1. შესავალი

ფუნქციონალური პროგრამირების მიზანია თითოეულ პროგრამას მისცეს მარტივი მათემატიკური ინტერპრეტაცია. ეს ინტერპრეტაცია უნდა იყოს შესრულების დეტალებისგან დამოუკიდებელი და ამავე დროს გასაგები მათთვის, ვისაც არ აქვს სამეცნიერო ხარისხი საგნობრივ არეში.

ლორენს პაულსონი

ფუნქციონალური პროგრამირების აღწერამდე გავიხსენოთ ზოგადად პროგრამირების განვითარების ისტორია.

XX საუკუნის 40–იან წლებში გაჩნდა პირველი ციფრული კომპიუტერები, რომლებიც, როგორც ცნობილია, პროგრამირდებოდა სხვადასხვა რიგის ტუმბლერების, გაყვანილობისა და ღილაკების საშუალებით. ასეთი გადამრთველობის რაოდენობა იყო რამდენიმე ასეული და მკეთრად იზრდებოდა პროგრამის სირთულესთან ერთად. ამიტომ პროგრამირების განვითარების შემდეგი ეტაპი გახდა ასემბლერის ენების შექმნა მარტივი მნემონიკით.

ვერც ასემბლერები გახდა ის ინსტრუმენტები, რომლებსაც ჩვეულებრივი ხალხი გამოიყენებდა. მნემოკოდები ჯერ კიდევ რჩებოდა ძალზე რთული, მით უმეტეს, რომ ყოველი ასემბლერი მყარად იყო მიბმული იმ არქიტექტურასთან, რომელზედაც ის სრულდებოდა.

მორიგი ნაბიჯი ასემბლერის შემდეგ გახდა ეგრეთ წოდებული მაღალი დონის იმპერატიული ენები (BASIC, Pascal, C, Ada და სხვა). ამ ენებს იმპერატიული ეწოდა იმ მარტივი მიზეზის გამო, რომ მათი მთავარი თვისება არის ორიენტირებულობა ინსტრუქციების თანმიმდევრულ შესრულებაზე, რომლებიც ოპერირებენ მეხსიერებაზე (ანუ მინიჭებები) და იტერაციული ციკლები. ფუნქციებისა და პროცედურების გამოძახებას ვერ გამოჰყავდა ასეთი ენები ცხადი იმპერატიულობიდან.

დავუბრუნდეთ ფუნქციონალურ პროგრამირებას... ფუნქციონალური პარადიგმის ქვაკუთხედს წარმოადგენს ფუნქცია. თუ ჩვენ გავიხსენებთ მათემატიკის ისტორიას, შეიძლება შევაფასოთ ცნება „ფუნქციის“ ასაკი. იგი უკვე ოთხასი წლისაა. მათემატიკამ ფუნქციებთან ოპერირებისთვის მოიგონა უამრავი რაოდენობა თეორიული და პრაქტიკული აპარატი დაწყებული ჩვეულებრივი დიფერენცირებიდან და ინტეგრირებიდან, დამთავრებული ფუნქციონალური ანალიზით, არამკაფიო სიმრავლეთა თეორიითა და კომპლექსური ცვლადის ფუნქციებით.

მათემატიკური ფუნქციები გამოხატავს კავშირს პროცესის პარამეტრებსა (შესასვლელი) და შედეგს (გამოსასვლელი) შორის. რადგან გამოთვლა ასევე

პროცესია, რომელსაც აქვს შესასვლელი და გამოსასვლელი, ფუნქცია სავსებით შესაფერისი და ადექვატური საშუალებაა გამოთვლების აღსაწერად. სწორედ ეს მარტივი პრინციპი არის ჩადებული ფუნქციონალური პარადიგმისა საფუძველში და ფუნქციონალური სტილით პროგრამირებაში. ფუნქციონალური პროგრამა წარმოადგენს ფუნქციების განსაზღვრებების ერთობლიობას. ფუნქცია განისაზღვრება სხვა ფუნქციებით, ან რეკურსიულად – თავისი თავით. პროგრამის შესრულების მომენტში ფუნქცია იღებს პარამეტრებს, ითვლის და აბრუნებს შედეგს, საჭიროების შემთხვევაში, ითვლის სხვა ფუნქციის მნიშვნელობებსაც. ფუნქციონალურ ენაზე პროგრამირებისას პროგრამისტი არ აღწერს გამოთვლების თანმიმდევრობას. მისთვის აუცილებელია მხოლოდ აღწეროს სასურველი შედეგი ფუნქციების სისტემის სახით.

ავღნიშნოთ, რონ ფუნქციონალური პროგრამირებამ, ისევე როგორც ლოგიკურმა პროგრამირებამ დიდი გამოყენება ჰპოვა ხელოვნურ ინტელექტსა და მის დანართებში. გავეცნოთ ფუნქციონალური პროგრამირების ისტორიას და ამოცანებს.

