პროგრამირების ენების თეორია და პრაქტიკა

განავრცობილი კონსპექტი: მარტივი ენა + მაგალითები + გამოცდის ტიპის კითხვები

თარიღი: 2026-01-11

როგორ წაიკითხო სწრაფად და სწორად

- ყოველ თემაზე გაიგე: (1) რა არის, (2) რისთვის გვჭირდება, (3) ერთი პატარა მაგალითი.
- BNF/EBNF + scoping + tokenization არის ყველაზე ხშირი კითხვები — ცალკე განყოფილებებიც აქვს.
- მეხსიერების თემაზე დაიმახსოვრე 3 სიტყვა: static / stack / heap.
- ბოლოშია სავარჯიშოები პასუხებით — თვითშემოწმებისთვის.

შინაარსი

- 1 კვირა — მიმოხილვა და პარადიგმები
- 2 კვირა — ფორმალური ენები, BNF/EBNF, parsing
- 3 კვირა — სემანტიკა (operational/denotational/axiomatic)
- 4 კვირა — ცვლადები, assignment, scope, binding
- 5 კვირა — ციკლები და control flow
- 6 კვირა — ტიპური სისტემები
- 7 კვირა — მონაცემთა სტრუქტურები და აბსტრაქცია (struct/union, ADT)
- 8 კვირა — შუალედურის „ჩეკლისტი“
- 9 კვირა — ფუნქციური პროგრამირება
- 10 კვირა — ლოგიკური პროგრამირება (Prolog)
- 11 კვირა — regex და DFA/NFA
- 12 კვირა — CFG და PDA
- 13 კვირა — თიურინგის მანქანა, Halting problem
- გამოცდის ფოკუს-თემები (ფოტოებიდან)
- სავარჯიშოები პასუხებით
- სწრაფი ლექსიკონი

1 კვირა — პროგრამირების ენების ზოგადი მიმოხილვა

პროგრამირების ენების საგანი ცდილობს აგიხსნას არა მხოლოდ „როგორ დავწეროთ კოდი“, არამედ — რატომ არის ენები ასეთები, როგორაა მოწყობილი და რა წესებით მუშაობს.

რა არის პროგრამირების ენა?

პროგრამირების ენა არის ფორმალური სისტემა, სადაც სიმბოლოების/სიტყვების წესით ვწერთ ინსტრუქციებს. ეს ინსტრუქციები შემდეგ უნდა გარდაიქმნას შესრულებად ფორმად: ან კომპილატორი თარგმნის, ან ინტერპრეტატორი ნაბიჯ-ნაბიჯ ასრულებს.

კომპილაცია vs ინტერპრეტაცია (ძალიან მარტივად)

- კომპილაცია: ერთხელ თარგმნი მთლიან პროგრამას (source -> executable/bytecode) და მერე სწრაფად უშვებ.
- ინტერპრეტაცია: თარგმნა/შესრულება მიდის ერთად, რიგრიგობით.
- რეალურად ბევრი სისტემა შერეულია (JIT, bytecode + VM).

პარადიგმები

პარადიგმა არის პროგრამირების „სტილი“. ერთი და იგივე ამოცანა შეიძლება სხვადასხვა პარადიგმით დაიწეროს, მაგრამ აზროვნება განსხვავდება.

Imperative

იდეა: გვაქვს მდგომარეობა (state) და ბრძანებები, რომლებიც მას ცვლიან. მაგალითად, x=x+1. ეს მოდელი ძალიან ახლოა კომპიუტერის რეალურ მუშაობასთან.

Functional

იდეა: პროგრამა არის ფუნქციების გამოყენება/კომპოზიცია. სასურველია side-effect ნაკლებად გვქონდეს, მონაცემები იყოს immutable, ხოლო განმეორება ხშირად კეთდება რეკურსიით.

