პროგრამირების ენების თეორია და პრაქტიკა

დეტალური კონსპექტი - მაქსიმალურად მარტივი ენით (ფინალისთვის)

კონსპექტი შედგენილია ატვირთულ სურათებზე დაყრდნობით. თითო თემაზე: რა არის, როგორ მუშაობს, პატარა მაგალითები და გამოცდაზე ხშირი ხაფანგები.

1. პროგრამირების ენა: სინტაქსი და სემანტიკა

პროგრამირების ენა არის წესებით განსაზღვრული საშუალება, რომლითაც კომპიუტერს ვაძლევთ ინსტრუქციებს. ყველაზე ხშირად გხვდება ორი მთავარი ცნება: სინტაქსი და სემანტიკა.

სინტაქსი vs სემანტიკა

- სინტაქსი - როგორ იწერება კოდი (გრამატიკა/ფორმა). მაგალითად, სად უნდა იყოს ფრჩხილი, სად - წერტილ-მძიმე.
- სემანტიკა - რას აკეთებს ეს კოდი შესრულებისას (მნიშვნელობა).

ხაფანგი: კოდი შეიძლება სინტაქსურად სწორი იყოს, მაგრამ სემანტიკურად (ლოგიკურად) არასწორი.

```
public int average(int a, int b) {
    int result = a + b / 2;
    return result;
}
// '/' სრულდება '+'-ზე ადრე, გამოდის: a + (b/2). ხშირად სწორი არის: (a+b)/2
```

2. ენების პარადიგმები (კლასიფიკაცია)

პარადიგმა არის პროგრამირების სტილი - როგორ 'ვფიქრობთ' და როგორ ვწერთ პროგრამას. ერთ ენას შეიძლება რამდენიმე პარადიგმა ჰქონდეს.

2.1 იმპერატიული (Imperative)

პროგრამა არის ბრძანებების მიმდევრობა: ჯერ ეს გააკეთე, მერე ის. ხშირად გამოიყენება ცვლადები და მათი მნიშვნელობის შეცვლა (state).

- ხშირად იყენებს: მინიჭება (=), if/else, for/while.
- მაგალითი ენები: C, Pascal; Java-ს დიდი ნაწილი.

2.2 პროცედურული (Procedural)

იმპერატიულის პრაქტიკული ვარიანტია: კოდი იყოფა ფუნქციებად/პროცედურებად, რომ მართვადი იყოს და გამეორება შემცირდეს.

2.3 ობიექტზე ორიენტირებული (OOP)

პროგრამა შედგება ობიექტებისგან: მონაცემი (fields) + ქცევა (methods). ძირითადი ცნებები: ენკაფსულაცია, მემკვიდრეობა, პოლიმორფიზმი.

2.4 დეკლარატიული (Declarative)

სწავლობ 'რას' გინდა მიაღწიო და არა 'როგორ' - სისტემა თვითონ არჩევს ნაბიჯებს (მაგალითად SQL).

2.5 ფუნქციური (Functional)

მთავარია ფუნქციებით აზროვნება, ნაკლები state-ის ცვლილება, map/filter/reduce, რეკურსია და ხშირად immutable მონაცემები.

2.6 ლოგიკური (Logic)

წერ ფაქტებს და წესებს, სისტემა (ინფერენსი) პოულობს პასუხს. მაგ: Prolog.

3. კომპილატორი, ინტერპრეტატორი და Java VM

კოდი რომ შესრულდეს, საჭიროა თარგმნა და გაშვება. ეს შეიძლება იყოს კომპილაციით, ინტერპრეტაციით ან ორივეს კომბინაციით.

3.1 კომპილატორი

- თარგმნის მთელ პროგრამას ერთიანად (source -> executable/bytecode), შემდეგ იშვება.
- ძლიერი: ბევრი შეცდომა ადრე იჭრება; შესრულება ხშირად სწრაფია.
- სუსტი: ცვლილებაზე ხელახალი კომპილაცია გჭირდება.

3.2 ინტერპრეტატორი

- კოდს კითხულობს და ასრულებს შესრულების დროს.
- ძლიერი: სწრაფი ტესტირება და debugging ხშირად მარტივია.
- სუსტი: შეიძლება ნელი იყოს (მაგრამ VM/JIT ასწრაფებს).

