

P175B124

Programavimo kalbų teorija

vacius.jusas@ktu.lt

***Don't just learn the language,
understand its culture.***

Anders Noras

Modulio medžiaga

- Viso semestro pateiktys lietuvių ir anglų kalbomis
- LD užduotys, ataskaitų įkėlimas

Auditoriniai užsiėmimai

- Paskaitos (2 val. per savaitę)
- Laboratoriniai darbai (kas antra savaitė):
 - 4 LD (3, 7, 11, 15 savaites) ir grupinis projektas (15 savaitę)

Atsiskaitymai

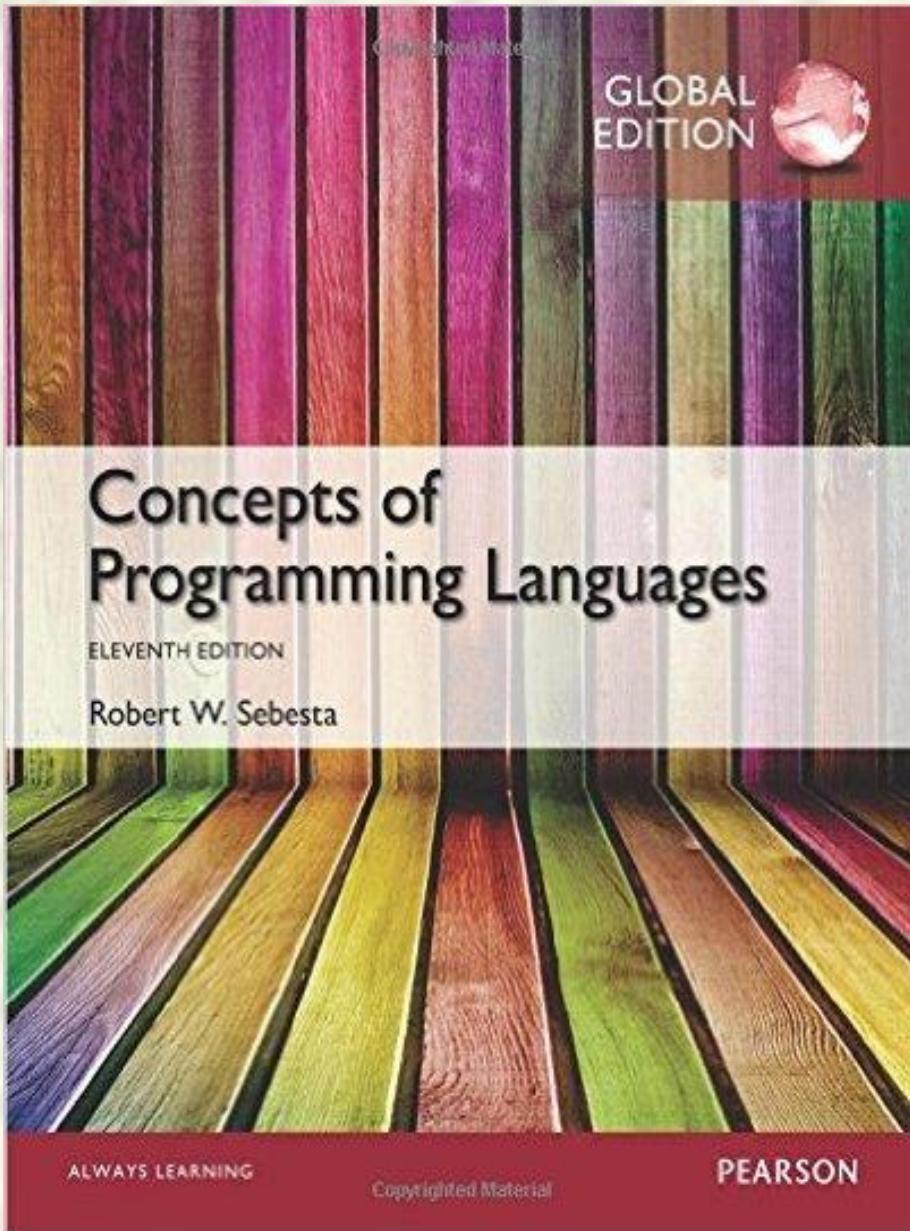
- 4 laboratoriniai darbai – po (7%, 8%, 8%, 7%) – 30%
- Probleminų sprendimų užduotis (dvi po 5%) – 10%
- Koliokviumas (9 savaitė) – 10%
 - Antras kartas (11 savaitė)
- Grupinis projektas – 20%
- Egzaminas – 30%

Projektas

- Jei darote projektą, kuris negeneruoja vykdomos programos (pvz.: Ohm.js), rašote tik funkcinėmis kalbomis tokiomis kaip F# ar kitomis.
- Jei netinka funkcinės kalbos, rašote projektą (kuo nori, t.y. C++), kuris generuotų vykdomą programą, ir būtų naudojami GCC arba LLVM kompiliatoriai.
- Visi kiti atvejai – maksimalus vertinimas 8

Literatūra

- R. W. Sebesta. Concepts of Programming Languages. 11th e., Science Research Publications; Global Edition edition, 2016, ISBN-13: 978-1292100555
- M.L. Scott. Programming Languages Pragmatics. 4th e. Morgan Kaufmann, 2015, ISBN-13: 978-0124104099
- P.Sestoft. Programming Language Concepts. 2nd e. Springer, 2017, ISBN 978-3-319-60788-7, Šifras:
D215822



Undergraduate Topics in Computer Science

Peter Sestoft

Programming Language Concepts



 Springer

Laboratoriniai darbai

- 6 programavimo kalbos
 - Python arba Ruby
 - Scala
 - Haskell arba F#
 - Prolog

Paskaitos 1

1. Įvadas
2. Istorija
3. Sintaksės ir semantikos aprašymas
4. Leksinė ir sintaksinė analizė
5. Vardai
6. Tipai
7. Išraiškos
8. Valdymo struktūros

Paskaitos 2

9. Paprogramės

10. Paprogramių įgyvendinimas

11. Abstraktusis duomenų tipas

12. Objektinis programavimas

13. Lygiagretumas

14. Išimtys

15. Funkcinis programavimas

16. Loginis programavimas

Realios paskaitos 1

1. Įvadas
2. Istorija
3. Sintaksės ir semantikos aprašymas
4. Leksinė ir sintaksinė analizė
- 5. Funkcinis programavimas**
6. Vardai
7. Tipai
8. Išraiškos

Realios paskaitos 2

9. Koliokviumas (20 min.), valdymo struktūros
10. Loginis programavimas
11. ASP. Paprogramės
12. Paprogramų įgyvendinimas
13. Abstraktusis duomenų tipas
14. Objektinis programavimas
15. Lygiagretumas
16. Išimtys

Pradedam teoriją

Temos klausimai

- Programavimo kalbu teorijos mokymo priežastys
- Programavimo sritys
- Kalbu įvertinimo kriterijai
- Įtaka kalbu projektavimui
- Kalbu kategorijos
- Kalbu projektų kompromisai
- Įgyvendinimo metodai
- Programavimo aplinkos

Priežastys

- Geresnė galimybė išreišksti idėjas
 - Kalbos konstrukcijos riboja mintis (natūrali, programavimo)
 - Didesnių galimybių žinojimas plečia minčių ribas
 - Galimybės modeliuojamos
- Pagerėjės pagrindas kalbų pasirinkimui
- Pagerėjusi galimybė mokytis naujų kalbų
 - Išmokti bendrąsias koncepcijas
 - Daugiau moki, lengviau įsisavini naujas kalbas
 - Kalbų pasaulis dinamiškas

Priežastys 2

- Geresnis įgyvendinimo reikšmės supratimas
 - Kalbu suprojektavimo būdą
 - klaidų taisymą
 - paprogramių kvietimą – efektyvių konstrukcijų parinkimą
- Geresnis jau žinomų kalbų naudojimas
- Geresnis bendras žinių pagrindas
 - Labiausiai paplitusi kalba nėra pati geriausia
 - Algol 60 vietoj Fortran

Tūkstančiai kalbų

- Kodėl tiek daug?:
 - Evoliucija
 - Specialūs tikslai
 - Asmeniniai nusistatymai
- Kas padaro kalbą sėkminga?:
 - Išreiškiamoji galia
 - Visos yra Tiuringo mašinos
 - Abstrakcijos
 - Panaudojimo lengvumas naujokui
 - Basic, Pascal, Java
 - Įgyvendinimo lengvumas
 - Basic, Pascal, Java

Tūkstančiai kalbų 2

- Standartizacija
 - Oficialus standartas
 - Standarde visos reikalingos priemonės
 - Pascal – atskiras kompiliavimas, eilutės, tiesioginė prieiga prie failų
- Atviras kodas
 - Kompiliatorius
 - Siejasi su kitomis atviromis priemonėmis (Unix, C, Linux)
- Puikūs kompiliatoriai
 - Fortran 90 (be rekursijos, rodyklių)
 - Common Lisp su papildomomis priemonėmis didelių projektų valdymui
- Ekonomika, parama ir inercija
 - PL/I – IBM
 - Cobol, Ada – DoD
 - C# – Microsoft
 - Swift - Apple

Pirmosios programavimo sritys

- Moksliniai tyrimai
 - Paprastos duomenų struktūros, operacijos su realaisiais skaičiais, masyvai, efektyvumas
 - Fortran (1956), varžovai – asembleris, Algol 60
- Verslo programos
 - Ataskaitų paruošimas, tikslus dešimtainių skaičių ir tekstinių duomenų naudojimas, dešimtainių skaičių operacijos
 - COBOL (1960)
- Dirbtinis intelektas
 - Dažniau simboliai, nei skaičiai, susietieji sąrašai
 - LISP (1959 ir iki 1990), Prolog, Scheme

Naujesnės programavimo sritys

- Sisteminis programavimas
 - Efektyvumo poreikis, nes pastoviai naudojam
 - Žemas lygmuo, nes bendrauja su įtaisais
 - Beveik be apribojimų – sisteminiai programuotojai
 - C Unix (1971)
- Interneto PJ
 - Eklektiškas kalbų rinkinys:
 - žymėjimo (XHTML),
 - scenarijų (JavaScript, PHP),
 - bendros paskirties (Java)
 - Dinaminis interneto turinys

Kalbos vertinimo kriterijai

- **Skaitomumas:** programų skaitymo ir supratimo lengvumas
- **Rašomumas:** programų rašymo lengvumas
- **Patikimumas:** atitikimas specifikacijai
- **Kaina:** galutinė bendra kaina

Skaitytomumas

Skaitomumo poslinkis

- Iki 1970 buvo efektyvumas ir skaitomumas kompiuteriui
- (1987 Booch) Pj gyvenimo ciklo koncepcija
- Kodavimui skiriamas mažesnis vaidmuo
- Priežiūra, bet ne kodavimas
- Nuo kompiuterio į žmogų
- Skaitomumas turi būti nagrinėjamas problemos kontekste

Bendrasis paprastumas

1. Aprépiama savybių ir konstrukcijų aibė:
 - Ignoruoti kai kurias savybes
 - Skirtingi poaibiai
2. Minimalus savybių daugiareikšmiškumas (daugiau nei vienas būdas operacijos įgyvendinimui, pvz.: +1 reikšmei C++)
3. Minimalus operatorių užklojimas (vienas simbolis, o prasmių daug, ypač, neleisti vartotojams)
4. Ne asemblerio kalbos paprastumas

Ortogonalumas

- Santykinai maža primityvių konstrukcijų aibė kombinuojama santykinai mažu būdų skaičiumi
- Ir kiekviena kombinacija teisēta
 - A reg1, atmintis
 - AR reg1, reg2 -- IBM
 - ADDL operand1, operand2 -- VAX
- Anglų kalbos išimtys
- a + b C kalboje
- C projektas – masyvai perduodami funkcijai per nuorodą
- Algol 68
- Mažai primityvų kombinacijų ir ribotas ortogonalumo naudojimas
- Funkcinės kalbos

Kitos skaitomumo savybės

- Duomenų tipai
 - Adekvatūs iš anksto apibrėžti duomenų tipai
- Sintaksės sutarimai:
 - Vardų formos: lanksčios kompozicijos (Fortran 77, Basic)
 - Specialūs žodžiai (static), sudėtiniai sakiniai. {} – užbaigia visas konstrukcijas, ar specialūs žodžiai gali būti kintamųjų vardais (Fortran 95), end if, end loop
 - Forma ir prasmė: save aprašančios konstrukcijos, prasmingi raktažodžiai (grep in Unix)

Rašomumas

- Matas kaip lengvai kalba gali būti naudojama pasirinktai probleminei sričiai (VB ir C). Lyginti tik probleminės srities rėmuose.
- Paprastumas ir ortogonalumas
 - Mažai konstrukcijų, mažai primityvų, maža taisyklių aibė jų apjungimui
- Abstrakcijos galimybės
 - Galimybė apibrėžti ir naudoti sudėtingas struktūras, ignoruojant jų realizavimo detales. Ypač svarbi abstrakcija šiandien
 - Proceso abstrakcija (paprogramės - rikiavimas) ir duomenų abstrakcija (dvejetainiai medžiai Fortran ir C++)
- Išraiškingumas
 - Patogių būdų aibė operacijų apibrėžimui (APL)
 - Operatorių ir iš anksto apibrėžtų funkcijų kiekis ir jėga (**kiek++**, **for**, **while**)

Patikimumas

- Programa patikima, jei dirba pagal specifikaciją prie bet kokių sąlygų
- Tipų tikrinimas
 - Tipų klaidų testavimas, ankstyvas aptikimas, vykdymo metu tipų klaidų tikrinimas brangus
- Išimčių valdymas
 - Perimti vykdymo klaidas ir imtis koregavimo priemonių
- Sinonimai ([alias](#))
 - Du ar daugiau būdų pasiekti tą pačią atminties įastelę
- Skaitomumas ir rašomumas
 - Jei nėra natūralių būdų, tenka naudoti nenatūralius būdus – mažėja patikimumas

Kalbu įvertinimo kriterijai

Charakteristika	Kriterijai		
	Skaitomumas	Rašomumas	Patikimumas
Paprastumas/Ortogonalumas	•	•	•
Valdymo struktūros	•	•	•
Duomenų tipai ir struktūros	•	•	•
Sintaksės projektas	•	•	•
Abstrakcija		•	•
Išraiškingumas		•	•
Tipų tikrinimas			•
Išimtys			•
Apriboti sinonimai			•

Kaina

1. Programuotojų apmokymas kalbos naudojimui
2. Programų rašymas – artumas taikymo sričiai. Dėl to kuriamos kalbos. Gera IDE.
3. Programų kompiliavimas
4. Programų vykdymas. Optimizavimas – pasirinkimas tarp kompiliavimo ir vykdymo.
5. Kalbos įgyvendinimo sistema: laisvi kompiliatoriai
6. Patikimumas: blogas patikimumas didina kainą
7. Programų priežiūra

Programų kūrimas, priežiūra, patikimumas – svarbiausi.

