თემა 2 შესავალი დაპროგრამების ენებში

* დაპროგრამების პარადიგმები
* დაპროგრამების ენის ელემენტები:

ენის ანბანი, სინტაქსი და სემანტიკა

ბეკუს–ნაურის ფორმები BNF და EBNF

მეცნიერებასა და ფილოსოფიაში, პარადიგმა (ბერძნელიდან - “თარგი, მაგალითი, შაბლონი, წარმოდგენა“) ნიშნავს კონცეფციების გარკვეულ ერთობლიობას, ე.ი. აზროვნების შაბლონს, თეორიების, კვლევის მეთოდების და სტანდარტების ჩათვლით, რომლის მიხედვითაც ხორციელდება შემდგომი განზოგადებები და ექსპერიმენტები მოცემულ სფეროში.

*დაპროგრამების პარადიგმა* არის ცნებათა, კონცეფციათა და აბსტრაქციათა ერთობლიობა, რომელიც განსაზღვრავს კომპიუტერული პროგრამების შექმნის სტილს. მარტივად, დაპროგრამების პარადიგმა - ეს არის მეთოდები, პრინციპები, ცნებები და მიდგომები, რომლებიც გამოიყენება ამა თუ იმ ენაზე პროგრამის შექმნის პროცესში.

სხვადასხვა გამოყენებითი სფეროს ამოცანების გადაწყვეტა მოითხოვს დაპროგრამების შესაბამის სტილს, ე.ი. შესაბამისი პარადიგმის გამოყენებას.

დაპროგრამების პარადიგმები განსხვავდებიან:

* ცნებებით და აბსტრაქციებით, რომლებიც გამოიყენება პროგრამის ელემენტების წარმოსადგენად (როგორიცაა ობიექტები, ფუნქციები, ცვლადები და სხვა)
* ბიჯებით, რომლებიც განსაზღვრავენ გამოთვლით პროცესს (მინიჭება, შეფასება, მონაცემთა ნაკადი, მართვის ნაკადი და ა.შ.).

**დაბალი დონის დაპროგრამების პარადიგმა**

თავდაპირველად, კომპიუტერული პროგრამები იწერებოდა მანქანურ ენაზე (ორობით კოდში). ეს იყო რთული პროცესი და წარმოადგენდა მრავალი შეცდომის წყაროს. მანქანურ–ორიენტირებული ენები დაბალი დონის დაპროგრამების პარადიგმის მქონეა და ითვლება პირველი თაობის ენებად.

პროგრამირების გამარტივების მიზნით შეიქმნა ასემბლერის ენები (Assembly languages). ასემბლერის ენები ითვლება დაბალი დონის პარადიგმის მქონე და წარმოადგენენ მეორე თაობის ენებს.

60-იანი წლებიდან ასემბლერის ენებს ჰქონდათ ბიბლიოთეკების მხარდაჭერა და მაკროსების წარმოქმნისა და წინასწარი დამუშავების საკმაოდ რთული შესაძლებლობები.

მათში ასევე განხორციელდა მოდულური დაპროგრამების საშუალებები, როგორიცაა ქვეპროგრამები, გარე და გლობალური ცვლადები, რაც კოდის განმეორებითი გამოყენების საშუალებას იძლევა.

ასემბლერის ენა პირდაპირაა დაკავშირებული პროცესორის ინსტრუქციებთან და მასზე, სწორად დაწერის შემთხვევაში, იქმნება ყველაზე სწრაფი და ოპტიმიზებული კოდი. თუმცა, არქიტექტურაზე დამოკიდებულების გამო ასემბლერის კოდი არაპორტაბელურია (არაა გადატანადი) – ერთი პლატფორმისთვის შექმნილი პროგრამა სხვა პლატფორმაზე ვერ იმუშავებს.

