LATVIJAS UNIVERSITĀTES
EKSAKTO ZINĀTŅU UN TEHNOLOĢIJU FAKULTĀTES
DATORIKAS NODAĻA

# INTERPRETATORS PROGRAMMĒŠANAS VALODAI IMMUTANT

KVALIFIKĀCIJAS DARBS DATORZINĀTNĒS

Autors: Rolands Frīdemanis
Studenta apliecības Nr.: rf23009
Darba vadītājs: asociētais profesors Dr. sc. comp. Aleksandrs Belovs

RĪGA, 2025

**Anotācija**

Mūsdienās liela daļa no vispārīga pielietojuma programmēšanas valodām, satur sematiku, kas paredz, ka datu mainība ir ierasta lieta, tomēr šim pastāv būtisks trūkums, kas ir datu neparedzamība. Lai spriešanu par programmas stāvokli padarītu paredzamu, šī darba ietvaros tiek izstrādāta programmēšanas valoda, kuras gramatika un sintakse ierosina noteikta datu nemainību. Šis darbs satur valodas specifikāciju un interpretatora arhitektūras dokumentāciju lekserim, parsētājam un abstraktā sintakses koka pārstaigāšanas algoritmam. Rezultātā, izmantojot valodu C, tiek izveidota augsta līmeņa, intepretējama, dinamiski tipizēta valoda ar atomāriem datu tipiem.

**Atslēgvārdi:** interpretators, AST, funkcionālā programmēšana, C valoda, datu nemainība

**Abstract**

Most modern general-purpose programming languages use grammar and syntax that suggests mutable data being an ordinary matter, which in reality complicates reasoning about program state. To make such reasoning predictable, this work explores the design and delelopment of a programming language with explicit syntax and semantics of immutable data. This work contains specification for such language and documentation of the architecture for the lexer, parser and AST-walker of the underlying interpreter. As a result, a high-level, interpreted, dynamically typed programming language with only atomic data types is created. The language is implemented in C.

**Keywords:** interpreter, AST, functional programming, C language, data immutability

# SATURS

# APZĪMĒJUMU SARAKSTS UN TERMINU SKAIDROJUMI

- **AST** - Abstraktās sintakses koks - koka datu struktūra.
- **mutabilitāte** - Datu īpašība, kas ļauj mainīt to vērtību pēc sākotnējās inicializācijas. Programmēšanas valodas kontekstā tā izpaužas kā interpretatora uzvedība caur valodas semantiku, kas ļauj mainīt datu vērtības izpildes laikā [1].
- **mutabls** - mutabilitātes īpašība, kas ļauj mainīt datu vērtību pēc sākotnējās inicializācijas.
- **nemutabls** - mutabilitātes īpašība, kas ļauj mainīt datu vērtību pēc sākotnējās inicializācijas.
- **Immutant** - Jauna pašizveidota interpretējama programmēšanas valoda ar datu nemainības semantiku, kas apskatīta šajā darbā.

# IEVADS

Datu mainība un to nepārtraukta plūsma no viena mainīga uz otru ir neapstrīdama daļa no lielas daļas programmatūras. Droši var apgalvot, ka datorsistēmu arhitektūru atmiņas koncepcija ļauj izmantot atmiņu vairākkārtēji. Atmiņu var relocēt, izdzēst, pieprasīt lielākus atmiņas gabalus u.t.t. No šī izriet datu mainības būtība, un līdz šai dienai tā ir iekalta lielā daļā, ja ne visu, programmēšanas un skriptēšanas valodu.

No otras puses, mainīgiem datiem pastāv īpašība būt neparedzamiem. Kamēr mazas sistēmas spēj tikt galā ar nelielu daudzumu mainīgo, tad lielajās sistēma tas var kļūt par acīmredzamu problēmu. Atmiņa datoram ir viena, tomēr atsaukties uz to var no dažādām vietām dažādos laika brīžos, kas padara spriešanu par datu stāvokli daudz sarežģītāk.

Programmētāji izmanto iespēju mainīt datus brīvā veidā, jo programmēšanas valodas un to abstrakcijas ir padarījušas to tik vienkāršu. Lai gūtu vairāk kontroles pār ipriekš minēto uzvedību, valodas no C saimes, Java, JavaScript un citas valodas satur gramatikas, kas ierobežo mainīgu datu inicializāciju un to turpmāko vērtību maiņu. Kā piemēru var minēt `const` atslēgvārdu no valodas C, kas fiksē doto mainīgo un liedz pārrakstīt tā vērtību, tomēr tas tikai daļēji attiecas uz atsauces tipa vērtībām. Kaut arī šāda tipa mainīgais vienmēr atsauksies tikai uz vienu konkrētu atmiņas apgabalu, nav noteikts, ka vērtība, kas tajā atrodas, nemainīsies. Līdz ar to, `const` atslēgvārds negarantē patiesu datu nemainību, bet tikai noslēdz to uz `readonly` pieeju. Neskatoties uz dažādiem programmēšanas valodu centieniem ierobežot datu mainību, tādas problēmas kā neparedzamas mainīgo vērtības pavedienu izpildes laikā, mainīgo aizstājējvārdu nepārdomāta ieviešana un izmantošana, mainīgo nejauša re-inicializācija u.c. joprojām sastāda lielu daļu programmu. [1]

Šis darbs izskata jaunas interpretējamas programmēšanas valodas izveidi, kurai pielietota datu nemainības semantika, cenšoties mazināt programmatūras izstrādes problēmas saistītas ar stāvoklu maiņu. Tas arī nozīmē, ka galvenā ideja kalpo far faktoru valodas nosaukuma izvēlei. Savukārt interpretatora sākotnējā versijā, kas šeit ir izskatīta, ietilpst apstrādes loģika sekojošām daļām: primitīvi datu tipi un to glabāšana mainīgos, ar to saistītā datu nemainības semantika, izteiksmes, tīras un netīras funkcijas, kontroles plūsma un cikli.