Kiti vertinimo kriterijai

- Mobilumas
 - Programų pernešimo lengvumas nuo vienos sistemos prie kitos
 - Standartizacija (C++ pradėjo 1989, baigė 1998)
- Bendrumas
 - Tinkamumas plačiam problemų ratui
- Apibrėžtumas
 - Oficialaus kalbos apibrėžimo pilnumas ir tikslumas

Kriterijų vertinimas

- Skaitomumas, rašomumas ir patikimumas nėra griežtai apibrėžti ir išmatuojami, bet svarbūs
- Kriterijai skirtingai vertinami iš skirtingų pozicijų:
 - Kalbos kūrėjams rūpi kalbos konstrukcijų įgyvendinimo sunkumas
 - Kalbos naudotojams – pirma rašomumas, po to skaitomumas
 - Kalbos projektuotojams – kalbos elegancija, kad kalba papilstų

Kiti faktoriai, veikiantys kalbu projektus

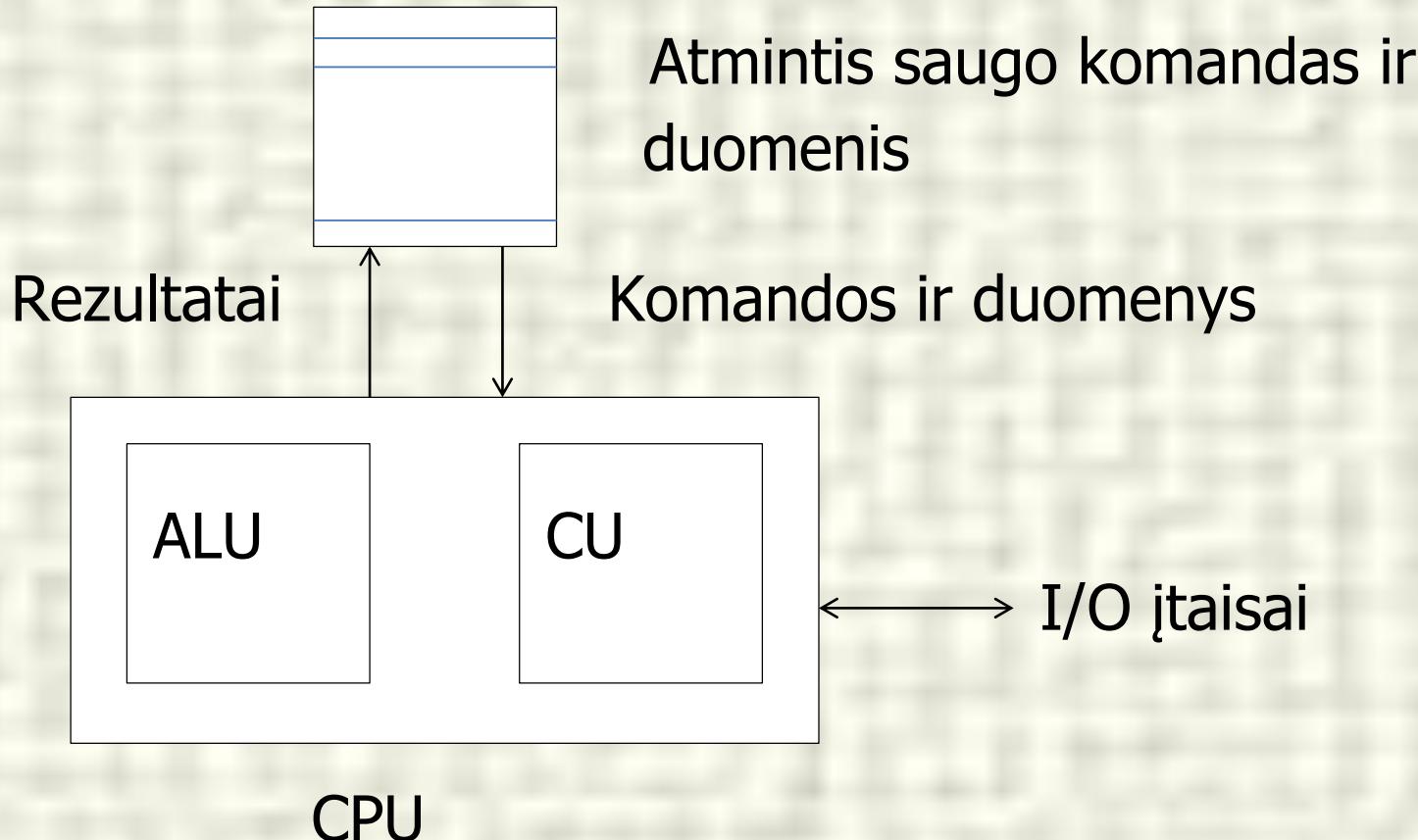
Kiti faktoriai

- Kompiuterio architektūra
 - Dauguma kalbų skirtos dominuojančiai von Neumann kompiuterio architektūrai
- Programavimo metodologijos
 - Naujos metodologijos (objektinis programavimas), naujos paradigmos ir naujos programavimo kalbos

Kompiuterio architektūros įtaka

- Vyraujanti architektūra: von Neumann
- Liepiamosios kalbos dominuoja dėl von Neumann kompiuterių
 - Duomenys ir programos saugomos atmintyje
 - Atmintis atskirta nuo CPU
 - Komandos ir duomenys yra siunčiami iš atminties į CPU
 - Liepiamujų kalbų bazė
 - Kintamieji modeliuoja atminties ląsteles
 - Priskyrimo sakiniai modeliuoja duomenų perkėlimą
 - Kartojimo – iteracijos efektyvumas, nes komandas šalia atminties ląstelėse.

Von Neumann architektūra



Vykdymo eiga

- Atnešti-vykdyti ciklas

Nustatyti programų skaitiklių

kartoti amžinai

atnešti komandą, kurią žymi skaitiklis

padidinti skaitiklio reikšmę

dekoduoti komandą

įvykdyti komandą

Kartojimo pabaiga

Programavimo metodologijos

- 1950 ir 1960 pradžia: paprastos programos – neefektyvi mašina
- 1960 pabaiga: žmonių efektyvumas tampa svarbus; skaitomumas, geresnės valdymo struktūros
 - Struktūrinis programavimas
 - Iš viršaus žemyn projektavimas, žingsninis detalizavimas
- 1970 pabaiga: procesus orientuotos į duomenis orientuotas
 - Duomenų abstrakcija
- 1980 vidurys: objektinis programavimas
 - Duomenų abstrakcija + paveldėjimas + polimorphizmas
- Lygiagretumas

Didžiausia kalbų kategorija

- Dažnai 4 kategorijos.
- Liepiamosios
 - Centrinės savybės yra kintamieji, priskyrimo sakiniai ir kartojimas
 - Patenka objektinės kalbos
 - Patenka vizualios kalbos (VB.NET) – atskirai neišskiriama, vadintos ketvirtos kartos kalbomis
 - Patenka scenarijų kalbos (Perl, JavaScript, Ruby), įgyvendinimo skirtumai – grynasis interpretatorius
 - Pavyzdžiai: C, Java, Perl, JavaScript, Visual BASIC .NET, C++

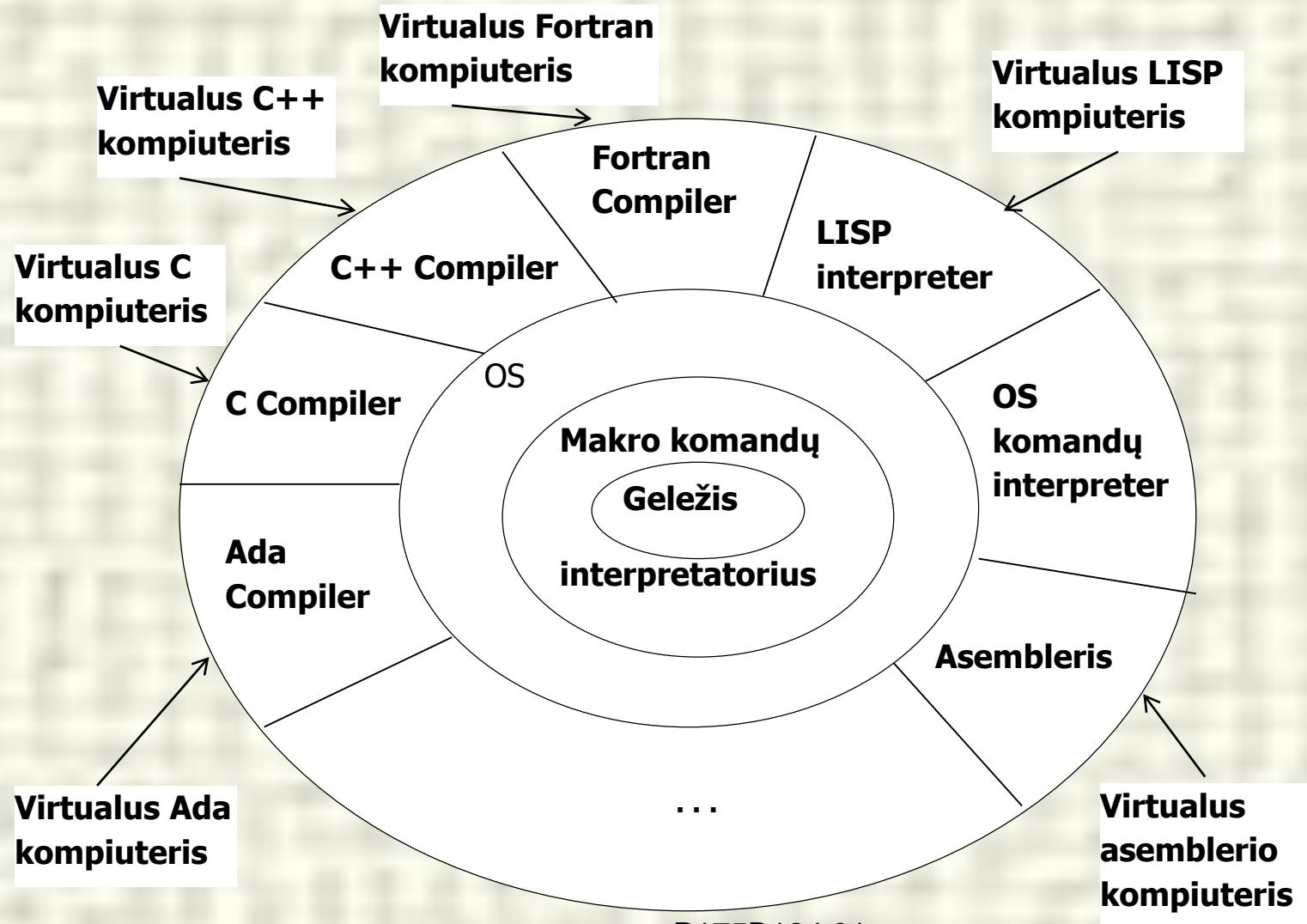
Kitos kalbų kategorijos

- Funkcinės
 - Pagrindinė skaičiavimų priemonė – funkcijos taikymas duotiems parametram
 - LISP, Scheme
- Loginės
 - Taisyklėmis bazuojamos (taisyklės nėra apibrėžiamos tam tikra tvarka)
 - Prolog
- Žymėjimo/programavimo hibridai
 - Išplėstos žymėjimo kalbos, palaikančios programavimą
 - JSTL, XSLT
- Specialios paskirties kalbos
 - S, R

Kalbų projektų kompromisai

- Patikimumas prieš vykdymo kainą
 - Java reikalauja, kad visi kreipiniai į masyvo elementus būtų tikrinami dėl neišėjimo už ribų. Akivaizdu, krinta vykdymo greitis
- Skaitomumas prieš rašomumą
 - APL pateikia daug galingų operatorių ir daug naujų simbolių, leidžiančių atlikti sudėtingus skaičiavimus, bet tai blogina skaitomumą
- Rašomumas (lankstumas) prieš patikimumą
 - C++ rodyklės yra galingas instrumentas ir labai lankstus, bet nepatikimas