ასემბლერებს აქვთ აბსტრაქციის შეზღუდული შესაძლებლობები, ამიტომაც მსხვილი/რთული პროგრამული უზრუნველყოფის განვითარებისთვის მათ არ იყენებენ. ასემბლერი გამოიყენებოდა და კვლავაც გამოიყენება დროის მიხედვით კრიტიკულ სისტემებში და ხშირად ჩაშენებულ სისტემებში. მათზე იწერება სისტემური პროგრამები, დრაივერები, ვირუსები/ანტივირუსები, ოპერაციული სისტემების ბირთვები. ასევე კოდის აჩქარების მიზნით შესაძლებელია ასემბლერზე დაწერილი კოდის გარკვეული ფრაგმენტის ჩაშენება მაღალი დონის ენით შექმნილ კოდში (შესაბამისი ბრძანებების არსებობის შემთხვევაში, მაგალითად, C –ენის \_asm{} დირექტივა).

მაგალითად, შემდეგ C –ს კოდში ჩაშენებული ასემბლერის ფრაგმენტი

#include <cstdio>

int main()

{

int n;

\_\_asm{

mov n, 10;

add n, 20;

};

printf("n = %d\n", n);

}

n = 30

Press any key to continue . . .

იძლევა შედეგს

მანქანაზე–ორიენტირებულ და ასემბლერის ენებს ხშირად მიაკუთვნებენ ე.წ. იმპერატიული დაპროგრამების ენებს.

**იმპერატიული დაპროგრამება** არის პარადიგმა, კომპიუტერული პროგრამის შექმნის სტილი, რომლისთვისაც დამახასიათებელია შემდეგი:

* პროგრამის საწყისი კოდი ჩაიწერება ინსტრუქციების (ბრძანებების) სახით
* ინსტრუქციები უნდა შესრულდეს თანმიმდევრულად
* ინსტრუქციის შესრულებით მიღებული მონაცემები შეგვიძლია ჩავწეროთ მეხსიერებაში
* მიმდინრე ინსტრუქციის შესრულებისას წინა ინსტრუქციების შესრულებით მიღებული მონაცემები შეგვიძლია წავიკითხოთ მეხსიერებიდან

იმპერატიულ სტილში დაწერილი პროგრამა მუშაობის პროცესში ცვლის მონაცემთა ერთობლიობას, ამბობენ – ცვლის მდგომარეობას. ჩავთვალოთ, რომ უკვე შესრულებული გვაქვს შეტანა-გამოტანის ოპერაციები, მაშინ პროგრამის მუშაობის პროცესი წარმოგვიდგება შემდეგნაირად:

პროგრამის საწყისი მონაცემები ქმნიან თავდაპირველ S მდგომარეობას, ხოლო პროგრამის შესრულების შემდეგ ვღებულობთ მდგომარეობას S’– შედეგს. თითოეული ინსტრუქციის შესრულებით მდგომარეობა იცვლება და თანმიმდევრულად გადადის ერთი მნიშვნელობიდან მეორეში (განსაზღვრულ, სასრულ რაოდენობაჯერ):

S = S0 → S1 → S2 → . . . → Sn = S’

მაგალითად, დალაგების ალგორითმის იმპლემენტაციაში თავდაპირველი მდგომარეობა არის რაიმე ტიპის მონაცემთა მასივი. პროგრამის მუშაობის განმავლობაში მდგომარეობა იცვლება, რის შედეგადაც მონაცემები ხდება დალაგებული – შედეგი. ამ დროს შუალედური მდგომარეობები წარმოადგენენ ცალკეულ ბიჯებს (სვლებს) ამ მიზნის მისაღწევად.

იმპერატიული პროგრამა ფართოდ იყენებს მინიჭებას, ე.ი. მდგომარეობები, საზოგადოდ, იცვლებიან მინიჭების შეტყობინების საშუალებით, რომელიც ჩაიწერება ასე:

v = E ან v := E, სადაც v არის ცვლადი, ხოლო E - გამოსახულება.

ხშირად გამოიყენება სხვა ინსტრუქციებიც, როგორიცაა პირობითი, ამორჩევის ან/და განმეორების (ციკლური) შეტყობინებები. ამრიგად, შეგვიძლია პროგრამა წარმოვადგინოთ როგორც მდგომარეობათა შეცვლის ინსტრუქციების ნაკრები.

იმპერატიული სტილის ნაკლი: მინიჭების შეტყობინების არსებობა ზრდის გამოთვლითი მოდელის სირთულეს და შეიძლება გახდეს სპეციფიური შეცდომების წყარო.