Object-Oriented

იდეა: ყველაფერი ორგანიზებულია ობიექტებად (მონაცემი + ქცევა). ინკაფსულაცია გვეხმარება დეტალების დამალვაში, პოლიმორფიზმი — ერთნაირი ინტერფეისით სხვადასხვა რეალიზაციაში.

Logic

იდეა: ვწერთ ფაქტებს/წესებს და ვკითხულობთ „მართალია თუ არა“. სისტემა თვითონ ეძებს დასკვნას (backtracking).

2 კვირა — ფორმალური ენები, BNF/EBNF და Parsing

ალფაბეტი, სტრიქონი, ენა

ალფაბეტი (Σ) არის სიმბოლოების ნაკრები. სტრიქონი არის ამ სიმბოლოების მიმდევრობა. ენა არის სტრიქონების ნაკრები. პროგრამირების ენაც სწორედ სტრიქონების გარკვეული ნაკრებია — „სწორი პროგრამები“.

გრამატიკა

გრამატიკა აღწერს, როგორ „გაიშლება“ პროგრამა წესებით. ამ წესებს ეწოდება productions. გვაქვს ტერმინალები (რეალური სიტყვები/სიმბოლოები) და არატერმინალები (გრამატიკის ცვლადები).

BNF

BNF-ში წესი იწერება: <A> ::= ... სადაც ::= ნიშნავს „განისაზღვრება როგორც“, ხოლო | ნიშნავს არჩევანს.

```
<assign> ::= <id> = <expr>
<id>     ::= x | y | z
<expr>   ::= <term> { (+|-) <term> }
<term>   ::= <factor> { (*|/) <factor> }
<factor> ::= <id> | <number> | ( <expr> )
```

მნიშვნელოვანი იდეა: <assign>, <expr>, <term> და ა.შ. არის არატერმინალები; =, +, *, ფრჩხილები და კონკრეტული სიმბოლოები — ტერმინალებია.

EBNF

EBNF ამატებს მოკლე ჩანაწერს: {X} ნიშნავს X-ის განმეორებას 0 ან მეტჯერ; [X] ნიშნავს optional-ს. ამის წყალობით გრამატიკა ნაკლებ წესს ითხოვს და უფრო მარტივი მოსახმარია.

Parsing (სინტაქსური ანალიზი)

Parser იღებს ტოკენებს და ამოწმებს, ჯდება თუ არა გრამატიკაში. თუ ჯდება, ააგებს parse tree-ს ან AST-ს.

- Parse tree: სრული ხე, სადაც ყველა არატერმინალიც ჩანს.
- AST: შეკუმშული ხე, სადაც მხოლოდ არსებითი ოპერაციებია.

Dangling else

dangling else არის ამბიგუიტეტი: else ვის ეკუთვნის? პრაქტიკული წესი: else ეკუთვნის უახლოეს if-ს, რომელსაც else ჯერ არ აქვს.

3 კვირა — სემანტიკა

სინტაქსი გვეუბნება „როგორ იწერება“, სემანტიკა კი — „რას ნიშნავს“. სემანტიკის სამი პოპულარული მიდგომაა:

Operational semantics

ხსნის შესრულების ნაბიჯებს: თითო ინსტრუქცია ცვლის მდგომარეობას. ამით მარტივია აზროვნება, განსაკუთრებით იმპერატიულ ენებზე.

Denotational semantics

პროგრამის ნაწილებს უსადაგებს მათემატიკურ ობიექტებს (ფუნქციებს). ძალიან მკაცრი განმარტებაა, მაგრამ აბსტრაქტულია.

Axiomatic semantics

იყენებს ლოგიკურ მტკიცებას (Hoare logic): {P} S {Q}. იდეაა, რომ პროგრამა დავამტკიცოთ პირობებით.

```
{ x = 2 }
x = x + 1;
{ x = 3 }
```

4 კვირა — ცვლადები, assignment, scope და binding

ცვლადი რას ნიშნავს რეალურად?