Kompiuterio sluoksniai



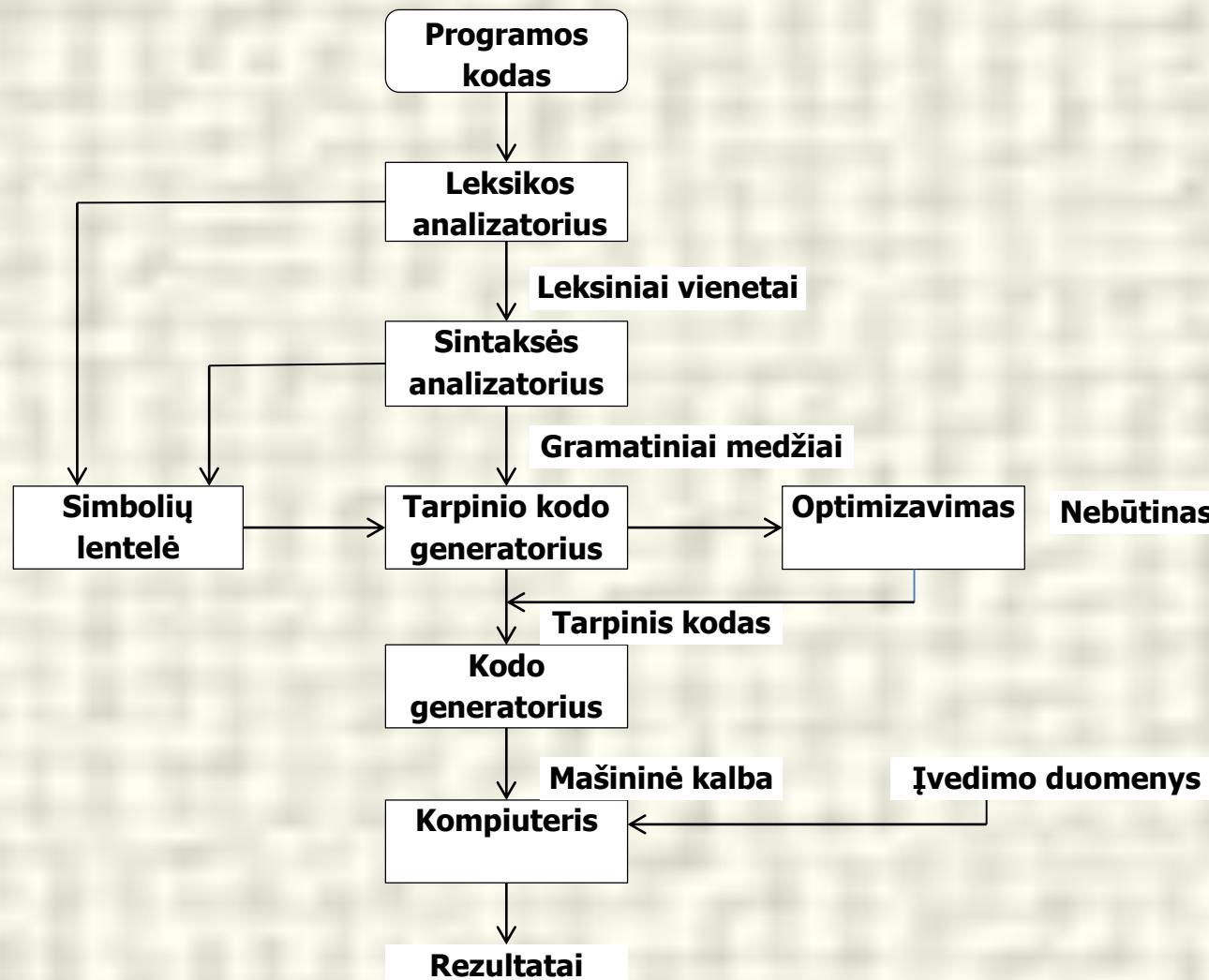
Įgyvendinimo metodai

- Kompiliavimas
 - Programos transliuojamos į mašininę kalbą
- Grynasis interpretatorius
 - Programas interpretuoja kita programa, vadinama interpretatoriumi
- Hibridinės sistemos
 - Kompromisas tarp kompiliatoriaus ir grynojo interpretatoriaus

Kompiliavimas

- Transliuoja aukšto lygmens kalbą į mašininę kalbą
- Lėtas transliavimas, greitas vykdymas
- Kompiliavimo procesą sudaro keli etapai:
 - Leksinė analizė: apjungia programos simbolius į leksinius vienetus
 - Sintaksinė analizė: transformuoja leksinius vienetus į gramatinį medį, kuris vaizduoja sintaksinę programos struktūrą
 - Semantinė analizė: generuoja tarpinį kodą ir tikrina papildomai klaidas
 - Kodo optimizavimas
 - Taikinio kodo generavimas: mašininis kodas
 - Simbolių lentelė – tai kompiliavimo DB: talpina – leksinė ir sintaksinė analizė, naudoja – semantinė analizė ir kodo generavimas

Kompiliavimo procesas



Papildomi kompliajavimo terminai

Nors kompliuota programa gali būti vykdoma kompiuteryje, jai reikia daugiau

- **Įkrovimo modulis** (vykdomasis paveikslas): vartotojo ir sistemos kodas kartu
- **Susiejimas ir įkrovimas:** sisteminių programinių vienetų surinkimo ir jų susiejimo su vartotojo programa procesas
- Ryšių redaktorius – prijungimas sisteminių resursų, bibliotekų, vartotojo modulių.

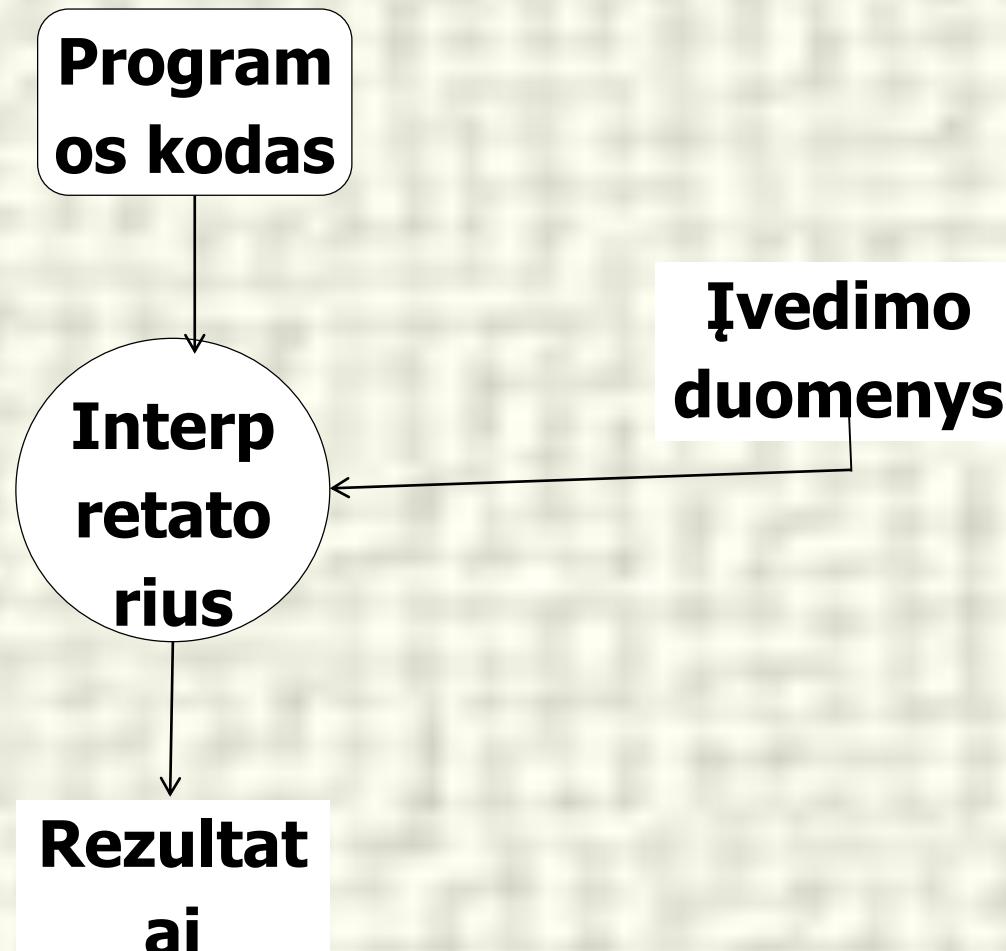
Siauroji von Neumann vieta

- Greitis tarp atminties ir jo procesoriaus apsprendžia kompiuterio greitį
- Programos komandos galėtų būti vykdomos dažnai greičiau, nei šis ryšio greitis – siauroji vieta, pirminis kompiuterių greičio ribojimo veiksnys
- Tai pirminis motyvas lygiagretiems kompiuteriams

Grynas interpretavimas

- Jokio transliavimo – tai akivaizdi virtuali mašina
- Lengvesnis programų įgyvendinimas (sprendimo laiko klaidos gali būti lengvai aptiktos ir parodytos)
- Lėtesnis vykdymas (nuo 10 iki 100 kartų lėčiau, nei kompiliuotos programos)
- Dažnai reikalauja daugiau erdvės
- Dabar retai naudojamas aukšto lygmens kalboms
- Naudojamas interneto scenarijų kalbose (JavaScript, PHP)
- Sakinių dekodavimas – siauroji vieta, nes kiekvieną sykį vykdymo metu reikia dekoduoti tą patį sakinį, taip pat reikia ir simbolių lentelės

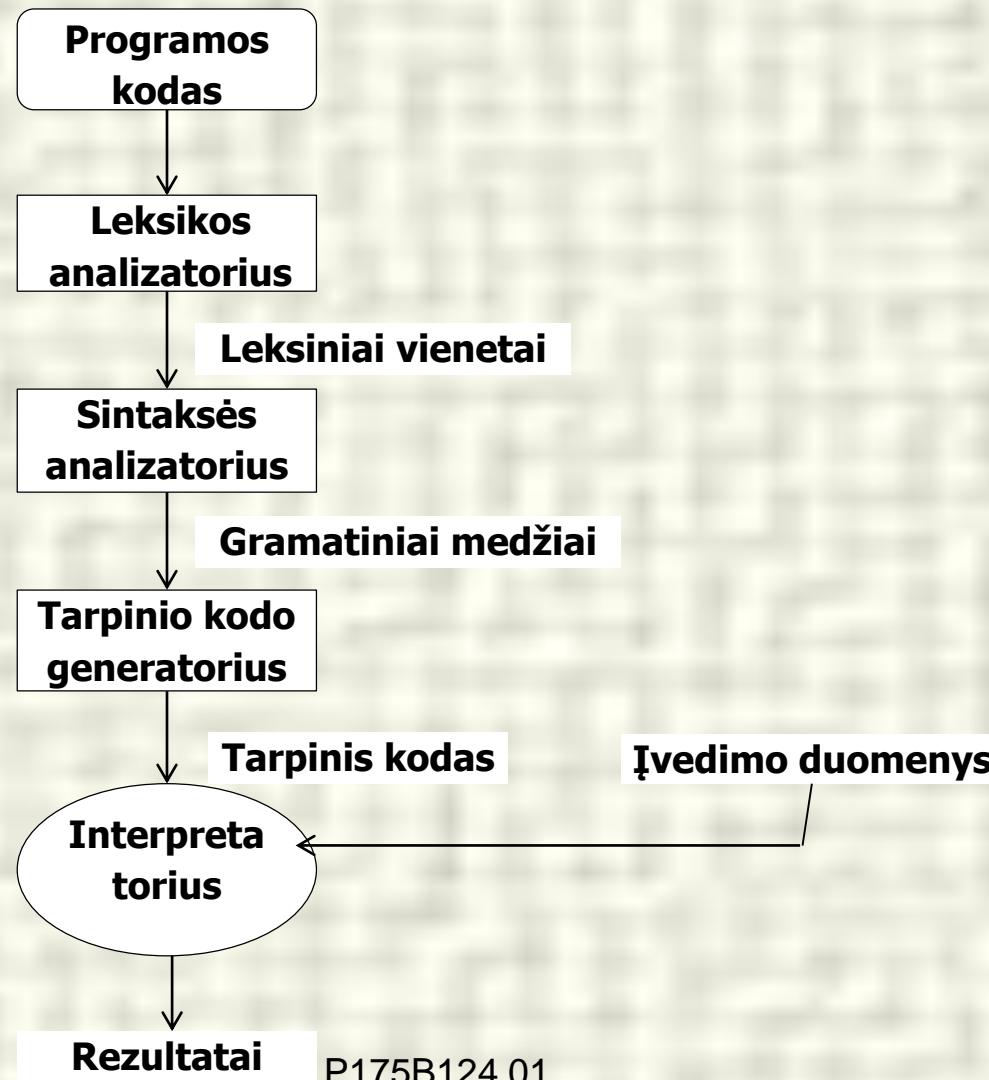
Grynojo interpretavimo procesas



Hibridinis įgyvendinimas

- Kompromisas tarp kompiliatorių ir grynuju interpretatorių
- Aukšto lygmens kalba, transliuojama į tarpinę kalbą, kuri interpretuojama
- Greitesnis nei grynasis interpretavimas, nes sakiniai dekoduojami tik kartą
- Pavyzdžiai
 - Perl programos yra dalinai kompiliuojamos, norint aptikti klaidas
 - Pradiniai Java įgyvendinimai, baitų kodas, Java virtuali mašina – interpretatorius

Hibridinis procesas



Just-in-Time įgyvendinimas

- Pradžioje programa transliuojama į tarpinį kodą
- Tuomet funkcijų, kai jos iškviečiamos, tarpinis kodas kompiliuojamas į mašininį kodą
- Mašininis kodas išsaugomas vėlesniems kvietimams
- JIT naudojamas Java programoms
- .NET kalbos įgyvendintos su JIT

Preprocesoriai

- Preprocesoriaus komandos naudojamos, norint ištraukti kodą iš kito failo
- Preprocesorius apdoroja programą prieš kompliaivimą ir ją išplečia
- Makro išplėtėjas
- C preprocesorius
 - `#include`, `#define`, ir kitos direktyvos

Kompiliatoriai per save

- C kompiliatorius C
- Ada kompiliatorius Ada
- **Bootstrapping** – pakelti save už raikštelių
- Pradedam nuo mažos versijos, po to ją plečiam
- Pascal kompiliatorius, parašytas Pascal, generuoja P kodą
- Pats kompiliatorius jau sutransliuotas į P kodą
- P kodo interpretatorius, parašytas Pascal
- P kodo interpretatorių rankiniu būdu transliuojame į vietinę kalbą.