3.3 Java VM (JVM)

Java-ში javac ქმნის bytecode-ს (.class). JVM ასრულებს bytecode-ს და ხშირად JIT-ით აჩქარებს. VM გვაძლევს პლატფორმათაშორისობას: ერთი bytecode - ბევრი სისტემა.

3.4 კომპილაციის ფაზები (სურათებში ჩარჩო თემებად ჩანს)

- ლექსიკური ანალიზი: ტექსტი იყოფა ტოკენებად (if, identifier, literal, symbol).
- Parsing: ტოკენები მოწმდება გრამატიკაზე და აიგება parse tree ან AST.
- სემანტიკური ანალიზი: ტიპები, სკოპი, წესები (მაგ: დაუდეკლარირებელი ცვლადი).
- კოდის გენერაცია: საბოლოო კოდის/შუალედურის შექმნა.

3.5 Binding time (როდის ფიქსირდება ინფორმაცია?)

დრო	რა ფიქსირდება
Language design time	ენის keyword-ები და სინტაქსის წესები.
Compile-time	სინტაქსური და ტიპების ბევრი შეცდომა, ბევრი ბმა.
Link-time	ბიბლიოთეკების/ფუნქციების მიბმა.
Run-time	მნიშვნელობები, დინამიკური გამოყოფა heap-ზე, late binding.

4. ლექსიკური ანალიზი: Tokens (ტოკენები)

ლექსერი ტექსტს ყოფს ტოკენებად - კატეგორიებად. ტოკენი არის: ტიპი + ხშირად ატრიბუტი.

ძირითადი ტოკენები

- keyword: if, else, while, return
- identifier: count, total (ცვლადის/ფუნქციის სახელები)
- literal: 10, 3.14, "hi"
- relop/operator: <=, ==, +, -, *, /
- symbol/delimiter: (,), {, }, ;

მაგალითი (ფოტოში მსგავსი კითხვა იყო):

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

ტოკენებად შეიძლება ჩაიწეროს ასე:

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

5. BNF და EBNF გრამატიკები

გრამატიკა არის სინტაქსის წესების ფორმალური აღწერა. ყველაზე ხშირად იყენებენ BNF-ს ან EBNF-ს.

5.1 BNF (Backus-Naur Form)

BNF-ის წესი იწერება: A ::= alpha. აქ A არის არა-ტერმინალი.

```
<assign> ::= <id> = <expr>
```

აქ '=' არის ტერმინალი, ხოლო , , - არა-ტერმინალები.

5.2 EBNF (Extended BNF)

EBNF არის BNF-ის გაფართოება: ამოკლებს ჩაწერას და აკეთებს გრამატიკას უფრო მოკლეს.

- [...] - optional (შეიძლება იყოს ან არ იყოს)
- { ... } - repetition (0 ან მეტი გამეორება)
- (...) - ჯგუფირება
- | - ან (ალტერნატივა)

5.3 სტრიქონების გენერაცია

```
<tri> ::= <id><id><id>
<id> ::= A | B | C | D | E | F
```

ამ წესით მიღებული ნებისმიერი სტრიქონი ზუსტად 3 ასოა და თითო ასო A-F-დანაა. მაგ: BCA სწორია.

6. Parsing, Parse Tree და AST

Parsing ამოწმებს: ტოკენების მიმდევრობა ემთხვევა თუ არა გრამატიკას. თუ ემთხვევა, აგებს სტრუქტურას (ხეს).

6.1 Parse Tree

- ფესვი - საწყისი არა-ტერმინალი (მაგ.).
- შიდა კვანძები - სხვა არა-ტერმინალები (,).
- ფოთლები - ტერმინალები (რეალური token-ები).

6.2 AST

AST არის უფრო 'აბსტრაქტული' ხე: ზოგ ზედმეტ დეტალს (მაგ. ზედმეტი ფრჩხილები) აღარ ინახავს და ტოვებს მთავარ ოპერაციებს.

