღვროს შემდეგი სახით:

```
MTree(A, B) = Ax MForest(A, J5);
MForest(A,B) = List(M Arc(A, B));
MArc(A, B) = Bx MTree(A, Б);
node = constructor MTree( A, B);
mRoot, mChildren = selectors MTree(A, B);
arc = constructor MArc(A, Б);
mArc, mTree = selectors MArc(A, B).
```

ტიპი `MTree(A,B)` ისევე განისაზღვრება, როგორც ტიპი `Tree(A)`. განსაზღვრებაში საჭიროა გამოვიყენოთ დამხმარე ტიპები, რადგანაც ხე თავის თავში შეიცავს უფრო მეტ ინფორმაციას. დამხმარე ტიპი `MForest(A,B)` წარმოადგენს მონიშნული რკალების სიმრავლეს, რომელიც გამოდის ხის წვეროდან. თავის მხრივ, დამხმარე ტიპი `MArc(A, B)` წარმოადგენს რკალის აღწერას, რომელიც მონიშნულია ტიპით `B`. ტიპებისთვის `MTree(A,B)` და `MArc(A,B)` მოყვანილია კონსტრუქტორები და სელექტორები.

ISSN 1512-1232

ბაზური ფუნქციების წარმოდგენილი ეს ერთობლიობა საკმარისია ამ ტიპის ნებისმიერი მნიშვნელობის დასამუშავებლად. ყველა სხვა ფუნქცია უნდა განისაზღვროს მათგან.

განსაზღვროთ შაბლონური ფუნქციები, რომლებიც საშუალებას გვაძლევს ნებისმიერი მიზნით დავამუშავოთ MTree(A,B) მნიშვნელობები. მონიშნული წვეროებითა და რკალებით ხეების დასამუშავებელი ნებისმიერი კონკრეტული ფუნქცია შეიძლება წარმოვადგინოთ ამ შაბლონური ფუნქციებით.

შემოვიღოთ შემდეგი ცნებები. ვთქვათ მონიშნული წვეროებითა და რკალებით ხის საწყისი წვერო, MForest სიის თავი და MTree–ის წვერო, რომელიც გამოდის MArc–დან, აღნიშნულია S_o, S_1 და S_2–ით, შესაბამისად. მათ დასამუშავებლად საჭიროა სამი ფუნქცია, f1 და f2, ამასთან f0 – საწყისი ფუნქციაა, ხოლო f1 და f2 – რეკურსიული. განვმარტოთ თითოეული ფუნქცია.

f0 არის ფუნქცია, რომელსაც აქვს ერთი პარამეტრი tree, რომელსაც შეესაბამება მისი საწყისი წვერო S_o.

f1 ღებულობს შემდეგ პარამეტრებს:

1) a — მიმდინარე წვეროს ჭდე;

2) к — პარამეტრი, რომელიც შეიცავს ხის განხილული ნაწილის დამუშავების შედეგს;

3) (x:xs) — ტყე, რომელიც ამ ფუნქციამ უნდა დაამუშავოს.

ამ ფუნქციის განსაზღვრას აქვს შემდეგი სახე:

```
f1 a к [] = gl a к
f1 a к (x:xs) = fl a (g2 (f2 a arc children k) a arc к) xs
where arc = mArc x
children = mTree xs
```

როგორც ვხედავთ, ფუნქცია ახორციელებს ხის გავლას თავიდან სიღრმისკენ, ამასთან, ფუნქციები gl და g2 არიან დამხმარე ფუნქციები, რომლებიც ცვლიან პარამეტრებს ამოცანის ლოგიკიდან გამომდინარე.

ფუნქცია f2 ღებულობს შემდეგ პარამეტრებს:

1) a — მიმდინარე წვეროს ჭდე;
2) b — მიმდინარე რკალის ჭდე;
3) к — ხის განხილული ნაწილის დამუშავების შედეგის პარამეტრი;
4) tree — ქვეხე შემდგომი დამუშავებისთვის.

ამ ფუნქციის განსაზღვრას აქვს შემდეგი სახე:

```
f2 a b k tree = f1 (mRoot tree)(g3 a b к)(mChildren tree)
```

აქაც ფუნქცია g3 დამოკიდებულია კონკრეტულ მიზნებზე. ის აუცილებელია პარამეტრი к–ს შემდეგი მნიშვნელობის მისაღებად.

f0 ფუნქცია კი ასე განისაზღვრება:

```
f0 tree = f1 (mRoot tree) k (mChildren tree)
```

სადაც პარამეტრ к–ს ენიჭება კონკრეტული მნიშვნელობა.