Šī darba pirmajā nodaļā tiks izskatīta programmēšanas valodas specifikācija, kurai pakārtosies interpretators. Savukārt otrā nodaļa būs veltīta interpretatora arhitektūras pārskatam un tā tehniskai specifikācijai, kā arī tehnoloģijām izmantotām porgrammatūras izstrādei. Trešā un ceturtā nodaļa virzīs uzmanību uz leksera un parsētāja izstrādes detaļām, bet piektā nodaļa būs par rezultējošā AST apstrādi. Tālākajā sestajā nodaļā būs apkopoti rezultāti par jauniegūto valodu un tās interpretatoru, un praktiski demonstrēts valodas pielietojums. Visbeidzot, septītajā nodaļā tiek izvirzītas idejas turpmākai valodas attīstībai, un izskatīts kopsavilkums par paveikto darbu.

# 1. VALODAS SPECIFIKĀCIJA

Šīs nodaļas nolūks ir formalizēt prasības, kas noteiktas uz valodu, kas attiecīgam interpretatoram ir jāspēj īstenot. Šī nodaļa neizklāsta implementācijas detaļas, bet gan drīzāk kalpo kā lietotāja rokasgramāta valodas lietošanā. Nodaļā izklāstītā informācija netikai palīdz lasītājiem iepazīties ar tās uzbūvi valodas lietošanas un attīstīstīšanas nolūkos, bet arī nosaka prasības, kas tiek attiecīgi nostādītas uz Immuntant programmēšanas valodas interpretatoru. Nodaļā ir skaidroti valodā definētie datu tipi, mainīgie funkcijas, un programmas vadības mehānismi, datu nemainības semantika, kas attiecas uz noteikta veida vērtībām, kā arī tiek uzskaitīti kļūdu paziņojumi un to attiecīgie skaidrojumi. Turpretim, lai iepazītos ar tehniskām implementācijas detaļām, skatīt nākamo nodaļu 2..

### Valodas nolūks

Valodas Immutant galvenais mērķis ir nodrošināt vienkāršu, viegli apgūstamu programmēšanas valodu, ar datu nemainības semantiku, kas ļauj izstrādāt programmatūru ar minimālu blakņu efektu risku, ko rada datu mainība. Valoda ir paredzēta gan iesācējiem programmētājiem, gan pieredzējušiem izstrādātājiem, kas vēlas izmantot datu nemainības priekšrocības savos projektos. Lai nodrošinātu šos mērķus, valoda Immutant piedāvā šādas galvenās iezīmes:

- **Datu nemainība:** Valoda Immutant ievieš stingru datu nemainības semantiku, kas nozīmē, ka pēc datu inicializācijas to vērtības nevar mainīt. Tas palīdz samazināt blakusefektu risku un padara programmas uzvedību paredzamu.
- **Vienkārša sintakse:** Valodas sintakse ir veidota tā, lai tā būtu viegli saprotama un lietojama, padarot to pievilcīgu gan iesācējiem, gan pieredzējušiem programmētājiem.
- **Dinamiskā tipizācija:** Valoda Immutant izmanto dinamisko tipizāciju, kas ļauj programmētājiem strādāt ar dažādiem datu tipiem bez nepieciešamības deklarēt to tipus pirms lietošanas.
- **Funkcionālā programmēšana:** Valoda atbalsta daļu no funkcionālās programmēšanas paradigmām, nodrošinot izpildlaikā verificējamas tīras funkcijas, kas palīdz samazināt blakusefektu risku.

## 1.1. Datu tipi

Programmēšanas valodas pamatu veidu datu tipi un ar to saistītie mehānismi. Valodā Immutant tiek definēti tikai primitīvie datu tipi, kas satur skaitļus, virknes un patiesuma vērtības. Datu tipizācija ir dinamiska, kas nozīmē, ka datu tipa pārbaude tiek veikta izpildes laikā.

### 1.1.1. Primitīvie datu tipi

Pirmatnējā Immutant valodas versijas tipu sistēma ietver tikai primitīvos datu tipus, t.i., pamata datu tipi, kas vēlāk var veidod citus tipus, piemēram, masīvus, objektus u.c. Salikti datu tipi netiek iekļauti valodas versijā, kas tiek aprakstīta šajā dokumentā, lai saglabātu interpretatora vienkāršību un vairāk koncentrētos uz datu nemainības semantikas īstenošanu.