ფუნქციონალური პროგრამირების ისტორია

როგორც ცნობილია, იმპერატიულ პროგრამირებას თეორიული საფუძველი ჩაეყარა ჯერ კიდევ XX საუკუნის 30–იან წლებში ალან ტიურინგისა და ჯონ ფონ ნეიმანის მიერ. თეორია, რომელიც ფუნქციონალური მიდგომის საფუძველია, ასევე დაიბადა 20–იან – 30–იან წლებში. მათ შორის, ვინც შეიმუშავა ფუნქციონალური პროგრამირების მათემატიკური საფუძვლები, შეიძლება დავასახელოთ მოზეს შენფინკელი (გერმანია და რუსეთი) და ჰასკელ ლარი (ინგლისი), რომელმაც კომბინატორული ლოგიკა დაამუშავა, ასევე ალონზო ჩერჩი (აშშ), რომელმაც შექმნა λ აღრიცხვა.

თეორია რჩებოდა თეორიად, სანამ წინა საუკუნის 50–იანი წლების დასაწყისში ჯონ მაკარტმა არ შეიმუშავა ენა Lisp, რომელიც გახდა პირველი ფუნქციონალური ენა და წლების განმავლობაში რჩებოდა ერთადერთად. ენა Lisp დღემდე გამოიყენება (მაგალითად, FORTRAN–ის მსგავსად), თუმცა ველარ აკმაყოფილებს ზოგიერთ თანამედროვე მოთხოვნას, რაც აიძულებს პროგრამების შემქმნელებს დიდი ძალისხმევა გადაიტანონ კომპილერზე. ამის აუცილებლობას იწვევს სულ უფრო მზარდი სირთულის პროგრამული უზრუნველყოფა.

ამ გარემოების გამო დიდ როლს თამაშობს ტიპიზაცია. XX საუკუნის 70–იანი წლების ბოლოსა და 80–იანი წლების დასაწყისში ინტენსიურად მუშავდებოდა ფუნქციონალური პროგრამირების შესაბამისი ტიპიზაციის მოდელები. მათი უმრავლესობა შეიცავს ისეთ ძლერ მექანიზმებს, როგორიცაა მონაცემთა აბსტრაქცია და პოლომორფიზმი. გაჩნდა მთელი რიგი ტიპიზირებული ფუნქციონალური ენები:

ML, Scheme, Hope, Miranda, Clean და ბევრი სხვა. დამატებით, მუდმივად იზრდებოდა დიალექტების რაოდენობაც.

და დადგა სიტუაცია, რომ პრაქტიკულად ყველა ჯგუფი, ვინც ფუნქციონალურ პროგრამირებაში მუშაობდა, იყენებდა საკუთარ ენას. ამან ხელი შეუშალა ასეთი ენების შემდგომ გავრცელებას და გააჩინა მთელი რიგი პრობლემები. ფუნქციონალური პროგრამირების სფეროში წამყვანი მკვლევარების გაერთიანებულმა ჯგუფმა სიტუაციის გამოსწორების მიზნით გადაწყვიტა სხვადასხვა ენის ღირსებები გამოეყენებინათ ახალი უნივერსალური ფუნქციონალური ენის შექმნისას. ასეთი ენის პირველი რეალიზაცია, რომელსაც დაერქვა Haskell ჰასკელ კარის საპატივსაცემოდ, შეიქმნა 90–იანი წლების დასაწყისში. ამჟამად ფუნქციონირებს სტანდარტი Haskell-98.

ფუნქციონალური ენების უმრავლესობისთვის, Lisp–ის ტრადიციებიდან გამომდინარე, რეალიზებულია ინტერპრეტატორი. ინტერპრეტორები მოსახერხებელია პროგრამის სწრაფი გამართვისთვის. ამ დროს კომპილაციის გრძელი პროცესი გამოიტოვება, რითაც ჩქარდება დამუშავების ჩვეულებრივი ციკლი. თუმცა, მეორეს მხრივ, ინტერპრეტატორები, კომპილერებთან შედარებით რამდენჯერმე აგებენ კოდის შესრულების სიჩქარეში. ამიტომაც, ინტერპრეტატორის გვერდით არსებობს კომპილერები, რომლებიც გენერირებენ მანქანურ კოდს (მაგალითად, Objective Caml) ანდა C/C++ კოდს (მაგალითად, Glasgow Haskell Compiler). ნიშანდობლივია ის, რომ პრაქტიკულად ყველა კომპილერი რეალიზებულია თვით ამ ენაზე.