Programavimo aplinkos

- Programinės įrangos kūrimo priemonių rinkinys – teksto redaktorius, kompiliatorius ir ryšių redaktorius
- UNIX
 - Senai žinoma OS ir priemonių rinkinys
 - Šiandien naudojama GVS (CDE, KDE, GNOME), kuri apima UNIX
- Microsoft Visual Studio.NET
 - Didelė sudėtinga vizuali aplinka
 - Naudojama interneto ir ne interneto programų sudarymui bet kurioje .NET kalboje
- NetBeans
 - Panašiai, kaip ir Visual Studio .NET, tačiau Java programoms, ir integruota kūrimo aplinka kūrimui (IDE) su [Java](#), [JavaScript](#), [PHP](#), [Groovy](#), [C](#), [C++](#), [Scala](#), [Clojure](#)

Santrauka

- Programavimo kalbų teorijos studijos suteikia tokį žinių:
 - Padidina mūsų galimybes naudoti įvairias konstrukcijas
 - Įgalina mus padaryti protingesnį kalbos pasirinkimą
 - Lengviau mokytis naujų kalbų
- Svarbiausi kalbų įvertinimo kriterijai: skaitomumas, rašomumas, patikimumas, kaina
- Didžiausią įtaką kalbų projektams turi kompiuterių architektūra ir programavimo metodologijos
- Pagrindiniai kalbų įgyvendinimo metodai: kompiliavimas, grynasis interpretavimas, hibridinis įgyvendinimas

P175B124

Programavimo kalbų istorija

***Know well more than two
programming languages.***

Russel Winder

Praėjusios temos klausimai

1. Kokie yra privalumai, kai kalbą įgyvendiname grynuoju interpretatoriumi?
2. Koks yra svarbiausias grynojo interpretatoriaus trūkumas?
3. Pateikite bent 2 kompromisų atvejus kalbų projektuose ir iliustruokite juos konkrečiomis kalbomis?
4. Kada kompiliatoriuje naudojama simbolių lentelė?
5. Ką reiškia, kad programa yra patikima?
6. Kas turėjo didžiausią įtaką programavimo kalbų projektams per prabėgusius 50 metų?
7. Kaip vadinama kalbų, kurių struktūra yra tiesiogiai įtakota von Neumann architektūros, kategorija?
8. Kokia programavimo kalbos konstrukcija suteikia proceso abstrakciją?
9. Kas yra sinonimai programavime?
10. Pateikite bent vieną ortogonalumo pažeidimo atvejį C kalbos projekte.

Temos klausimai - 1

- Zuse's Plankalkül
- Minimalus aparatūros programavimas: pseudo kodai
- IBM 704 ir Fortran
- Funkcinis programavimas: LISP
- Pirmas žingsnis link tobulėsnio programavimo: ALGOL 60
- Verslo įrašų kompiuterizavimas: COBOL
- Laiko pasidalijimo pradžia: BASIC

Temos klausimai - 2

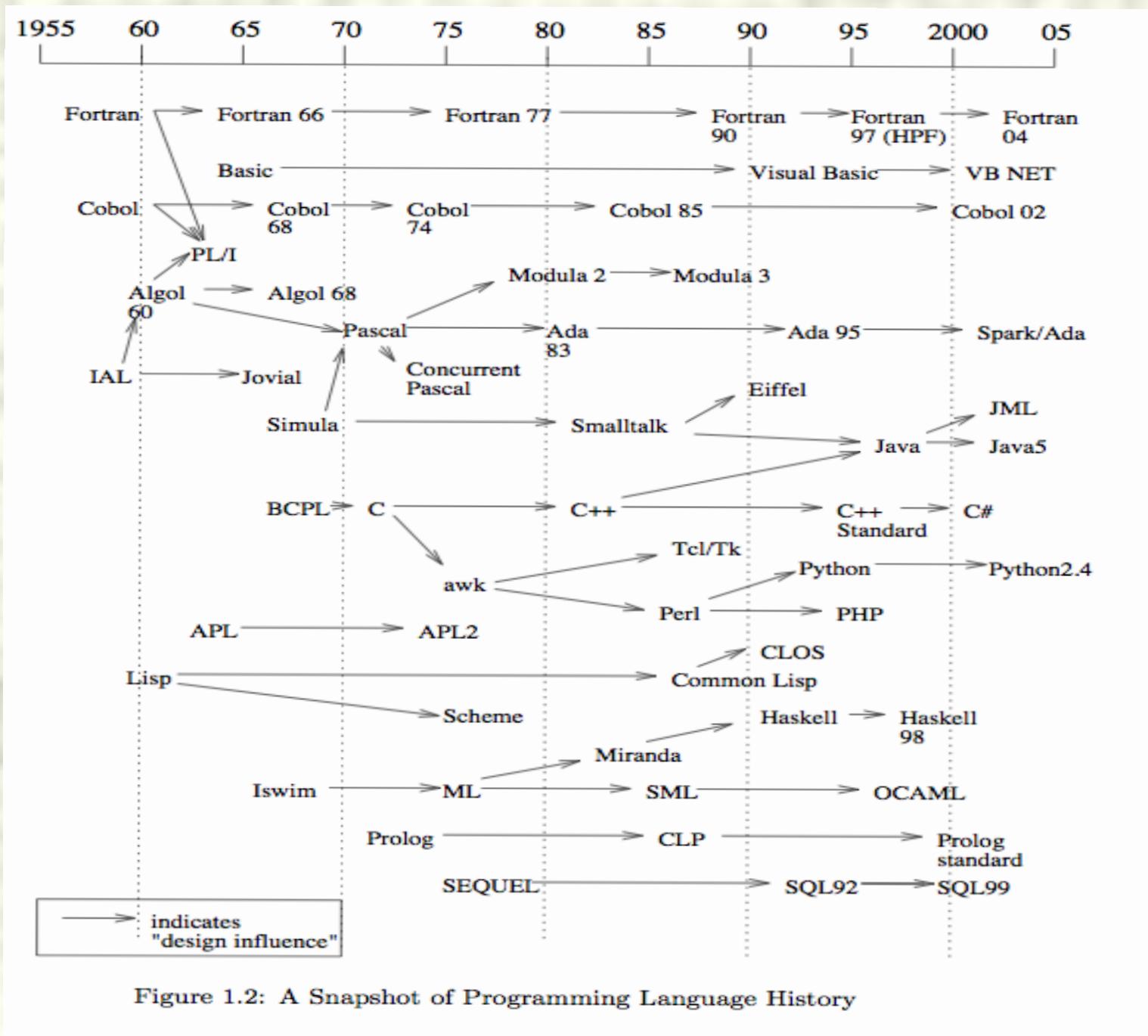
- Viskas kiekvienam: PL/I
- Dvi ankstyvosios dinaminės kalbos: APL ir SNOBOL
- Duomenų abstrakcijos pradžia: SIMULA 67
- Ortogonalus projektas: ALGOL 68
- Pirmieji ALGOL pasekėjai
- Loginis programavimas: Prolog
- Didžiausios projektavimo pastangos: Ada

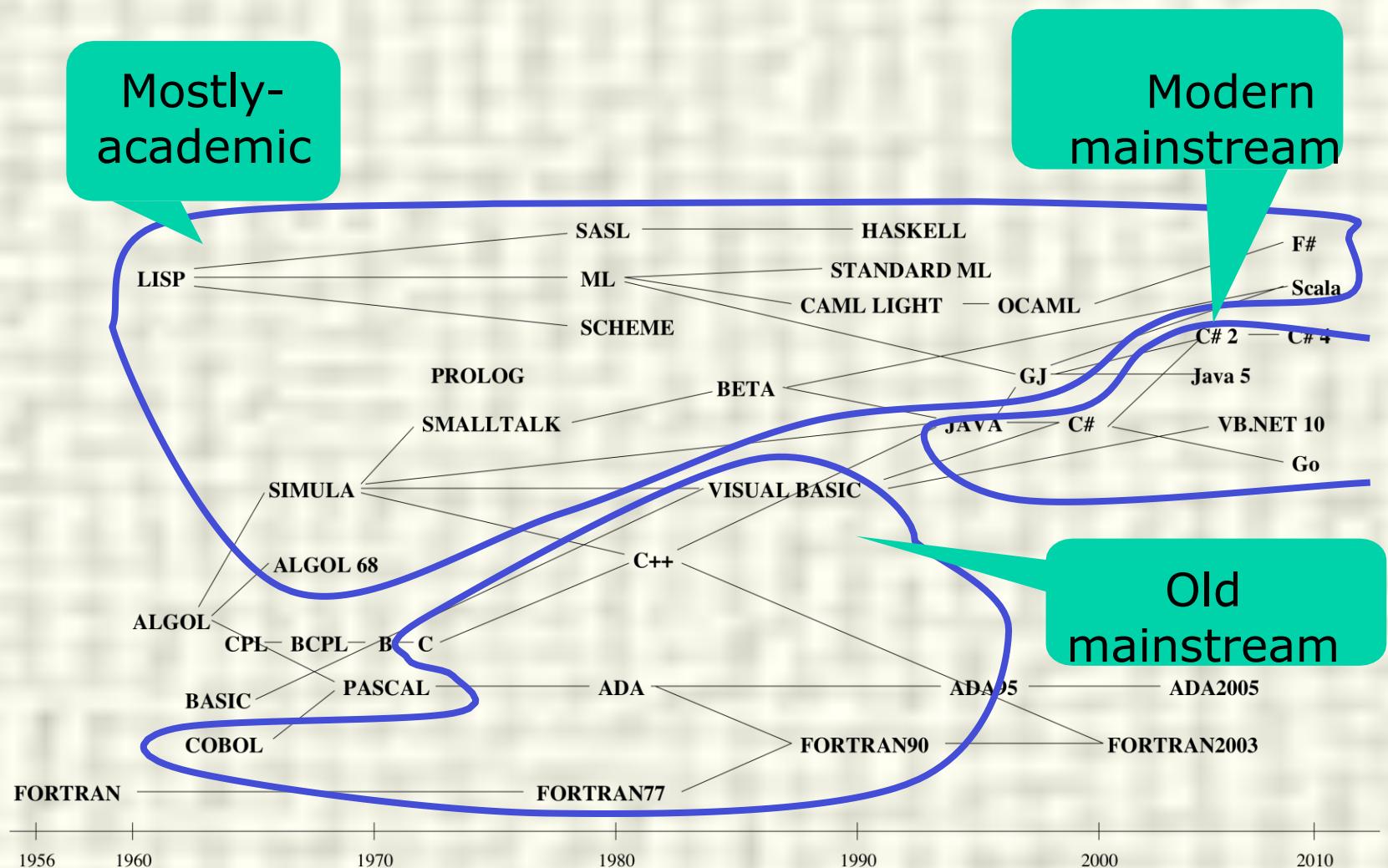
Temos klausimai - 3

- Objektinis programavimas: Smalltalk
- Liepiamojo (**imperative**) ir objektinio kombinacija: C++
- Liepamoji objektinė kalba: Java
- Scenarijų kalbos
- C++ bazuojama naujojo tūkstantmečio kalba: C#
- Žymėjimo (**markup**)/programavimo hibridinės kalbos

Kalbų kūrimo aplinka

- Kalbų vystymas ir naujų savybių atsiradimas
- Kompiuterių vartotojų bendruomenės:
 - Dirbtinio intelekto
 - Kompiuterių mokslo mokymo
 - Mokslo ir inžinerijos
 - Informacinių sistemų
 - Sistemų ir tinklų
 - Interneto





Peter Sestoft

Zuse's Plankalkül

- Suprojektuota 1945, bet ne publikuota iki 1972 (Konrad Zuse), sukurta izoliacijoje.
- Nuo 1936 iki 1944 Z4 kompiuteris, vėliau dirbo vienas
- Nebuvo įgyvendinta
- Paprasčiausias tipas – bitas
- Sudėtingi sakiniai (ciklas for, assert sakinys)
- Sudėtingos duomenų struktūros
 - Realieji skaičiai, masyvai, įrašai
- Rūšiavimas, kvadratinės šaknies traukimas, grafo susietumo nustatymas, šachmatų žaidimas

Plankalkül sintakse

- Užrašo forma – dvi ar trys eilutės vienam sakiniui
- Priskyrimo sakinys: A[4] + 1 priskirti A[5]

		A + 1	=>	A	
V		4		5	(indeksai)
S		1.n		1.n	(duomenų tipai, n bitų)

Pseudokodas

- Kompiuteriai nepatikimi, lèti, brangūs, nèra PI, net asemblerio (1940 pabaiga – 1950 pradžia)
- Trūkumai:
 - Blogas skaitomumas – tik skaitmenys, nèra tekstinių vardų
 - Sudètingas modifikavimas – absolutus adresavimas, išmetimas, įterpimas, NOP
 - Varginantis išraiškų kodavimas
 - Mašinos trūkumai – nèra indeksavimo ir realiuju skaičių

