

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: Dr. dat. Aleksandrs Belovs

RĪGA, 2025

## **Anotācija**

Mūsdienās liela daļa no vispārīga pielietojuma programmēšanas valodām, satur semantiku, 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 skenerim, 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 development 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-walk evaluation 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 . . . . .	3
Ievads . . . . .	4
<b>1. Vispārējs apraksts</b>	<b>6</b>
1.1. Esošā stāvokļa apraksts . . . . .	6
1.2. Pasūtītājs . . . . .	6
1.3. Produkta perspektīva . . . . .	6
1.4. Produkta funkcijas . . . . .	6
1.5. Darījumprasības . . . . .	7
1.6. Sistēmas lietotāji . . . . .	7
1.7. Vispārējie ierobežojumi . . . . .	8
1.8. Pieņēmumi un atkarības . . . . .	8
<b>2. Valodas specifikācija</b>	<b>9</b>
2.1. Datu tipi . . . . .	9
2.2. Mainīgie . . . . .	11
2.3. Operatori un izteiksmes . . . . .	14
2.4. Izvērtēšanas kārtība . . . . .	17
2.5. Funkcijas . . . . .	18
2.6. Kontroles plūsma . . . . .	20
2.7. Atslēgas vārdi . . . . .	22
2.8. Formālā gramatika . . . . .	23
<b>3. Programmatūras Prasību specifikācija</b>	<b>26</b>
3.1. Funkcionālās prasības . . . . .	26
3.2. Skenēšanas modulis . . . . .	28
3.3. Parsēšanas modulis . . . . .	32
3.4. Izvērtēšanas modulis . . . . .	35
3.5. Interpretatora kodola modulis . . . . .	37
3.6. Lietotāja saskarnes modulis . . . . .	39
3.7. Nefunkcionālās prasības . . . . .	41
<b>4. Programmatūras projektējuma apraksts</b>	<b>42</b>
4.1. Daļējs funkciju projektējums . . . . .	42

---

<b>5. Datu struktūras un algoritmi</b>	<b>48</b>
5.1. Datu struktūras . . . . .	48
5.2. Algoritmi . . . . .	51
<b>6. Testēšana</b>	<b>57</b>
6.1. Testēšanas dokumentācija . . . . .	57
<b>7. Projekta Organizācija</b>	<b>60</b>
<b>8. Kvalitātes nodrošināšana</b>	<b>61</b>
<b>9. Konfigurācijas pārvaldība</b>	<b>62</b>
<b>10. Darbietilpības novērtējums</b>	<b>63</b>
<b>11. Secinājumi</b>	<b>64</b>

## 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 neļauj mainīt datu vērtību pēc sākotnējās inicializācijas.
- **Immutable** - Jauna pašizveidota interpretējama programmēšanas valoda ar datu nemainības semantiku, kas apskatīta šajā darbā.
- **lvalue** - Angļu valodā “left value” jeb saīsināti lvalue - izteiksme, kas apzīmē atmiņas vietu, kurā var glabāt vērtību.
- **REPL** - Angļu valodā “Read-Eval-Print Loop” - interaktīva vide, kas ļauj lietotājam ievadīt kodu, to izpildīt un redzēt rezultātu tūlītēji.
- **UAT** - Angļu valodā “User Acceptance Testing” - lietotāja akceptēšanas testēšana - programmatūras testēšanas veids, kurā galalietotāji pārbauda sistēmu, lai pārliecinātos, ka tā atbilst viņu prasībām un ir gatava lietošanai.
- **LSP** - Angļu valodā “Language Server Protocol” - valodas servera protokols - standarts, kas ļauj izstrādes rīkiem un redaktoriem sazināties ar programmēšanas valodu serveriem, lai nodrošinātu funkciju automātisko pabeigšanu, sintakses izcelšanu un kļūdu pārbaudi.

## 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 pārvietot, 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 valodu.

No otras puses, mainīgiem datiem pastāv īpašība būt neparedzamiem. Kamēr mazas sistēmas spēj tikt galā ar nelielu daudzumu mainīgo, tad lielajās sistēmā 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 iepriekš 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 aizstājvārdu 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 stāvokļu maiņu. Tas arī nozīmē, ka galvenā valodas ideja ir faktors 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.

### Valodas nolūks

Valodas Immutant galvenais mērķis ir nodrošināt vienkāršu, viegli apgūstamu programmēšanas valodu, ar datu nemainības semantiku, kas ļauj izstrādāt programmatūru ar minimālu blakus efektu risku, ko rada datu mainība. Valoda ir paredzēta gan iesācējiem programmētājiem, gan pieredzējušiem izstrādātājiem, kas vēlas izmantot datu nemainības priekšrocības savos projektos.

---

## **Pārskats**

Šī darba pirmajā nodaļā lasītāji tiek iepazīstināti ar produkta potenciālu un tiek sniegti īsu vispārēju pārskatu. Savukārt otrā nodaļa nostāda prasības uz valodas specifikāciju. Trešā un ceturtnā nodaļa raksturo programmatūras projektējumu un piektā nodaļa satur testēšanas plānu. Pārējās nodaļas nesatur tehnisku informāciju, bet gan raksturo projekta īstenošanas procesu. Pēdējā nodaļa ir veltīta secinājumiem un turpmākajām darba iespējām.

# 1. VISPĀRĒJS APRAKSTS

## 1.1. Esošā stāvokļa apraksts

Uz doto brīdi eksistē vairākas programmēšanas valodas, kas piedāvā dažādus līmeņus datu nemainības atbalstam. Piemēram, funkcionālās programmēšanas valodas kā Haskell un Clojure ir veidotas ap datu nemainības principiem, nodrošinot spēcīgu atbalstu nemutabliem datiem un tīrām funkcijām [4, 7]. No otras puses, imperatīvās valodas kā Python un JavaScript piedāvā iespējas strādāt ar nemutabliem datiem, bet tās nav tik stingras kā funkcionālajās valodās [2, 6].

## 1.2. Pasūtītājs

Sistēma tiek izstrādāta pēc autora iniciatīvas kvalifikācijas darba ietvaros.

## 1.3. Produkta perspektīva

Jaunā programmēšanas valoda Immutant ir paredzēta kā patstāvīgs produkts, kas var tikt izmantots kā rīks dažādiem programmēšanas projektiem. Tā nav atkarīga no jebkuras konkrētas platformas vai sistēmas.

## 1.4. Produkta funkcijas

Valoda Immutant piedāvā šādas galvenās funkcijas:

- **Datu nemainība:** Valoda Immutant ievieš stingru datu nemainības semantiku, kas nozīmē, ka pēc datu inicializācijas to vērtības nevar mainīt. Tas palīdz samazināt blakus-sefektu 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 tiem 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.

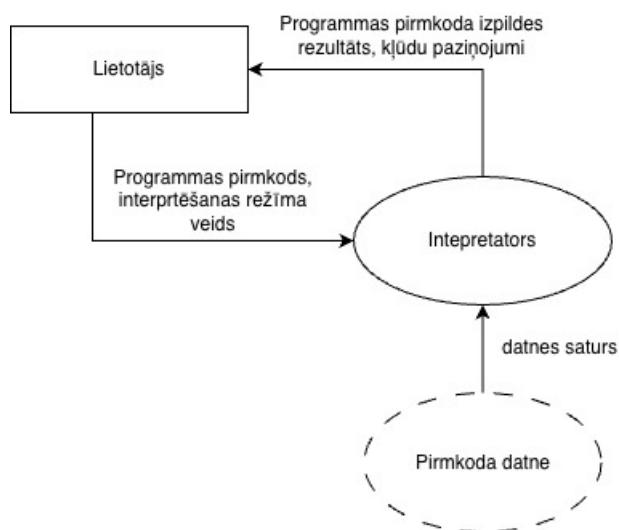
## 1.5. Darījumasprasības

Programmatūrai ir izveidotas šādas darījumasprasības:

- Valodas sintakse un semantika jābūt viegli saprotamai un lietojamai, lai veicinātu plašu pieņemšanu starp programmētājiem.
- Intepretatoram ir jāatbalsta gan interarktvās vides režīmu, gan skriptu izpildi no datnēm.

## 1.6. Sistēmas lietotāji

Sistēmas galvenie lietotāji ir programmētāji, kas vēlas izmantot datu nemainības priekšrocības savos projektos. Tie var būt gan iesācēji, kas mācās programmēšanu, gan pieredzējuši izstrādātāji, kas meklē jaunu rīku saviem projektiem. Spēkā ir pieņēmums, ka lietotājam ir pamata izpratne par programmēšanas jēdzieniem un terminoloģiju un tas ir iepazinies ar programmēšanas valodu Immutant dokumentāciju.



1.1. Att. 0. ģimeņa datu plūsmas diagramma

---

## **1.7. Vispārējie ierobežojumi**

Interpretators ticis izstrādāts Mac OS platformai, tāpēc tiek pieņemts, ka lietotājam darbojas šajā operētājsistēmā. Tomēr interpretatora kods ir rakstīts tā, lai to būtu viegli pārnest uz citām platformas ar minimālām izmaiņām.

## **1.8. Pieņēmumi un atkarības**

- Pirmkoda datnes, kas tiek padotas izpildei interpretatoram nav aizsargātas pret lasīšanu.
- Pirmkods ir rakstīts atbilstoši valodas Immutant gramatikai un sintaksei.

## 2. VALODAS SPECIFIKĀCIJA

Šīs nodaļas nolūks ir formalizēt prasības, kas noteiktas uz valodu, kas interpretatoram ir jāspēj īstenot. Šī nodaļa neizklāsta implementācijas detaļas, bet gan drīzāk kalpo kā lietotāja rokasgrāmata valodas lietošanā. Nodaļā izklāstītā informācija ne tikai palīdz lasītājiem iepazīties ar tās uzbūvi valodas lietošanas un attīstības nolūkos, bet arī nosaka prasības, kas tiek attiecīgi nostādītas uz Immutant 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.

### 2.1. 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.

#### 2.1.1. Primitīvie datu tipi

Pirmatnējā Immutant valodas versijas tipu sistēma ietver tikai primitīvos datu tipus, t.i., pamata datu tipi, no kuriem potenciāli var veidot citus saliktus 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.

#### Virtnes

Virtnes ir jebkurš simbolu kopums, kas ir ietverts starp pēdīnām. Tās var saturēt jebkādas simbolus, tostarp spec-simbolus, kā @, ^, & u.c. Proti, virknēs nav liegts izmantot speciālos atslēgas vārdus, ko pati valoda ir definējusi, piemēram, mut (vairāk par to minēts sadaļā 2.2.). Piemērs dažādām virknēm seko zemāk. Tāpat, arī tikai vienu rakstzīmi ir atļauts uzdot ar virknes tipu.

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

---

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

Piemērs tādai virknei:

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

Rezultātā atkārtots pēdiņu simbols netiek uztvers kā virknes beigu indikators, 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 konstantēm, kas satur aproksimētu matemātiskās konstantes  $\pi$  vērtību ( $\sim 3.1415926535$ ).

Valodas gramatika atļauj izmantot vadošo punktu priekšā skaitlim, kas apzīmē decimāldaļu skaitlim 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 2.3.).

---

## 2.2. Mainīgie

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

### 2.2.1. 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 `RUNTIME_ERR-2`.

Mainīgos deklarē sekojošā formā:

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

Mainīgā deklarācija sākas ar vienu no 2 atslēgvārdiem `<mutability>` - `mut` un `imut` -, 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, `mut`, `imut`, `if`, `else` u.c. (pilnu atslēgvārdu sarakstu skatīt sadaļā 2.7.).

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.

### 2.2.2. 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 `mut` un `imut` 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 `mut`.

Jebkuram mainīgajam, kas deklarēts ar `mut`, var tikt piešķirta jauna 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 gadījumos, 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 `imut`. Atslēgvārds `imut` ir saīsinājums no `immutable`, kas apzīmē mainīgo, kas nav maināms. Tas atbilst nemutabilitātes semantikai, kas izmantota funkcionālajā programmēšanā.

`Nemutabls` mainīgos nedrīkst re-inicializēt. Tas nozīmē, ka sekojoša programma izmetīs kļūdu `RUNTIME_ERR-11`

```
imut x = 10;  
x = 20; // kļūda
```

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

### **2.2.3. 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 jāpārveido uz skaitli vai skaitlis par virkni.

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 veidā

---

liek smieties, par to, kā Javascript valodā izteiksmes "5" + 3 rezultāts ir "53", nevis 8, tomēr tam arī atrodams loģisks skaidrojums.

Tā kā programmēšanas valodas Immutable 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 Immutable 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 `RUNTIME_ERR-12`. 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 skaitlim 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, "" (tukša virkne) un `false` tiek pārvērstas par aplamu vērtību, bet viss pārējais par patiesu.
- `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", "function".

#### 2.2.4. Mainīgo redzamība

Mainīgo redzamība nosaka kurās programmas daļas ir pieejams konkrētais mainīgais. Valodā Immutable 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:

```
imut 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 `Immutable` 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() {  
    imut localVar = 5;  
    if(true) {  
        mut anotherLocalVar = localVar + 10;  
    }  
}  
doMagic();
```

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

Mainīgie, kurus mēģina izmantot ārpus to deklarācijas redzesloka, izmet kļūdu `RUNTIME_ERR-2`. Pēc savas būtības tā ir nedeklarēto mainīgo izmantošana. Skatīt piemēru zemāk.

```
fn setLocalVar() {  
    imut localVar = 5;  
}  
setLocalVar();  
print(localVar); // Kļūda RUNTIME_ERR-2
```

Līdzīgi situācija rodas, izmantojot mainīgo pirms tā deklarācijas:

```
print(globalVar); // Kļūda RUNTIME_ERR-2  
imut globalVar = 10;
```

## 2.3. Operatori un izteiksmes

### Izteiksmes

Izteiksme ir valodas konstrukcija, kas atgriež kādu vērtību. Piemēram, funkcijas 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.

### 2.3.1. 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.

### 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 kreisais operands ir nulle, tad tiek izmesta kļūda `RUNTIME_ERR-5`
- % 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

- $\geq$  lielāks vai vienāds ar, atgriež patiesu vērtību, ja kreisā operanda vērtība ir lielāka vai vienāda ar labā operanda vērtību
- $\leq$  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. Vienmēr atgriež aplamu vērtību, ja abi operandi ir ar dažādiem datu tiem.
- $!=$  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.