ISSN 1512-1232

1.3 რეკურსიული სტრუქტურა. ხის წარმოდგენა ფუნქციონალურ ენა F#-ში

ხე არის რეკურსიული სტრუქტურა Tree<T> კლასის ობიექტი, სადაც T არის მოცემული ხის წევრი, ხე შედგება წვეროსგან და ორი ქვეხისგან [4]:

F# კოდი:

```
type Tree<'T> =
   | Node of Tree<'T> * 'T * Tree<'T>
   | Leaf
```

სადაც, Leaf– ფოთოლი არის ტერმინალური ელემენტი ქვეხეების გარეშე, ხოლო Node–შტო არის ელემენტი, რომელიც შეიცავს მნიშვნელობას და ქვეხეს.

ზემოთ განხილული (1) ხე F#–ზე ასე წარმოდგება:

```
let tr=Node ('ა', [
          Node ('ა',[
            Node ('ლ',[
               Node ('დ',[
                 Leaf ('ა')]);
               Node ('ე',[
              Node ('ბ',[
                Node ('ა',[
                 Node ('დ',[
                   Leaf ('ი')]); ])
                Node ('ე',[
                 Node ('ბ',[
                   Leaf ('ი')]);
                Node ('უ',[
                 Node ('ლ',[
                   Leaf ('ი')]) ]) ])  ]) ]) ]) ]) ]) ;
```

ხეზე ძებნის ალგორითმების რეალიზაცია შემოწმებული იყო ხეებზე, რომლებიც აიგო ხისქვეშა მიმართულებით შემთხვევითი 50/50 F# ენის მეთოდებით.

შემდეგი პროცედურა საშუალებას იძლევა ისე გავიაროთ ხე, რომ მის თითოეულ ელემენტთან მხოლოდ ერთხელ აღვმოჩნდეთ:

```
let rec iter f=function
  Leaf(T) -> f(T)
  |Node(T,l) -> (f(T);for t in L do iter f t done);;
let iterh f =
  let rec itr n = function
     Leaf (T) -> f n T
  |Node(T,L) ->(f n T; for t in L do itr (n+1) t done in
   itr 0;;
let print_tree = iter
   (fun h x -> printf "%s%A\n" (space(h*3))x)T;;
```

რეკურსიული tree ტიპის ფუნქციებთან ჩვენ გამოვიყენებთ ფუნქცია map–ს, რომელიც ასე განიმარტება:

```
map:(A->B)->A tree->B tree          --ფუნქცია map–ის ტიპი
```

ISSN 1512-1232

```
    let rec map f=function
        Leaf (T) -> Leaf (f T)
        |Node (T,L) ->
           Node(f T, list.map(fun t ->map f t)L);;
```

ხის აგება ხდება მოცემული ელემენტების მაქსიმალური რაოდენობისაგან, რეკურსიული ფუნქციით MakeTree.

ამ ფუნქციის F# კოდი:

```
let makeTree nodeCount density =
if nodeCount < 1 then invalidArg "nodeCount""out of range"
if not (0.0 < density && density <= 1.0) then invalidArg
"density""out of range"

letrec makeTreeUtil nodeCount density offset (r:Random) =
let flip1 = r.NextDouble() > density
let flip2 = r.NextDouble() > density
let newCount = nodeCount - 1
let count1, count2 =
let c = if flip1 && flip2 then newCount / 2 elif flip1 then
newCount else 0
if r.NextDouble()>0.5 then c, newCount - c else newCount - c, c
let l = if count1 > 0 then makeTreeUtil count1 density (offset +
1) r else Leaf
let r = if count2 > 0 then makeTreeUtil count2 density (offset + 1
+ count1) r else Leaf
        Node(l, offset.ToString(), r)
        makeTreeUtil nodeCount density 0 (new Random())
```

სადაც, nodeCount არის დარჩენილი ელემენტების რაოდენობა, Offset - მიმდინარე ელემენტის მნიშვნელობა. r - შემთხვევითი რიცხვების გენერატორი, რის მეშვეობითაც გამოითვლება ხისქვეშა მიმართულება. როგორც ჩანს, მათი კოდები რეკურსიულად გამოიძახება მანამდე, სანამ დარჩენილი ასაწყობი ელემენტების რაოდენობა 0-მდე არ მივა.

2. ხეზე ახალი სიტყვის დამატების მეთოდი

ავღწეროთ მეთოდი, რომელიც ამატებს ხეში ახალ სიტყვას. მეთოდში ვაბრუნებთ გადმოცემულ კვანძს როგორც შედეგს, კონვეინული სისტემის რეალიზაციის მიზნით.