იმპერატიული დაპროგრამების ერთ–ერთ ფორმას წარმოადგენს

**სტრუქტურული დაპროგრამება**

სტრუქტურული დაპროგრამების იდეის ჩამოყალიბება და განვითარება უკავშირდება ედსგერ დეიქსტრას სახელს, შემოთავაზებულია 70–იან წლებში და დაფუძნებულია შემდეგ პრინციპებზე:

1. აკრძალულია უპირობო გადასვლის goto ოპერატორის გამოყენება.
2. ნებისმიერი პროგრამა აგებულია სამი ძირითადი მმართველი კონსტრუქციისგან: თანმიმდევრობა, განშტოება და განმეორება, სადაც

* თანმიმდევრობა – ინსტრუქციების ერთჯერადი შესრულება იმ რიგით, რომლითაც ისინი ჩაწერილია პროგრამის ტექსტში
* განშტოება – პროგრამის გაგრძელების ორი ან მეტი ალტერნატივიდან ერთის ამორჩევა, რაც დამოკიდებულია მოცემული პირობის შესრულებაზე
* განმეორება – ერთი და იგივე ინსტრუქციების ჯგუფის მრავალჯერადი განხორციელება მანამ, სანამ სრულდება მოცემული (განმეორების გაგრძელების) პირობა.

1. პროგრამაში ძირითადი მმართველი კონსტრუქციები შეიძლება იყოს ერთმანეთში ნებისმიერად ჩადგმული
2. ლოგიკურად დაკავშირებული ინსტრუქციების ჯგუფი უნდა გაფორმდეს ბლოკის სახით.

ბლოკი – პროგრამის საწყისი კოდის ნაწილი, რომელშიც გაერთიანებულია ინსტრუქციები და, შესაძლოა, განაცხადები. დაშვებულია ბლოკების ჩადგმა. ბლოკის საზღვრები მკაცრად განსაზღვრულია. მაგალითად, პასკალში ბლოკი ჩაიდგმევა BEGIN ... END –ში, C/C++ –ში ფიგურულ ფრჩხილებში {...}.

1. ყველა ჩამოთვლილ კონსტრუქციას აქვს ერთი შესასვლელი და ერთი გამოსასვლელი. ამით მიიღწევა ნებისმიერი სირთულის ალგორითმების აღწერის შესაძლებლობა მარტივი და საიმედო მექანიზმების გამოყენებით.

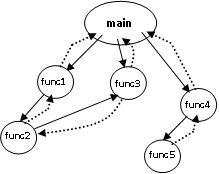
**პროცედურული დაპროგრამება**

წარმოადგენს სტრუქტურული დაპროგრამების ფორმას, ხშირად ამბობენ „მემკვიდრეს“, იმ თვალსაზრისით, რომ ეყრდნობა სტრუქტურული დაპროგრამების ყველა პრინციპს და ასევე ეფუძნება ქვეპროგრამის გამოყენების მეთოდოლოგიას.

პროცედურული დაპროგრამების ძირითადი აზრი ეფუძნება პრინციპს ”გათიშე და იბატონე”: აღწერისთვის რთული ამოცანა უნდა დაიყოს რამდენიმე ქვეამოცანად და ეს დაყოფა უნდა გაგრძელდეს მანამ, სანამ არ მიიღება საკმარისად მარტივი ამოცანები. კომპიუტერულ პროგრამაში ყოველი ამოცანის ალგორითმი კი ფორმდება ცალკეული პროგრამული ერთეულის – ქვეპროგრამის – სახით.

ქვეპროგრამის ქვეშ იგულისხმება პროცედურა, ფუნქცია ან მეთოდი, რომელიც შეიცავს შესრულებადი გამოთვლითი ბიჯების ნაკრებს.