ავღნიშნოთ, რომ ქვემოთ მოყვანილ ფუნქციონალური პროგრამირების მაგალითებში გამოვიყენებთ ან აბსტრაქტული ფუნქციონალური ენას, რომელიც ახლოა მათემატიკურ ნოტაციასთან, ან Haskell-ს, რომლის უფასო კომპილერები შეიძლება გადმოწერილი იყოს გვერდიდან www.haskell.org.

ფუნქციონალური ენების თვისებები

შეიძლება მოკლედ ჩამოვთვალოთ ფუნქციონალური ენების ძირითადი თვისებები:

- მოკლე და მარტივი კოდი;
- მკაცრი ტიპიზაცია;
- მოდულირება;
- ფუნქცია – ეს მნიშვნელობაა;
- სისუფთავე (გვერდითი ეფექტების არარსებობა);
- გადატანილი (ზარმაცი) გამოთვლები.

მოკლე და მარტივი კოდი

პროგრამა ფუნქციონალურ ენაზე საზოგადოდ უფრო მოკლეა და მარტივი, ვიდრე იგივე პროგრამა იმპერატიულ ენაზე. შევადაროთ პროგრამები C–ზე და აბსტრაქტულ ფუნქციონალურ ენაზე ჰოარეს სწრაფი დახარისხების მაგალითზე. (ეს მაგალითი გახდა კლასიკური მაგალითი ფუნქციონალური ენების უპირატესობების საჩვენებლად).

მაგალითი 1. ჰოარეს სწრაფი დახარისხება C–ზე.

```
void quickSort (int a[], int l, int r)
{
	int i = l;
	int j = r;
	int x = a[(l + r) / 2];
	do
	{
		while (a[i] < x) i++;
		while (x < a[j]) j--;
		if (i <= j)
		{
			int temp = a[i];
			a[i++] = a[j];
			a[j--] = temp;
		}
	}
	while (i <= j);
	if (l < j) quickSort (a, l, j);
	if (i < r) quickSort (a, i, r);
}
```

მაგალითი 2. ჰოარეს სწრაფი დახარისხება აბსტრაქტულ ფუნქციონალურ ენაზე.

```
quickSort ([]) = []
quickSort ([h : t]) = quickSort (n | n   t, n <= h) + [h] +
quickSort (n | n   t, n > h)
```

მაგალითი 2 შეიძლება წავიკითხოთ ასე:

1. თუ სია ცარიელია, მაშინ შედეგიც იქნება ცარიელი სია.

2. წინააღმდეგ შეთხვევაში (ანუ სია როცა არ არის ცარიელი) გამოიყოფა თავი (პირველი ელემენტი) და კუდი (დარჩენილი ელემენტების სია, რომელიც შეიძლება იყოს ცარიელი). ამ შემთხვევაში შედეგი იქნება სამი სიის კონკატენაცია. პირველი არის დახარისხებული სია, რომელიც შედგება კუდის (t) ყველა იმ ელემენტისგან, რომლებიც ნაკლებია ან ტოლი თავის (h). მეორე არის ერთელემენტიანი სიას, რომელიც შეიცავს თავს (h), ხოლო მესამე არის დახარისხებული სია, რომელიც კუდის (t) ყველა ელემენტისგან შედგება, რომელიც მეტია თავზე (h).

მაგალითი 3. ჰოარეს სწრაფი დახარისხება ენა Haskell–ზე.

```
quickSort [] = []
quickSort (h : t) = quickSort [y | y <- t, y < h] ++ [h] ++
quickSort [y | y <- t, y >= h]
```

ამ მარტივ მაგალითზეც ჩანს, თუ როგორ იგებს ფუნქციონალური პროგრამირების სტილი როგორც კოდის რაოდენობაში, ასევე მის ელეგანტურობაში.

ამას გარდა, ყველა ოპერაცია მეხსიერებასთან სრულდება ავტომატურად. ნებისმიერი ობიექტის შექმნისას მას ავტომატურად გამოეყოფა მეხსიერება. მას შემდეგ, რაც ობიექტი თავის დანიშნულებას შეასრულებს, ის ავტომატურადვე განადგურდება დამლაგებლის მიერ, რომელიც არის ნებისმიერი ფუნქციონალური ენის ნაწილი.

კიდევ ერთი სასარგებლო თვისება, რომელიც იძლევა პროგრამის შემცირების საშუალებას, არის ნიმუშთან შედარების მექანიზმი. ეს იძლევა საშუალებას აღიწეროს ფუნქცია, როგორც ინდუციური განსაზღვრება. მაგალითად:

მაგალითი 4. ფიბონაჩის N–ური რიცხვის განსაზღვრა.

```
fibb (0) = 1
fibb (1) = 1
fibb (N) = fibb (N – 2) + fibb (N – 1)
```

როგორც ჩანს, ფუნქციონალური ენები ადის უფრო მაღალ აბსტრაქტულ დონეზე, ვიდრე ტრადიციული იმპერატიული ენები.