- $!$  negācija, 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.

### 2.3.2. 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 negācijas operators)
- $*, /, \%$  (aritmētiskie reizināšanas, dalīšanas un modula operatori)
- $+, -$  (aritmētiskie saskaitīšanas un atņemšanas operatori)
- $>, <, \geq, \leq$  (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 kreisajā pusē), kas rezultātā atgriež patiesu vērtību. Pēc tam tiek izvērtēta otra apakšizteiksme

```
(true && false)
```

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

## 2.4. Izvērtēšanas kārtība

Līdzīgi kā daudzās citās programmēšanas valodās, piemēram Java un Ruby, arī valodā Immutant izteiksmes izvērtēšanas kārtība ir no kreisās uz labo pusi. Tas nozīmē, ka izteiksmes kreisais operands tiek izvērtēts pirms labā operanda.

Zemāk ir dots piemērs, kas ilustrē šo izvērtēšanas kārtību:

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

```
imut sum = add(2, 3) + 4; // sum vērtība ir 9
```

Dotajā piemēra mainīgajā (skat. 2.2.) `sum` tiek piešķirta vērtība, kas pēc būtības ir izvērtēts izteiksmes `add(2, 3) + 4` rezultāts. Tā kā izteiksmes kreisais operands ir funkcijas izsaukums (skat. 2.5.) `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.

### 2.4.1. Saīsinātā izvērtēšana

Interpretatora optimizācijas nolūkos, valodas 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 paties. Tā kā valodā Immutant izteiksmes tiek izvērtētas no kreisās puses uz labo, tad ir pietiekami izvērtēt tikai kreiso operandu, ja tas ir paties.

Piemēram, izteiksmi `true || (someFunction() + 5 > 10)` izvērtē kā patiesu, pie tam interpretators ignorēs (neizvērtēs) labo operandu.

## 2.5. Funkcijas

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

Funkcijas definē sekojošā formā:

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

<purity> apzīmē funkcijas īpašību mainīt datus ārpus tās konteksta. Datorzinātnēs "impure" jeb tulkojumā no angļu valodas ņetīrsāpazīmē 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 funkciju izmantošana pirmkodā ir ieteicama, jo tā padara programmu prognozējamāku un vieglāk saprotamu, jo zināms ka tīras funkcijas var izsaukt tikai citas tīras funkcijas vai izteiksmes, kas nozīmē, ka tās nevienā brīdī neizmaina ārējo stāvokli. Tātad, pure atslēgvārds tiek lietots, lai definētu tīru funkciju, bet impure - netīru.

### 2.5.1. Tīras un netīras funkcijas

Ja funkcija ir definēta kā tīra, tad tās argumentu vērtības tiek padotas pēc to vērtībām. Tas nozīmē, ka, ja kāds arguments ir mainīgais, funkcijā mainot tā vērtību, mainīgā vērtība mainīsies tikai funkcijas ietvaros, bet ne citā apkārtnē. Citiem vārdiem, jebkurš mainīgais, kas padots kā arguments tīrai funkcijai tiek pa jaunam deklarēts un inicializēts funkcijas iekšējā kontekstā. Pie tam, tīras funkcijas ķermenī ir aizliegts izsaukt netīras funkcijas.

---

Savukārt netīras funkcijas argumenti tiek padoti kā atsauces uz to oriģinālajām vērtībām. Tas nozīmē, ka, ja kāds arguments ir mainīgais, funkcijā mainot tā vērtību, mainīgā vērtība mainīsies arī ārējā kontekstā. Netīras funkcijas ķermenī ir atļauts izsaukt gan tīras, gan netīras funkcijas.

Atslēgvārda `pure` lietošana nav obligāta, jo valodā `Immutable` pēc noklusējuma lietotāja definētās funkcijas tiek uzskatītas par tīrām.

### 2.5.2. Funkcijas definīcija un izsaukums

```
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 definīcija. `<identifier>` satur funkcijas nosaukumu, kas ir unikāls identifikators atmiņā. Uz funkcijas nosaukumu attiecas tie paši noteikumi, kas minēti sadaļā 2.2. par mainīgo identifikatoriem.

Pēc funkcijas nosaukuma seko apaļās iekavas, kas satur parametru sarakstu `<parameters>`. Parametri ir iekšējā funkcijas 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:

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

Funkcijas izsaukums ir izteiksme, kas satur funkcijas 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

---

virrke. Ja funkcijai nav argumentu, tad iekavas ir tukšas. Argumentus padod funkcijai secīgi, tādā pašā secībā kā tie ir definēti funkcijas parametru sarakstā.

Piemērs funkcijas izsaukumiem:

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

Pie nekorektas rīcības ar funkcijām parādās kāda no sekojošām izpildlaika kļūdām:

- `RUNTIME_ERR-8`
- `RUNTIME_ERR-13`

### 2.5.3. Iebūvētas funkcijas

Valodā `Immutant` ir definētas sekojošas iebūvētas funkcijas:

- `print(value)` - izvada dotā operanda vērtību uz konsoli. pieņem jebkura datu tipa vērtību kā argumentu, un izvada tās reprezentāciju kā virkni uz konsoli.
- `time()` - izvada pašreizējo laiku milisekundēs kopš 1970. gada 1. janvāra 00:00:00 UTC.
- `toString(value)`, `toNumber(value)`, `toBoolean(value)`, `typeof(value)` - datu tipu konversijas funkcijas, kas aprakstītas sadaļā 2.2.3.
- `input()` - nolasa lietotāja ievadi no standarta ievades kā virkni. Funkcija gaida, kamēr lietotājs ievada datus un nospiež `Enter` taustiņu, pēc tam atgriež ievadīto vērtību kā virkni.

## 2.6. 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`.

### 2.6.1. 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.

Piemērs programmai, kas atkarība no skaitļa vērtības izvada atbilstošu ziņojumu:

```
imut num = 10;  
  
if (num > 0) {  
    print("Number is greater than zero");  
} else if (num < 0) {  
    print("Number is less than zero");  
} else {  
    print("Number is zero");  
}
```

## 2.6.2. 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.

Ciklisku darbību var nodemonstrēt ar fibonacci skaitļu virknes ģenerēšanu:

```
// Demonstrē mainīgo izmantošanu ciklā  
mut prev = 0;  
mut curr = 1;  
mut count = 0;  
imut n = 10;  
  
while (count < n) {  
    imut val = curr;  
    print(val);  
    imut next = prev + curr;  
    prev = curr;  
    curr = next;  
    count = count + 1;  
}
```

## 2.7. Atslēgas vārdi

Valodā Immutant ir definēti sekojoši atslēgas vārdi, kas ir rezervēti valodas sintaksei un tiem nevar tikt piešķirta cita nozīme:

- `imut` - tiek lietots, lai deklarētu nemutablus mainīgos
- `mut` - tiek lietots, lai deklarētu mutablus mainīgos
- `fn` - tiek lietots, lai definētu funkcijas
- `pure` - tiek lietots, lai definētu tīras funkcijas
- `impure` - tiek lietots, lai definētu netīras funkcijas
- `if` - tiek lietots, lai definētu nosacījuma apgalvojumu

- `else` - tiek lietots, lai definētu nosacījuma apgalvojuma alternatīvu
- `while` - tiek lietots, lai definētu ciklu
- `return` - tiek lietots, lai atgrieztu vērtību no funkcijas
- `true` - patiesuma vērtība
- `false` - aplamas vērtība

## 2.8. Formālā gramatika

Valodas Immutant formālā, kas aprakstīta BNF formā, definē valodas specifikāciju - kas var izvērsties par kaut ko citu un uz ko tieši. Termināļi ir apzīmēti ar lielajiem burtiem, bet netermināļi ar mazajiem burtiem.

PROGRAMMA:

`program`  $\rightarrow$  `statement* EOF`

PRIEKŠRAKSTI:

`statement`  $\rightarrow$  `var_declaration`  
 $\mid$  `function_declaration`  
 $\mid$  `if_statement`  
 $\mid$  `while_statement`  
 $\mid$  `return_statement`  
 $\mid$  `block_statement`  
 $\mid$  `expression_statement`

`var_declaration`  $\rightarrow$  `mutability IDENTIFIER`  
 $\quad$  `( "=" expression )? ";"`

`mutability`  $\rightarrow$  `"mut" | "imut"`

`function_declaration`  $\rightarrow$  `purity? "fn" IDENTIFIER "("`  
 $\quad$  `parameters? ")" block_statement`

`purity`  $\rightarrow$  `"pure" | "impure"`

`parameters`  $\rightarrow$  `IDENTIFIER ( "," IDENTIFIER )*`

`if_statement`  $\rightarrow$  `"if" "(" expression ")"`

---

```

        block_statement
        ( "else" block_statement )?

while_statement    -> "while" "(" expression ")"
                    block_statement

return_statement   -> "return" expression? ";"

block_statement    -> "{" statement* "}"

expression_statement -> expression ";"

IZTEIKSMES:

expression         -> assignment

assignment         -> logical_or
                    ( "=" logical_or )?

logical_or         -> logical_and
                    ( "||" logical_and )*

logical_and        -> equality
                    ( "&&" equality )*

equality           -> comparison
                    ( ( "!=" | "==" ) comparison )*

comparison         -> additive
                    ( ( ">" | ">=" | "<" | "<=" )
                      additive )*

additive           -> multiplicative
                    ( ( "-" | "+" ) multiplicative )*