#### Virknes

Virkne ir jebkurš simbolu kopums, kas ir ietverts starp dubultpēdiņām. Tās var saturēt jebkādus simbolus, tostarp specsimbolus, kā `@,^,&` u.c. Proti, vikrnēs nav liegts izmantot speciālos atslēgas vārdus, ko pati valoda ir definējusi, piemēram, `mutant` (vairāk par to minēts sadaļā 1.5.). Piemērs dažādām viknēm seko zemāk. Tāpat, arī tikai vienu rakstzīmi ir atļauts uzdot ar virknes tipu.

```
"", "someString", "garumzīmes īāē", " ", ,"\""
```

Izņēmuma gadījums, ir dubultpēdiņu simbola izmantošana iekš vikrnes. Lai neuztvertu to kā virknes beigas, tas ir jāekranē ar vadošo simbolu \. Realitātē neekranizētas virknes, kas satur specsimbolus ir pamata sintakses kļūda, ko programmētāji bieži vien pieļauj neuzmanības dēļ.

Piemērs tādai virknei:

```
"They call themselves \"the wolves\""
```

Rezultātā atkārtots dubultpēdiņu simbols netiek uztvers kā virknes beigu indenkators, bet gan kā daļa no virknes vērtības, un tiek panākts sekojošs teksts:

```
They call themselves "the wolves"
```

Uz ekranizēšanu attiecas arī pats ekranizēšanas simbols \. Ja ir nepieciešamība to iekļaut virknē, tas arī ir jāekranizē sekojošā veidā:

```
"There is something mysterious about the \"\\\" character."
```

Dotās virknes rezultējošais teksts ir:

```
There is something mysterious about the "\" character.
```

#### Skaitļi

Skaitlis ir jebkura vērtība no Reālo skaitļu kopas. Ir svarīgi uzsvērt, ka iracionāli skaitļi tiek implementēti pēc IEEE 754 standarta, tāpēc realitātē tie nav līdz galam iracionāli, bet gan ir to aproksimētas vērtības. Daļskaitļu decimālo daļu atdala ar punkta (.) rakstzīmi.

Piemērs skaitliskām vērtībām seko zemāk:

```
102, -9, -0.3, 421.5632, PI
```

Dotajā piemērā `PI` ir viena no valodā eksistējošām konstatēm, kas satur aproksimētu matemātiskās konstantes $\pi$ vērtību (`~3.1415926535`).

Valodas gramatika atļauj izmantot vadošo punktu priekšā skaitlim, kas apzīmē decimāldaļu skailim nulle. Piemēram, decimālskaitlis `.34` ir identisks ar `0.34`.

#### Patiesuma vērtības

Patiesuma vērtības valodā Immutant seko divvērtīgai loģikai, kur ir tikai divas iespējamās vērtības: patiesa un aplama. Patiesu vērtību apzīmē ar `true`, bet aplamu ar `false`. Patiesuma vērtības ir pamats loģiskām izteiksmēm, kas tiek izmantotas kontroles plūsmas konstrukcijās, piemēram, nosacījumos un ciklos.

Patiesuma vērtību var iegūt vai nu to tieši uzdodot, vai arī piemērojot loģiskos operatorus (skat. sadaļu 1.3.).

### 1.2. Konstrukcijas

#### 1.2.1. Izteiksmes

Izteiksmes ir konstrukcija, kas atgriež kādu vērtību. Piemēram, funckijas izsaukums ir izteiksme, kas rezultātā atgriež izpildītas funkcijas vērtību. Pie tam, arī aritmētiska vai loģiska operācija ir izteiksmes konstrukcijas, jo atgriež skaitlisku vai patiesuma vērtību attiecīgi.

#### 1.2.2. Apgalvojumi

Apgalvojumi jeb deklarācijas ir izteiksmes, kas veic kādu darbību, bet pašas par sevi nedod nekādu vērtību. Piemēram, mainīgo inicializācija ir apgalvojums, jo tā veic darbību - piešķir vērtību mainīgajam, bet pati par sevi nedod nekādu vērtību.

Funkcijas definīcija ir apgalvojums, jo tā arī veic darbību - reģistrē funkciju atmiņā, bet pati par sevi nedod nekādu vērtību.

## 1.3. Operatori

Operatorus iedala unāros un bināros. Unārs operators tiek pielietots tikai vienam operandam, kur tai pat laikā bināri operatori ir pielietoti 2 operandiem. Piemēram, negācijas operators, kas apvērš patiesuma vērtību ir pielietots vienam operatoram, bet saskaitīšanas operators ir pielietots diviem skaitļiem, producējot to summu.

Vērts precizēt, ka gadījumā, ja operators ir binārs, tad tas tiek rakstīts abiem operandiem starpā. Turpmāk tekstā kreisas operands tiek definēts kā šāda bināra operatora operands, kas atrodas pa kreisi no operatora, un labais kas attiecīgi ir pa labi no tā. Piemēram, izteiksmē `23 + 45` skaitlis `23` ir kreisas operands, `45` ir labais operands, un + ir binārais summas operators.