ცვლადი არის: სახელი + ტიპი + მნიშვნელობა + მისამართი მეხსიერებაში. ბევრ შეცდომას რომ აიცილო, უნდა გესმოდეს სად „ცხოვრობს“ ცვლადი და როდის ჩანს.

Binding time

Binding არის მიბმა (სახელი -> ტიპზე, სახელი -> მისამართზე, ოპერატორი -> ოპერაციაზე). Binding time არის როდის ხდება ეს.

- compile-time: ტიპების შემოწმება (სტატიკურ ენებში), ოპერატორების გადაწყვეტა.
- link-time: სხვადასხვა ფაილების/მოდულების დაკავშირება.
- run-time: heap allocation, dynamic typing, dynamic dispatch.

Scope: static vs dynamic

Static (lexical) scoping: ცვლადი იძებნება კოდის მდებარეობით. Dynamic scoping: ცვლადი იძებნება გამოძახების ჯაჭვით (call stack).

გამოცდის კლასიკური მაგალითი (static scoping)

```
int globalVar = 5;

int f(){
  return globalVar;
}

int g(){
  int globalVar = 10;
  return f();
}

print(g());  // პასუხი: 5 (static scoping)
```

რატომ 5? რადგან f() განსაზღვრულია გლობალურ scope-ში და იქ ხედავს globalVar=5-ს. g()-ის შიდა globalVar გავლენას ვერ ახდენს f()-ზე static scoping-ში.

5 კვირა — ციკლები და control flow

ციკლი არის ერთი და იგივე ბლოკის განმეორება. ციკლების სწორ არჩევანს დიდი გავლენა აქვს კოდის სისუფთავეზე.

while / do-while / for

- while: ჯერ პირობა, მერე ბლოკი (0-ჯერაც შეიძლება).
- do-while: ჯერ ბლოკი, მერე პირობა (მინ. 1-ჯერ).
- for: როცა გვაქვს ინდექსი ან წინასწარ ცნობილია iteration count.

break და continue

- break: ციკლიდან მთლიანად გამოსვლა.
- continue: მიმდინარე iteration-ის გამოტოვება და შემდეგზე გადასვლა.

რატომ არის ციკლები PL-ში მნიშვნელოვანი?

ენის დიზაინში ციკლები აჩვენებს, როგორ ფიქრობდა ენის შემქმნელი control flow-ზე. მაგალითად, ზოგ ენაში არის foreach, ზოგში iterator-ები, ზოგში recursion არის მთავარი.

6 კვირა — ტიპური სისტემები

ტიპური სისტემა იცავს პროგრამას „არასწორი მოქმედებებისგან“: მაგალითად, რომ სტრინგს არ მოვექცეთ როგორც მისამართს. ტიპები ასევე ეხმარება IDE-ს და ოპტიმიზაციას.

სტატიკური vs დინამიკური ტიპიზაცია

- Static: ტიპი ცნობილია compile-time-ზე (C/Java).
- Dynamic: ტიპი შეიძლება გაირკვეს run-time-ზე (Python/JS).

ძლიერი vs სუსტი ტიპიზაცია

Strong typing ჩვეულებრივ ნიშნავს ნაკლებ implicit გარდაქმნას და უფრო მკაცრ წესებს. Weak typing-ში შეიძლება ბევრი coercion იყოს და უცნაური შედეგებიც.

შეცდომების ტიპები

- compile-time error: ტიპების შეუსაბამობა (static ენებში).
- run-time error: მაგალითად divide by zero, null dereference, index out of bounds.

7 კვირა — მონაცემთა სტრუქტურები და აბსტრაქცია

struct vs union

struct-ში ყველა ველი თავის ადგილას დევს. union-ში ყველა ველი იზიარებს ერთსა და იმავე მეხსიერებას — ერთდროულად ერთ-ერთი უნდა იყოს აქტიური.