multiplicative     -> unary
                    ( ( "/" | "*" | "%" ) unary )*

unary              -> ( "!" ) unary
                    | call

```

---

call	-> primary ( "(" arguments? ")" )*
arguments	-> expression ( "," expression )*
primary	-> NUMBER   STRING   "true"   "false"   IDENTIFIER   "(" expression ")"

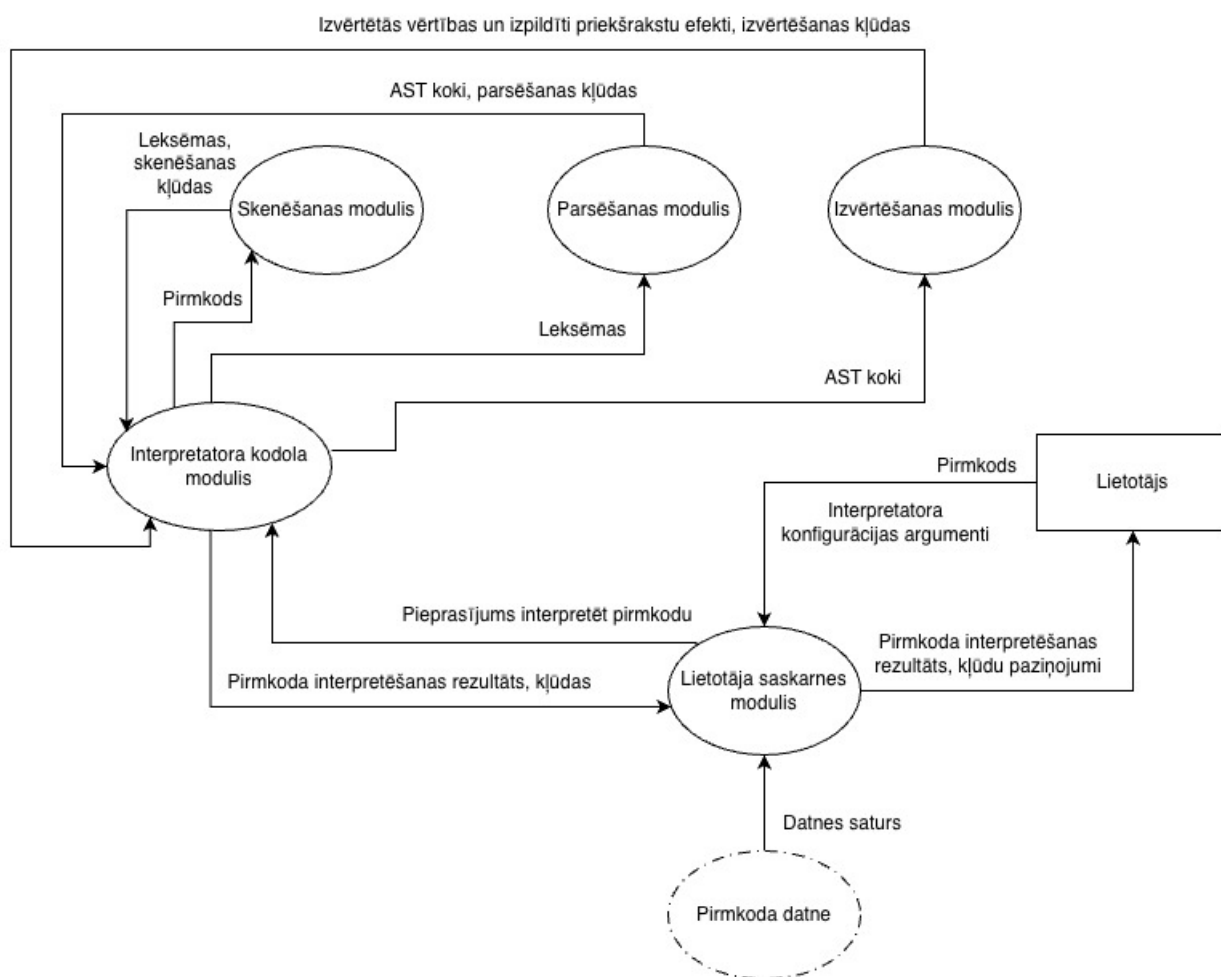
Apzīmējumu saraksts no formālās gramatikas:

- | apzīmē alternatīvu (vai)
- \* apzīmē nulli vai vairākas reizes (*Kleene star*)
- ? apzīmē nulli vai vienreiz (opcionālītāte)
- ( ) apzīmē grupu

### 3. PROGRAMMATŪRAS PRASĪBU SPECIFIKĀCIJA

#### 3.1. Funkcionālās prasības

Interpretatora programmatūru iedala 5 moduļos, katrs no tiem atbild par savu funkcionalitāti. Skenēšanas, parsēšanas un izvērtēšanas moduļi veido pašu pamatu interpretēšanai, taču ir nepieciešami arī citi moduļi, lai nodrošinātu pilnvērtīgu programmu - lietotāja saskarnes modulis un interpretatora kodola modulis.



3.1. Att. 1. līmeņa datu plūsmas diagramma

SYNTAX_ERR-1	Ja skeneris sagaida virknes beigu simbolu, bet to neatrod, parāda paziņojumu: “Unterminated string literal”.
SYNTAX_ERR-2	Ja skaitlis neatbilst valodas specifikācijai, parāda paziņojumu: “Invalid number”.
SYNTAX_ERR-3	Ja skeneris neatpazīst leksēmas tipu, t.i., tā neeksistē valodas alfabētā, parāda paziņojumu: “Invalid token: <lexeme>”. <lexeme> tiek aizstāts ar neatpazītās leksēmas vērtību.
SYNTAX_ERR-4	Ja parsētājs neatrod atverošo iekavu nosacījuma izteiksmei, parāda paziņojumu: “Expected ‘(’ after condition_source“. <condition_source> ir nosacījuma izteiksmes lietotājs - if vai while priekšraksti.
SYNTAX_ERR-5	Ja izteiksme ir grupēta iekavās vai ir daļa no nosacījuma un pēc tās neseko aizverošā iekava, parāda paziņojumu: “Expected ‘)’ after expression“.
SYNTAX_ERR-6	Ja priekšraksta beigās nav atrodamas semikola zīmes, parāda paziņojumu: “Expected ‘;’ after <expr_type>”. ‘<expr_type>’ ir vai nu expression statement vai variable declaration atkarībā no priekšraksta.
SYNTAX_ERR-7	Ja priekšraksts ir bijis atverošais pirmkoda bloks, bet aizverošā figūriekava netika atrasta, tad parāda paziņojumu: “Expected ‘}’ after block statement“.
SYNTAX_ERR-8	Ja funkcijas izsaukumā padotais argumentu saraksts netiek noslēgts ar aizverošo iekavu, tad parāda paziņojumu: “Expect ‘)’ after arguments.“.
SYNTAX_ERR-9	Ja parsētājs neatrod izteiksmi, kur tai būtu jābūt, parāda paziņojumu: “Expected expression“.
SYNTAX_ERR-10	Ja piešķiršanas operatora kreisās operands nav derīgs, lai izpildītu operāciju (nav lvalue), tad parāda paziņojumu: “Invalid assignment target.“.
RUNTIME_ERR-1	Ja dalīšanas operācijai padots dalītājs 0, tad parāda paziņojumu: “Division by zero is illegal“.
RUNTIME_ERR-2	Ja lietotājs izmanto mainīgo vidē, kur tas nav deklarēts un neeksistē, tad parāda paziņojumu: “Undefined variable: <t>“. <t> tiek aizvietots ar attiecīgā mainīgā nosaukumu.

RUNTIME_ERR-3	Ja operācijas operandu tipi neatbilst valodas specifikācijai, tad parāda paziņojumu “Expected a <t>value“. <t> tiek aizvietots ar atbilstošu datu tipu, ko izvērtēšanas modulis sagaidīja.
RUNTIME_ERR-4	Ja operatoram + abi operandi nav vienādā tipa un nav ne virknes ne skaitļu tipa, tad parāda paziņojumu: “Operands to ‘+’ must be both numbers or both strings“.
RUNTIME_ERR-5	Ja abi dalīšanas operācijas operandi nav skaitļi, izmet paziņojumu: “Operands to ‘/’ must be numbers“.
RUNTIME_ERR-6	Ja funkcijas izsaukuma saucējs nav derīga funkcija, tad parāda paziņojumu: “Callee is not a function“.
RUNTIME_ERR-7	Ja mainīgais ar tādu pašu nosaukumu ir jau iepriekš deklarēts, parāda paziņojumu: “Variable already defined: <t>“. <t> tiek aizstāts ar attiecīgu mainīgā nosaukumu.
RUNTIME_ERR-8	Ja funkcija tiek izsaukta ar argumentu skaitu, kas atšķirās no parametru skaita funkcijas definīcijā, parāda paziņojumu: “Incorrect number of arguments passed to function“.
RUNTIME_ERR-11	Ja nemutablam mainīgam pēc inicializācijas vēlreiz piešķir kādu vērtību, parāda paziņojumu “Cannot assign to constant variable: <t>“. <t> tiek aizstāts ar attiecīgu mainīgā nosaukumu.
RUNTIME_ERR-12	Ja datus nav iespējam pārvērst uz citu datu tipu, parāda paziņojumu: “Invalid type conversion“.
RUNTIME_ERR-13	Ja tīra funkcija izsauc netīru funkciju, parāda paziņojumu: “Pure function cannot call impure function“.
MISC_ERR-1	Ja doto failu neizdodas nolasīt, parāda paziņojumu: “Error: Could not open file“.

### 3.2. Tabula Kļūdu paziņojumu apraksts

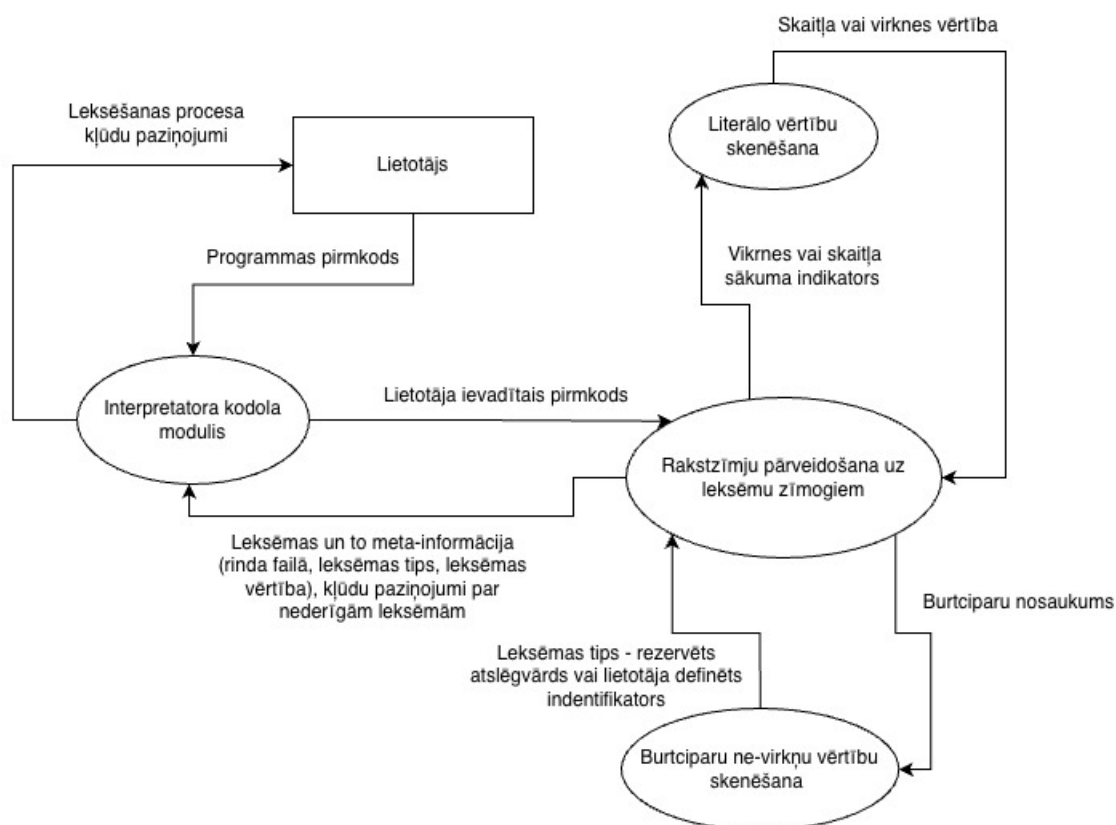
### 3.2. Skenēšanas modulis

Skenēšanas modulis ir atbildīgs par pirmkoda pārvēršanu leksēmās. Skenēšanas modulis izmanto leksisko analīzi, lai identificētu un grupētu pirmkoda elementus, piemēram, at-

Modulis	Funkcija	Identifikators
Lietotāja saskarnes modulis	Pirmkoda interpretēšana no datnes	CLI-1
	REPL režīms	CLI-2
Interpretatora kodola modulis	Interpretēšanas piespiedu apstādināšana	CORE-1
Skenēšanas modulis	Rakstzīmju pārveidošana uz leksēmu zīmogiem	LEX-1
	Burtciparu ne-virkņu vērtību skenēšana	LEX-2
	Literālo vērtību skenēšana	LEX-3
Parsēšanas modulis	Leksēmu parsēšana uz AST kokiem	PARSE-1