* პროგრამის განმეორებადი ფრაგმენტი შეიძლება გაფორმდეს ქვეპროგრამის სახით (პროცედურა ან ფუნქცია). ლოგიკურად დასრულებული პროგრამის ფრაგმენტი შეიძლება გავაფორმოთ ქვეპროგრამად მაშინაც, როდესაც იგი არ განმეორდება.
* პროგრამის დაპროექტება ხდება ეტაპობრივად, „ზევიდან ქვემოთ“ მეთოდის გამოყენებით:

თავდაპირველად იწერება ძირითადი პროგრამის ტექსტი, რომელშიც ცალკე გაფორმებული პროგრამის ნაწილის (ქვეპროგრამის) ნაცვლად ჩაიდგმება მისი გამოძახება – ფიქტიური ნაწილი (stub, ე.ი. საცობი). საცობი აკმაყოფილებს ჩადგმადი ფრაგმენტის ინტერფეისის მოთხოვნებს, მაგრამ ჯერ ვერ ასრულებს თავის დანიშნულებას. მას შემდეგ, რაც პროგრამისტი დარწმუნდება, რომ პროგრამის ზოგადი სტრუქტურა სწორია, ქვეპროგრამები გაიწერება და საცობები შეიცვლება რეალურად მომუშავე კოდით.

ნებისმიერი ქვეპროგრამის გამოძახება შესაძლოა პროგრამის შესრულების ნებისმიერ მომენტში, მათ შორის სხვა ქვეპროგრამის მიერ ან რეკურსიულად. მაგალითად, C–პროგრამის ფუნქციების ურთიერთქმედების ერთ-ერთი შესაძლო სქემაა :

პროცედურული დაპროგრამების ენა საშუალებას აძლევს პროგრამისტს ზუსტად განსაზღვროს თითოეული ბიჯი ამოცანების განხორციელებაში. პროგრამისტმა იცის, რა უნდა გაკეთდეს და უზრუნველყოფს ეტაპობრივ ინსტრუქციებს, თუ როგორ უნდა შესრულდეს ამოცანა.

შესაძლო უპირატესობები:

* უკეთესი არჩევანია, ვიდრე უბრალო თანმიმდევრული ან არასტრუქტურირებული დაპროგრამება, ზომიერი სირთულის ამოცანებში და იმ შემთხვევაში, თუ პროგრამის მოვლა (შეცდომების პოვნა/გასწორება, შემდგომი განვითარება) მოითხოვს მნიშვნელოვან სიმარტივეს.
* კოდის განმეორებითი გამოყენების შესაძლებლობა პროგრამის სხვადასხვა ადგილას, მისი კოპირების გარეშე.
* პროგრამის გააზრებისა და მის ცვლილებაზე დაკვირვების უკეთესი საშუალება, ვიდრე GOTO ან JUMP შეტყობინებების გამოყენების დროს (ე.წ. სპაგეტი–კოდი).
* უნარი იყოს მოდულური ან სტრუქტურირებული.

ნაკლი:

* რადგან მონაცემები და ქვეპროგრამები გამოყოფილია ერთმანეთისაგან, აუცილებელია თვალყურის მიდევნება იმაზე, თუ რა რიგით გამოიძახება ქვეპროგრამები და რა ცვლილებებს განიცდის ამ დროს მონაცემები. ეს პროცესი მით უფრო რთულდება, რაც უფრო იზრდება პროგრამის მოცულობა (რაც უფრო რთულია გადასაწყვეტი ამოცანა).
* პროცედურულ დაპროგრამებაში არასაკმარისად არის განხორციელებული პროგრამული კოდის განმეორებადი გამოყენება.
* პროცედურული მიდგომის მთავარი ნაკლი კი ისაა, რომ მონაცემებისა და ქვეპროგრამების ერთმანეთისაგან გამოყოფა ცუდად აღწერს რეალური სამყაროს სურათს. რეალური სამყაროს ობიექტების (ადამიანების, მანქანების და სხვა) თვისებები და ქცევა განუყოფელია და მოიაზრება მთლიანობაში.

**განზოგადებული დაპროგრამება**

ეს დაპროგრამების პარადიგმა მდგომარეობს მონაცემებისა და ალგორითმების ისეთ აღწერაში, რომ ალგორითმის გამოიყენება შესაძლებელია განსხვავებულ მონაცემთა ტიპებთან ალგორითმის იმპლემენტაციის შეცვლის გარეშე.

იგი ეფუძნება მონაცემთა სტრუქტურების და ალგორითმების ერთმანეთისგან გამოყოფას აბსტრაქტული აღწერების შემოღებით.