- struct: უფრო უსაფრთხო და მარტივი.
- union: ზოგავს მეხსიერებას, მაგრამ მოითხოვს ფრთხილ გამოყენებას.

ADT (Abstract Data Type)

ADT ნიშნავს: ვგეგმავთ ტიპს ინტერფეისით (ოპერაციებით), არა შიდა რეალიზაციით. მომხმარებელი იცნობს push/pop-ს, მაგრამ არ აინტერესებს მასივი გამოიყენე თუ linked list.

მეხსიერება: static / stack / heap

- static: გლობალური/სტატიკური ცვლადები.
- stack: ფუნქციის ლოკალური ცვლადები და დაბრუნების ინფორმაცია (activation record).
- heap: new/malloc-ით შექმნილი ობიექტები.

8 კვირა — შუალედურის ჩეკლისტი

- BNF/EBNF: ::=, |, terminal vs nonterminal, { } და [ ].
- Parsing: parse tree vs AST, lexer vs parser.
- Scoping: static vs dynamic; შეძლო მსგავსი კოდის პასუხის დადგენა.
- Binding time: compile/link/run მაგალითებით.
- Stack vs heap: სად ინახება ლოკალური და new/malloc ობიექტები.

9 კვირა — ფუნქციური პროგრამირება

Lambda calculus (საერთო იდეა)

ლამბდა კალკულუსში გამოთვლა განიხილება როგორც ფუნქციების გამოყენება. λx. expr ნიშნავს ფუნქციას, რომელიც იღებს x-ს და აბრუნებს expr-ს.

```
(λx. x + 1) 5   => 6
```

map/filter/reduce (ფოტოს ტიპის კითხვა)

```
numbers.map(x -> x*x).filter(y -> y>10).reduce((acc,z)->acc+z, 0)
```

მნიშვნელობა: ავიღოთ კვადრატები, დავტოვოთ >10 და დავაჯამოთ.

10 კვირა — ლოგიკური პროგრამირება (Prolog)

ფაქტები და წესები

```
parent(alice, bob).
parent(bob, charlie).

grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
```

შემდეგ ვსვამთ შეკითხვას: ?- grandparent(alice, Z). სისტემა ზუსტად აქ აკეთებს unification-ს და backtracking-ს.

Unification და Backtracking

Unification არის „დამთხვევა ცვლადებით“: როგორ მივანიჭოთ X,Y,Z ისე, რომ წესები შესრულდეს. Backtracking არის ალტერნატიული გზების გამოცდა, თუ პირველი ვერ გამოვიდა.

11 კვირა — რეგულარული ენები, regex, DFA/NFA

რეგულარული ენები ყველაზე მარტივი კლასია. მათ აღწერს რეგულარული ექსპრესიები და ამოიცნობს DFA/NFA.

DFA vs NFA

- DFA: თითო სიმბოლოზე ერთი გადასვლა.
- NFA: შეიძლება რამდენიმე გადასვლა ან ε-გადასვლა.
- ფაქტი: ორივე ერთნაირ ენებს ცნობს.

Regex-ის ძირითადი ოპერატორები

- | (არჩევანი)
- კონკატენაცია (მიმდევრობა)
- * (0 ან მეტჯერ)
- ( ) (ჯგუფირება)

12 კვირა — CFG და PDA

CFG საჭიროა მაშინ, როცა „ბალანსი“ უნდა დავიჭიროთ (მაგ: ფრჩხილები). DFA ამას ვერ გააკეთებს, რადგან არ აქვს მეხსიერება. PDA-ს აქვს stack.

CFG მაგალითი (ბალანსირებული ფრჩხილები)

<S> ::= (<S>) <S> | ε

რატომ stack?

stack-ში „ვიმახსოვრებთ“ რამდენი (გავხსენით და რამდენი) უნდა დავხუროთ.