	Parsētāja sinhronizēšana pēc notikušās parsēšanas kļūdas	PARSE-2
Izvērtēšanas modulis	AST koka izvērtēšana	EVAL-1
	Izteiksmes AST koka izvērtēšana	EVAL-2
	Priekšraksta AST koka izvērtēšana	EVAL-3

### 3.1. *Tabula* Funkciju sadalījums pa moduļiem

slēgvārdus, operatorus, atomārās vērtības un identifikatorus.



3.2. Att. Skenēšanas moduļa 2.līmeņa DPD

<b>Identifikators:</b>	LEX-1
<b>Nosaukums:</b>	Rakstzīmju pārveidošana uz leksēmu zīmogiem
<b>Mērķis:</b>	Pārvērst pirmkodu leksēmās
<b>Ievaddati:</b>	Lietotāja pirmkods
<b>Apstrāde:</b>	Skeneris dotajā secībā iziet cauri katram simbolam un pārvērš to atbilstošā leksēmā, veicot nepieciešamās papildfunkcijas pilnu leksēmu vērtību nolasīšanai, ja leksēma nesastāv tikai no vienas rakstzīmes.
<b>Izvaddati:</b>	Leksēmas un skenēšanas kļūdas, ja tādas radušās procesā.
<b>Kļūdu paziņojumi:</b>	SYNTAX_ERR-1
	SYNTAX_ERR-2

### 3.3. *Tabula* Funkcijas LEX-1 apraksts

<b>Identifikators:</b>	LEX-2
<b>Nosaukums:</b>	Literālo vērtību skenēšana
<b>Mērķis:</b>	Noskaidrot pilnas leksēmu vērtības kā literālus
<b>Ievaddati:</b>	Lietotāja pirmkods, leksēmas vērtības sākuma simbols, kas indicē, ka leksēma varētu būt noteikta tipa vērtība
<b>Apstrāde:</b>	Tiek noskaidrots vai leksēma var būt skaitlis vai virkne. Ja tas ir skaitlis, tad skeneris iteratīvi nolasa ciparu aiz ciparu, ļaujot vienam punkta simbolam atrasties šajā skaitlī, kas apzīmētu decimāldaļu. Virknes gadījuma skeneris noalasa visus simbolus sākot at atverošo un beidzot ar aizverošo pēdiņas simbolu.
<b>Izvaddati:</b>	Leksēmas pilna vērtība un skenēšanas kļūdas, ja tādas radušās procesā.
<b>Kļūdu paziņojumi:</b>	SYNTAX_ERR-1 SYNTAX_ERR-2

### 3.4. *Tabula* Funkcijas LEX-2 apraksts

<b>Identifikators:</b>	LEX-3
<b>Nosaukums:</b>	Burtciparu ne-virkņu vērtību skenēšana

---

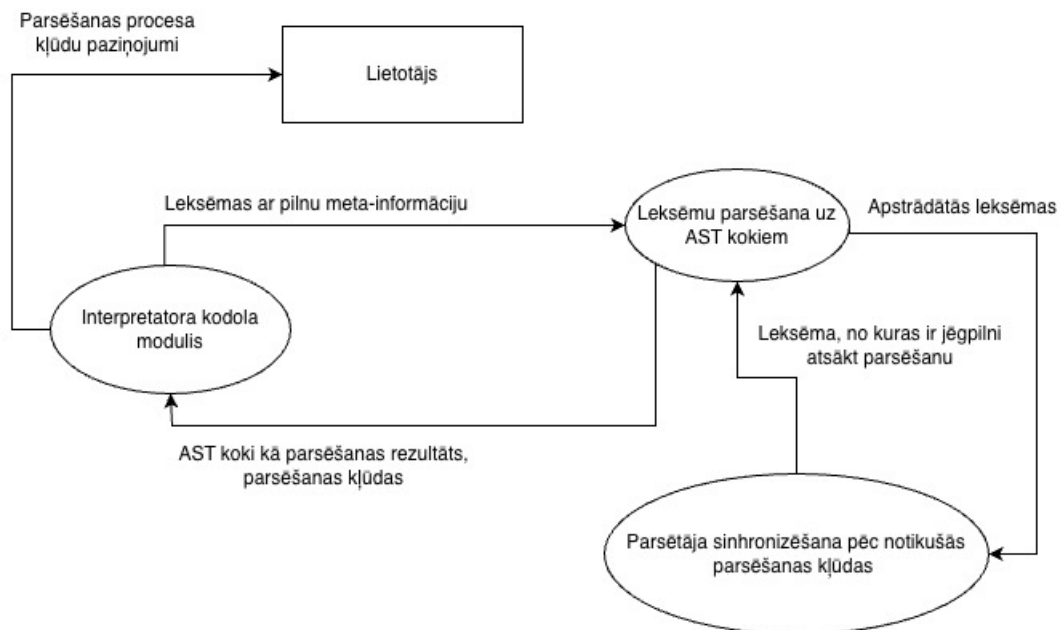
<b>Mērķis:</b>	Noskaidrot vai dotā vērtība ir valodā rezervēts atslēgvārds vai lietotāja definēts identifikators
<b>Ievaddati:</b>	Lietotāja pirmkods, leksēmas vērtības sākuma simbols, kas indicē, ka leksēmai jābūt burtciparu ne-virknes vērtībai
<b>Apstrāde:</b>	Funkcija pārbauda, vai dotā vērtība neatbilst precīzi kādam no valodā rezervētiem atslēgvārdiem. Pretējā gadījumā tiek atgriezta informācija, ka leksēma ir lietotāja definēts identifikators.
<b>Izvaddati:</b>	Burtciparu ne-virkņu datu tips un vērtība
<b>Kļūdu paziņojumi:</b>	-

---

### 3.5. *Tabula* Funkcijas LEX-3 apraksts

### 3.3. Parsēšanas modulis

Parsēšanas modulis ir atbildīgs par leksēmu pārvēršanu abstraktā sintakses kokā (AST). Parsēšanas modulis izmanto sintaktisko analīzi, lai identificētu un strukturētu pirmkoda elementus, piemēram, izteiksmes, apgalvojumus un priekšrakstus.



**3.3. Att. Parsēšanas moduļa 2.līmeņa DPD**

<b>Identifikators:</b>	PARSE-1
<b>Nosaukums:</b>	Leksēmu parsēšana uz AST kokiem
<b>Mērķis:</b>	Pārvērst leksēmas abstraktā sintakses kokā (AST), lai pirmkoda izvērtēšana būtu iespējama
<b>Ievaddati:</b>	Leksēmas no lietotāja pirmkoda
<b>Apstrāde:</b>	Parsētājs izmanto rekursīvo nolaišanās algoritmu, parsējot leksēmas no gramatikas sākuma produkcijas līdz pat dziļākajai produkcijai, lai nodrošināti viennozīmīgu AST koka uzbūvi, respektējot uzdotās izteiksmju prioritātes.
<b>Izvaddati:</b>	AST koki un parsēšanas kļūdas, ja tādas radušās procesā.
<b>Kļūdu paziņojumi:</b>	SYNTAX_ERR-4
	SYNTAX_ERR-5
	SYNTAX_ERR-6

---

SYNTAX\_ERR-7

---

SYNTAX\_ERR-8

---

SYNTAX\_ERR-9

---

SYNTAX\_ERR-10

---

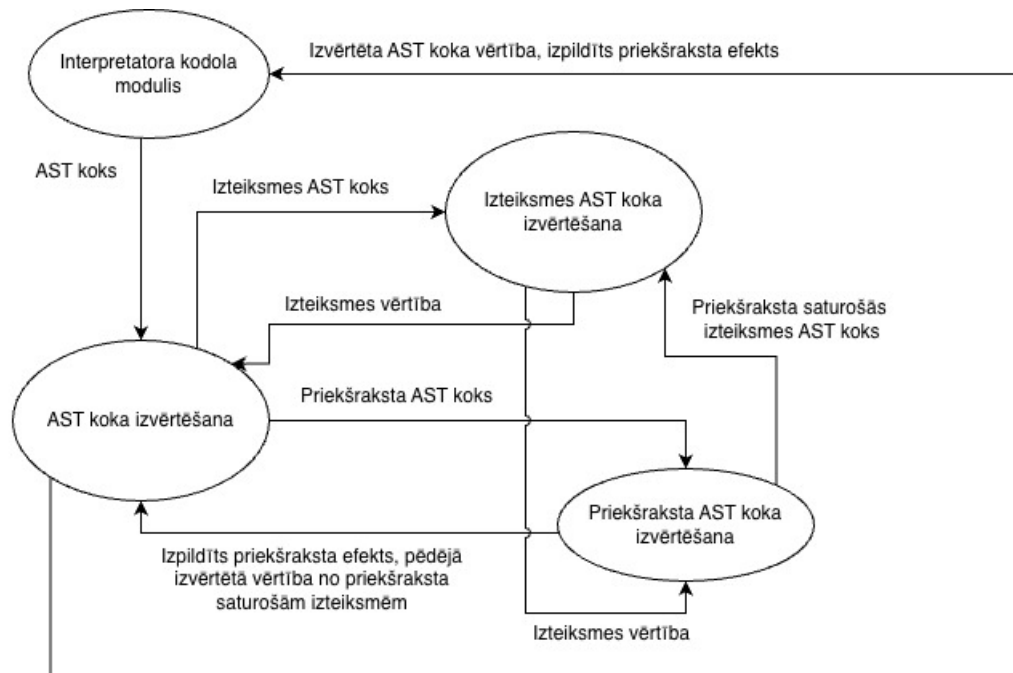
### 3.6. *Tabula* Funkcijas PARSE-1 apraksts

<b>Identifikators:</b>	PARSE-2
<b>Nosaukums:</b>	Parsētāja sinhronizēšana pēc notikušās parsēšanas kļūdas
<b>Mērķis:</b>	Parsēšanas kļūdas gadījumā nodot parsētājam informāciju, no kuras leksēmas ir jāpilni atsākt parsēšanu. Tas tiek darīts, lai lietotājs uzzinātu pēc iespējas vairāk kļūdu, ko ir veicis pirmkoda sintaksē, bet taj pat laikā nodrošināt, ka turpmākā parsēšana nerada viltotus AST vai neloģiskas kļūdas.
<b>Ievaddati:</b>	Leksēmu masīvs un ziņa par pēdējo apstrādāto leksēmu
<b>Apstrāde:</b>	Funkcija iet pāri leksēmu masīvam, kamēr neatrod leksēmu, no kuras ir sagaidāms, ka sākas jauns priekšraksts, piemēram, mainīgā deklarācija.
<b>Izvaddati:</b>	Jauna pozīcija leksēmu masīvā, no kuras ir jāpilni atsākt parsēšanu
<b>Kļūdu paziņojumi:</b>	-

### 3.7. *Tabula* Funkcijas PARSE-2 apraksts

### 3.4. Izvērtēšanas modulis

Izvērtēšanas modulis ir atbildīgs par AST koku izvērtēšanu un programmas izpildi. Šis modulis ir pēdējais solis interpretācijas procesā, kur AST koks tiek pārvērsts par faktiskām darbībām un rezultātiem, kas atspoguļo lietotāja rakstītā pirmkoda ieceres.



3.4. Att. Izvērtēšanas moduļa 2.līmeņa DPD

<b>Identifikators:</b>	EVAL-1
<b>Nosaukums:</b>	AST koka izvērtēšana
<b>Mērķis:</b>	Izvērtēt AST koku un veikt darbības, kas atbilst lietotāja rakstītā pirmkoda loģikai un semantikai.
<b>Ievaddati:</b>	pirmkods noparsēts AST kokos
<b>Apstrāde:</b>	Funkcija novērtē, vai dots AST koks atbilst priekšrakstam vai izteiksmei, un nodot kontroli atbilstošai apstrādes funkcijai.