აბსტრაქტული აღწერები არის აბსტრაქტული მონაცემთა ტიპის კონცეფციის განვითარება. ცალკეული ტიპის აღწერის ნაცვლად განზოგადებულ დაპროგრამებაში შემოდის ისეთი ტიპების ოჯახის აღწერა, რომლებსაც აქვთ საერთო ინტერფეისი და სემანტიკური ქცევა. მოთხოვნების ერთობლიობას, რომელიც განსაზღვრავს ინტერფეისს და სემანტიკურ ქცევას ეწოდება კონცეფცია (concept).

ამგვარად, განზოგადებული სტილით დაწერილი ალგორითმი შეიძლება გამოყენებულ იქნეს ნებისმიერ ტიპთან, რომელიც აკმაყოფილებს მას თავისი კონცეფციებით. ამ შესაძლებლობას პოლიმორფიზმი ეწოდება.

განზოგადებული დაპროგრამება ამა თუ იმ სახით მხარდაჭერილია სხვადასხვა დაპროგრამების ენებში. პირველად გაჩნდა 70 წლებში ენაში Ada განზოგადებული ფუნქციების სახით. დღესდღეობით განხორციელებულია ბევრ ობიექტზე ორიენტირებულ ენაში, როგორიცაა C++, Java, Object Pascal, D, Eiffel, .NET პლატფორმის ენებში და სხვა.

**ობიექტზე ორიენტირებული** **დაპროგრამება**

მიეკუთვნება იმპერატიული დაპროგრამების პარადიგმას და ცალკე იქნება განხილული ჩვენ კურსში.

**დეკლარატიული დაპროგრამება**

დეკლარატიული დაპროგრამება არის პროგრამირების პარადიგმა, რომელიც განსაზღვრავს ამოცანის გადაჭრის სპეციფიკას, ანუ, აღწერს *რაში მდგომარეობს ამოცანა და როგორია მოსალოდნელი შედეგი.*

არის იმპერატიული დაპროგრამების საპირისპირო, რომელშიც ამა თუ იმ დეტალიზების დონეზე აღწერილია *როგორ უნდა გადაწყდეს ამოცანა და წარმოდგენილ იქნეს შედეგი*. საზოგადოდ, დეკლარატიული დაპროგრამება „მიდის“ ადამიანიდან მანქანამდე, ხოლო იმპერატიული – მანქანიდან ადამიანამდე. როგორც შედეგი, დეკლარატიული დაპროგრამება არ იყენებს მდგომარეობის ცნებას, ე.ი. არ შეიცავს მინიჭების ოპერატორს.

ამ პარადიგმის მქონე ენებს შორის უნდა აღინიშნოს ფუნქციონალური და ლოგიკური დაპროგრამების ენები.

**ფუნქციონალური** და **ლოგიკური** დაპროგრამების სტილს ჩვენ ასევე მოგვიანებით განვიხილავთ.

ზოგიერთ ენაში მხარდაჭერილია ერთი კონკრეტული პარადიგმა:

Smalltalk–ში – აღიწერება ობიექტზე ორიენტირებული პარადიგმით

Haskell–ი – წმინდა ფუნქციონალური დარპოგრამების ენაა.

არსებობს დაპროგრამების ენები, რომლებშიც მხარდაჭერილია რამდენიმე პარადიგმა:

Object Pascal, C++, C#, Visual Basic, Common Lisp, Scheme, Perl, Python, Ruby, Oz, F#.

რამდენიმე პარადიგმის მქონე ენების არსებობის მიზანია – საშუალება მისცეს პროგრამისტებს გამოიყენონ საუკეთესო ინსტრუმენტარი პროგრამის შექმნისას, იმის გათვალისწინებით, რომ ვერც ერთი ცალკეული პარადიგმა ვერ გადაჭრის ყველა პრობლემას ყველაზე მარტივი და ეფექტური გზით.

**დაპროგრამების ენების ელემენტები**

* ***ენის ანბანი, სინტაქსი და სემანტიკა***

მაღალი დონის ყოველი დაპროგრამების ენა ხასიათდება შემდეგი კომპონენტებით: ანბანი, სინტაქსი და სემანტიკა.

ენის **ანბანი** არის იმ და მხოლოდ იმ ატომური (განუყოფელი) სიმბოლოების სიმრავლე, რომელთა გამოყენებაც დაშვებელია მოცემული ენის წინადადებების ასაგებად. ზოგჯერ, ანბანში შემავალ სიმბოლოებს ასოებს უწოდებენ ბუნებრივი ენების ასოების ანალოგიურად.