Trumpas kodas

- Trumpą kodą sukūrė Mauchly 1949 BINAC kompiuteriams, po to UNIVAC I
 - Išraiškos buvo koduojamos iš kairės į dešinę
 - Grynasis interpretatorius, žodis – 72 bitai
 - 50 kartų lėčiau, nei mašininis kodas
 - Operacijų kodai:
 - 01 – 06 abs value 1n (n+2)nd power
 - 02) 07 + 2n (n+2)nd root
 - 03 = 08 pause 4n if <= n
 - 04 / 09 (58 print and tab

Kodas: 00 X0 03 20 06 Y0

P175B124 02

13

Greitas kodas

- Greitą kodą sukūrė Backus 1954 IBM 701
 - Pseudo operacijos aritmetinėms ir matematinėms funkcijoms
 - Salyginis ir nesalyginis šakojimas
 - Adreso registru automatinis didinimas masyvo reikšmių išrinkimui
 - 4.2 milisekundės pridėjimo operacijai
 - Tik 700 žodžių vartotojo programai

Tobulesnės sistemos

- UNIVAC kompiliavimo sistema (1951 – 1953)
 - Grace Hopper vadovaujama komanda
 - Pseudo kodas išplėstas į mašininį kodą
- David J. Wheeler (Cambridge University)
 - Sukūrė keičiamų adresų blokų naudojimo metodą, norint išspręsti absolutaus adresavimo problemą
 - Asemblerio kalbos idėja (M. V. Wilkes, 1951)

IBM 704 ir Fortran

- Fortran 0: 1954 - neįgyvendintas
- Fortran I: 1957
 - Suprojektuotas IBM 704, kuris turėjo indekso registrus ir realiujų skaičių aparatinę įrangą
 - Paskatino programavimo kalbų kompiliavimo idėją, nes interpretavimo kaina (nebuvo realiujų skaičių programinės įrangos)
 - Kūrimo aplinka
 - Kompiuteriai turėjo mažai atminties ir nepatikimi
 - Programos moksliniams skaičiavimams
 - Nebuvo programavimo metodologijos ir priemonių
 - Mašinos efektyvumas buvo didžiausias rūpestis

Fortran projektavimo procesas

- Aplinkos įtaka Fortran I projektui
 - Nėra poreikio dinaminei atminčiai
 - Reikia gero masyvų valdymo ir skaičiavimo ciklų
 - Nėra eilučių, dešimtainės aritmetikos ir gero ivedimo išvedimo
- Iki IBM 704 paskelbimo, Fortran projektavimas prasidėjo – J. Backus

Fortran I apžvalga - 1

- Pirmoji įgyvendinta Fortran versija
 - Vardai iki 6 simbolių
 - Skaičiavimo ciklo sąlyga tikrinama pabaigoje
 - Formatuotas įvedimas išvedimas
 - Vartotojo apibrėžtos paprogramės
 - Trijų kelių sąlygos operatorius (aritmetinis **IF**)
 - Ciklo operatorius Do loop
 - Nėra duomenų tipų sakinių
 - I, J, K, L, M, N – sveikasis tipas

Fortran I apžvalga - 2

- Pirmoji įgyvendinta Fortran versija
 - Nėra atskiro kompiliavimo
 - Kompiliatorius išleistas 1957 balandžio mėn.
 - Programas, didesnes už 400 eilučių, retai sėkmingai kompliuodavo, nes kompas sutrikdavo
 - Kodas buvo labai greitas (18 tarnautojo metų buvo skirta optimizavimui)
 - Greitai plačiai paplito

Fortran II

- Pasirodė 1958
 - Nepriklausomas paprogramių kompiliavimas
 - Ištaisytos klaidos
- Programas, didesnes už 400 eilučių, retai sėkmingai kompiliuodavo, nes kompas sutrikdavo, o dabar galėdavo naudoti sukompiliuotą kodą

Fortran IV

- Sukurtas 1960-62
 - Išreikštas tipų paskelbimas
 - Loginis išrinkimo sakiny
 - Paprogramių vardai galėjo būti parametrai
 - ANSI 1966

Fortran 77

- Naujas standartas 1978
 - Eilučių apdorojimas
 - Loginis ciklo valdymas
 - **IF-THEN-ELSE** sakiny

Fortran 90

- Žymiausi pokyčiai nuo Fortran 77
 - Moduliai
 - Dinaminiai masyvai
 - Rodyklės
 - Rekursija
 - Varianto ([Case](#)) sakinys
 - Parametru tipų tikrinimas
 - Nuimtas reikalavimas atskiroms simbolių pozicijoms
 - Tapo Fortran

Vėliausios Fortran versijos

- Fortran 95 – nedaug papildymų, kai kas panaikinta
- Fortran 2003 – OP, bendravimas su C kalba

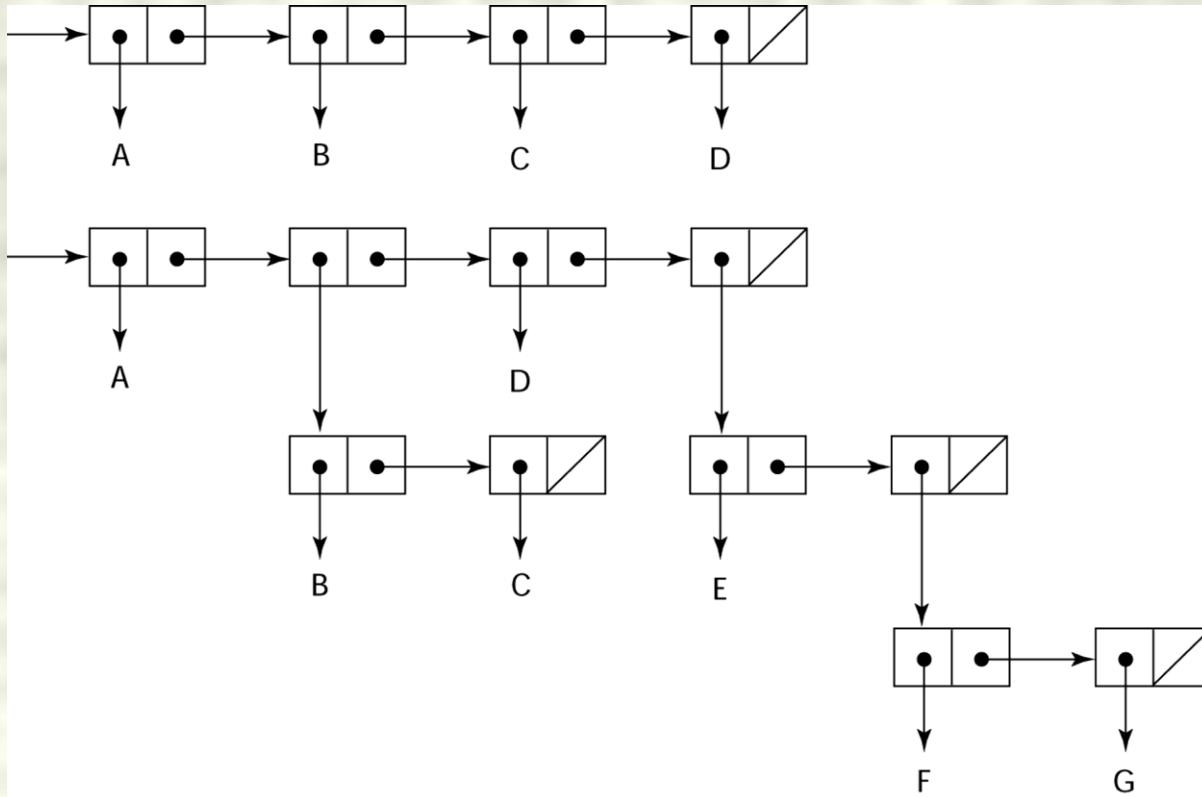
Fortran įvertinimas

- Gerai optimizuoti kompiatoriai (visos versijos iki 90)
 - Visų kintamųjų tipai ir atmintis fiksuoti iki vykdymo – nėra dinamiški
 - Nebuvo rekursijos
 - Pakeitė amžiam kompiuterių naudojimo būdą
- Vadinas kompiuterių pasaulio *lingua franca*

Funkcinis programavimas: LISP

- Sąrašų apdorojimo kalba
 - Suprojektavo MIT McCarthy (prasidėjo 1958)
- Dirbtinio intelekto tyrimams reikėjo kalbos:
 - Duomenų apdorojimui sąrašuose (ne masyvuose)
 - Simboliniam skaičiavimui (ne aritmetiniams)
- Tik 2 duomenų tipai: atomai ir sąrašai
- Sintaksė remiasi *lambda skaičiavimais*

Du LISP sarašai



- Du sarašai: (A B C D) ir (A (B C) D (E (F G)))

LISP įvertinimas

- Pradėjo funkcinį programavimą
 - Nereikia kintamuųjų ir priskyrimu
 - Valdymas per rekursiją ir sąlygines išraiškas
- Vis dar dominuojanti dirbtinio intelekto kalba
- COMMON LISP ir Scheme yra šiuolaikiniai LISP dialektai
- ML (1979) (ne vien funkcinė), Miranda (1986), Haskell (1992), F#, Erlang

Scheme

- Sukurta MIT 1970 metų viduryje
- Maža
- Intensyvus statinio akiračio naudojimas
- Funkcijos – pirmosios klasės esybės
- Paprasta sintaksė – ideali mokymo programoms

COMMON LISP

- Apjungti įvairių LISP dialektų savybes į vieną kalbą (1996)
- Didelė, sudėtinga
- Statinis ir dinaminis akiračiai

ALGOL 60

- Kūrimo aplinka
 - FORTRAN buvo skirtas IBM 70x
 - Buvo sukurta daugiau kalbų, visos atskiroms mašinoms
 - Nei vienos pernešamos kalbos, visos priklausomos nuo mašinų
 - Nei vienos universalios kalbos algoritmų žymėjimui
- ALGOL 60 buvo projektavimo pastangų sukurti universalią kalbą rezultatas

ALGOL 60 projekto pradžia

- ACM ir GAMM (Vokietijos matematikų bendrija) suorganizavo 4 dienų susitikimą 1958 metais Ciuriche)
- Kalbos tikslai
 - Artima matematiniam užrašui
 - Gera algoritmų aprašams
 - Transliuojama į mašininį kodą
- Kompromisai tarp asmenybių, abiejų Atlanto pusiu

ALGOL 58

- Formalizuota tipų koncepcija
- Vardai bet kokio ilgio
- Masyvai su daug indeksų
- Parametrai skirstomi pagal kryptį (in, out)
- Indeksai rašomi laužtiniuose skliaustuose
- Sudėtiniai sakiniai (**begin** . . . **end**)
- Kabliataškis sakinių atskyrimui
- Priskyrimo operatorius išraišk => kint virto kint := išraišk
- **if** turėjo **else-if** dalį
- Be I/O, nes būtų priklausomas nuo mašinos

ALGOL 58 įgyvendinimas

- Neketinta įgyvendinti, bet variacijų buvo (MAD, NELIAC, JOVIAL)
- IBM émési entuziastingai, bet 1959 m. meté

ALGOL 60 apžvalga

- Modifikuotas ALGOL 58 6 dienų susitikime Paryžiuje
- Naujos savybės
 - Blokinė struktūra – lokalus akiratis
 - Du parametru perdavimo metodai (pagal reikšmę, pagal vardą)
 - Paprogramių rekursija
 - Dėklo dinaminiai masyvai
 - Vis dar nėra I/O ir eilučių apdorojimo
 - Backus pristatė kalbos aprašymo formą, vėliau tapusią BNF. Prie jos daug dirbo danų mokslininkas P.Naur. Jis pateikė visą Algol aprašą BNF

ALGOL 60 įvertinimas

- Sėkmė
 - Algoritmų užrašymui naudotas daugiau kaip 20 metų
 - Visos vėlesnės liepiamosios kalbos juo rėmėsi
 - **Pirmaoji** suprojektuota tarptautinės grupės
 - **Pirmaoji** nepriklausoma nuo mašinos kalba
 - **Pirmaoji** kalba, kurios sintaksė buvo formaliai apibrėžta (BNF)
 - Inicijavo mokslo sritis: formaliasias kalbas, kompiliatorių teoriją, BNF paremtus kompiliatorius

ALGOL 60 įvertinimas - 2

- Nesėkmė
 - Niekada plačiai nenaudotas, ypač JAV
 - Priežastys
 - I/O stoka ir simbolių aibė pavertė programas nepernešamomis
 - Per daug lanksti – sunku įgyvendinti
 - Fortran įsitvirtinimas
 - Formalus sintaksės aprašas
 - IBM nepalaikė

COBOL istorinė aplinka

- Vis dar JAV populiarūs
- Maža įtaka kitoms kalboms, nes nebuvo kuriamos naujos kalbos, kadangi COBOL atitiko savo tikslą.
- Didelio verslo kalba, daugiausiai vystymo mažam versle, kuris perka „nuo lentynų".
- Kūrimo aplinka
 - UNIVAC buvo pradedantis naudoti FLOW-MATIC
 - USAF buvo pradedantis naudoti AIMACO
 - IBM kūrė COMTRAN

COBOL projektavimo procesas

- Pirmasis projekto susitikimas Pentagone – 1959 m.
- Projekto tikslai
 - Turi atrodyti kaip anglų kalba
 - Turi būti lengva naudoti, net jei prarasim galią
 - Privalo turėti platesnį vartotojų rataj
 - Projekto metu nekreipti dėmesio į įgyvendinimo problemas
- Projekto komiteto nariai visi buvo iš kompiuterių gamintojų ir DoD
- Projekto problemos: aritmetinės išraiškos, indeksai, kova tarp gamintojų