თავიდან ვეძებთ ადგილს, სადაც უნდა ჩავსვათ ახალი სიტყვა. ვთქვათ, გვაქვს სიტყვების მიმდევრობა, რომლებიც უკვე არიან დამატებულები ხეში: apple, application, appearance და დავამატოთ სიტყვა „apple-store“. იმის შემდეგ, როცა გამოვიძახებთ მეთოდს FindNode, ჩვენ დაგვიბრუნდება კვანძი ასოთი „e“ სიტყვიდან „apple“ და სტრიქონი „-store“. რადგან სტრიქონის ნაშთი „-store“ არ არის ცარიელი, დავამატებთ შვილობილ სიაში კიდევ ერთ კვანძს, რომელსაც შევქმნით სტრიქონის ნაშთისთვის კონსტრუქტორის გამოძახებით. კონსტრუქტორი შექმნის ჯაჭვს ასოებისგან -_s_t_o_r_e. იმისთვის, რომ ხე დარჩეს დალაგებული ანბანის მიხედვით, ახალი შვილებისთვის ვიყენებ ფუნქციას List.SortBy. თუ სიტყვის ნაშთი გახდა ცარიელი, მაშინ ვანიჭებთ ბოლო კვანძს სიტყვის დამთავრების ნიშანს isword <- true. ფუნქცია ასე განისაზღვრება:

ISSN 1512-1232

```
member node.Add (str : string) =
    let (n, s) = node.FindNode(str)
    match s.Length with
    | 0 -> n.isword <- true
    | _ -> n.childs <- List.sortBy (fun n -> n.letter) (new Node
(s) :: n.childs)
     node
```

შემდეგი ფუნქცია არის ფუნქცია, რომელიც კითხულობს სიტყვების მიმდევრობას კონსოლიდან. ყოველი სტრიქონი აღიქვება როგორც ცალკე სიტყვა. ყველა შეყვანილი სიტყვიდან ვადგენთ სიას, სანამ მომხმარებელი არ შეიყვანს ცარიელ სტრიქონს. ცარიელი სტრიქონი სიაში არ ემატება.

```
let rec readList () =
    match stdin.ReadLine() with
    | w when w.Length = 0 -> []
    | w -> w :: readList()
```

მომხმარებელსა და სისტემას შორის დიალოგს აქვს შემდეგი სახე:

```
Enter source list of words:
apple
application
apple-store
appearance

Find : app
appearance
apple
apple-store
application

Find : appl
apple
apple-store
application

Find : appli
application

Find : finger
```

3. ხეზე ძებნის „ტალღური მეთოდი“

ინტერნეტის პარადოქსია ის, რომ ყოველ წამს ინფორმაცია ხდება სულ უფრო და უფრო მეტი, მაგრამ საჭირო ინფორმაციის მოძიება – სულ უფრო რთულდება. ინფორმაციის მოძიება ერთ–ერთი ყველაზე გავრცელებული და ამავე დროს ყველაზე რთული ამოცანაა, რომელსაც ეხება ქსელში ყოველი მომხმარებელი.

ინფორმაციის ძებნის აუცილებლობამ და პრობლემის მნიშვნელობამ შექმნა დარგი – საძიებო სამსახური, რომელიც შეიძლება პირობითად დაიყოს კითხვარებად (directories) და საძიებო სისტემებად (search engines).

იმის გათვალისწინებით, რომ ინფორმაციული ტექნოლოგიები სწრაფად იზრდება, ფართოდ გამოიყენება მრავალპროცესორიანი კომპიუტერები, აქტუალურ

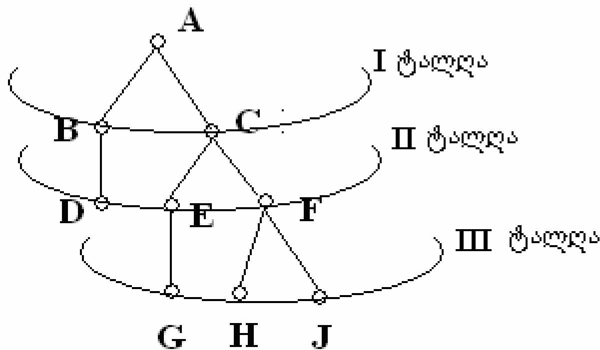
ISSN 1512-1232
მნიშვნელობას იძენს ძებნის თანამედროვე მეთოდები, რომლებიც მრავალპროცესორიანი კომპიუტერებისთვის გამოიყენება.