13 კვირა — თიურინგის მანქანა და Halting problem

Church–Turing thesis

თეზა ამბობს: რაც ალგორითმულად გამოთვლადია, თიურინგის მანქანითაც გამოთვლადია.

Halting problem

ზოგადად არ არსებობს ალგორითმი, რომელიც ყველა პროგრამისთვის იტყვის გაჩერდება თუ არა. ეს გვაჩვენებს, რომ ყველა პრობლემას ვერ გადავჭრით ავტომატურად.

გამოცდის ფოკუს-თემები (ფოტოებიდან)

1) Tokenization — როგორ დავთვალოთ ტოკენები

```
if (count <= 10) total = total + 1;
```

- keyword: if
- symbol: (
- identifier: count
- relop: <=
- literal: 10
- symbol:)
- identifier: total
- symbol: =
- identifier: total
- symbol: +
- literal: 1
- symbol: ;

2) Lisp: car/cdr/cadr

- car = პირველი ელემენტი
- cdr = კუდი (პირველის გარეშე)
- cadr = მეორე ელემენტი = car(cdr(list))

```
(cadr '(A B C D))  ; => B
```

3) „საშუალოს“ ტიპის შეცდომა პრიორიტეტში

თუ წერ average = a + b/2, ჯერ b/2 ითვლება. სწორი საშუალოა (a+b)/2. ასევე, integer division-ს მიაქციე ყურადღება (Java/C-ში).

სავარჯიშოები პასუხებით

1. რა განსხვავებაა lexer-სა და parser-ს შორის?

პასუხი: lexer აკეთებს ტოკენებს; parser ამოწმებს გრამატიკას და აგებს ხეს.

2. BNF-ში ::= რას ნიშნავს?

პასუხი: „განისაზღვრება როგორც“.

3. BNF-ში | რას ნიშნავს?

პასუხი: ალტერნატივა/არჩევანი.

4. EBNF-ში {X} რას ნიშნავს?

პასუხი: X განმეორდება 0 ან მეტჯერ.

5. Static scoping-ში f() რომელ x-ს ხედავს?

პასუხი: იმ x-ს, რომელიც ფუნქციის განსაზღვრის ადგილას არის ხილვადი.

6. Stack-ზე რა ინახება?

პასუხი: ფუნქციის გამოძახების ჩარჩოები და ლოკალური ცვლადები.

7. Heap-ზე რა ინახება?

პასუხი: დინამიკური ობიექტები (new/malloc).

8. DFA და NFA ძალით?

პასუხი: ეკვივალენტური (ერთნაირ ენებს ცნობენ).

9. CFG რისთვის არის კარგი?

პასუხი: ნესტირებული/ბალანსირებული სტრუქტურებისთვის.

10. Halting problem რას ამბობს?

პასუხი: ყველა პროგრამისთვის გაჩერებადობის უნივერსალური გადაწყვეტა არ არსებობს.

სწრაფი ლექსიკონი

ტერმინი	მარტივი განმარტება
Terminal	რეალური სიმბოლო/ტოკენი (მაგ: if, =, +).
Nonterminal	გრამატიკის ცვლადი (მაგ: <expr>).
Token	ლექსერის მიერ გამოყოფილი ერთეული.
Parse tree	სინტაქსური სტრუქტურის ხე.
AST	შეკუმშული სინტაქსური ხე.
Static scope	ცვლადის მოძებნა კოდის მდებარეობით.
Dynamic scope	ცვლადის მოძებნა call stack-ით.
Binding time	როდის ხდება მიბმა.
Stack	ფუნქციის ჩარჩოები.
Heap	დინამიკური ობიექტები.
DFA/NFA	საბოლოო ავტომატები რეგულარული ენებისთვის.
CFG/PDA	კონტექსტ-თავისუფალი გრამატიკა და stack-ავტომატი.
Turing machine	გამოთვლის ძლიერი მოდელი.
Halting problem	გადაუწყვეტადობის კლასიკური მაგალითი.