COBOL bazė

- Rēmēsi FLOW-MATIC
- FLOW-MATIC savybēs
 - Vardai iki 12 simbolių su pabraukimo brūkšneliu
 - Angliški vardai aritmetinėms operacijoms (nėra aritmetinių išraišķu)
 - Duomenys ir kodas buvo pilnai atskirti
 - Pirmasis kiekvieno sakinio žodis buvo veiksmažodis

COBOL įvertinimas

- Indėlis
 - Pirmą kartą makro galimybė aukšto lygmens kalboje
 - Hierarchinės duomenų struktūros – įrašai
 - Vienas į kitą įdėti išrinkimo sakiniai
 - Ilgi vardai (iki 30 simbolių) su brūkšneliais
 - Atskira duomenų sritis

COBOL – DoD įtaka

- Pirmoji kalba, kurios reikalavo DoD
 - Būtų nepasisekusi be DoD
- Vis dar plačiai naudojama verslo programų kalba

BASIC

- Suprojektavo Kemeny & Kurtz (1963)
- Projekto tikslai:
 - Lengva išmokti ir naudoti ne kompiuterių studentams
 - Privalo būti „maloni ir draugiška“
 - Lengva naudoti namuose
 - Laisva ir privati prieigos
 - Vartotojo laikas svarbiau, nei kompiuterio laikas
- Esamas populiarus dialektas: Visual Basic (VB.NET)
- Pirmoji plačiai naudojama kalba su laiko pasidalinimu

PL/I: istorinė aplinka

- 1963 m.
 - Mokslo skaičiavimams reikėjo daugiau I/O galimybių, nei turėjo COBOL;
 - Verslo vartotojams reikėjo realiųjų skaičių ir masyvų
 - Dvi skirtinges bendruomenės – per brangu
- Akivaizdus sprendimas
 - Sukurti naują kompiuterį abiems grupėms
 - Suprojektuoti naują kalbą abiems grupėms

PL/I

- Suprojektavo IBM ir SHARE
- Skaičiavimo situacija IBM požiūriu 1964 m.
 - Moksliniai skaičiavimai
 - IBM 1620 ir 7090 kompiuteriai
 - FORTRAN
 - SHARE vartotojų grupė
 - Verslo skaičiavimas
 - IBM 1401, 7080 kompiuteriai
 - COBOL
 - GUIDE vartotojų grupė

PL/I: projektavimo procesas

- Suprojektuota per 5 mėn. 3 X 3 komitetas
 - 3 nariai iš IBM, 3 nariai iš SHARE, susitikdavo kiekvieną savaitę 3 ar 4 dienas.
- Pradinė koncepcija
 - Fortran IV išplėtimas
- Iš pradžių vadinta NPL (New Programming Language)
- Pakeista į PL/I 1965 m.

PL/I įvertinimas

- PL/I indėlis
 - Apjungė, kas geriausia iš Fortran, COBOL, Algol 60
 - **Pirmasis** vienetų lygmens lygiagretumas
 - **Pirmasis** išimčių valdymas (23)
 - Pasirenkama rekursija
 - **Pirmasis** rodyklės duomenų tipas
 - **Pirmosios** bendros masyvų dalys (matricos eilutė kaip masyvas)
- Rūpesčiai
 - Daug naujų savybių buvo blogai suprojektuotos
 - Per didelė ir per sudėtinga

APL ir SNOBOL

- Dinaminiai tipai, dinaminis atminties skyrimas
- Kintamieji be tipų
 - Kintamasis gauna tipą pagal priskirtą reikšmę
- Atminties paskiriama, kai kintamasis gauna reikšmę

APL

- Suprojektuota kaip aparatūros aprašymo kalba IBM, Ken Iverson apie 1960 m.
 - Išraiškinga, daug operatorių skaliarams ir masyvams
 - Sunkiai skaitomos programos
- Vis dar naudojama; minimalūs pakeitimai

SNOBOL

- Suprojektuota kaip eilučių manipuliavimo kalba Bell Labs, Farber, Griswold ir Polonsky 1964 m.
- Galingi operatoriai eilučių paieškai
- Teksto redaktorių realizavimui
- Lėtesnė, nei alternatyvios kalbos, todėl jau plačiai nenaudojama, bet dar kai kur teksto apdorojimo uždaviniuose išlikusi

SIMULA 67

- Suprojektuota sistemų modeliavimui Norvegijoje
- Rėmėsi ALGOL 60 ir SIMULA I
- Svarbiausias indėlis
 - Paprogramių rūšis - koprogramė
 - Klasės, objektais, paveldimumas
- Koprogramė – paprogramių apibendrinimas su daugybe jėjimo taškų, stabdymu ir vėlesniu tėsimu.

ALGOL 68

- Iš ALGOL 60, bet ne šios kalbos super aibė
- Naujų idėjų šaltinis, nors pati kalba nebuvo plačiai paplitusi
- Projektas remiasi ortogonalumo koncepcija

ALGOL 68 įvertinimas

- Indėlis
 - Vartotojo apibrėžtos duomenų struktūros
 - Nuorodų tipai
 - Dinaminiai masyvai
- Pastabos
 - Ta pati problema, kaip ir su Algol 60 – kalbos aprašas pateiktas formalia nežinoma kalba
 - Mažiau naudojama, nei ALGOL 60
 - Stipri įtaka vėlesniems kalboms: Pascal, C ir Ada

Algol kalbos palikuonys

Pascal 1971

- Sukūrė Wirth, buvęs ALGOL 68 komiteto narys
- Struktūrinio programavimo mokymui
- Maža, paprasta, nieko naujo
 - Nebuvo atskiro modulių kompiliavimo
- Didžiausia įtaka programavimo mokymui
 - Nuo 1970 vidurio iki 2000 plačiausiai naudojama kalba programavimo mokymui
- Turbo Pascal

C 1972

- Sukurta sisteminiam programavimui (Bell Labs, Dennis Richie)
- Išvystė iš BCLP, B ir ALGOL 68
- Galinga operatorių aibė, bet bloga tipų kontrolė
- Pradžioje paplito per UNIX
- Daug taikymo sričių
- Yra mėgstantys, yra nemėgstantys (2017 m. populiariausia kalba)

Prolog

- Sukūrė Comerauer ir Roussel (University of Aix-Marseille), padėjo Kowalski (University of Edinburgh)
- Remiasi formaliaja logika
- Ne procedūrinė
- Lyg protinė duomenų bazė, naudojanti išvadų procesą, norint išvesti tiesą iš duotų užklausų
- Labai neefektyvi, siaura taikymo sritis

Ada

- Didžiulės projektavimo pastangos, daug žmonių, pinigų, 8 metai, DoD, įterptinės sistemos
 - Strawman reikalavimai (April 1975)
 - Woodman reikalavimai (August 1975)
 - Tinman reikalavimai (1976)
 - Ironman reikalavimai (1977)
 - Steelman reikalavimai (1978)
- Pavadinta Augusta Ada Byron garbei, pirmoji programuotoja

Ada įvertinimas

- Indėlis
 - Paketai – duomenų abstrakcija
 - Išimčių valdymas
 - Bendriniai programų vienetai
 - Lygiagretumas
- Pastabos
 - Geras projektas
 - Įtraukė viską, kas buvo žinoma programų inžinerijoje ir kalbu projektuose
 - Pirmieji kompiliatoriai buvo sudėtingi; pirmasis tinkamas kompiliatorius pasirodė po 5 metų nuo projekto užbaigimo
 - Nepaplito dėl kompiliatorių stokos – buvo tik mokami

Ada 95

- Ada 95 (prasidejo 1988)
 - Palaiko OP per tipu išvedimą – paveldėjimas
 - Geresni bendruju duomenų valdymo mechanizmai – dinaminis susietumas
 - Naujos lygiagretumo savybės
 - Daugiau įvairių bibliotekų
- Populiarumas nukentėjo, nes DoD daugiau nereikalauja jos privalomumo ir dėl C++

Smalltalk–80

- Sukurtas Xerox PARC, iš pradžių Alan Kay, vėliau Adele Goldberg
- Pirmasis pilnas objektinės kalbos įgyvendinimas (duomenų abstrakcija, paveldėjimas ir dinaminis susiejimas)
- Pradėjo grafinę vartotojo sąsają
- Paskatino OP

C++

- Sukurta at Bell Labs, B. Stroustrup, 1980 m.
- Išsivystė iš C ir SIMULA 67 (objektinė dalis)
- Išimčių valdymas
- Didelė ir sudėtinga kalba, nes palaiko procedūrinį ir objektinį programavimą
- Išpopuliarėjo kartu su OP
- Pirmasis ANSI standartas 1997, naujas standartas 2011

Susijusios OP kalbos

- Eiffel (Bertrand Meyer - 1992)
 - Nėra tiesioginio ryšio į kitas kalbas
 - Mažesnė ir paprastesnė, nei C++, bet pakankamai galinga
 - Nebuvo populiarūs, nes C++ gerbėjai jau buvo C programuotojai
- Delphi (Borland)
 - Pascal su OP savybėmis
 - Elegantūs ir saugesnės, nei C++
- Anders Hejlsberg – Turbo Pascal, Delphi, C#

Java

- Sukurta Sun (nupirko Oracle 2011) 1990 m. pradžioje
 - C ir C++ netenkino įterptinės įrangos mažuose prietaisuose
- Remiasi C++
 - Žymiai supaprastinta (neturi **struct**, **union**, **enum**, rodyklių aritmetikos ir dalies C++ tipų konversijos)
 - Palaiko *tik* OP
 - Turi nuorodas, bet ne rodykles
 - Palaiko lygiagretumą

Java įvertinimas

- Pašalino daugumą nesaugiu C++ savybių
- Lygiagretumas
- Programėlių, GVS, DB prieigos bibliotekos
- Mobili: JVM konsepcija, JIT kompiliatoriai
- Plačiai naudojama interneto programavime
- Naudojimas augo greičiau, negu bet kurios kalbos iki jos
- Kuriamos naujos versijos

C#

- .NET platformos dalis (2000)
- Bazuojama C++ , Java ir Delphi
- Turi komponentus
- Visos .NET kalbos naudoja Common Type System (CTS), kuri yra bendroje bibliotekoje

Objective-C

- 1984, Brad Cox, Apple kompiuteriai
- Objektinė C kalbos versija, bazuojama Smalltalk, reikia C kalbos žinių
- Nėra operatorių užklojimo.
- Buvo naudota NextStep operacinės sistemos parašymui Next kompiuteryje.
- Naudojama iPhone, iPad programavimui (2008), 2012 paėmė didžiausią rinkos dalį
- Programų karkasas Cocoa, objektu biblioteka

Swift

- 2014 birželį Apple paskelbė. Kūrė slaptai 4 metus.
- Paradigmos – objektinė, funkcinė, liepiamoji.
- Funkcijos – pirmos klasės tipas, stipriai tipizuota, automatinis atminties valdymas
- Tikslai:
 - Lengviau kurti programas, naudojant šiuolaikinių kalbų savybes;
 - Kurti saugesnį kodą, išvengiant bendrujų programavimo kalbų klaidų
 - Kurti lengvai skaitomą kodą
 - Būti suderinamai su Objective-C karkasais (Cocoa, Cocoa Touch)

Scenarijų kalbos

Scenarijų kalbos: Perl

Larry Wall, 1987 m.

Kintamujų tipas statinis, tačiau skelbiami numanomai

Trys skirtinges vardų erdvės, kurias atskiria pirmasis kintamojo simbolis

Galinga, bet pavojinga

- Plačiai paplito dėl CGI programavimo interne
- Taip pat naudojama kaip UNIX administravimo kalba

JavaScript ir PHP

- JavaScript (1995 m.)
 - Prasidėjo Netscape, bet vėliau tapo bendru Netscape ir Sun Microsystems (nebéra, Oracle nupirko) kūriniu
 - Kliento pusės HTML įterptinė kalba dėl puslapių dinamiškumo
 - Gynasis interpretatorius
 - Siejasi su Java tik per panašią sintakę
- PHP (1994 m.)
 - PHP: Rasmus Lerdorf
 - Serverio pusės HTML įterpta scenarijų kalba, skirta formų apdorojimui ir bendravimui su DB
 - Gynasis interpretatorius
 - Palaiko daugumą DBVS
 - Kuriamos naujos versijos

Python ir Lua

- Python (1989 m.)
 - Objektinė interpretuojama scenarių kalba
 - Tipas tikrinamas, tačiau dinaminis
 - Naudojama CGI programavimui ir formų apdorojimui
 - Palaiko sąrašus, aibes, maišos lenteles
 - Vartotojas gali išplėsti
- Lua (1993 m.)
 - Objektinė interpretuojama scenarių kalba
 - Tipas tikrinamas, tačiau dinaminis
 - Naudojama CGI programavimui ir formų apdorojimui
 - Palaiko sąrašus, aibes, maišos lenteles, ir viskas vienintelėje duomenų struktūroje – lentelėje
 - Lengvai išplečiama

Ruby

- Autorius Yukihiro Matsumoto (a.k.a, “Matz”, 1993 m.)
- Prasidėjo kaip Perl ir Python pakeitimas
- Grynai objektinė scenarijų kalba
- Visi duomenys yra objektai
- Populiariausi operatoriai yra įgyvendinti kaip metodai, kuriuos vartotojas gali pakeisti
- Grynasis interpretatorius

Hibridinės kalbos

- XSLT
 - Išplečiama žymėjimo kalba (XML): meta žymėjimo kalba
 - eXtensible Stylesheet Language Transformation (XSTL) transformuoja XML dokumentus dėl parodymo
 - Programavimo konstrukcijos (ciklas)
- JSP
 - Java Server Pages: technologijų rinkinys dinaminių interneto puslapių palaikymui
 - servlet: Java programa, esanti interneto serveryje, aktyvuojama, kai kviečia HTML dokumentas; servleto išvedimas parodomos naršyklėje
 - JSTL turi programavimo konstrukcijas (HTML elementai)

Santrauka

- Svarbiausios programavimo kalbos
- Kalbų savybių įtaka būsimoms kalboms

P175B124

Sintaksė ir semantika

Don't rely on “magic happens here”.