<b>Izvaddati:</b>	Izpildīts priekšraksta efekts, pēdējā izvērtētā vērtība no priekšraksta saturošām izteiksmēm

---

**Kļūdu paziņojumi:**

-

---

### 3.8. *Tabula* Funkcijas EVAL-1 apraksts

---

**Identifikators:**

EVAL-2

---

**Nosaukums:**

Izteiksmes AST izvērtēšana

---

**Mērķis:**

Izvērtēt izteiksmes AST

---

**Ievaddati:**

Izteiksme AST formātā

---

**Apstrāde:**

Funkcija nosaka izteiksmes veidu un atbilstoši tam palaiž dotās izteiksmes izvērtēšanas algoritmu.

---

**Izvaddati:**

Izteiksmes vērtība, izpildīts efekts, ja tāds ir, piemēram, vērtības piešķiršana mainīgam

---

**Kļūdu paziņojumi:**

RUNTIME\_ERR-1

---

RUNTIME\_ERR-2

---

RUNTIME\_ERR-3

---

RUNTIME\_ERR-4

---

RUNTIME\_ERR-5

---

RUNTIME\_ERR-6

---

RUNTIME\_ERR-7

---

RUNTIME\_ERR-8

---

RUNTIME\_ERR-11

---

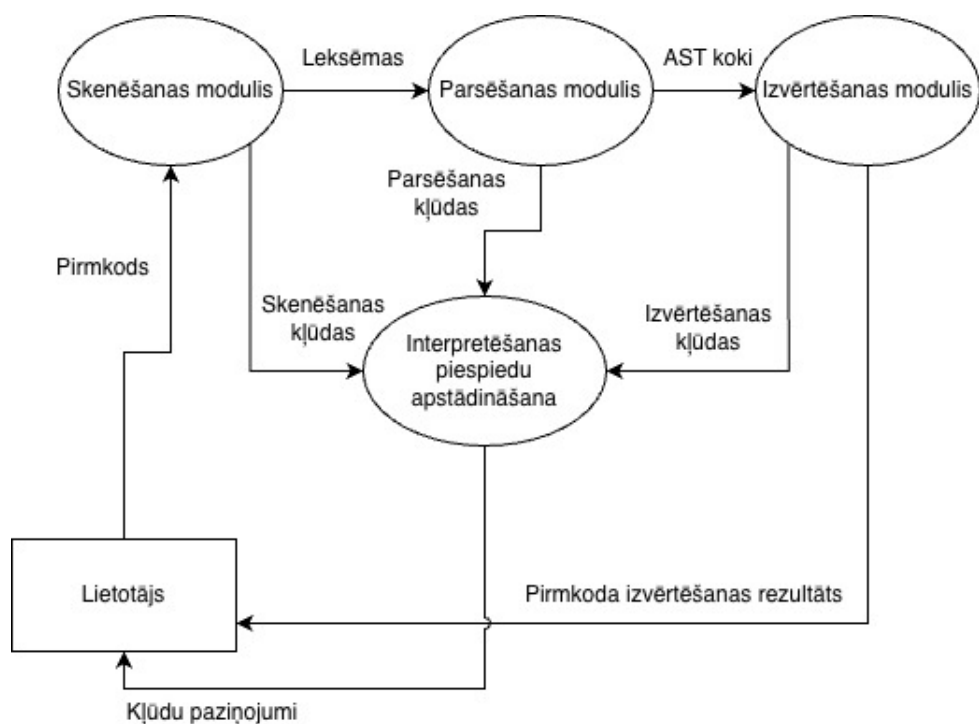
### 3.9. *Tabula* Funkcijas EVAL-2 apraksts

<b>Identifikators:</b>	EVAL-3
<b>Nosaukums:</b>	Priekšraksta AST izvērtēšana
<b>Mērķis:</b>	Izvērtēt priekšraksta AST
<b>Ievaddati:</b>	Priekšraksta AST koka izvērtēšana
<b>Apstrāde:</b>	Funkcija nosaka priekšraksta veidu un palaiž tam atbilstošu izvērtēšanas algoritmu. Tā kā priekšraksti sastāv no izteiksmēm, tas izmanto funkciju EVAL-2, lai sekmīgi izvērtētu visu priekšrakstu.
<b>Izvaddati:</b>	Izpildīts priekšraksta efekts, pēdējā izvērtētā vērtība no priekšraksta saturošām izteiksmēm
<b>Kļūdu paziņojumi:</b>	Funkcijas daļa, kas dara unikālu darbu, neveido nevienas kļūdas, tomēr iekšēji tā arī izmanto izteiksmju izvērtēšanu, tāpēc šī funkcijas realitātē rada visus paziņojumus no funkcijas EVAL-2

### 3.10. *Tabula* Funkcijas EVAL-3 apraksts

## 3.5. Interpretatora kodola modulis

Interpretatora kodols ir atbildīgs par interpretēšanas procesa kontroli un pārvaldību. Tas nodrošina, ka visi 3 interpretēšanas procesa moduļi (skenēšana, parsēšana un izvērtēšana) padot saražotos datus viens otram novērš šo citu moduļu darbību, ja citā interpretēšanas modulī ir notikusi kāda kļūda.



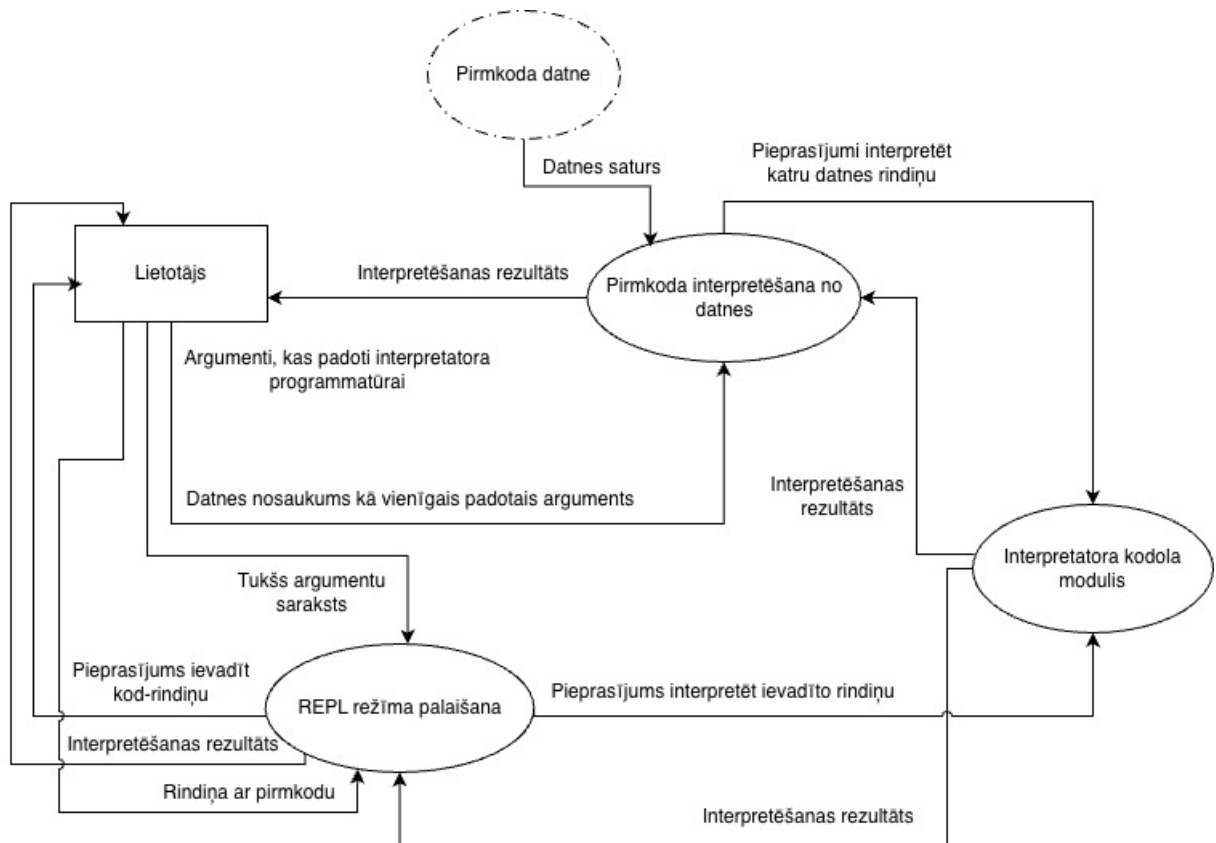
**3.5. Att. Interpretatora kodola moduļa 2.līmeņa DPD**

<b>Identifikators:</b>	CORE-1
<b>Nosaukums:</b>	Interpretēšanas piespiedu apstādināšana
<b>Mērķis:</b>	Neļaut sekojošiem interpretēšanas procesa soļiem turpināties, ja kādā no moduļiem ir notikusi kļūda. Tas ir, nākamie soļi, piemēram, parsēšana vai izvērtēšana, ir bezjēdzīgi, ja dati, kas tiem tiktu padoti nav pilnvērtīgi.
<b>Ievaddati:</b>	Kļūdas no skenēšanas, parsēšanas vai izvērtēšanas moduļa
<b>Apstrāde:</b>	Funkcija izseko kļūdas katrā no moduļiem un novērš turpmāko soļu izpildi, ja tādas radušās.
<b>Izvaddati:</b>	Paziņojums par kļūdu un uz kuras rindiņas tā ir notikusi
<b>Kļūdu paziņojumi:</b>	-

### 3.11. Tabula Funkcijas CORE-1 apraksts

### 3.6. Lietotāja saskarnes modulis

Lietotāja saskarnes modulis ir atbildīgs par mijiedarbību starp lietotāju un interpretatoru. Tas nodrošina veidu, kā lietotājs var ievadīt pirmkodu un saņemt rezultātus vai kļūdu ziņojumus.



3.6. Att. Lietotāja saskarnes moduļa 2.līmeņa DPD

<b>Identifikators:</b>	CLI-1
<b>Nosaukums:</b>	Pirmkoda interpretēšana no datnes
<b>Mērķis:</b>	Palaist interpretatoru ar pirmkodu no datnes
<b>Ievaddati:</b>	Datnes ceļš

---

<b>Apstrāde:</b>	Funkcija atver failu un nodot tās saturu interpretatora kodolam. Tas savukārt secīgi noskenē visas rindīņas un tikai tad nodot izveidotās leksēmas nākamajam interpretēšanas solim.
------------------	---

---

<b>Izvaddati:</b>	Interpretēšanas rezultāts, kļūdas, ja tādas radījušās
-------------------	---

---

<b>Kļūdu paziņojumi:</b>	MISC_ERR-1
--------------------------	------------

---

### 3.12. *Tabula* Funkcijas CLI-1 apraksts

---

<b>Identifikators:</b>	CLI-2
------------------------	-------

---

<b>Nosaukums:</b>	REPL režīms
-------------------	-------------

---

<b>Mērķis:</b>	Nodrošināt interaktīvu pirmkoda interpretēšanas vidi
----------------	--

---

<b>Ievaddati:</b>	Viens vai vairāki (vienā blokā) pirmkoda priekšraksti
-------------------	---

---

<b>Apstrāde:</b>	Ja ievadītā pirmkoda rindīņa ir izvērtējama viena pati, tad uzreiz nodod kontroli kodolam, kas veic skenēšanu, parsēšanu un izvērtēšanu tūlītēji. Tomēr, ja ievadītā rindīņa atver priekšrakstu bloku, funkcija uzkrāj pirmkodu buferī līdz lietotājs neaizver bloku. Gadījumā ar <code>if</code> priekšrakstu, funkcija pieprasa lietotājam turpināt ievadīt vēl pirmkodu un sagaida <code>else</code> bloku pirms sākas interpretēšana, jo <code>if</code> un <code>else</code> kalpo kā viens loģisks bloks. Ja priekšraksts <code>if</code> tiek aizvērts un seko tukša rindīņa, funkcija to uztver kā <code>if</code> priekšrakstu bez noklusējuma izpildes, tāpēc beidz pieprasīt ievaddatus un izvērtē priekšrakstu, ja vien tas neatrodas citā blokā, kura aizvēršana arī ir jāsagaida.
------------------	---

---

<b>Izvaddati:</b>	Interpretēšanas rezultāts, kļūdas, ja tādas radījušās
-------------------	---

---

<b>Kļūdu paziņojumi:</b>	-
--------------------------	---

---