მკაცრი ტიპიზაცია

პრაქტიკულად ყველა თანამედროვე პროგრამირების ენა წარმოადგენს ტიპიზირებულ ენას (შესაძლებელა JavaScript–ისა და მისი დიალექტების გამოკლებით. არ არსებობს იმპერატიული ენა, რომელშიც არ იყოს ცნება „ტიპი“). მკაცრი ტიპიზაცია უზრუნველყოფს უსაფრთხოებას. პროგრამა, რომელიც ამოწმებს

ტიპებს, არ შეწყდება ოპერციული სისტემის შეტყობინებით "access violation". ეს განსაკუთრებით ეხება ისეთ ენებს, როგორიცაა C/C++ და Object Pascal, სადაც მიმთითებლების გამოყენება ხშირად ხდება. ფუნქციონალურ ენებში შეცდომების დიდი ნაწილის გასწორება ხდება კომპილაციის ეტაპზე, ამიტომ გამართვის სტადია და პროგრამის დამუშავების მთლიანი დრო მცირდება. და კიდევ, მკაცრი ტიპიზაცია კომპილერს აძლევს უფრო ეფექტური კოდის გენერირების საშუალებას და ამით აჩქარებს პროგრამის შესრულების დროს.

თუ განვიხილავთ ჰოარეს სწრაფი დახარისხების მაგალითს, შეიძლება დავინახოთ, რომ უკვე ნახსენებ განსხვავებების გარდა C ენაზე ვარიანტსა და აბსტრაქტულ ფუნქციონალურ ენაზე ვარიანტს შორის, არის კიდევ ერთი ძირითადი განსხვავება: ფუნქცია C–ზე ახდენს int ტიპის (მთელი რიცხვების) დახარისხებას, ხოლო აბსტრაქტულ ფუნქციონალურ ენაზე – ნებისმიერი ტიპის მნიშვნელობების სიის, რომელიც ეკუთვნის დალაგებული სიდიდეების კლასს. ამიტომ, ბოლო ფუნქციას შეუძლია დაახარისხოს მთელი რიცხვების სია, ასევე ნამდვილი რიცხვების სია და სტრიქონების სია. შეიძლება ავღწეროთ ახალი ტიპი. მისთვის განვსაზღვროთ შედარების ოპერაცია და შემდეგ გამოვიყენოთ ხელახალი კომპილაციის გარეშე quickSort ამ ახალი ტიპის მნიშვნელობების სიისთვისაც. ამ სასარგებლო თვისებას ეწოდება პარამეტრული ანუ ჭეშმარიტი პოლიმორფიზმი და მას მხარს უჭერს ფუნქციონალური ენების უმრავლესობა.

პოლიმორფიზმის კიდევ ერთი სახეობაა ფუნქციების გადატვირთვა, რომელიც სხვადასხვა ფუნქციებს, მაგრამ რაღაცით მსგავსს, აძლევს ერთიდაიგივე სახელებს. გადატვირთული ოპერაციის ტიპიური მაგალითია შეკრების ოპერაცია. მთელი რიცხვებისა და ნამდვილი რიცხვების შეკრების ფუნქციები სხვადასხვაა, მაგრამ მოხერხებულობისთვის ისინი ატარებენ ერთიდაიგივე სახელს. ზოგიერთი ფუნქციონალური ენა, პოლიმორფიზმის გარდა მხარს უჭერს ოპერაციების გადატვირთვასაც.

ენა C++ არის ისეთი ცნება, როგორიცაა შაბლონი, რომელიც იძლევა საშუალებას განისაზრვროს პოლომორფული ფუნქციები, მსგავსი quickSort–ის. C++–ის სტანდარტულ ბიბლიოთეკაში STL შედის ასეთი ფუნქცია და კიდევ მრავალი სხვა პოლიმორფული ფუნქცია. მაგრამ C++–ის შაბლონები და Ada–ს გვაროვნული ფუნქციები, ბადებს გადატვირთული ფუნქციების სიმრავლეს, რომლებსაც კომპილერი ყოველ ჯერზე აკომპილირებს, რაც უარყოფითად მოქმედებს კომპილაციის დროსა და კოდის ზომაზე. ხოლო ფუნქციონალურ ენებში პოლიმორფული ფუნქცია quickSort – ეს ერთი, ერთადერთი ფუნქციაა.