Alan Griffiths

Istorijos paskaitos klausimai

1. Kokie pseudo kodo trūkumai?
2. Kokie faktoriai lėmė FORTRAN I kalbos paplitimą ir jos ilgalaikę sėkmę?
3. Kokie faktoriai lėmė COBOL kalbos ilgalaikę sėkmę?
4. Kas labiausiai sukliudė Algol60 kalbos platesniam paplitimui?
5. Kokiomis savybėmis pasižymi dinaminės kalbos?
6. Kodėl BASIC kalba buvo reikšminga 1980 – 1985 metais?
7. Kokį C varianto sakinio trūkumą pašalino C#?
8. Kokiam tikslui buvo projektuojama Ada pirmiausiai?
9. Išvardinkite 5 scenarijų kalbas?
10. Kokia Ruby aritmetinių operatorių charakteristika skiria juos nuo kitų kalbų aritmetinių operatorių?

Temos klausimai

- Sakinys, leksema (**lexem**), žetonas (**token**)
- Bekontekstės (**context-free**) gramatikos – BNF, EBNF
- Atributų gramatika
- Veiklos (**operational**) semantika
- Žymėjimo (**denotational**) semantika
- Aksiomų (**axiomatic**) semantika

Įvadas

- Programavimo kalbos aprašo svarba (Algol 60, Algol 68)
- Kalbos aprašo vartotojai
 - Pradiniai vertintojai
 - Įgyvendintojai
 - Vartotojai-programuotojai
 - Kitų kalbų projektuotojai
- **Sintaksė:** išraiškų, sakinių ir programų vienetų forma ar struktūra
- **Semantika:** išraiškų, sakinių ir programų vienetų prasmė, pvz.: ciklo sakiny: *while (loginė išraiška) sakiny*
- Sintaksė ir semantika yra artimai susietos. Sintakse aprašyti yra paprasčiau

Sintaksės terminologija

- *Kalba* – sakinių aibė
- *Sakinys* – alfabeto simbolių eilučių kombinacija
- *Leksema* – žemiausiojo lygmens sintaksinis vienetas (*, sum, begin)
- *Žetonas* – leksemų kategorija, trumpiausias sintaksinis vienetas, turintis prasmę (vardai, konstantos, raktažodžiai)
- Programa – leksemų rinkinys

Leksemos ir žetonai

*Reikšmė = 2 * skaitiklis + 17;*

Leksema	Žetonas
<i>Reikšmė</i>	Kintamojo vardas
=	Priskyrimo ženklas
2	Sveikoji konstanta
*	Daugybos operacija
<i>skaitiklis</i>	Kintamojo vardas
+	Sumos operacija
17	Sveikoji konstanta
;	Kabliataškis

Sintaksės lygmenys

Trys lygmenys:

- Leksinė sintaksė = visi baziniai kalbos simboliai (vardai, reikšmės, operatoriai, ...)
- Konkreti sintaksė = išraiškų, sakinių ir programų rašymo taisyklės
- Abstrakti sintaksė = vidinis programas atvaizdavimas, kai svarbiau turinys, o ne forma.

Formalus kalbos apibrėžimas - 1

- Atpažinimas (**recognition**)
 - Atpažinimo įtaisas skaito įvedimo eilutes, sudarytas iš kalbos alfabeto, ir nusprendžia, ar įvedimo eilutė priklauso kalbai
 - Veikia kaip filtras
 - Kalbos yra begalinės
 - Kompiliatoriaus sintaksės analizės dalis – gramatinis analizatorius (**parser**)
 - Atpažiniklis (**recognizer**) néra naudingas kaip kalbos aprašymas

Formalus kalbos apibrėžimas - 2

- **Generavimas**

- Paspausti mygtuką – įtaisas, generuojantis kalbos sakinius
- Galima nustatyti, ar sakinio sintaksė yra gera, palyginant ją su generatoriaus struktūra
- Artimas ryšys tarp kalbos generavimo ir atpažinimo – labai svarbus (**seminal**) atradimas

Sintaksės aprašymas

Sąvokos

- *Gramatika* – formalus kalbos apibrėžimo mechanizmas
- *Metakalba* – kalba, naudojama apibrėžti kitas kalbas.
- *Gramatika* išreiškiama metakalba.
- *Gramatikos tikslas*: apibrėžti programavimo kalbos sintakse.

Gramatikų lygmenys

Lingvistas Noam Chomsky, 1957

Gramatikos – kalbu generatoriai

Keturi gramatikos lygmenys:

- Reguliari – žetonų sintaksės aprašymui
- Bekontekstė – visos kalbos sintaksės aprašymui
- Kontekstinė
- Neapribota

Bet kokia eilučių aibė, kuri gali būti apibrėžta, jei pridedam rekursiją, vadinama bekontekste kalba. Bekontekstes kalbas generuoja bekontekstės gramatikos.

BNF ir bekontekstės gramatikos

- Bekontekstės gramatikos
 - Kalbu generatoriai, skirti aprašyti natūralių kalbu sintakse
- Backus-Naur Form (1959)
 - Sukūrė John Backus aprašyti Algol 58 sintakse
 - BNF – stilizuota bekontekstės gramatikos versija
 - Pirmą kartą panaudota, aprašant Algol 60 sintakse

BNF gramatika

- *Taisyklių aibė*: P
- *Terminaliniai simboliai*: T
- *Neterminaliniai simboliai*: N
- *Pradžios simbolis*: $S \in N$

• Taisyklės forma

$A \rightarrow \omega$ (LHS (**left hand side**) ir RHS)

- Kur $A \in N$ ir $\omega \in (N \cup T)^*$

BNF terminai

- Abstrakcijos naudojamos žymėti sintaksinių struktūrų klasėms – sintaksiniai kintamieji vadinami neterminaliniais simboliais
- *Terminaliniai* yra leksemos arba žetonai
- Taisyklė turi kairiąją pusę (LHS), kuri yra netermininas, ir dešiniąją pusę (RHS), kuri yra terminalų ir/arba neterminalu eilutė
- Neterminai dažnai rašomi kampiniuose skliaustuose
- Gramatika: baigtinė netuščia taisyklių aibė
- *Pradžios simbolis* – specialus gramatikos netermininas

Pavyzdžiai

- Gramatika:

`<binaryDigit> → 0`

`<binaryDigit> → 1`

- ekvivalentu:

`<binaryDigit> → 0 | 1`

`<if_stmt> → if (<logic_expr>) <stmt>`
`| if (<logic_expr>) <stmt>`
`else <stmt>`

Rekursija

- Kintamo ilgio sintaksiniai elementai aprašomi, naudojant rekursiją

$$\begin{aligned} <\text{ident_list}> \rightarrow & \quad <\text{ident}> \\ & | \quad <\text{ident}>, \quad <\text{ident_list}> \end{aligned}$$

Gramatikos pavyzdys

$\langle \text{program} \rangle \rightarrow \text{begin } \langle \text{stmt_list} \rangle \text{ end}$

$\langle \text{stmt_list} \rangle \rightarrow \langle \text{stmt} \rangle$

$| \langle \text{stmt} \rangle ; \langle \text{stmt_list} \rangle$

$\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$

$\langle \text{var} \rangle \rightarrow A | B | C$

$\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle + \langle \text{var} \rangle$

$| \langle \text{var} \rangle - \langle \text{var} \rangle$

$| \langle \text{var} \rangle$

Išvedimas

- Gramatika – tai generavimo įtaisas kalbos apibrėžimui.
- Kalbos sakiniai generuojami, naudojant išvedimą
- Išvedimas ([derivation](#)) – tai kartotinis taisyklių taikymas, pradedant starto simboliu ir baigiant sakiniu, kurj sudaro tik terminalai

Gramatikos pavyzdys

- Gramatika:

$$\begin{aligned} <\text{Integer}> &\rightarrow <\text{Digit}> \mid <\text{Integer}> <\text{Digit}> \\ <\text{Digit}> &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

- Galime išvesti iš šios gramatikos bet koki beženkli sveikajį skaičių (352).

352 išvedimas

- 6 žingsnių procesas, pradedant
Integer

352 išvedimas (1)

- Naudojam gramatikos taisykľę:

$\langle \text{Integer} \rangle \rightarrow \langle \text{Integer} \rangle \langle \text{Digit} \rangle$

352 išvedimas (1-2)

- Dešiniają pusę keičiam viena iš taisyklių:

$$\begin{aligned} <\text{Integer}> &\rightarrow <\text{Integer}> <\text{Digit}> \\ &\rightarrow <\text{Integer}> 2 \end{aligned}$$

352 išvedimas (1-3)

- Kiekvienas žingsnis susietas su prieš tai buvusiu.

```
<Integer> → <Integer> <Digit>
          → <Integer> 2
          → <Integer> <Digit> 2
```

352 išvedimas (1-4)

```
<Integer> → <Integer> <Digit>
          → <Integer> 2
          → <Integer> <Digit> 2
          → <Integer> 5 2
```

352 išvedimas (1-5)

```
<Integer> → <Integer> <Digit>
          → <Integer> 2
          → <Integer> <Digit> 2
          → <Integer> 5 2
          → <Digit> 5 2
```

352 išvedimas (1-6)

Baigiamo, kai lieka vien tik terminaliniai simboliai

```
<Integer> → <Integer> <Digit>
          → <Integer> 2
          → <Integer> <Digit> 2
          → <Integer> 5 2
          → <Digit> 5 2
          → 3 5 2
```

Išvedimai

- Kiekviena išvedimo simbolių eilutė yra *sakinio formoje (sentential form)*
- Sakinys yra sakinio formoje, jei jis turi tik terminalus
- Kairinis (**leftmost**) išvedimas – tai išvedimas, kuriame kiekvienoje sakinio formoje keičiamas kairiausias neterminalas
- Išvedimas gali būti nei kairinis, nei dešininis

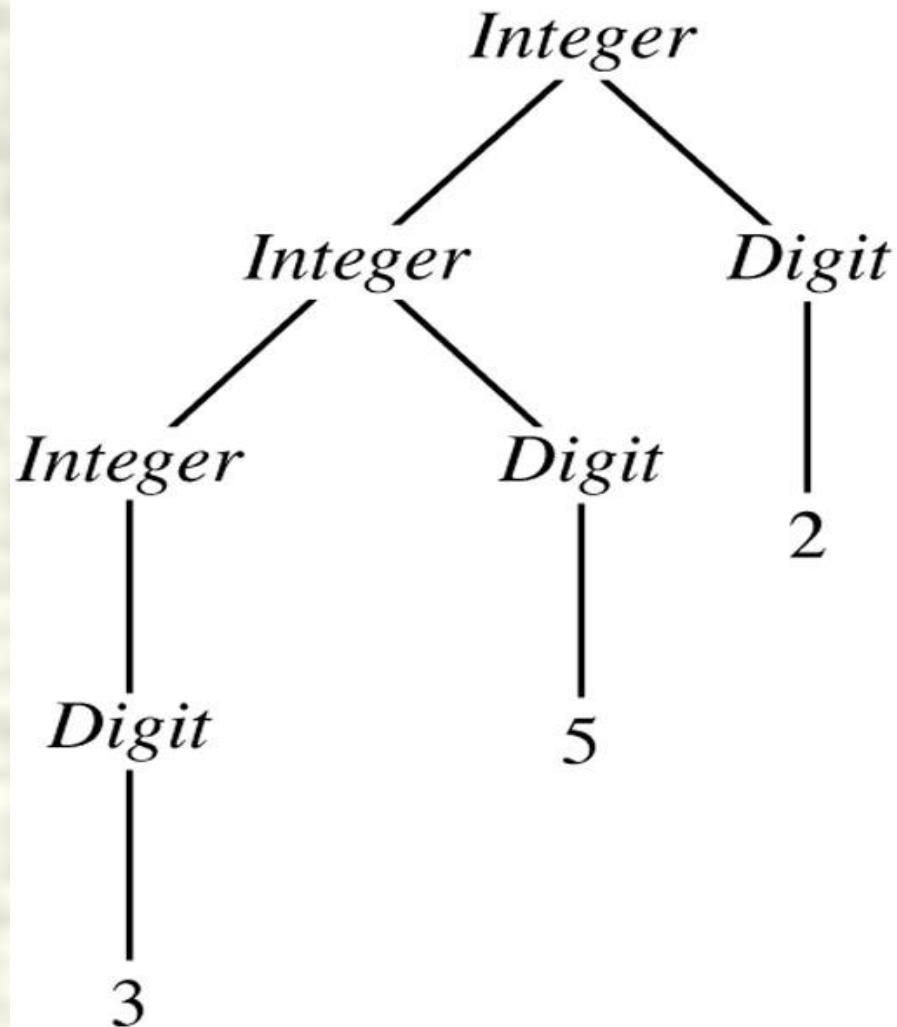
Kitas 352 išvedimas

```
<Integer> → <Integer> <Digit>
          → <Integer> <Digit> <Digit>
          → <Digit> <Digit> <Digit>
          → 3 <Digit> <Digit>
          → 3 5 <Digit>
          → 3 5 2
```

- Tai kairinis išvedimas.
- Prieš tai buvo *dešininis išvedimas*.
- Gramatinis medis ([parse tree](#))