### 3.13. *Tabula* Funkcijas CLI-2 apraksts

---

### **3.7. Nefunkcionālās prasības**

#### **3.7.1. Veiktspēja**

Sistēmai ir jāspēj interpretēt pirmkodu ar vidējo ātrumu vismaz 1000 līnijas sekundē standarta mūsdienu datora konfigurācijā.

#### **3.7.2. Drošība**

Sistēma pēc būtības nevar garantēt, ka interpretējamās programmas kods nenodarīs kaitējumu sistēmai. Lietotājam ir jāapzinās šie riski un jāievēro piesardzība, interpretējot nezināmas izcelsmes pirmkodu.

#### **3.7.3. Uzturēšana**

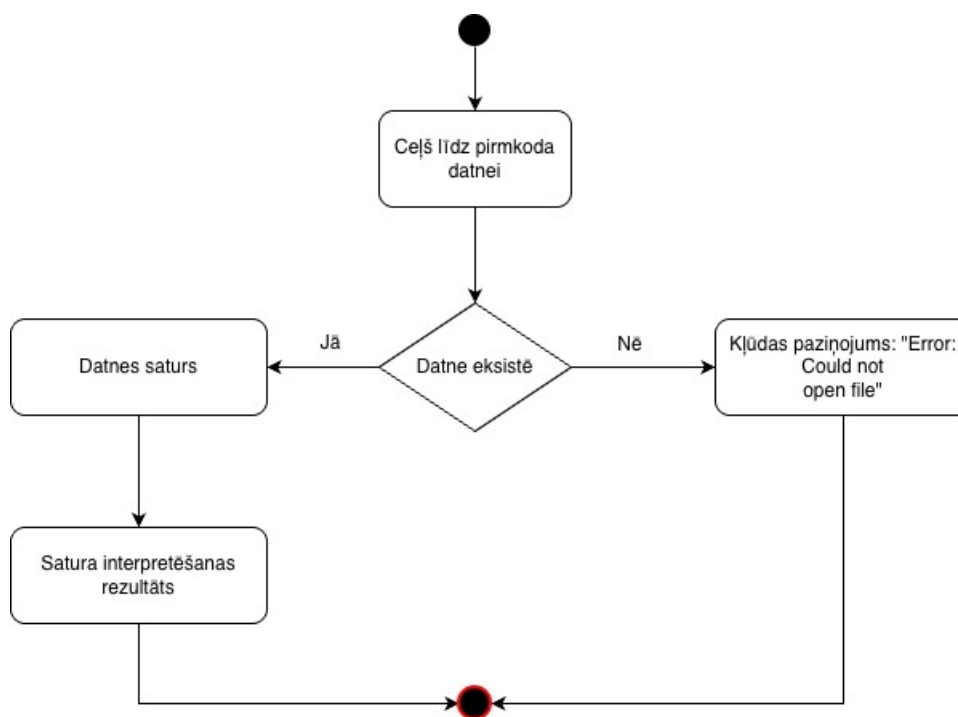
Sistēmas kodam jābūt labi dokumentētam un strukturētam loģiskos moduļos, lai atvieglotu turpmāko uzturēšanu un attīstību. Ir jāveido vienības un integrācijas testi, lai nodrošinātu sistēmas stabilitāti un uzticamību pie veiktām izmaiņām.

## 4. PROGRAMMATŪRAS PROJEKTĒJUMA APRAKSTS

### 4.1. Daļējs funkciju projektējums

#### 4.1.1. CLI-1 funkcijas projektējums: Pirmkoda interpretēšana no datnes

Pirmkoda datnes ir primārais veids, kā viegli izplatīt lietojumprogrammas. Tas ļauj lietotājiem saglabāt un koplietot savu kodu, kā arī palaist to dažādās vidēs. Šī funkcija nodrošina interpretēšanas palaišanu tieši no pirmkoda datnes.

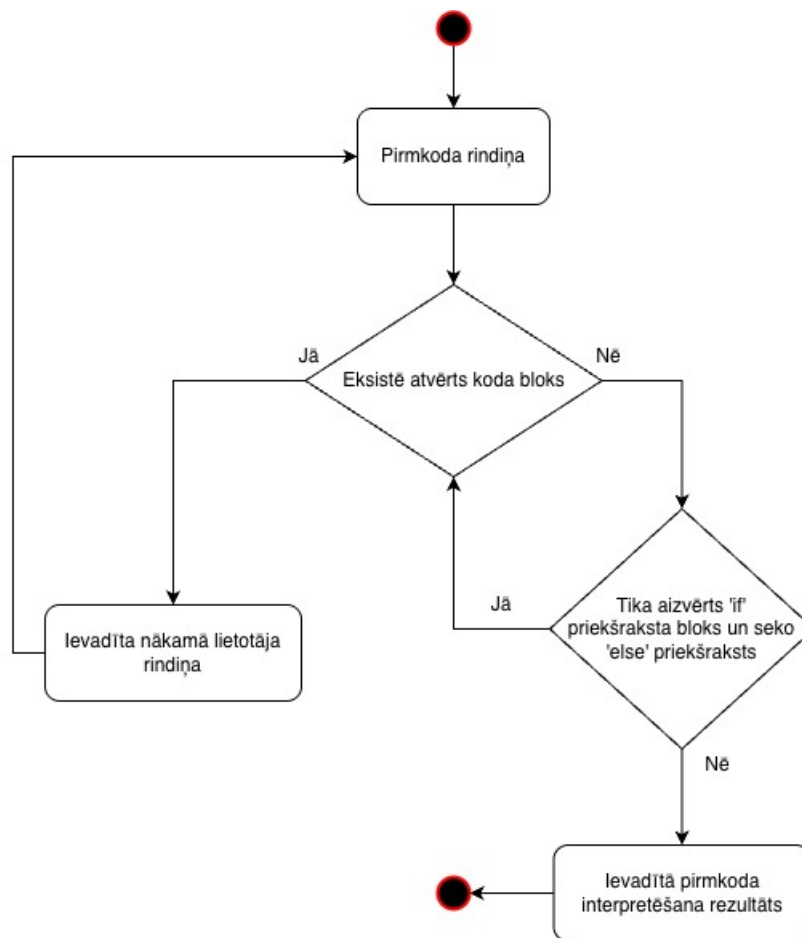


4.1. Att. Funkcijas CLI-1 projektējums

#### 4.1.2. CLI-2 funkcijas projektējums: REPL režīms

Dažkārt ir noderīgi ātri pārbaudīt nelielu koda fragmentu un eksperimentēt interaktīvā veidā - tam domāta REPL (Read-Eval-Print Loop) funkcija. Tā ļauj lietotājam ievadīt pirmkoda

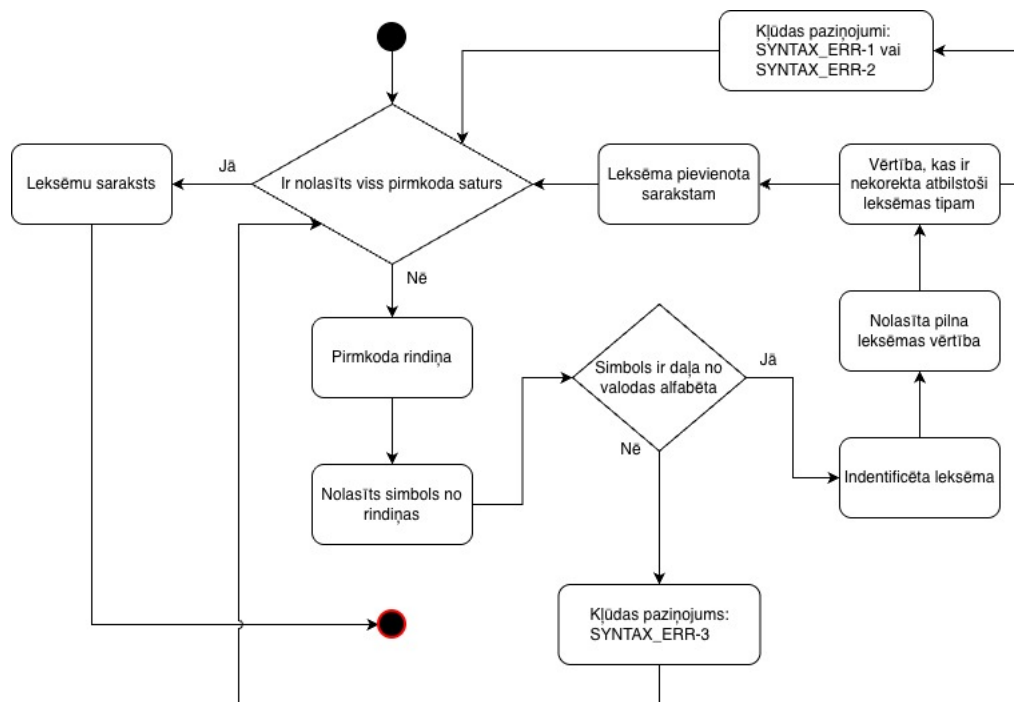
rindiņu, un uzreiz ieraudzīt rezultātu.



**4.2. Att. Funkcijas CLI-2 projektējums**

#### **4.1.3. LEX-1 funkcijas projektējums: Rakstzīmju pārveidošana uz leksēmu zīmogiem**

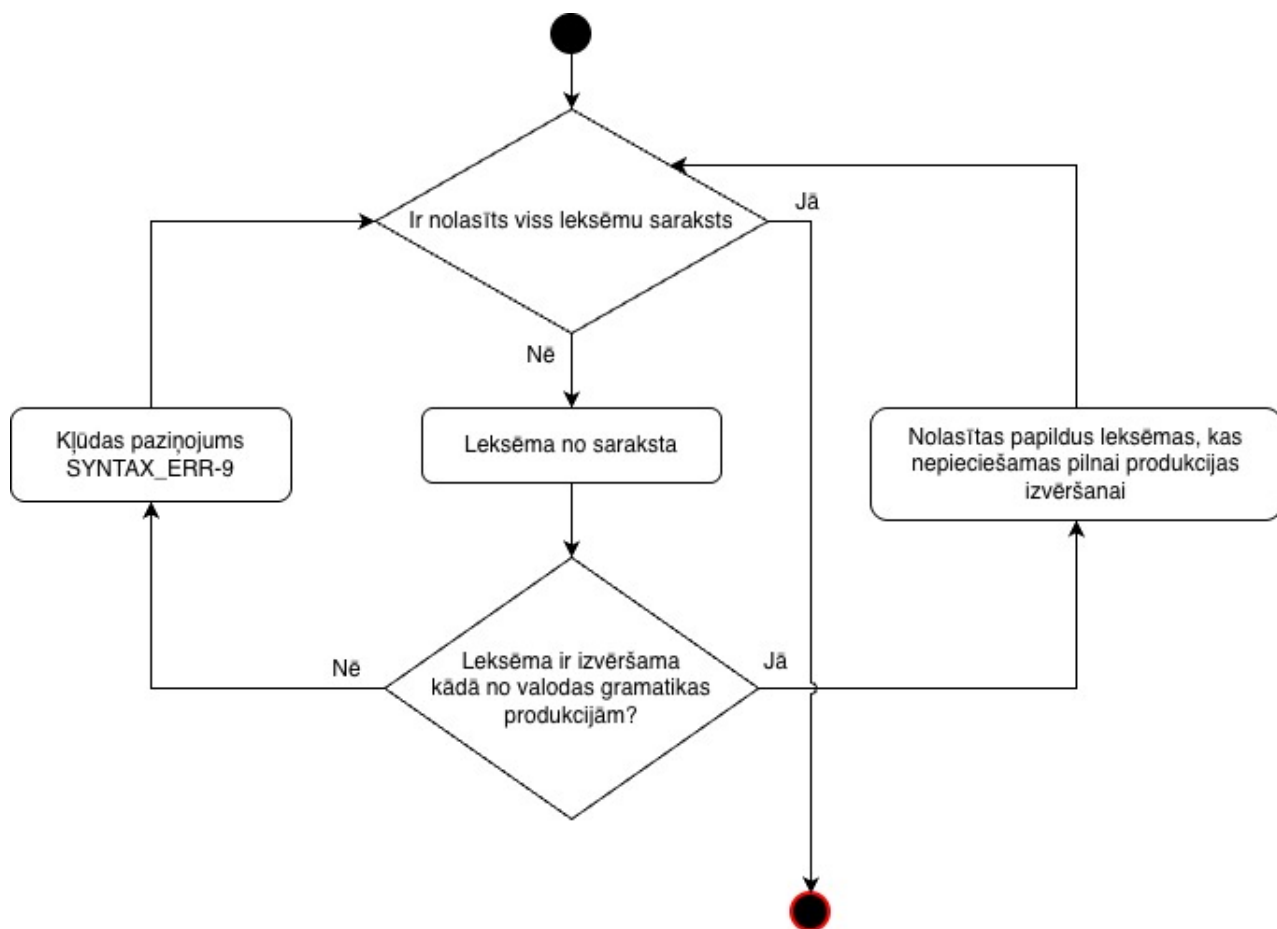
LEX-1 funkcija ir atbildīga par pirmkoda pārvēršanu leksēmās, kas ir interpretēšanas pamatvienības. Tā izskrien cauri katram simbolam pirmkodā un tās sasaista kopā ar meta informāciju - leksēmas tipu, vērtību un atrašanās vietu pirmkodā.



**4.3. Att. Funkcijas LEX-1 projektējums**

#### **4.1.4. PARSE-1 funkcijas projektējums: Leksēmu parsēšana uz AST kociem**

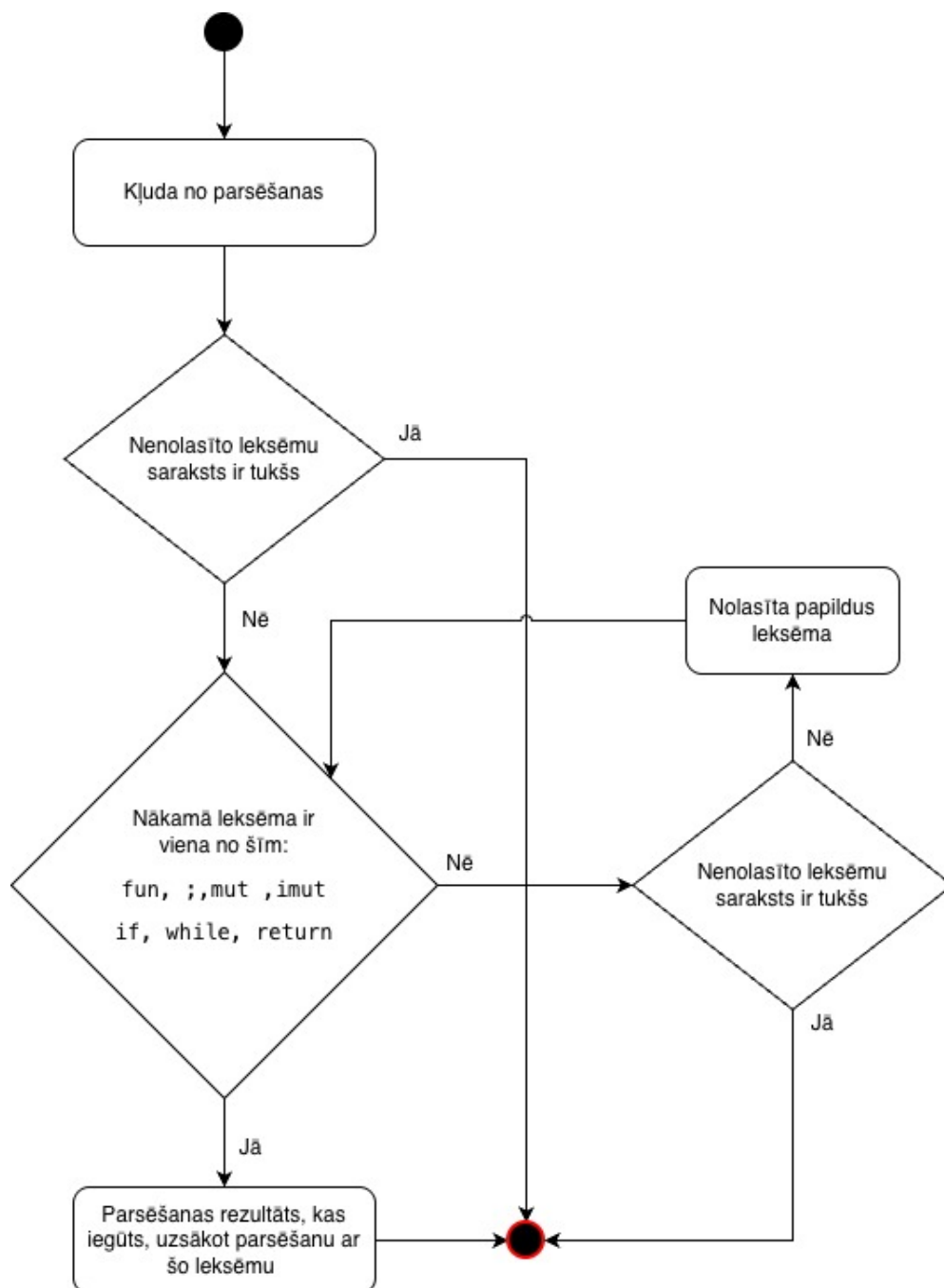
PARSE-1 funkcija ir atbildīga par leksēmu pārvēršanu abstraktos sintakses kokos. Tā rekursīvi nolaižas cauri gramatikas produkcijām, ievērojot viennozīmību, un veidojot koku struktūru, kas atspoguļo pirmkoda loģiku un hierarhiju.



4.4. Att. Funkcijas PARSE-1 projektējums

#### 4.1.5. PARSE-2 funkcijas projektējums: Parsētāja sinhronizēšana pēc notikušās parsēšanas kļūdas

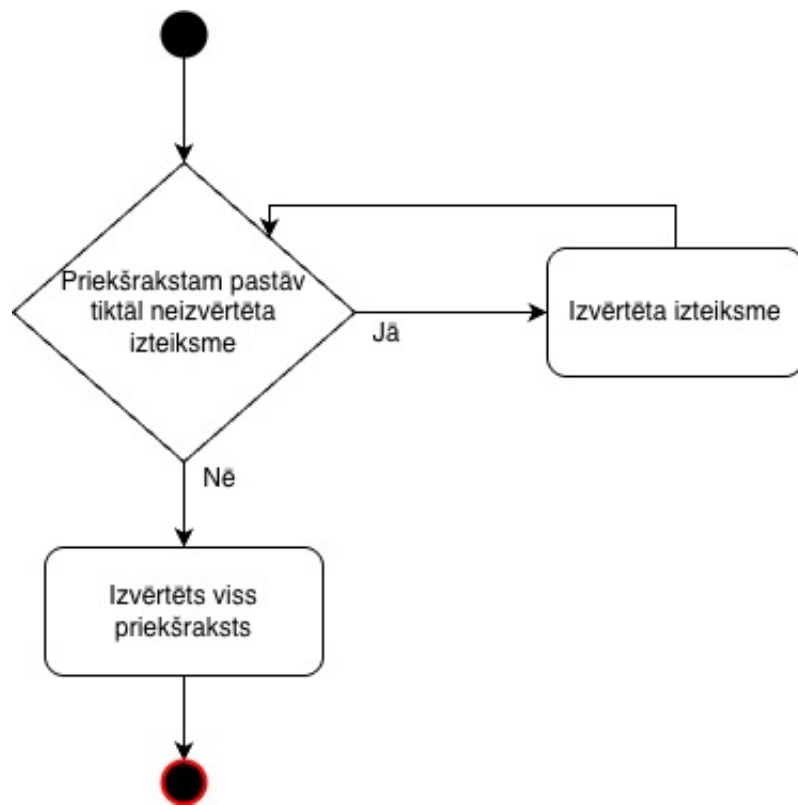
PARSE-2 funkcija ir izstrādāta, lai nodrošinātu parsētāja stabilitāti un spēju turpināt darbu pēc parsēšanas kļūdas. Tā meklē piemērotu vietu leksēmu secībā, no kuras var atsākt parsēšanu, tādējādi ļaujot lietotājam uzzināt pēc iespējas vairāk kļūdu vienā interpretēšanas reizē.



**4.5. Att. Funkcijas PARSE-2 projektējums**

#### **4.1.6. EVAL-1 funkcijas projektējums: AST koka izvērtēšana**

EVAL-1 funkcija ir atbildīga par AST koka izvērtēšanu un, visbeidzot, programmas izpildi. Tā nosaka, vai dotais AST koks atbilst priekšrakstam vai izteiksmei, un nodod kontroli atbilstošai apstrādes funkcijai, lai veiktu nepieciešamās darbības.



**4.6. Att. Funkcijas EVAL-1 projektējums**

## **5. DATU STRUKTŪRAS UN ALGORITMI**

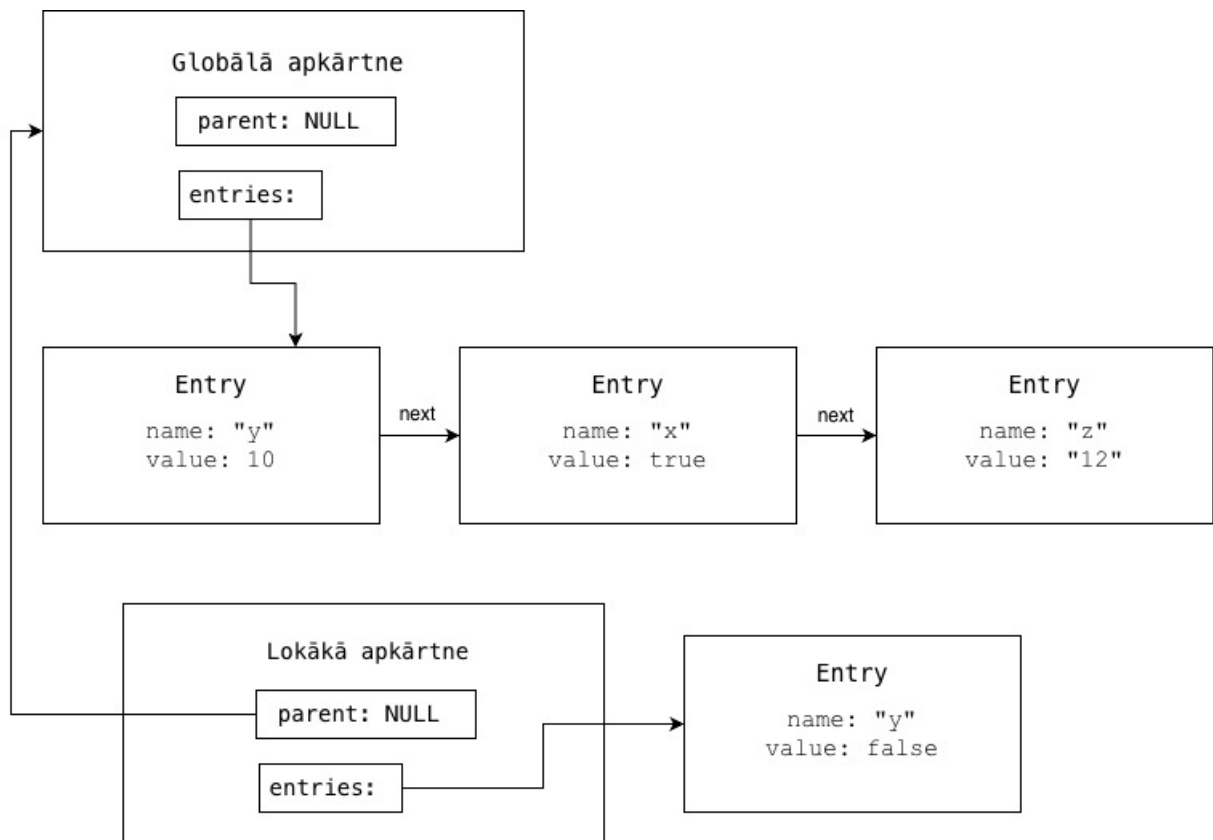
Šajā nodaļā tiek aprakstītas galvenās datu struktūras un algoritmi, kas izmantoti interpretatora implementācijā. Detalizēta izpratne par šīm struktūrām un algoritmiem ir būtiska sistēmas iekšējās darbības izpratnei.

### **5.1. Datu struktūras**

#### **5.1.1. Vides struktūra (Environment)**

Vides datu struktūra tiek izmantota, lai glabātu mainīgo un funkciju vērtības izpildes laikā. Struktūra ir realizēta kā saistīts saraksts ar norādi uz vecāka vidi, lai atbalstītu leksisko tvērumu. Šeit vide un apkārtne tiek lietoti kā sinonīmi.

Leksiskais tvērums nozīmē, ka funkcijas var piekļūt mainīgajiem no tās vides, kurā tās tika definētas, nevis no vides, kurā tās tiek izsauktas. Tas ir būtiski funkciju darbībai un mainīgo redzamībai.



**5.1. Att. Vides struktūra**

Katrs apkārtnes mezgls satur norādi uz vecāka vidi (`parent`) un uz saistīto sarakstu ar mainīgajiem (`entries`). Kad tiek meklēts mainīgais, sistēma vispirms pārbauda pašreizējo vidi, un, ja mainīgais nav atrasts, meklē vecāka vidē rekursīvi līdz pat globālajai videi. Šī pieeja nodrošina leksisko tvērumu un ļauj iekšējām funkcijām piekļūt ārējo tvērumu mainīgajiem.

Visefektīvākais veids kā glabāt mainīgo vērtības ir izmantojot jaucējtabulu, kas nodrošina ātru piekļuvi pēc atslēgas (mainīgā nosaukuma). Neskatoties uz to, šajā interpretatorā ir izvēlēta saistīta saraksta pieeja, lai saglabātu vienkāršību un vieglāku implementāciju. To, protams, būtu ieteicams mainīt nākotnē.

### 5.1.2. Abstraktā sintakses koka struktūras

Abstraktais sintakses koks (AST) reprezentē programmas struktūru pēc parsēšanas. AST sastāv no divām galvenajām hierarhijām: izteiksmēm (`Expr`) un priekšrakstiem (`Stmt`).

`Expr` (pamata tips)