დაპროგრამების ენის **სინტაქსი** არის ამ ენაზე ჩაწერილი პროგრამის ან მისი ფრაგმენტის წინადადებების აგების წესების ერთობლიობა. ენის წინადადებას ქმნიან ანბანში შემავალი სიმბოლოების მიმდევრობები (კომბინაციები).

**სემანტიკა** კი განსაზღვრავს ენის სწორად აგებული წინადადებების აღქმის წესების ერთობლიობას. ე.ი., დაპროგრამების ენის სემანტიკა ადგენს შესაბამისობას პროგრამის ენობრივ კონსტრუქციებსა და იმ კონკრეტულ ქმედებებს შორის, რომლებსაც ასრულებს კომპიუტერი პროგრამის განხორციელების პროცესში. ფაქტობრივად, სემანტიკა განმარტავს ენის წინადადებების აზრს.

ამავე დროს, სინტაქსი და სემანტიკა დამოუკიდებელი ენობრივი მახასიათებლებია. სინტაქსულად სწორად ჩაწერილმა წინადადებამ შეიძლება ვერ გადმოსცეს სწორი აზრი.

ვინაიდან დაპროგრამების ენა წარმოადგენს კომპიუტერული პროგრამების ჩაწერის ფორმალურ ენას, მისი სინტაქსი შეიძლება აღიწეროს ფორმალურად. სინტაქსის ყველაზე გავრცელებული ფორმალური ნოტაციები ცნობილია სახელწოდებით ბეკუს–ნაურის ფორმა *BNF* და გაფართოებული ბეკუს–ნაურის ფორმა *EBNF*. პირველი შემოთავაზებული იყო ჯონ ბეკუსის (ამერიკელი ინფორმატიკოსი, Fortran–ის ავტორი) და პეტერ ნაურის (დანიელი მეცნიერი ინფორმატიკის დარგში, სტრუქტურული ენა Algol 60 –ის ერთ–ერთი ავტორი) მიერ 60-იან წლებში. ხოლო *EBNF* შემოიღო ნიკლაუს ვირტმა (შვეიცარიელი ინფორმატიკოსი, Pascal–ის ავტორი) და აღწერა მასზე Pascal–ის სინტაქსი.

*BNF* წესებისჩაწერაში მონაწილეობს შემდეგი აღნიშვნები:

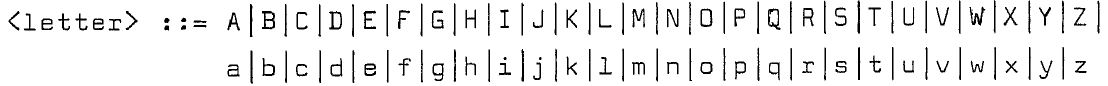
**::=** – შედგება (is composed of)

**[]** – არააუცილებელი (optional)

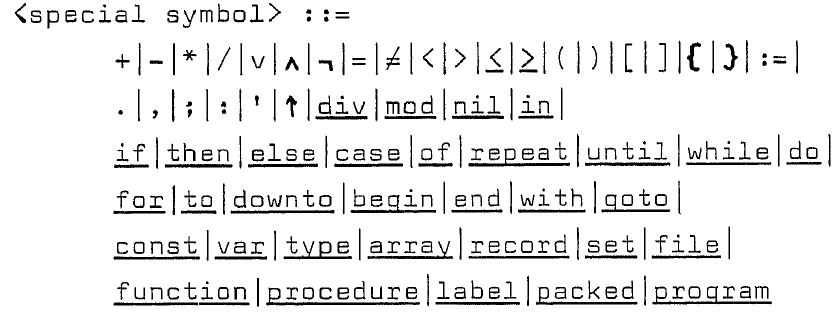
**{}** – ნული ან მეტი გამეორება (zero or more repetitions)