6.3 while-ის გრამატიკის მაგალითი (ფოტოებიდან)

```
<program> ::= <loop>
<loop> ::= while ( <var> <relop> <var> ) do <assign>
<relop> ::= < | > | =
<assign> ::= <var> = <digit> / <digit>
<var> ::= x | y | z
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

ხაფანგი გამოცდაზე: გრამატიკაში რაც არ წერია (მაგ. '{' '}' თუ არ არსებობს), ის სინტაქსურად დაუშვებელია.

ასევე, შესაძლოა სინტაქსურად სწორი იყოს, მაგრამ runtime შეცდომა ჰქონდეს. მაგ: 5/0 - გაყოფა ნულზე.

7. სკოპინგი: static (lexical) და dynamic

სკოპი პასუხობს კითხვას: 'როცა ვწერ x-ს, რომელი x იგულისხმება?' ანუ რომელ დეკლარაციას ვუკავშირდებით.

7.1 Static (lexical) scoping

lexical სკოპინგში ცვლადს ვპოულობთ კოდის სტრუქტურით (სად არის ფუნქცია დაწერილი).

```
int x = 10;
int f() { return x; }
int g() {
  int x = 20;
  return f();
}
print(g());  // lexical -> 10
```

აქ f() ხედავს გლობალურ x-ს (10), რადგან f() იქ არის განსაზღვრული, სადაც x=10 ჩანს. ამიტომ პასუხია 10.

7.2 Dynamic scoping

dynamic სკოპინგში f() ეძებს x-ს გამოძახებების ჯაჭვში (call stack). ამ მაგალითში პასუხი იქნებოდა 20.

7.3 d ცვლადის მაგალითი (ფოტოებიდან)

```
var d = 2;
function f(x){ return x + d; }
function g(){
  var d = 1;
  return f(3);
}
print(g());
```

- lexical/static: f ხედავს გლობალს d=2 -> 3+2=5
- dynamic: f იღებს d-ს g()-დან -> 3+1=4

8. მეხსიერება: global, local და dynamic (heap)

ამ თემაში მნიშვნელოვანია: სად ინახება სხვადასხვა ტიპის ცვლადი მეხსიერებაში.

8.1 Global variable

```
int globalVar = 5;
```

არსებობს პროგრამის დაწყებიდან დასრულებამდე. ხშირად data segment-შია.

8.2 Local variable

```
void func(){
  int localVar = 10;
}
```

არსებობს მხოლოდ ფუნქციის გამოძახების განმავლობაში. ჩვეულებრივ stack-ზეა.

8.3 Dynamic (heap)

დინამიკური ობიექტები (new/malloc) ჩვეულებრივ heap-ზე ინახება და მათი სიცოცხლე runtime-ზე მართდება.

9. ფუნქციური ოპერატორები: map, filter, reduce

მარტივი წესი: map ცვლის, filter არჩევს, reduce აერთიანებს/აჯამებს.

```
numbers
  .map(x => x*x)        // კვადრატები
  .filter(y => y > 10)  // მხოლოდ >10
  .reduce((acc,z)=>acc+z, 0) // ჯამი
```

შედეგი: იმ კვადრატების ჯამი, რომლებიც 10-ზე მეტია.

10. Lisp: cadr და რეკურსიული სია

10.1 cadr

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

cadr ნიშნავს: car(cdr(list)) ანუ სიის მეორე ელემენტი.

10.2 სიის სიგრძე (რეკურსიული იდეა)

```
(define len (lst)
  (cond
    ((null? lst) 0)
    (else (+ 1 (len (cdr lst))))))
```

თუ სია ცარიელია -> 0, სხვა შემთხვევაში -> 1 + დარჩენილი ნაწილის სიგრძე.

11. Dangling else problem

კლასიკური პრობლემა: else ვის ეკუთვნის, თუ if-ები ჩაშენებულია?

```
if (E1)
  if (E2) S1
  else S2
```

უმეტეს ენებში მოქმედებს წესი: else ეკუთვნის უახლოეს დაუმთავრებელ if-ს (nearest if).

12. რიცხვების BNF ისე, რომ '02' არ დაიშვას

მოთხოვნა: '0' დაშვებულია, მაგრამ მრავალნიშნა რიცხვი არ უნდა იწყებოდეს 0-ით (მაგ: 02 აკრძალულია).

```
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<nonzero> ::= 1|2|3|4|5|6|7|8|9
<number> ::= 0 | <nonzero><more>
<more> ::= ε | <digit><more>
```

აქ '02' ვერ მიიღება, რადგან მრავალნიშნა რიცხვი მხოლოდ -ით იწყება. '0' ცალკე წესითაა დაშვებული.