ზოგიერთი ენაში, მაგალითად, Ada–ში მკაცრი ტიპიზაცია ითხოვს პროგრამისტისგან ცხადად აღწეროს ყველა მნიშვნელობის და ფუნქციის ტიპი. ამის თავიდან ასაცილებლად, მკაცრად ტიპიზირებულ ფუნქციონალურ ენებში ჩადგმულია სპეციალური მექანიზმი, რომელიც კომპილერს საშუალებას აძლევს განსაზღვროს კონსტანტის, გამოსახულების და ფუნქციის ტიპი კონტექსტიდან

გამომდინარე. ამ მექანიზმს უწოდებენ ტიპების გამოყვანის მექანიზმს. ცნობილია რამდენიმე ასეთი მექანიზმი, თუმცა მათი უმრავლესაბა წარმოადგენს სახესხვაობებს **ჰინდლი–მილნერის** ტიპიზაციის მოდელისა, რომელიც XX საუკუნის 80–იანი წლების დასაწყისში დამუშავდა. ამრიგად, უმრავლეს შემთხვევებში შეიძლება არ მივუთითოთ ფუნქციის ტიპი.

მოდულირება

მოდულირების მექანიზმი საშუალებას იძლევა პროგრამა დავყოთ რამდენიმე დამოუკიდებელ ნაწილად (მოდულად) მათ შორის მკვეთრად განსაზღვრული კავშირებით. ამით მარტივდება დიდი პროგრამული სისტემების პროექტირებისა და შემდგომი მხარდაჭერის პროცესები. მოდულირების მხარდაჭერა არ წარმოადგენს კონკრეტულად ფუნქციონალური ენების თვისებას, თუმცა მას მხარს უჭერს ფუნქციონალური ენების უმრავლესობა. არსებობს ძალზე განვითარებული მოდულური იმპერატიული ენები. ასეთი ენებია, მაგალითად, Modula-2 და Ada-95.

ფუნქცია – ეს მნიშვნელობაა

ფუნქციონალურ ენებში (ისევე, როგორც, საზოგადოდ, პროგრამირებასა და მათემატიკაში) ფუნქციები შეიძლება გადაეცეს სხვა ფუნქციებს არგუმენტად ან დაბრუნდეს როგორც შედეგი. ფუნქციებს, რომლებიც ფუნქციონალურ არგუმენტებს იღებს, უწოდებენ მაღალი რიგის ფუნქციებს ანუ ფუნქციონალებს. ყველაზე ცნობილი ფუნქციონალი არის ფუნქცია map. ეს ფუნქცია იყენებს მოცემულ ფუნქციას სიის ყველა ელემენტთან და აფორმირებს შედეგად სხვა სიას. მაგალითად, განვსაზღვროთ ფუნქცია, რომელსაც აჰყავს მთელი რიცხვი კვადრატში, ასე:

```
square (N) = N * N
```

შეიძლება გამოვიყენოთ ფუნქცია map ნებისმიერი სიის ყველა ელემენტის კვადრატში ასაყვანად:

```
squareList = map (square, [1, 2, 3, 4])
```

ამ ინსტრუქციის შესრულების შედეგი იქნება სია [1, 4, 9, 16].

```
squareList x = map (square, x )
```

სისუფთავე (გვერდითი ეფექტების არ არსებობა)

იმპერატიულ ენებში ფუნქციამ მისი შესრულების პროცესში, შეიძლება წაიკითხოს ან შეცვალოს გლობალური ცვლადების მნიშვნელობები და შეასრულოს შეტანა–გამოტანის ოპერაციები. ამიტომ, თუ გამოვიძახებთ ერთიდაიგივე ფუნქციას ორჯერ ერთიდაიგივე არგუმენტებით, შეიძლება მოხდეს, რომ შედეგად გამოითვალოს ორი სხვადასხვა მნიშვნელობა. ასეთ ფუნქციას უწოდებენ ფუნქციას გვერდითი ეფექტებით.

ფუნქციის აღწერა გვერდითი ეფექტების გარეშე პრაქტიკულად შესაძლებელია ყველა ენაში, მაგრამ ზოგიერთი ენა მხარს უჭერს, ითხოვს გვერდით ეფექტებს. მაგალითად, მრავალ ობიექტ–ორიენტირებულ ენაში კლასის წევრ ფუნქციას გადაეცემა ფარული არგუმენტი (ხშირად მას უწოდებენ `this` ან `self`), რომელსაც ეს ფუნქცია არაცხადად მოდიფიცირებს.

წმინდა ფუნქციონალურ ენაში მინიჭების ოპერატორი არ არსებობს. ობიექტები არ შეიძლება შეიცვალოს და განადგურდეს, შესაძლოა მხოლოდ ახალი შეიქმნას არსებულების დეკომპოზიციითა და სინთეზით. არასაჭირო ობიექტებზე ზრუნავს ენაში ჩადგმული დამლაგებელი. ამის წყალობით წმინდა ფუნქციონალურ ენებში ყველა ფუნქცია თავისუფალია გვერდითი ეფექტებისგან. თუმცა, ეს არ უშლის ხელს მოხდეს ამ ენებში ზოგიერთი სასარგებლო იმპერატიული თვისების იმიტირება, ისეთის, როგორიცაა გამონაკლისი სიტუაცია და მასივები. ამისთვის არსებობს სპეციალური მეთოდები.