ჩვენ ყურადღებას გავამახვილებთ "ტალღური ძებნის" მეთოდის პრინციპზე, რომლის რეალიზაცია Lisp-ზე აღწერილია [5]–ში.

3.1 "ტალღური" ძებნა ხეზე

„ტალღური" ძებნის პრინციპი მდგომარეობს შემდეგში: ძებნა იწყება ხის წვეროდან. თუ საძებნი სიდიდე ემთხვევა წვეროზე მოთავსებულ მნიშვნელობას, მაშინ საძიებო პროცესი წარმატებულად დამთავრდება, თუ არა და შედარება გაგრძელდება წვეროდან გამომავალ "პირველ ტალღაზე", ანუ წვეროს შვილობილ წვეროებზე. აქ ხაზი უნდა გაესვას იმას, რომ მიმდინარე ტალღაზე ძებნის პროცესი ხდება პარალელურად. საჭირო წვეროს მოძებნის შემთხვევაში პროცესი მთავრდება, თუ არა და შედარების პროცესი გრძელდება შვილობილი წვეროების შვილობილ წვეროებზე ("მეორე ტალღაზე") ხდება და ა.შ. წარმოდგენილი პროცესი მთავრდება, როცა იქნება მოძებნილი საჭირო წვერო ან როცა მიიღწევა ბოლო წვეროები (ფოთლები).

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.

ამ მეთოდის უპირატესობა მდგომარეობს იმაში, რომ მისი რეალიზაცია შესაძლებელია მრავალპროცესორიან კომპიუტერზე. [1]-ში განხილულია ალგორითმი, სადაც ასეთი ხეები წარმოდგენილია ფუნქციონალური დაპროგრამების ენა Lisp-ის სიების საშუალებით. 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.

F#-ში მეთოდის რეალიზებისთვის გამოიყენება mapcar ფუნქციონალი, რომელსაც შემდეგი სახე აქვს:

(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

ISSN 1512-1232
იქნება მოძებნილი საჭირო წვერო ან როცა მიიღწევა ბოლო წვეროები (ფოთლები). შიდა რეალიზაცია mapcar ფუნქციონალისა F#-ში იყენებს მოცემული კომპიუტერის ყველა პროცესორს.

3.2 „ტალღური ძებნა" ქსელზე

„ტალღური ძებნა" შეიძლება განვაზოგადოდ ქსელებისთვისაც.

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.

ცოდნის სემანტიკური ქსელებით წარმოდგენისთვის ვუდსის [6,7] 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, არ 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 დამუშავება და შედეგის უკან გადაცემა.

მრავალპროცესორიანი კომპიუტერებზე პროგრამირებისას, 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.

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

ISSN 1512-1232

გამოყენებული ლიტერატურა

[1] ნ.არჩვაძე, მ.ფხოველიშვილი, ლ. შეწირული. მონაცემთა წარმოდგენა სიის სტრუქტურებით. "მეცნიერება და ტექნოლოგიები", №7-9, 2008, გვ. 18-24.

[2] Natela Archvadze.Templates Processing Lists in Haskell. Abstracts II International Conference Deducated to the 70th Anniversary of the Georgian National Academy of Sciences & the 120th Birthday of First President Academician Nikoloz (Niko) Muskhelishvili. September 15-19, 2011, Batumi, Georgia. pp. 64. http://rmi.ge/~gmu/II_Annual_Conference/E_II_Annual.htm .

[3] Душкин Р. В. Функциональное программирование на языке Haskell. — М.: ДМК-Пресс, 2007. — 608 стр. — ISBN 5-94074-335-8.

[4] Visual F#. http://msdn.microsoft.com/ru-ru/library/dd233154.aspx

[5] Archvadze N., Pkhovelishvili M., Shetsiruli L. The Methods of the Effective Date Search for the Listed Structures. International Conference on Modern Problems in Applied Mathematics Dedicated to the 90th Anniversary of the Iv/Javakhishvili Tbilisi State University and 40th Anniversary of the I.Vekua Institute of Applied Mathematics. Book of Abstracts. Tbilisi, 2008. p.16. http://www.viam.science.tsu.ge/viam40/tezisebis_krebuli.pdf .

[6] 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.

[7] Вудс В.А. Сетевые грамматики для анализа естественных языков. // Кибернетический сборник. Новая Серия. Вып. 13. М.: Мир, 1978. С. 120-158.

ნაშრომი მიღებულია: 2012-05-12