**Aritmētiskie operatori:** tiek pielietoti skaitļiem, un atgriež skaitlisku vērtību.

- + saskaitīšana, atgriež divu skaitļu summu
- - atņemšana, atgriež pirmā skaitļa vērtību mīnus otrā skaitļa vērtību
- * reizināšana, atgriež divu skaitļu reizinājumu
- / dalīšana, atgriež kreisā operanda dalījumu ar labo operandu. Ja labais operands ir nulle, tad tiek izmests izņēmums `DivisionByZeroException`
- % modulis, atgriež kreisā skaitļa operanda moduli ar labā operanda vērtību

**Aritmētiskās salīdzināšanas operatori:** tiek pielietoti skaitļiem, un atgriež patiesuma vērtību.

- > lielāks par, atgriež patiesu vērtību, ja kreisā operanda vērtība ir lielāka par labā operanda vērtību
- < mazāks par, atgriež patiesu vērtību, ja kreisā operanda vērtība ir mazāka par labā operanda vērtību
- >= lielāks vai vienāds ar, atgriež patiesu vērtību, ja kreisā operanda vērtība ir lielāka vai vienāda ar labā operanda vērtību
- <= mazāks vai vienāds ar, atgriež patiesu vērtību, ja kreisā operanda vērtība ir mazāka vai vienāda ar labā operanda vērtību

**Ekvivalences operatori:** tiek pielietoti jebkura datu tipa operandiem un atgriež patiesuma vērtību.

- == ir vienāds, atgriež patiesu vērtību, ja abu operandu vērtības ir vienādas. Proti, atgriež aplamnu vērtību, ja abi operandi ir ar dažādiem datu tipiem.

- `!=` nav vienāds ar, atgriež patiesu vērtību, ja abu operandu vērtības nav vienādas, tai skaitā arī ja to datu tipi ir dažādi.

**Loģiskie operatori:** Pielietoti patiesuma vērtībām, un atgriež patiesuma vērtību.

- `!` nēgācija, apvērš patiesuma vērtību. Ir unārs operators.
- `&&` konjukcija, atgriež patiesu vērtību, ja abu operandu vērtības ir patiesas
- `||` disjunkcija, atgriež patiesu vērtību, ja vismaz viena no abu operandu vērtībām ir patiesa.

### 1.3.1. Operatoru darbības kārtība

Katram operatoram ir noteikta sava prioritāte, tas ir, kārtība, kādā apakšizteiksmes ar vairākiem operatoriem tiek izpildītas. Piemēram, reizināšanas operatoram ir augstāka prioritāte, nekā saskaitīšanas operatoram, tāpēc izteiksme `2 + 3 * 4` tiek izskaitļota kā `2 + (3 * 4)`, nevis `(2 + 3) * 4`. Zemāk ir uzskaitīti operatori to prioritātes secībā, no augstākās uz zemāko. Operatori ar vienādu prioritāti tiek minēti kopīgi vienā saraksta apakšpunktā.

- `!` (unārs nēgācijas operators)
- `*`, `/`, `%` (aritmētiskie reizināšanas, dalīšanas un modula operatori)
- `+`, `-` (aritmētiskie saskaitīšanas un atņemšanas operatori)
- `>`, `<`, `>=`, `<=` (aritmētiskie salīdzināšanas operatori)
- `==`, `!=` (ekvivalences operatori)
- `&&` (konjukcija)
- `||` (disjunkcija)

Ir kritiski atzīmēt, ka dažādu operatoru izpildes kārtību var mākslīgi paaugstināt, izmantojot apaļās iekavas. Izteiksmes iekavās tiek izpildītas pirmās, neatkarīgi no tajās esošo operatoru prioritātes.

Skaidrības labad, tiek dots piemērs ar saliktu izteiksmi un tās izvērtēšanas kārtību:

```
3 + 4 % 2 == 3 && true || false
```

Alternatīvi, to var pārrakstīt ar iekavām, kas ilustrē izvērtēšanas kārtību:

```
((3 + (4 % 2)) == 3) && (true || false)
```

Tātad, apakšizteiksme

```
((3 + (4 % 2)) == 3)
```

tiek izvērtēta pirmā (jo atrodas labajā pusē), kas rezultātā atgriež patiesu vērtību. Pēc tam tiek izvērtēta apakšizteiksme

```
(true || false)
```

kas arī atgriež patiesu vērtību. Beigās tiek pielietots disjunkcijas operators `||`, kas atgriež patiesu vērtību, jo vismaz viens no abiem operandiem (kas īstenībā ir izteiksmes) ir patiess.

## 1.4. Izvērtēšanas kārtība

Līdzīgi kā daudzās citās programmēšanas valodās, piemēram Java un Ruby, arī valodā Immutant izteiksmju izvērtēšanas kārtība ir no kreisās uz labo pusi. Tas nozīmē, ka izteiksmes kreisais operands tiek izvērtēts pirms labā operanda. Tas, gan, negarantē ka jebkuras izteiksmes pirmie kreisējie operandi tiks izvērtēti pirmie, jo saliktas izteiksmes ar vairāk par vienu operatoru ņem vērā operatoru prioritātes un asociativitāti.