რა უპირატესობა აქვთ წმინდა ფუნქციონალურ ენებს? გარდა პროგრამების გამარტივებული ანალიზისა, არსებობს კიდევ ერთი ძლიერი უპირატესობა – პარალელიზმი. ვინაიდან ფუნქცია გამოთვლისას იყენებს მხოლოდ თავის პარამეტრებს, ჩვენ შეგვიძლია გამოვთვალოთ დამოუკიდებელი ფუნქციები ნებისმიერი რიგით ან პარალელურად, ეს შედეგზე ასახვას ვერ ჰპოვებს. ამასთან, პარალელიზმი შეიძლება განხორციელდეს არა მხოლოდ ენის კომპილატორის დონეზე, არამედ არქიტექტურის დონეზეც. ზოგიერთ ლაბორატორიაში უკვე შემუშავებულია და გამოიყენება ექსპერიმენტალური კომპიუტერები, რომლებიც მსგავს არქიტექტურას ეყრდნობა. მაგალითისთვის შეიძლება დავასახელოთ `Lisp`-მანქანა.

გადატანილი გამოთვლები

ტრადიციულ პროგრამიების ენებში (მაგალითად,`C++`–ში) ფუნქციის გამოძახება იწვევს ყველა არგუმენტის გამოთვლას. ფუნქციის გამოძახების ამ მეთოდს უწოდებენ გამოძახებას–მნიშვნელობით. თუ ფუნქციაში რომელიღაც არგუმენტი არ

გამოიყენება, მაშინ გამოთვლის შედეგი იკარგება. აქედან გამომდინარე, გამოთვლები ამაოდ ჩატარდა. რაღაც აზრით, მნიშვნელობით გამოძახების საწინააღმდეგოა გამოძახება საჭიროების მიხედვით. ამ შემთხვევაში არგუმენტი გამოიძახება, თუ საჭიროა შედეგის გამოთვლისათვის. ასეთი გამოთვლების მაგალითად შეიძლება დავასახელოთ კონიუქციის ოპერატორი (&&) C++–დან, რომელიც არ ითვლის მეორე არგუმენტს, თუ პირველ არგუმენტს აქვს მცდარი მნიშვნელობა.

თუ ფუნქციონალური ენა მხარს არ უჭერს გადატანილ გამოთვლებს, მას უწოდებენ მკაცრ ენას. მართლაც, ასეთ ენებში გამოთვლების რიგი მკაცრად არის განსაზღვრული. მკაცრი ენების მაგალითად შეიძლება დავასახელოთ Scheme, Standard ML და Caml.

ენებს, რომლებიც იყენებენ გადატანილ გამოთვლებს, უწოდებენ არამკაცრს. Haskell — არამკაცრი ენაა, ისევე, როგორც Gofer და Miranda. არამკაცრი ენები ამასთანავე არის სუფთა.

ძალზე ხშირად მკაცრი ენები შეიცავს ზოგიერთი ისეთი შესაძლებლობების მხარდაჭერას, რაც ახასიათებს არამკაცრ ენებს. მაგალითად, უსასრულო სიებს. Standard ML–ში სპეციალური მოდულია, რომელიც გადატანილ გამოთვლებს უჭერს მხარს. ხოლო Objective Caml, ამის გარდა, შეიცავს დარეზერვირებულ სიტყვა lazy და კონსტრუქციას მნიშვნელობათა სიისთვის, რომელიც გამოითვლება აუცილებლობის მიხედვით.

ამოსახსნელი ამოცანები

ფუნქციონალური პროგრამირების კურსებში, ტრადიციულად განხილული ამოცანებიდან, შეიძლება გამოვყოთ შემდეგი:

1. დარჩენილი პროცედურის მიღება

თუ მოცემულია შემდეგი ობიექტები:

$P\ (x_1, x_2, \ldots, x_n)$ — რაღაც პროცედურა.

$x_1 = a_1,\ x_2 = a_2$ — პარამეტრების ცნობილი მნიშვნელობები.

$x_3, \ldots, x_n$ — პარამეტრების უცნობი მნიშვნელობები.

მოითხოვება დარჩენილი პროცედურის მიღება $P1\ (x_3, \ldots, x_n)$. ეს ამოცანა იხსნება პროგრამების მხოლოდ ვიწრო კლასისთვის.