**| –** ან (or)

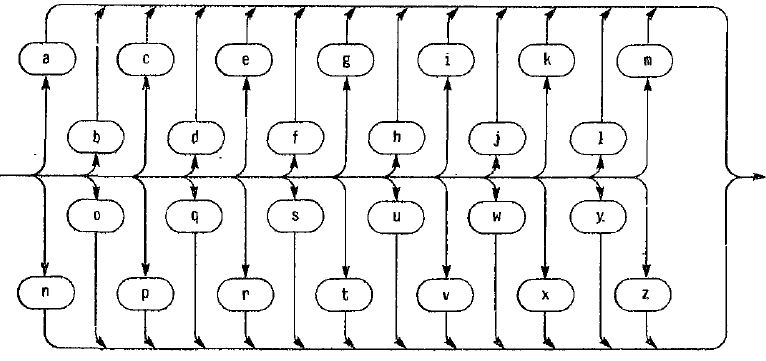
მაგალითად, Pascal–ენის ანბანის, სპეციალური და რეზერვირებული სიტყვების ფორმალიზმია (BNF):

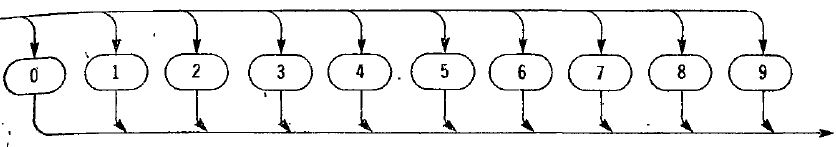


C:\Users\irina\Desktop\________MS leqcia\digit.JPG



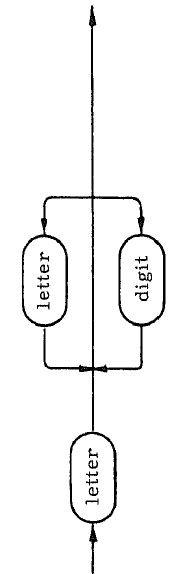
შესაბამისი სინტაქსური დიაგრამები:





სახელის (იდენტიფიკატორის) ფორმულა (EBNF) და დიაგრამა:

identifier = letter | "\_" {, letter | digit | "\_" } ;

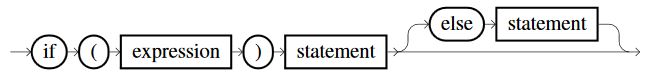


ცვლადზე განაცხადის სინტაქსის ფორმალიზება C –ში:

C:\Users\irina\Desktop\________MS leqcia\variable.JPG

if შეტყობინების ფორმალიზმი და შესაბამისი სინტაქსური დიაგრამა:

C:\Users\irina\Desktop\________MS leqcia\if.JPG



სიმბოლოების მიმდევრობა, რომელიც მიიღება დიაგრამის თანმიმდევრულად გავლისას, წარმოადგენს სინტაქსურად სწორ პროგრამას.

რაც შეეხება if შეტყობინების სემანტიკას (აღქმას და შესრულების წესს), იგი აღიწერება მათემატიკური ლოგიკის გამოყენებით. დავუშვათ, რომ მონაცემთა ნებისმიერი s მდგომარეობისთვის ცნობილია S1 და S2 შეტყობინებების აზრი. აღვნიშნოთ ეს ფაქტი ფორმულებით p (S1, s) და p (S2, s) შესაბამისად. მაშინ if კონსტრუქციის

if ( C ) S1 else S2

აზრი მოიცემა ფორმულით:

C:\Users\irina\Desktop\________MS leqcia\New folder\IF semantics.JPG

ე.ი. თუ s მდგომარეობაში C –ის გამოთვლა შედეგად გვაძლევს *ჭეშმარიტს*, მაშინ if შეტყობინების აზრი იგივეა, რაც S1 –ისა; წინააღმდეგ შემთხვევაში C –ის გამოთვლის შედეგია *არა ჭეშმარიტი*,და if –ს იგივე აზრი აქვს რაც S2 –ს.

თუ დავკმაყოფილდებით if კონსტრუქციის სემანტიკის არაფორმალური განსაზღვრით, მივიღებთ ახსნას, რომელიც გვხვდება დაპროგრამების ენის ყველა სახელმძღვანელოში: მოწმდება if –ის პირობა C; თუ პირობა ჭეშმარიტია, მაშინ უნდა შესრულდეს S1, ხოლო თუ მცდარია, მაშინ – S2.