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

AST	Abstraktās sintakses koks
immutability	Pilnīga datu nemainība (koncepts)
immutable	Nemainīgs (datu īpašība)
read-only	Tikai lasāmi (dati)

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 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 dažāiem 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, izteikumi, tīras un netīras funckijas, 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..

1.1. Datu tipi

Immutant valodā tiek definēti tikai primitīvi datu tipi, kurus iedala 3 kategorijās: patiesuma, virknes tipa, un skaitļi. Par katru no šiem zemāk seko savs skaidrojums.

1.1.1. Virknes

Definīcija

Vikrne ir jebrkuš simbolu kopums, kas ir ietverts starp dubultpēdiņām.

Virknes var saturēt jebkādus simbolus, tostarp specsimbolus, kā `@`, `^`, `&` u.c. Proti, vikrnēs arī nav liegts izmantot speciālos atslēgas vārdus, ko pati valoda ir definējusi, piemēram, `mutant` (vairāk par to minēts sadaļā 1.2.3.). Piemērs dažādām viknēm seko zemāk. Tāpat, arī vienu rakstzīmi ir atļauts uzdot ar virknes tipu.

```
"", "someString", "sentence with spaces", " ", ,"\""
```

Izņēmuma gadījums, ir dubultpēdiņu simbola izmantošana iekš vikrnes. Lai nezutvertu 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.
```

1.1.2. Skaitļi

Definīcija

Skaitlis ir jebkura vērtība no Reālo skaitļu kopas. Ir svarīgi uzsvērt, ka irracionāli skaitļi tiek implementēti pēc IEEE 754 standarta, tāpēc realitātē tie nav līdz galam irracionāli, 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 π vērtību (`~3.1415926535`).

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

1.1.3. Patiesuma vērtības

Definīcija

Patiesuma vērtības ir tādas vērtības, kuras ir vai nu patiesas vai nu aplamas.

No augstāk redzamās definīcijas iziriet, ka valodai Immutant tiek definēta divvērtīga loģika. Tātad, patiesu vērtību apzīmē ar `true`, bet aplamu ar `false`.

1.2. Konstrukcijas

1.2.1. Izteikumi

Izteikumi mēdz būt dažādi - tie var gan neproducēt neko, gan arī producēt kādu vērtību. Piemēram, mainīgo inicializācijas izteikums pēc būtības ir izpidāma komanda, toties tā rezultātā pati par sevi nedod nekādu vērtību. No otras puses jebkurš salīdzināšanas operators, piemēram īr vienāds"jeb == (detalizētāk par to skat. 1.2.2.), producē patiesuma vērtību, kas izsaka salīdzināšanas rezultātu,

1.2.2. Operatori

Operatorus iedala unāros un bināros. Unārs operators tiek pielietots tikai vienam operandam, kur tai pat laikā bināri operatori ir pielietoti 2 operandiem.

Saraksts ar unāriem un bināriem operatoriem seko zemāk.

Unāri operatori:

- `!` negācija, agriež operanda pretējo patiesuma vērtību.

Bināri operatori:

- = piešķiršana, piešķir noteiktam identifikatoram, piemēram, mainīgajam, vērtību
- + saskaitīšana, atgriež divu skait

Simbols	Vārdisks nosaukums	Efekts	Atgrieztā vērtība	Operanda tips
`asd`	Negācija	-	Prētējā operanda vērtība. `true` pārtop par false un otrādi.	Būla tips

1.1. *Tabula* Unāru operatoru skaidrojumi

1.2.3. Mainīgie

Valoda nodala 2 veidu manīgos: tie kas nepaļaujas datu mutācijai jeb mainībai un tie, kas tai paļaujas.

```
mutant a = 3;
immutant b = 3;
```

1.3. 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.