2. ფუნქციის მათემატიკური აღწერის მიღება

ვთქვათ, გვაქვს P პროგრამა. მისთვის განსაზღვრულია შესასვლელი მნიშვნელობები და გამოსასვლელი მნიშვნელობები. მოითხოვება აიგოს ფუნქციის მათემატიკური აღწერა

$f : Dx_1, \ldots, Dx_n \rightarrow Dy_1, \ldots, Dy_m.$

3. პროგრამირების ენის სემანტიკის ფორმალური აღწერა.
4. მონაცემთა დინამიური სტრუქტურების აღწერა.
5. პროგრამის „მნიშვნელოვანი“ ნაწილის აგება მონაცემთა სტრუქტურის აღწერით, რომლებსაც ამუშავებს ასაგები პროგრამა.
6. პროგრამის ზოგიერთი თვისების არსებობის დამტკიცება.
7. პროგრამების ექვივალენტური ტრანსფორმაცია.

ყველა ეს ამოცანა საკმაოდ მარტივად იხსნება ფუნქციონალური პროგრამირების საშუალებებით, მაგრამ პრაქტიკულად გადაუწყვეტია იმპერატიულ ენებზე.

საცნობარო მასალა

ფუნქციონალური პროგრამირების ენები

მოვიყვანოთ ზოგიერთი ფუნქციონალური ენის მოკლე აღწერა. დამატებითი ინფორმაცია იხილეთ ქვემოთ ჩამოთვლილ რესურსებში.

Lisp (List processor). ითვლება, რომ არის პირველი ფუნქციონალური ენა. არატიპიზირებულია. შეიცავს იმპერატიულ თვისებებსაც, თუმცა საზოგადოდ წაახალისებს ფუნქციონალური პროგრამირების სტილს. გამოთვლებისას გამოიყენება გამოძახება მნიშვნელობით. არსებობს ენის ობიექტ–ორიენტირებული დიალექტი CLOS.

ISWIM (If you See What I Mean). ფუნქციონალური ენა–პროტოტიპი. დამუშავებულია ლანდიმის მიერ XX საუკუნის 60-იან წლებში იმის სადემონსტრაციოდ, თუ როგორი შეიძლება იყოს ფუნქციონალური ენა. ენასთან ერთად ლანდინმა შექმნა სპეციალური ვირტუალური მანქანა, რომელიც ასრულებდა პროგრამებს ISWIM–ზე. ამ ვირტუალურმა მანქანამ, რომელიც მუშაობდა მნიშვნელობით გამოძახებაზე, მიიღო სახელწოდება SECD-მანქანისა. ISWIM ენის სინტაქსს იყენებს მრავალი ფუნქციონალური ენა. ISWIM სინტაქს ჰგავს ML–ის სინტაქსი, განსაკუთრებით Caml–ის სინტაქსი.

Scheme. Lisp–ის დიალექტი, რომელიც დანიშნულია სამეცნიერო კვლევებისთვის computer science დარგში. Scheme–ის შექმნისას ყურადღება გამახვილდა ელეგენტურობასა და სიმარტივეზე. ამის გამო ენა უფრო პატარა გამოვიდა, ვიდრე Common Lisp.

ML (Meta Language). მკაცრი ენების ოჯახი განვითარებული ტიპების პოლიმორფული სისტემით და პარამეტრიზირებული მოდულებით. ML ისწავლება დასავლეთის მრავალ უნივერსიტეტში (ზოგან, პროგრამირების პირველ ენადაც კი).

Standard ML. ერთ–ერთი პირველი ტიპიზირებული ფუნქციონალური პროგრამირების ენაა. შეიცავს ზოგიერთ იმპერატიულ თვისებას, ისეთს, როგორიცაა მიმთითებლები (გამოიყენება მნიშვნელობების შესაცვლელად) და ამიტომაც, არ არის სუფთა ენა. ძალზე საინტერესოდ არის რეალიზებული მოდულურობა. გამოთვლებისას იყენებს გამოძახებას მნიშვნელობით. აქვს ტიპების ძლიერი პოლიმორფული სისტემა. ენის ბოლო სტანდარტია Standard ML-97, რომლისთვისაც არსებობს სინტაქსის, ასევე ენის სტატიკური და დიმანიკური სემანტიკის ფორმალური მათემატიკური აღწერები.

Caml Light და **Objective Caml.** როგორც Standard ML მიეკუთვნება ML ოჯახს. Objective Caml განსხვავდება Caml Light–გან იმით, რომ მხარს უჭერს კლასიკურ ობიექტ–ორიენტირებულ პროგრამირებას. როგორც Standard ML, ისიც მკაცრია, მაგრამ აქვს ჩადგმული მექანიზმი გადატანილი გამოთვლებისთვის.