Zemāk ir dots piemērs, kas ilustrē šo izvērtēšanas kārtību:

```
pure fn add(a, b) {
    return a + b;
}

immutant sum = add(2, 3) + 4;
```

Dotajā piemēra mainīgajā (skat. 1.5.) `sum` tiek piešķirta vērtība, kas pēc būtības ir izvērtēts izteiksmes `add(2, 3) + 4` rezultāts. Tā kā izteiksmes kreisais operands ir funkcijas izsaukums (skat. 1.6.) `add(2, 3)`, tad vispirms tiek izvērtēta šī apakšizteiksme, kas rezultātā atgriež skaitli `5`. Pēc tam tiek izvērtēts labais operands `4` (tas ir parasts skaitlis, tāpēc šeit netiek padarīts daudz darba), un beigās tiek pielietots saskaitīšanas operators `+`, kas atgriež summu `9`. Tātad, mainīgajam `sum` tiek piešķirta vērtība `9`.

### 1.4.1. Saīsinātā izvērtēšana

Interpretatora optimizācijas nolūkos, valoas specifikācijā ietilpst arī saīsinātās izvērtēšanas mehānisms, kas ļauj izvairīties no nevajadzīgu apakšizteiksmju izvērtēšanu, loģiskajās izteiksmēs ar disjunkciju [3].

Pēc definīcijas, loģiskā izteiksme ar disjunkciju ir patiesa, ja vismaz viens no tās operandiem ir patiess. Tā kā valodā Immutant izteiksmes tiek izvērtētas no kreisās puses uz labo, tad ir pietiekami izvērtēt tikai kreiso operandu, ja tas ir patiess.

Piemēram, izteiksme `true || (someFunction() + 5 > 10)` izvērtējas kā patiesa, pie tam interpretators noignorēs (neizvērtēs) labo operandu.

## 1.5. Mainīgie

Mainīgie ir programmēšanas valodas konstrukcija, kas ļauj saglabāt datus atmiņā un atsaukties uz tiem vēlāk programmā. Immutant valodā nošķir 2 veidu manīgos: tie kas nepaļaujas datu mutācijai jeb mainībai un tie, kas tai paļaujas.

### 1.5.1. Mainīgo deklarācija un inicializācija

Mainīgo deklarācija ir apgalvojums, kas reģistrē mainīgā identifikatoru atmiņā, bet inicializācija ir apgalvojums, kas piešķir mainīgajam sākotnējo vērtību.

Mainīgo deklarācija un inicializācija var notikt vienā solī, vai arī tās var izpildīt atsevišķi, tomēr jātur prātā ka mainīgajam ir jābūt deklarētam pirms tā inicializācijas vai jebkādas citas izmantošanas programmā, pretējā gadījumā tiek izmests izņēmums `UndeclaredVariableException`.

Mainīgos deklarē sekojošā formā:

```
<mutability> <identifier>;
```

Mainīgā deklarācija sākas ar vienu no 2 atslēgvārdiem <mutability> - `mutant` un `immutant` -, kas nosaka mainīgā mutācijas īpašību.

Mainīgo deklarāciju nobeidz patvaļīgs identifikators <identifier> jeb mainīgā nosaukums. Identifikators var saturēt burtus, ciparus un pasvītrojuma simbolu (_), bet nedrīkst sākties ar ciparu. Tāpat, identifikators nedrīkst sakrist ar kādu no valodas atslēgvārdiem, piemēram, `mutant`, `immutant`, `if`, `else` u.c. (pilnu atslēgvārdu sarakstu skatīt sadaļā **??**).

Mainīgā inicializācijai ir jāseko vienai no sekojošām formām:

```
<identifier> = <expression>;
```

```
<mutability> <identifier> = <expression>;
```

Pirmo gadījumu lieto, ja mainīgā deklarācija ir veikta iepriekš, bet otro, ja mainīgā deklarācija un inicializācija notiek vienlaicīgi.

Lai inicializētu mainīgo, t.i. piešķirtu tam vērtību, ir jāizmanto piešķiršanas operatora simbols (=), kam seko izteiksme <expression>, kas rezultātā atgriež vērtību, ko piešķir mainīgajam.

### 1.5.2. Mainīgo mutabilitāte

Mainīgo mutabilitāte jeb mainības īpašība nosaka, vai mainīgā vērtību ir iespējams mainīt pēc tā inicializācijas. Valodā immutant ir divu veidu mainīgie: **mutable** un **immutable**, tos deklarē izmantojot atslēgvārdus `mutant` un `immutant` attiecīgi. Turpmāk šajā darbā tiks izmantoti šo terminu latviešu atvasinājumi, t.i., **mutabilitāte, mutabls** un **nemutabilitāte, nemutabls** lai raksturotu datu mainības īpašības.

#### Mutabilitāte `mutable`

Mutabls, atvasināts no angļu valodas vārda `mutable`, ir tāds mainīgais, kura vērtību ir iespējams mainīt pēc tā inicializācijas. Tādus mainīgos deklarē ar atslēgvārdu `mutant`.

Jebkurš mainīgais, kas deklarēts ar `mutable`, var tikt piešķirts jaunu vērtība jebkurā programmas izpildes brīdī pēc tā inicializācijas. Tas ļauj programmētājiem veidot dinamiskākas programmas, kurās dati var mainīties atkarībā no programmas loģikas un lietotāja ievades. Proti, to ir ieteicams darīt tikai gaījumos, kad risinājumi ar nemutabliem mainīgajiem ir pārāk sarežģīti vai neefektīvi.