```

|-- LiteralExpr
|   |-- NumberExpr (double value)
|   |-- StringExpr (char* value)
|   +-- BoolExpr (int value)
  
```

---

```

|-- VariableExpr (char* name)
|-- AssignExpr (char* name, Expr* value)
|-- UnaryExpr (TokenType operator, Expr* operand)
|-- BinaryExpr (Expr* left, TokenType operator, Expr* right)
|-- LogicalExpr (Expr* left, TokenType operator, Expr* right)
+-- CallExpr (Expr* callee, Expr** arguments, size_t arg_count)

```

Stmt (pamata tips)

```

|-- ExprStmt (Expr* expression)
|-- VarDeclStmt (char* name, Expr* initializer, MutabilityType)
|-- FunDeclStmt (char* name, char** params, Stmt* body)
|-- ReturnStmt (Expr* value)
|-- IfStmt (Expr* condition, Stmt* then_branch, Stmt* else_branch)
|-- WhileStmt (Expr* condition, Stmt* body)
+-- BlockStmt (Stmt** statements, size_t count)

```

Katrs mezgls satur type lauku, kas norāda mezgla tipu, un line lauku kļūdu ziņojumiem. Izteiksmju mezgli atgriež vērtības, bet priekšrakstu mezgli izpilda darbības un maina stāvokli. Šī struktūra ļauj rekursīvi apstrādāt programmas kodu, sākot no saknes mezgla un ejot dziļumā pa katru apakšroku.

### 5.1.3. Vērtību reprezentācija

Vērtības tiek reprezentētas ar Value struktūru, kas izmanto union efektīvai atmiņas izmantošanai. Katra vērtība satur tipu un atbilstošo datu lauku.

Value struktūra (32 baiti):