Miranda. შემუშავებულია დევიდ ტერნეტის მიერ, როგორც სტანდარტული ფუნქციონალური ენა, რომელიც იყენებს გადატანილ გამოთვლებს. აქვს მკაცრი ტიპების პოლიმორფული სისტემა. მსგავსად ML-ისა, ისწავლება მრავალ უნივერსიტეტში. დიდი ზეგავლენა იქონია Haskell ენის მკვლევარებზე.

Haskell. ერთ–ერთი ყველაზე გავრცელებული არამკაცრი ენა. აქვს ტიპიზაციის ძალზე განვითარებული სისტემა. უფრო ცუდად მუშაობს მოდულებთან. ენის ბოლო სტანდარტია — Haskell 98.

Gofer (GOod For Equational Reasoning). Haskell–ის გამარტივებული დიალექტი. გამიზნულია ფუნქციონალური პროგრამირების შესასწავლად.

Clean. სპეციალურად დანიშნულია პარალელური და დანაწილებული პროგრამირებისთვის. სინტაქსით მიაგავს Haskell–ს. სუფთაა. იყენებს გადატანილ გამოთვლებს. კომპილატორთან ერთად მოყვება ბიბლიოთეკები (I/O libraries), რომელიც იძლევა საშუალებას დაპროგრამდეს გრაფიკული ინტერფეისის Win32–თვის ან MacOS–თვის.

Internet-რესურსები ფუნქციონალურ პროგრამირებაში

www.haskell.org — საიტი, რომელიც ეხება ფუნქციონალურ პროგრამირებას ზოგადად და ენა Haskell – კონკრეტულად. შეიცავს სხვადასხვა საცნობარო

ინფორმაციას, ინტერპრეტატორების სიას და Haskell–ის კომპილერს (ამჟამად, ყველა ინტერპრეტატორი და კომპილერი არის უფასო). ამას გარდა, საიტზე ნახავთ უამრავი რესურსის მისამართს ფუნქციონალური პროგრამირების თეორიისა და სხვა ენების შესახებ (Standard ML, Clean).

cm.bell-labs.com/cm/cs/what/smlnj – Standard ML of New Jersey. ძალზე კარგი კომპილერია. უფასო დისტრიბუციაში კომპილერის გარდა ასევე შედის MLYacc და MLLex უტილიტები და ბიბლიოთეკა Standard ML Basis Library. შეიძლება გაეცნოთ კომპილერისა და ბიბლიოთეკის დოკუმენტაციასაც.

www.harlequin.com/products/ads/ml/–Harlequin MLWorks, კომერციული კომპილერი Standard ML–ის. თუმცა, არაკომერციული მიზნით შესაძლოა ისარგებლოთ უფასო, ცოტათი შეზღუდული ვერსიით.

caml.inria.fr – ინსტიტუტი INRIA. საშინაო საიტი ენების Caml Light–ის და Objective Caml–ის მკვლევართა ჯგუფის. შესაძლოა უფასოდ გადმოქაჩოთ Objective Caml, რომელიც შეიცავს ინტერპრეტატორს, კომპილერს ბაიტ–კოდში და მანქანურ კოდში, Yacc და Lex Caml–ისთვის, გამმართველს და პროფაილერს, დოკუმენტაციას, მაგალითებს. კომპილირებული კოდის ხარისხი ამ კომპილერს აქვს ძალზე კარგი, სიჩქარით სჯობნის Standard ML–აც კი New Jersey–დან.

www.cs.kun.nl/~clean/ – შეიცავს ენა Clean–ის კომპილატორის დისტრიბუციას. ეს კომპილერი არის კომერციული, მაგრამ უშვებს უფასო გამოყენებას არაკომერციული მიზნებისთვის. იქიდან გამომდინარე, რომ კომპილერი არის ფასიანი, გამომდინარეობს მისი ხარისხი (ძალზე სწრაფია), აქვს დამუშავების გარემო, მოყვება კარგი დოკუმენტაცია და სტანდარტული ბიბლიოთეკები.

ლიტერატურა

1. Хювёнен Э., Сеппенен И. Мир Lisp'a. В 2-х томах. М.: Мир, 1990.
2. Бердж В. Методы рекурсивного программирования. М.: Машиностроение, 1983.
3. Филд А., Харрисон П. Функциональное программирование. М.: Мир, 1993.
4. Хендерсон П. Функциональное программирование. Применение и реализация. М.: Мир, 1983.
5. Джонс С., Лестер Д. Реализация функциональных языков. М.: Мир, 1991.
6. Henson M. Elements of functional languages. Dept. of CS. University of Sassex, 1990.

7. Fokker J. Functional programming. Dept. of CS. Utrecht University, 1995.
8. Thompson S. Haskell: The Craft of Functional Programming. 2-nd edition, Addison-Wesley, 1999.
9. Bird R. Introduction to Functional Programming using Haskell. 2-nd edition, Prentice Hall Press, 1998.