#### Mutabilitāte `immutable`

Nemutabls, atvasināts no angļu valodas vārda `immutable`, ir tāds mainīgais, kura vērtību nav iespējams mainīt pēc tā inicializācijas. Tādus mainīgos deklarē ar atslēgvārdu `immutant`. Šeit ir pamanāma acīmredzama vārdu spēle, jo `immutant` ir neoloģisms angļu valodā, kas apvieno vārdus `immutable` un `mutant` radot jaunu vārdu, kas apzīmē mainīgo, kas nav maināms. Tas atbilst nemutabilitātes semantikai, kas izmantota funkcionālajā programmēšanā, un tādēļ arī sekojošs nosaukums izvēlēts šai programmēšanas valodā.

Nemutablus mainīgos nedrīkskt re-inicializēt, tas nozīmē, ka sekojoša programma izmetīs kļūdu `ImmutableVariableModificationException`:

```
immutant x = 10;
x = 20; // Izmests izņēmums: ImmutableVariableModificationException
```

Nemutabli mainīgie nedrīkt arī tik padoti kā argumenti netīrām funkcijām (skat. sadaļu 1.6.), jo tas pārkāpj datu nemainības principu. Pretējā gadījumā netīras funckijas varētu mainīt nemutabla mainīgā vērtību, kas noved pie neparedzamas programmas uzvedības. Šo uzvedību demonstrē šāds piemērs:

```
immutant num = 5;

impure fn modify(value) {
    value += 10; // Mēģinājums mainīt nemutabla mainīgā vērtību
}
```

```
modify(num); // Izmests izņēmums: ImmutableVariableModificationException
```

Turpretim, tīras funkcijas (skat. nodaļu 1.6.) var pieņemt gan mutablus, gan nemutablus mainīgos kā argumentus, jo tīras funkcijas pēc definicījas neietekmē ārējo stāvokli un nemaina dotās vērtības (nenotiek blakus efekti).

Tā kā valodā Immutant netiek definēti sarežģītāki datu tipi, piemēram, masīvi vai objekti, tad nav nepieciešams apskatīt mutabilitātes īpašības attiecībā uz šādiem datu tipiem.

### 1.5.3. Datu tipa konversija

Pastāv gadījumi, kad parādās nepieciešamība konvertēt viena datu tipa vērtību uz cita datu tipa vērtību. Piemēram, ja kādu aritmētisku operatoru, piemēram, saskaitīšanu, pielieto virknei un skaitlim. Tad ir nepieciešams konvertēt vienu no operandiem uz otra datu tipu, lai veiktu šo operāciju, šajā gadījumā virkne jāparveido uz skaitli vai skaitlis par virni.

Daudzās valodās eksistē netiešā datu tipu konversija jeb tā, ko interpretators pats veic pēc vajadzības. Piemēram, valodā Javascript izteiksme `"5"+ 3` rezultātā atgriež virkni `"53"`, jo interpretators veic netiešo tipu konversiju, pārvēršot skaitli `3` par virkni `"3"` un pēc tam veicot virkņu konkatenāciju [2]. Šī ir gan priekšrocība, jo operators tiek netieši pārslogots uz darbību ar virknēm, nevis skaitļiem, tomēr tas var novest pie neparedzamas uzvedības, kas zināmā sabiedzrībā rada jokus, par to kā Javascript "5"+ 3 ir "53", nevis 8.

Tā kā programmēšanas valodas Immutant galvenais mērķis ir palīdzēt programmētājiem rakstīt paredzamāku kodu, datu tipu konversija tiek atbalstīta tikai tiešā veidā. Tas nozīmē, ka saskaitot virkni un skaitli, nav jāpiedomā vai rezultātā radīsies virkne vai skaitlis, jo zināms, ka neviennozīmīgu operāciju izpilde izmetīs izpildlaika kļūdu.

No otras puses, ja programmas autors tiešām vēlas pārveidot vienu datu tipu uz citu, tad valoda Immutant piedāvā iebūvētas funkcijas, kas veic šo darbību. Šīs funkcijas ir aprakstītas zemāk:

- `toString(value)` - Atgriež ‘value‘ vērtību pārvēstu virknes datu tipā. Pārvēršanas loģika - doto vērtību pārvērš par virkni, izmantojot tās bāzes reprezentāciju. Piemēram, skaitlis `42` tiek pārvērsts par virkni `"42"`, bet patiesuma vērtība tiek pārvērsta par virkni `"true"` vai `"false"` attiecīgi.

- `toNumber(value)` - Atgriež vērtības skaitlisko reprezentāciju. Ja virkne satur derīgu skaitlisko vērtību, tad tā tiek pārvērsta par atbilstošu skaitli. Piemēram, virkne `"123.45"` tiek pārvērsta par skaitli `123.45`. Ja virkne nesatur derīgu skaitlisko vērtību, tiek izmests izņēmums `InvalidTypeConversionException`. patiesa vērtība `true` tiek pārvērsta par skaitli `1`, bet `false` par skaitli `0`.