```

+-----+
| ValueType type          | (4 baiti)
+-----+
| union {                 | (16 baiti)
|   long double number;   |
|   int boolean;          |
|   char* string;         |
|   Callable* callable;   |
| }                       |
+-----+
| MutabilityType mut      | (4 baiti)
| PurityType purity       | (4 baiti)
+-----+

```

---

ValueType enum:

- VAL\_NUMBER - skaitliska vērtība (long double)
- VAL\_BOOL - būla vērtība (int)
- VAL\_STRING - vikrnes vērtība (char\*)
- VAL\_FN - lietotāja definētā funkcija (Callable\*)
- VAL\_NATIVE - iebūvēta funkcija (Callable\*)

Union konstrukcija ļauj vienā atmiņas vietā glabāt dažādus datu tipus, izmantojot tikai tik daudz atmiņas, cik nepieciešams lielākajam tipam.

## 5.2. Algoritmi

### 5.2.1. Leksiskā analīze

Leksiskā analīze pārveido pirmkodu uz leksēmām, apstrādājot katru rakstzīmi un atpazīstot atslēgvārdus, identifikatorus, literālus un operatorus.

Algoritms: scan\_tokens(source\_code)

1. Inicializē token\_buffer
2. Kamēr nav sasniegts koda beigas:
  - a. Izlaiž tukšumus un komentārus
  - b. Nosaka nākamās leksēmas tipu:
    - Ja cipars -> skenē NUMBER
    - Ja burts -> skenē IDENTIFIER vai KEYWORD
    - Ja pēdiņa -> skenē STRING
    - Ja operators -> atpazīst (+ - \* / = == != < > <= >=)
    - Ja pieturzīme -> atpazīst (; , ( ) { })
  - c. Izveido Token{type, lexeme, line}
  - d. Pievieno token\_buffer
3. Pievieno EOF leksēmu
4. Atgriež token\_buffer

Atslēgvārdu atpazīšana:

- Pēc identifikatora nolasīšanas pārbauda, vai tas ir atslēgvārds (imut, mut, fn, if, else, while, return)
- Ja atbilst -> TOKEN\_KEYWORD
- Citādi -> TOKEN\_IDENTIFIER

---

### 5.2.2. Parsēšana

Parsēšana pārveido leksēmas uz AST kokiem, izmantojot rekursīvo nolaišanās metodi, ievērojot gramatikas noteikumus un operatoru prioritātes.

Algoritms: `parse(tokens)`

1. Inicializē `token_index = 0`
2. Izveido `program_statements = []`
3. Kamēr nav sasniegtas programmas beigas:
  - a. Parsē priekšrakstu un pievieno `program_statements` masīvam
4. Atgriež `BlockStmt{program_statements}`

Algoritms: `parse_statement()`

1. Nosaka nākamās leksēmas tipu:
  - Ja `"imut"` vai `"mut"` -> `parse_var_declaration()`
  - Ja `"fn"`, `"pure"` vai `"impure"` -> `parse_function_declaration()`
  - Ja `"if"` -> `parse_if_statement()`
  - Ja `"while"` -> `parse_while_statement()`
  - Ja `"{"` -> `parse_block_statement()`
  - Ja `"return"` -> `parse_return_statement()`
  - Citādi -> `parse_expression_statement()`

Algoritms: `parse_expression()`

Izmanto operatoru prioritātes (no zemākās uz augstāko):

- `parse_or_expression()`
- + - `parse_and_expression()`
- + - `parse_equality_expression()`
- + - `parse_comparison_expression()`
- + - `parse_additive_expression()`
- + - `parse_multiplicative_expression()`
- + - `parse_unary_expression()`
- + - `parse_call_expression()`
- + - `parse_primary_expression()`

Algoritms: `parse_or_expression()`

1. Parsē `left = parse_and_expression()`
2. Kamēr nākamā leksēma ir `"||"`:
  - a. patērē `"||"`
  - b. Parsē `right = parse_and_expression()`
  - c. `left = LogicalExpr{left, OR, right}`

---

### 3. Atgriež left

Algoritms: parse\_and\_expression()

Identisks parse\_or\_expression(), bet ar "&&" operatoru

Algoritms: parse\_equality\_expression()

1. Parsē left = parse\_comparison\_expression()
2. Kamēr nākamā leksēma ir "==" vai "!=":
  - a. Operators = nākamā leksēma
  - b. patērē operatoru
  - c. Parsē right = parse\_comparison\_expression()
  - d. left = BinaryExpr{left, operators, right}
3. Atgriež left

Algoritms: parse\_comparison\_expression()

1. Parsē left = parse\_additive\_expression()
2. Kamēr nākamā leksēma ir "<", ">", "<=", ">=":
  - a. Operators = nākamā leksēma
  - b. patērē operatoru
  - c. Parsē right = parse\_additive\_expression()
  - d. left = BinaryExpr{left, operators, right}
3. Atgriež left

Algoritms: parse\_additive\_expression()

1. Parsē left = parse\_multiplicative\_expression()
2. Kamēr nākamā leksēma ir "+" vai "-":
  - a. Operators = nākamā leksēma
  - b. patērē operatoru
  - c. Parsē right = parse\_multiplicative\_expression()
  - d. left = BinaryExpr{left, operators, right}
3. Atgriež left

Algoritms: parse\_multiplicative\_expression()

1. Parsē left = parse\_unary\_expression()
2. Kamēr nākamā leksēma ir "\*", "/" vai "%":
  - a. Operators = nākamā leksēma
  - b. patērē operatoru
  - c. Parsē right = parse\_unary\_expression()
  - d. left = BinaryExpr{left, operators, right}
3. Atgriež left

---

Algoritms: parse\_unary\_expression()

1. Ja nākamā leksēma ir "!":
  - a. patērē "!"
  - b. Parsē operand = parse\_unary\_expression()
  - c. Atgriež UnaryExpr{NOT, operand}
2. Citādi parsē call\_expression()

Algoritms: parse\_call\_expression()

1. Parsē expr = parse\_primary\_expression()
2. Kamēr nākamā leksēma ir "(":
  - a. patērē "("
  - b. Parsē argumentu sarakstu un tos pievieno arguments masīvam:
    - arguments = []
  - c. Ja nākamā leksēma nav ")":
    - arguments += parse\_expression()
    - Kamēr nākamā leksēma ir ",":
      - \* patērē ","
      - \* arguments += parse\_expression()
  - d. patērē ")"
  - e. expr = CallExpr{expr, arguments}
3. Atgriež expr

Algoritms: parse\_primary\_expression()

1. Nosaka nākamās leksēmas tipu:
  - Ja NUMBER -> Atgriež NumberExpr{value}
  - Ja STRING -> Atgriež StringExpr{value}
  - Ja "true" -> Atgriež BoolExpr{true}
  - Ja "false" -> Atgriež BoolExpr{false}
  - Ja IDENTIFIER -> Parsē variable\_or\_assignment()
  - Ja "(" -> parse\_grouped\_expression()

Parsēšana rekursīvi nolaižas cauri izteiksmju hierarhijai, ievērojot operatoru prioritātes. Katra funkcija atbild par vienu prioritātes līmeni, sākot no zemākās (loģiskie operatori) līdz augstākajai (primāras izteiksmes). Rekursīvā nolaišanās nodrošina pareizu asociativitāti. Ja parsēšana neizdodas, tiek izvirzīta sistēmas kļūda ar norādi uz leksēmas pozīciju pirmkodā.

---

### 5.2.3. Izvērtēšana

Izvērtēšana pārveido AST kokus par izpildlaika darbībām, izmantojot koka apstaigāšanas interpretācijas metodi. Katra AST mezgla veids tiek apstrādāts atbilstošā veidā, un mainīgo vērtības tiek uzglabātas vidē.

Algoritms: `evaluate(ast, environment)`

1. Inicializē `context{environment, error_flag}`
2. Izvērtē `ast root` mezglu
3. Ja kļūda `flags` ir uzstādīta, atgriež `null`
4. Citādi atgriež pēdējo izvērtēto vērtību

Algoritms: `eval_statement(stmt, context)`

1. Nosaka priekšraksta tipu:
  - Ja `ExprStmt` -> Izvērtē izteiksmi
  - Ja `VarDeclStmt` -> Deklarē mainīgo vidē
  - Ja `FunDeclStmt` -> Reģistrē funkciju vidē
  - Ja `BlockStmt` -> Izveido jaunu vidi un izvērtē bloku
  - Ja `IfStmt` -> Novērtē nosacījumu, izpilda attiecīgo zaru
  - Ja `WhileStmt` -> Atkārtoti izpilda, kamēr nosacījums ir patiess
  - Ja `ReturnStmt` -> Nostāda `return_value` un iziet

Algoritms: `eval_expression(expr, context)`

1. Nosaka izteiksmes tipu:
  - Ja `Literal` -> Atgriež vērtību
  - Ja `Variable` -> Meklē mainīgo vidē, atgriež vērtību
  - Ja `Assign` -> Izvērtē izteiksmi, atjaunina mainīgā vērtību vidē
  - Ja `Binary` -> Izvērtē abus operandus, pielieto operatoru
  - Ja `Logical` -> Izvērtē loģisko izteiksmi, ņemot vērā loģisko saīsināšanu
  - Ja `Unary` -> Izvērtē operandu, pielieto operatoru
  - Ja `Call` -> Izvērtē saucēju un argumentus, izsauc funkciju

Algoritms: `eval_block(statements, context)`

1. Izveido jaunu apkārtni `new_env{parent: context.env}`
2. Priekš katra priekšraksta `s` in `statements`:
  - a. `eval_statement(s, new_context)`
  - b. Ja `return_value` ir uzstādīts, iziet
  - c. Ja kļūdas karogs ir uzstādīts, iziet

Algoritms: `eval_call_expr(callee_expr, arguments, context)`

- 
1. Izvērtē funkciju `function = eval_expression(callee_expr, context)`
  2. Pārbauda, vai `function` ir callable tipa
  3. Pārbauda argumentu skaitu
  4. Priekš noteikta callable tipa:
    - a. Ja `VAL_FN`:
      - Izveido `call_env{parent: function.closure_env}`
      - Izveido parametru mainīgos ar argumentu vērtībām iekš `call_env` apkārtne
      - Izpilda funkcijas ķermeni ar iekš `call_env` apkārtne
      - Atgriež `return_value`
    - b. Ja `VAL_NATIVE`:
      - Izsauc C funkciju ar arguments
      - Atgriež rezultātu

Algoritms: `lookup_variable(name, env)`

1. `current_env = env`
2. Kamēr `current_env != null`:
  - a. Meklē mainīgā nosaukumu `name` `current_env.entries` sarakstā
  - b. Ja atrasts, atgriež `entry.value`
  - c. `current_env = current_env.parent`
3. Ja nav atrasts, atgriež kļūdu `RUNTIME_ERR-2`

Algoritms: `apply_binary_operator(left, op, right)`

1. Pēc operatora tipa pārbauda operandu tipus
2. Ja aritmētiskais operators:
  - Abiem operandiem jābūt `NUMBER`
  - Atgriež skaitlisku rezultātu
3. Ja salīdzinājuma operators:
  - Abiem operandiem jābūt `NUMBER`
  - Atgriež `BOOL` rezultātu
4. Ja vienādības operators:
  - Abiem operandu tipiem ir jāsakrīt
  - Atgriež `BOOL` rezultātu

Mainīgo un funkciju vērtības tiek uzglabātas vidē, kas tiek dinamiski izveidota pie katra funkcijas izsaukuma. Videi ir norāde uz vecāka apkārtni, kas ļauj iedarbināt leksisko tvērumu, kur iekšējās vides var piekļūt ārējo apkārtni mainīgajiem. Kļūdas tiek pamanītas katrā posmā, un ja kļūda ir konstatēta, interpretācija apstājas.

## 6. TESTĒŠANA

### 6.1. Testēšanas dokumentācija

Funkcijas identifikators	Testēšanas datums	Soli	Sagaidāmais rezultāts	Testa rezultāts
LEX-1	19.12	1. Izveido pirmkoda virkni "3 + @ - \$()" 2. Skenē doto pirmkoda virkni	leksēmas NUMBER(3), PLUS, MINUS, LEFT_PAREN, RIGHT_PAREN 2 kļūdainas leksēmas	OK
LEX-1	19.12	1. Izveido pirmkoda virkni "&&" 2. Skenē doto pirmkoda virni	Loģiskā tipa leksēma "&&" 2 kļūdainas leksēmas	OK
LEX-2	15.12	1. Izveido virkni "else" 2. Skenē izveidoto virkni	Atslēgvārda tipa leksēma ar vērtību "else"	FAIL
	17.12			OK
LEX-2	15.12	1. Izveido virkni "39.456723asd=as" 2. Skenē izveidoto virkni	Skaitliskā tipa leksēma ar vērtību "39.456723"	OK
LEX-3	15.12	1. Izveido virkni "_immutant" 2. Skenē izveidoto virkni	Identifikatora tipa leksēma ar vērtību "_immutant"	OK
PARSE-1	18.12	1. Izveido aritmētisku izteiksmi ar leksēmām: 2, +, 4, /, 5 2. Izsauc parsētāju ar dotām leksēmām 3. Pārbauda AST koka formu	BinaryExpr(+) [NUMBER(2), BinaryExpr(/) [NUMBER(4), NUMBER(5)]]	OK

Funkcijas identifi-kators	Testēšana datums	Soļi	Sagaidāmais rezultāts	Testa rezultāts
PARSE-1	27.12	1. Parsē binārās izteiksmes ar dažādiem operatoriem 2. Pārbauda, ka operāciju kārtība tiek ievērota 3. Pārbauda AST struktūru	BinaryExpr(-) [BinaryExpr(+) [NUMBER(1), BinaryExpr(*) [NUMBER(2), NUMBER(3)]], NUMBER(4)]	FAIL
	28.12			FAIL
	30.12			OK
PARSE-1	02.01	1. Parsē priekšraksta bloku no leksēmām: { , }	Izvedots pirmkoda bloka AST mezgls ar 0 priekšrakstiem	OK
EVAL-2	18.12	1. Saskaita virknes: “from the ” + “outside world” 2. Izvērtē izteiksmes vērtību	virkne “from the outside world”	OK
EVAL-2	22.12	1. Izvērtē aritmētiskās operācijas: 10 + 32.5 2. Pārbauda rezultāta vērtību un tipu	Skaitliskā vērtība 42.5	OK
EVAL-2	28.12	1. Izvērtē atņemšanas operāciju: -100 - 58.5 2. Pārbauda rezultāta vērtību	Skaitliskā vērtība -158.5	OK
EVAL-2	30.12	1. Izvērtē reizināšanas operāciju: 6 * 7 2. Pārbauda rezultāta vērtību	Skaitliskā vērtība 42	OK
EVAL-2	29.12	1. Izvērtē dalīšanas operāciju: 84 / 2 2. Pārbauda rezultāta vērtību	Skaitliskā vērtība 42	OK
	01.01			FAIL
	02.01			OK

Funkcijas identifikators	Testēšanas datums	Soli	Sagaidāmais rezultāts	Testa rezultāts
EVAL-2	03.01	1. Mēģina dalīt ar nulli: 42 / 0 2. Pārbauda kļūdas stāvokli	Kļūda RUNTIME_ERR-1	OK
EVAL-2	02.01	1. Piešķir mainīgajam x vērtību 42 2. Pārbauda, ka vērtība tiek saglabāta vidē	Mainīgā x vērtība ir 42	OK
EVAL-2	03.01	1. Mēģina piešķirt vērtību nedefinētam mainīgajam y 2. Pārbauda kļūdas stāvokli	Kļūda RUNTIME_ERR-2	OK
EVAL-3	18.12	1. Izvērtē IF/ELSE izteiksmi ar patieso nosacījumu 2. Pārbauda, ka izpildās THEN bloks 3. Pārbauda, ka ELSE bloks netiek izpildīts	IF bloks izpildīts, ELSE netiek	OK
	20.12			OK
EVAL-3	26.12	1. Izvērtē WHILE ciklu ar nosacījumu, kurš kļūst aplams pēc 3 iterācijām 2. Pārbauda, ka bloks izpildās 3 reizes 3. Apstiprina pareizu cikla apstāšanos	Cikls izpildās 3 reizes, pēc tam apstājas	FAIL
	28.12			OK
EVAL-3	03.01	1. Izvērtē funkciju izsaukumu ar nepareizu argumentu skaitu 2. Funkcija sagaida 2 argumentus, bet tiek padots 1 3. Pārbauda kļūdas kodu	Kļūda RUNTIME_ERR-8	OK

## **7. PROJEKTA ORGANIZĀCIJA**

Vispirms tika izstrādāta pilna valodas specifikācija un tad sāka programmatūras izstrāde, kas tika veikta individuāli. Kvalifikācijas darba tēmas izvēlē bija tīri no personīgās iniciatīvas. Programmatūras pirmkods ir rakstīts valodā C pēc C11 standarta, izmantojot GNU Compiler Collection (GCC) kompilatoru.

Koda testēšana notika vienlaicīgi ar galvenās programmatūras izstrādi, nodrošinot modulāru arhitektūru un plašu testu klāstu. Pilns kvalifikācijas darba dokuments ir sagatavots pēc programmatūras izstrādes un testēšanas pabeigšanas.

## 8. KVALITĀTES NODROŠINĀŠANA

Programmatūras kvalitātes nodrošināšanai lielāko uzsvaru tika likts uz vienībtestēšanu un integrācijas testēšanu. Katrs izstrādātais modulis tika pārbaudīts, lai nodrošinātu tā atbilstību specifikācijai un pareizu darbību dažādos scenārijos. Testēšanas rezultāti tika dokumentēti, un visi atklātie defekti tika novērsti pirms galīgās programmatūras versijas izveides.

Lai programmpirmkods pats par sevi būtu kvalitatīvs, tika ievērotas labās programmšanas prakses, piemēram, koda lasāmības uzturēšana, komentāru rakstīšana un modulāru funkciju izstrāde. Izmantotā valoda C ir stingri tipizēta, kas palīdz novērst daudzas potenciālās kļūdas jau kompilācijas laikā. Lai paātrinātu izstrādes procesu, tika izmantota LSP (Language Server Protocol) integrācija iekš koda redaktora, kas nodrošināja automātisku koda formatēšanu un sintakses pārbaudi vēl pirms kompilatora palaišanas.

Programmatūras dokumentācija tika izstrādāta, balstoties uz LVS 72:1996 un LVS 68:1996 vadlīnijām. Lai nodrošinātu lietotājam noderīgu programmatūru, interpretatora izstrādes laikā tika veikta UAT testēšana, pārbaudot izstrādāto moduļu lietderību reālos scenārijos, kas veicināja prasību pielāgošanu lietotāja vajadzībām. Akcepttestēšanas grupa sastāv tikai no paša autora un darba vadītāja.

Papildus manuālai vienībtestu un integrācijas testu palaišanai, tika izveidota automatizēta testēšanas sistēma, izmantojot GitHub Actions, kas mākonī palaida visu testu izpildi, katru reizi, kad tiešsaistes repozitorijs tika atjaunināts. Tas nodrošināja, ka jaunie labojumi vai jaunizveidotās funkcijas neietekmē jau esošo funkcionalitāti neparedzamā veidā.

## 9. KONFIGURĀCIJAS PĀRVALDĪBA

Programmatūras un tās dokumentācijas pirmkods ir versionēti, izmantojot Git versiju kontroles sistēmu. Versiju kontrolei ir piesaistīts tiešsaistes repozitorijs GitHub platformā, kur drošā veidā tiek glabātas visas izmaiņas un versijas. Katrs izstrādes posms ir dokumentēts ar atbilstošiem komentāriem un izmaiņu aprakstiem, kas ļauj viegli izsekot izmaiņām un atgriezties pie iepriekšējām versijām, ja nepieciešams.

Programmas kompilācijas un testēšanas rīka konfigurācijas ir definētas katra savā failā (Makefile un project.yml attiecīgi), kas arī ir glabāti versiju kontroles sistēmā.

## 10. DARBIETILPĪBAS NOVĒRTĒJUMS

Darba uzdevums	Optimistiskais laiks (dienas)	Reālistiskais laiks (dienas)	Pesimistiskais laiks (dienas)	Faktiskais laiks (dienas)
Prasību ievākšana un valodas specifikācijas sastādīšana	7	14	18	15
Kompilatoru teorijas un testēšanas rīka izmantošanas apgūšana	10	13	20	13
Interpretatora kodola izstrāde	20	25	35	20
Lietotāja saskarnes izstrāde un kodola integrācija	4	6	8	4
Testēšana un kļūdu novēršana	8	12	15	12
Kopā	49	70	96	64

### 10.1. Tabula Darbietilpības novērtējums

Vadoties pēc 10. tabulas, var secināt, ka kopējais faktiskais darba laiks bija 64 dienas, kas ir pārspēj pat reālistisko laika novērtējumu 70 dienas. Tas liecina par efektīvu laika plānošanu un izpildi projekta gaitā.

Tā kā viena mēneša ietvaros ir aptuveni 21 darba diena, tad kopējais projekta izstrādes laiks ir aptuveni 3 mēneši.

Lielāko daļu laika aizņēma prasību ievākšana un valodas specifikācijas sastādīšana, kā arī interpretatora kodola izstrāde. Šie posmi bija kritiski projekta veiksmīgai īstenošanai, un tiem tika pievērsta īpaša uzmanība, lai nodrošinātu augstu kvalitāti un atbilstību lietotāja vajadzībām. Proti, kompilatoru teorijas un testēšanas rīka izmantošanas apgūšana aizkavēja projekta izstrādi, taču bez šīm zināšanām nebūtu iespējams izstrādāt interpretatoru prātīgā laikā, tāpēc šis projekta posms nav mazāk svarīgs par pārējiem.

## 11. SECINĀJUMI

Šajā kvalifikācijas darbā tika izstrādāts vienkāršs interpretators imperatīvai programmēšanas valodai, kas atbalsta pamata datu tipus, mainīgos, izteiksmes, priekšrakstus un funkcijas. Interpretators tika izstrādāts, izmantojot modulāru pieeju, sadalot to skenēšanas, parsēšanas un izvērtēšanas moduļos, kas ļauj viegli uzturēt un paplašināt programmatūru nākotnē.

Jaunizveidotā valoda ir radīta ar mērķi popularizēt konstantu datu plūsmu caur programmatūru, kas tiek panākts ar tīru funkciju un nemutablu mainīgo izmantošanu. Akceptētēšana parādīja, ka arī bez mutablu mainīgo un netīru funkciju izmantošanas ir ļoti sarežģīti vai par dažreizi neiespējami izveidot noteiktus algoritmus, piemēram, Fibonacci skaitļu virknes ģenerēšanu. Tas pierāda, ka valodas specifikācija arī ir veiksmīgi sasniegusi savu mērķi, jo lietotājam ir dotas visas galvenās programmēšanas konstrukcijas, lai varētu izveidot funkcionālu programmu, vienlaicīgi liekot lietotājam domāt par to, kā viņa kods varētu tikt uzlabots, lai tas būtu uzturamāks un efektīvāks.

Tā kā Immutant valodas sākotnējā versija (aprakstīta šajā dokumentā) nav tik bagātīga kā citas populāras programmēšanas valodas, tai ir daudz iespējas tālāk attīstīties. Viena no šādām iespējām ir paplašināt valodas sintaksi un semantiku, pievienojot jaunas datu struktūras, piemēram, masīvus vai vārdnīcas, kā arī paplašināt esošo funkciju klāstu ar jauniem iebūvētiem rīkiem. Vēl viena iespēja ir uzlabot veiktspēju, izstrādājot interpretatora virtuālo mašīnu un īstenojot optimizācijas paņēmienus, piemēram, neizmantotās atmiņas automātisko atbrīvošanu vai izteiksmju priekšapstrādi.

## 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.
- [4] Simon Peyton Jones, Lennart Augustsson, Dave Barton, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK and New York, NY, 2003. Formal language definition of Haskell 98.
- [5] Robert Nystrom. *Crafting Interpreters*. Genever Benning, 2021. Available online.
- [6] Python Core Team. *Python Language Reference, release 3.x*. Python Software Foundation, Hampton, NH, 2025. Official Python language specification and reference manual.
- [7] Rich Hickey and Clojure Contributors. Clojure official documentation, 2025. Official documentation for the Clojure programming language.

Kvalifikācijas darbs **Interpretators programmēšanas valodai Immutant** izstrādāts Latvijas Universitātes Eksakto zinātņu un tehnoloģiju fakultātē, Datorikas nodaļā.

Ar savu parakstu apliecinu, ka darbs izstrādāts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai un/vai recenzentam uzrādītajai darba versijai.

Autors: ***Rolands Frīdemanis*** \_\_\_\_\_ **.01.2026.**

Rekomendēju darbu aizstāvēšanai

Darba vadītājs: ***Dr. dat. Aleksandrs Belovs*** \_\_\_\_\_ **.01.2026.**

Recenzents: ***Dr. dat. Zane Bičevska***

Darbs iesniegts **05.01.2026.**

Kvalifikācijas darbu pārbaudījumu komisijas sekretārs (elektronisks paraksts)