Virkne ir uzskatāma par derīgu skaitlisko vērtību, ja tā satur tikai skaitli vai decimālskaitli. Ja skailim apkārt ir tukšuma simboli, tie tiek ignorēti. Piemēram, virknes `"42 "` un `3.14"` ir derīgas skaitliskās vērtības, bet virknes `"123abc"` un `"hello"` nav derīgas.

- `toBoolean(value)` - Pārveido dotā operanda vērtību par patiesuma vērtību. Vērtības `0`, `"0"`, `""` (tukša virkne) un `false` tiek pārvērstas par aplamu vērtību `false`. Visas pārējās vērtības tiek pārvērstas par patiesu vērtību `true`.
- `typeOf(value)` - Atgriež dotā operanda datu tipu kā virkni. Pieņem jebkura datu tipa vērtību kā argumentu, un atgriež vienu no šādām virkņu vērtībām: `"number"`, `"string"`, `"boolean"`.

Pie tam, datu tipu konversijas funkcijas neietekmē dotā operanda sākotnējo vērtību un datu tipu, bet gan atgriež jaunu vērtību ar konvertēto tipu. Ja datu tipa konversija nav iespējama, tiek izmests izņēmums `InvalidTypeConversionException`.

### 1.5.4. Mainīgo redzamība

Mainīgo redzamība ir tāda īpašība, kas nosaka kurās programmas daļas ir pieejams konkrētais mainīgais. Valodā Immutant mainīgo redzamība tiek noteikta pēc to deklarācijas vietas. Mainīgā redzesloks ir tā programmas daļa, kurā ir pieejams attiecīgais mainīgais.

Piemērs globāla mainīgā deklarācijai un tā izmantošanai funkcijā:

```
immutant globalVar = 10;

fn printGlobalVar() {
  print(globalVar);
}

printGlobalVar(); // Izvada: 10
```

Mainīgie, kas deklarēti ārpus jebkuras funkcijas ķermeņa, ir pieejami visā programmas kodā, ieskaitot visas funkcijas. Šādi mainīgie tiek saukti par globāliem mainīgajiem, t.i. to redzesloks aptver visu programmu.

Funkcijas un jebkādas citas konstrukcijas ar ķermeni (valodā Immutant tie arī ir cikli un nosacījumi) izveido savu redzesloku, un attiecīgi mainīgie deklarēti šajā redzeslokā ir pieejami tikai šajā konstrukcijā un tās iekšējos redzeslokos.

Piemērs lokāla mainīgā deklarācijai un tā izmantošanai funkcijā:

```
fn doMagic() {
  immutant localVar = 5;
```

```
  if(true) {
    mutant anotherLocalVar = localVar + 10;
  }
}
doMagic();
```

Kā var redzēt, localVar eksistē visā `doMagic` funckijas redzeslokā, tai skaitā arī nosacījuma `if` ķermenī. Šajā konkrētajā gadījumā `anotherLocalVar` ir pieejams tikai `if` ķermenī.

Mainīgie, kurus mēģina izmantot ārpus to deklarācijas redzesloka, izraisa izņēmumu `UndeclaredVariableException`. Pēc savas būtības tā ir nedeklarēto mainīgo izmantošana. Skatīt piemēru zemāk.

```
fn setLocalVar() {
  immutant localVar = 5;
}
setLocalVar();
print(localVar); // Izmests izņēmums: UndeclaredVariableException
```

Līdzīgi situācija rodas, izmantojot mainīgo pirms tā deklarācijas:

```
print(globalVar); // Izmests izņēmums: UndeclaredVariableException
immutant globalVar = 10;
```

### 1.6. Funkcijas

Funkcijas ir pilnībā vai daļēji izoltēts program-kods, kas var tikt izsaukts un izmantots ar dažādiem parametriem, dažādās vietās un dažādos laika brīžos. Funkcijas pašas par sevi ir daļa no programkoda, tomēr, lai tās varētu tikt izpildītas, ir nepieciešams tās izsaukt.

Funkcijas definē sekojošā formā:

```
<purity> fn <identifier>(<parameters>) {
    <statements>
}
```

<purity> apzīmē funckcijas īpašību mainīt datus ārpus tās lokālā konteksta. Matemātikā īmpure"jeb tiešā tulkojumā no angļu valodas ņetīrsāpzīmē funkcijas, kas maina ārējos stāvokļus vai ir atkarīgas no tiem. Savukārt "pure"jeb "tīras"funkcijas ir tādas, kas neietekmē ārējos stāvokļus un ir atkarīgas tikai no to parametriem. Tīru funckiju izmantošana programkodā ir ieteicama, jo tā padara programmu prognozējamāku un vieglāk saprotamu, jo zināms ka tīras funckijas var izsaukt tikai citas tīras funkcijas vai izteiksmes, kas nozīmē, ka tās nevienā brīdī neizmaino ārējo stāvokli. Tātad, `pure` atslēgvārds tiek lietots, lai definētu tīru funkciju, bet

impure - netīru. Ja funkcija ir definēta kā tīra, tad tās ķermenī nav atļauts izsaukt netīras funkcijas vai veikt citas darbības, kas maina ārējo stāvokli. No otras puses, šī atslēgvārda lietošana nav obligāta, jo valodā Immutant pēc noklusējuma visas funkcijas tiek uzskatītas par tīrām.

```
fn <identifier>(<parameters>) {
    <statements>
}
```

Augstāk redzamajā piemērā funkcija tiek definēta kā tīra, jo nav lietots neviens no tīrības īpašības atslēgvārdiem. Atslēgvārds fn apzīmē, ka pēc tā seko funkcijas definicija. <identifier> satur funckcijas nosaukumu, kas ir unikāls identifikators atmiņā. Uz funckiajs nosaukumu attiecas tie paši noteikumi, kas minēti sadaļā 1.5. par mainīgo identifikatoriem.

Pēc funkcijas nosaukuma seko apaļās iekavas, kas satur parametru sarakstu <parameters>. Parametri ir iekšējā funckijas konteksta mainīgie, kas tiek nodoti funkcijai tās izsaukšanas brīdī, un kuru vērtības funkcija izmanto savā darbībā. Parametru saraksts ir komatu atdalītu parametru virkne. Ja funkcijai nav parametru, tad iekavas ir tukšas.

Funkcijas ķermenis <statements> ir ietverts figūriekavās, un satur apgalvojumus un izteiksmes, kas tiek izpildītas, kad funkcija tiek izsaukta. Funkcijas ķermenī ir jābūt vismaz vienam apgalvojumam vai izteiksmei. Ja funkcija ir tīra, tad tās ķermenī nav atļauts izsaukt netīras funkcijas vai veikt citas darbības, kas maina ārējo stāvokli.

Piemērs tīrai funkcijai, kas aprēķina divu skaitļu summu:

```
pure fn add(a, b) {
    return a + b;
}
```

Piemērs netīrai funkcijai, kas pieskaita jaunu skaitli globālam mainīgajam:

```
mutant num = 4;
impure fn addGlobal(newNum) {
    num += newNum;
}
```

Funkcijas izsaukums ir izteiksme, kas satur funckijas nosaukumu, kam seko apaļās iekavas ar argumentu sarakstu. Argumenti ir vērtības, kas tiek nodotas funkcijai tās izsaukšanas brīdī, un kuru vērtības funkcija izmanto savā darbībā. Argumentu saraksts ir komatu atdalītu argumentu virkne. Ja funkcijai nav argumentu, tad iekavas ir tukšas. Argumentus padod funckijai secīgi, tādā pašā secībā kā tie ir definēti funkcijas parametru sarakstā.

Piemērs funkcijas izsaukumiem:

```
addGlobal(5);
immutant sum = add(3, 7);
```

## 1.7. Kontroles plūsma

Kontroles plūsma ir mehānisms, kas nosaka programmas izpildes secību. Valodā Immutant ir definēti sekojoši kontroles plūsmas mehānismi: `if-else` zarošanās un cikls `while`.

### 1.7.1. Apgalvojums if-else

Apgalvojums `if-else` ļauj izpildīt dažādas koda daļas atkarībā no nosacījuma izpildes. Sintakse ir sekojoša:

```
if (<condition>) {
    <statements>
} else {
    <statements>
}
```

<condition> ir loģiska izteiksme, kas rezultātā atgriež patiesuma vērtību. Ja izteiksmes vērtība ir patiesa, tad tiek izpildīti apgalvojumi <statements> iekš pirmās figūriekavas. Ja izteiksmes vērtība ir aplama, tad tiek izpildīti apgalvojumi <statements> iekš otrās figūriekavas pēc atslēgvārda `else`.

Ja ir nepieciešams izpildīt vairākus nosacījumus, tad var izmantot vairākas `else if` zarošanās. Sintakse ir sekojoša:

```
if (<condition1>) {
    <statements1>
} else if (<condition2>) {
    <statements2>
} else {
    <statements3>
}
```

Pie tam, apgalvojums `else` nav obligāts, un to var izlaist, ja nav nepieciešams izpildīt kodu gadījumā, ja neviens no nosacījumiem nav izpildīts.

### 1.7.2. Cikli

Cikls `while` ļauj izpildīt koda daļu atkārtoti, kamēr nosacījums ir izpildīts. Sintakse ir sekojoša:

```
while (<condition>) {
    <statements>
}
```

<condition> ir loģiska izteiksme, kas rezultātā atgriež patiesuma vērtību. Ja izteiksmes vērtība ir patiesa, tad tiek izpildīti apgalvojumi <statements> iekš figūriekavām. Pēc apgalvojumu izpildes, nosacījums tiek pārbaudīts vēlreiz, un ja tas joprojām ir patiess, tad apgalvojumi tiek izpildīti vēlreiz. Šis process turpinās, kamēr nosacījums kļūst aplams. Ja nosacījums sākotnēji ir aplams, tad cikla ķermenis netiek izpildīts ne reizi.

## 1.8. Kļūdu veidi un apzīmējumi

# 2. INTERPRETATORA ARHITEKTŪRA

Šeit būs info par lekseri, parsētāju un tree walker.

# BIBLIOGRĀFIJA


[1] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. Exploring language support for immutability. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 736–747. IEEE/ACM, 2016.

[2] ECMA International. Ecmascript® language specification (ecma-262 15th edition), 2024.

[3] Daan J.C. Staudt Jan A. Bergstra, Alban Ponse. Short-circuit logic. *arXiv preprint arXiv:1010.3674*, 2013.