Gramatinis medis

- Hierarchinis išvedimo atvaizdavimas



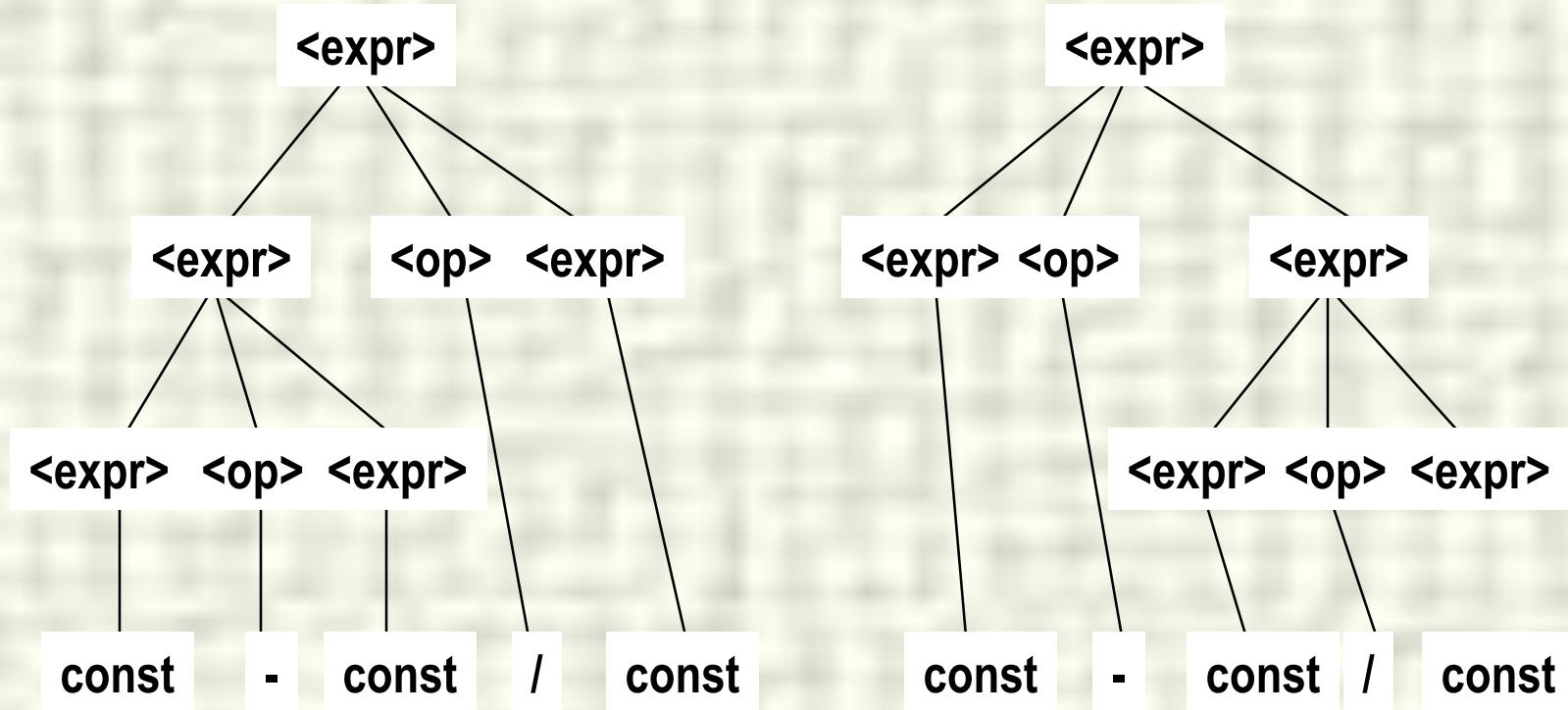
Gramatikos dviprasmiškumas

- Gramatika yra dviprasmiška (**ambiguous**) tada ir tik tada, jei ji generuoja sakinio formą, kuriai galima sudaryti keletą skirtinį gramatinių medžių

Dviprasmiška gramatika

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \mid \text{const}$

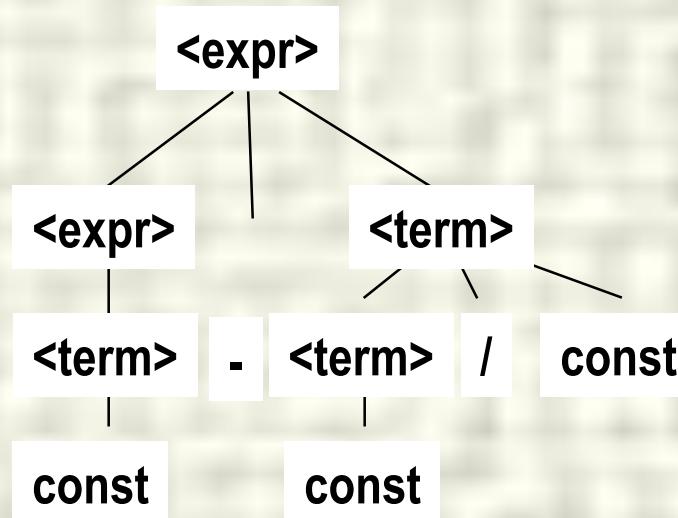
$\langle \text{op} \rangle \rightarrow / \mid -$



Nedviprasmiška gramatika

- Reikia papildomų simbolių ir taisyklių, norint nurodyti prioritetus (**precedence**)

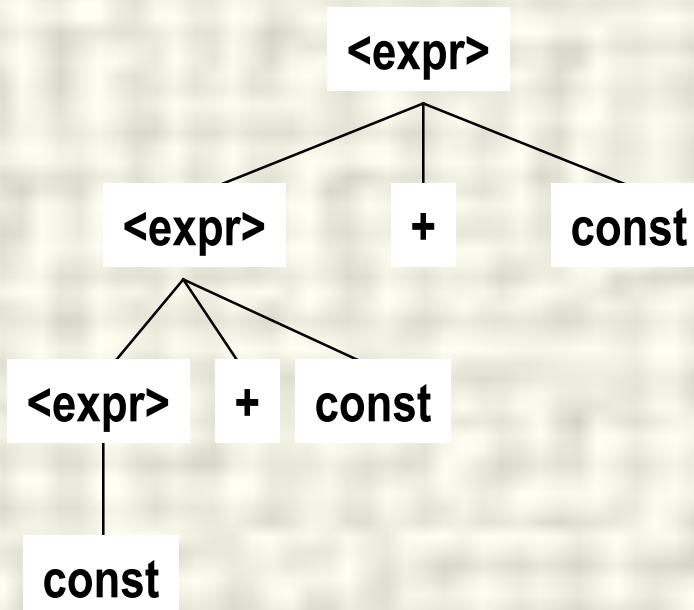
```
<expr> → <expr> - <term> | <term>
<term> → <term> / const | const
```



Operatorių asociatyvumas

- Operatorių asociatyvumą (**associativity**) taip pat gali nurodyti gramatika

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \text{const}$ (dviprasmiška)
 $\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \text{const} \mid \text{const}$ (nedviprasmiška)



Išplėsta BNF (EBNF)

- Neprivalomos dalys rašomos tarp []
 $\text{<proc_call>} \rightarrow \text{ident} [(\text{<expr_list>})]$
- Alternatyvios dalys rašomos tarp () ir atskiriamos |
 $\text{<term>} \rightarrow \text{<term>} (+|-) \text{ const}$
- Pasikartojimai (0 ir daugiau) rašomi tarp { }
 $\text{<ident>} \rightarrow \text{letter} \{ \text{letter|digit} \}$

BNF ir EBNF

- BNF

$$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle$$
$$| \quad \langle \text{expr} \rangle - \langle \text{term} \rangle$$
$$| \quad \langle \text{term} \rangle$$
$$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle$$
$$| \quad \langle \text{term} \rangle / \langle \text{factor} \rangle$$
$$| \quad \langle \text{factor} \rangle$$

- EBNF

$$\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle \{ (+ \mid -) \langle \text{term} \rangle \}$$
$$\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ (* \mid /) \langle \text{factor} \rangle \}$$

EBNF naujienos

- Alternatyvios RHSs yra atskirose eilutėse
- Naudojamas : vietoj =>
- Naudojama opt nebūtinoms dalims
- Naudojama oneof, kai yra pasirinkimas

Gramatika ir atpažinimas

- Artimas ryšys tarp atpažinimo ir generavimo
- Jei yra kalbos bekontekstė gramatika, atpažinimą galima sukonstruoti algoritmiškai – automatiškai
- Leidžia greitai sukurti kalbos sintaksės analizatorių. Labai reikšminga naujai kalbai.
- yacc, Johnson, 1975

Atributų gramatika

Statinė semantika

- Bekontekstės gramatikos (CFG) negali aprašyti visos programavimo kalbos sintaksės
- Kelia rūpestj:
 - Bekontekstė, bet varžanti (išraiškų operandų tipai)
 - Kontekstinė (kintamieji privalo būti paskelbti prieš naudojimą)
- Statinė – atliekama kompiliavimo metu (tipų apribojimai). Daugiau sintaksė, nei semantika

Atributų gramatikos

- Atributų gramatikos (AG) (Knuth – 1968) papildo CFG, įnešdamos tam tikrą semantiką į gramatinį medį
- Bekontekstė gramatika + atributai, atributų skaičiavimas (**computation**) ir predikatų funkcijos
- Atributai – lyg kintamieji, nes jiems galima priskirti reikšmes.
- AG privalumai:
 - Statinės semantikos specifikavimas
 - Kompiliatoriaus projektas (statinės semantikos tikrinimas)

Atributų gramatikos apibrėžimas

- Atributų gramatika yra bekontekstė gramatika $G = (S, N, T, P)$ su tokiais papildymais:
 - Kiekvienam gramatikos simboliui x yra aibė atributo reikšmių $A(x)$ ($S(x)$ ir $I(x)$ – nesusikertančios aibės)
 - Kiekviена taisyklė turi aibę funkcijų, kurios skaičiuoja tam tikrus neterminalo atributus taisyklėje
 - Kiekvienna taisyklė turi (gali būti tuščia) aibę predikatų, kad patikrintų atributų suderinamumą

Atributų gramatikos atributai

- Tegul $X_0 \rightarrow X_1 \dots X_n$ yra taisyklė
- Funkcijos, kurių forma $S(X_0) = f(A(X_1), \dots, A(X_n))$, apibrėžia *sintezuojamus* (*synthesized*) atributus. Kelia semantinę informaciją į viršų medžiu.
- Funkcijos, kurių forma $I(X_j) = f(A(X_0), \dots, A(X_n))$, for $1 \leq j \leq n$, apibrėžia *paveldėtus* (*inherited*) atributus. Nuleidžia semantinę informaciją žemyn medžiu.
- *Dažnai paveldėti atributai apriboti dėl ciklų (j priklauso tik nuo 0..j-1)*
- Pradžioje lapai turi *vidinius* (*intrinsic*) atributus – sintezuojamus

Predikatas

- Galimi tik tie išvedimai, kuriuose kiekvienas predikatas susietas su kiekvienu neterminalu įgyja tiesos reikšmę

Atributų gramatikos pavyzdys

- Sintaksė
 - $\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$
 - $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle + \langle \text{var} \rangle \mid \langle \text{var} \rangle$
 - $\langle \text{var} \rangle \rightarrow A \mid B \mid C$
- **actual_type**: sintezuojamas dėl $\langle \text{var} \rangle$ ir $\langle \text{expr} \rangle$
- **expected_type**: paveldėtas dėl $\langle \text{expr} \rangle$, nustatomas pagal kaireę pusę.
- Du tipai: integer ir real

Atributų gramatikos taisyklės

- Sintaksės taisyklė: $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle[1] + \langle \text{var} \rangle[2]$

Semantikos taisyklės:

$$\langle \text{expr} \rangle.\text{actual_type} \leftarrow \langle \text{var} \rangle[1].\text{actual_type}$$

Predikatas:

$$\langle \text{var} \rangle[1].\text{actual_type} == \langle \text{var} \rangle[2].\text{actual_type}$$
$$\langle \text{expr} \rangle.\text{expected_type} == \langle \text{expr} \rangle.\text{actual_type}$$

- Sintaksės taisyklė: $\langle \text{var} \rangle \rightarrow \text{id}$

Semantikos taisyklė:

$$\langle \text{var} \rangle.\text{actual_type} \leftarrow \text{lookup } (\langle \text{var} \rangle.\text{string})$$

Medžio dekoravimas

- Jei visi atributai buvo paveldėti, medis gali būti dekoruojamas iš viršaus-žemyn
- Jei visi atributai buvo sintezuoti, medis gali būti dekoruojamas iš apačios į viršų
- Daugelyje atvejų abi atributų rūšys yra naudojamos, todėl turi būti kombinuojami abu būdai: iš viršaus-žemyn, iš apačios į viršų

Atributų reikšmių skaičiavimas

`<expr>.expected_type ← inherited from parent`

`<var>[1].actual_type ← lookup (A)`

`<var>[2].actual_type ← lookup (B)`

`<var>[1].actual_type == <var>[2].actual_type`

`<expr>.actual_type ← <var>[1].actual_type`

`<expr>.actual_type == <expr>.expected_type`

Įvertinimas

- Statinės semantikos taisyklių tikrinimas – svarbi kompiliatorių dalis
- Dydis ir sudėtingumas
- Sudėtinga rašyti ir skaityti
- Mažiau formalios gramatikos naudojamos

Dinaminė semantika

Semantika

- Nėra vienintelio plačiai paplitusio būdo formaliam semantikos aprašui
- Įvairūs poreikiai:
 - Programuotojams reikia žinoti, ką reiškia sakiniai
 - Kompiliatorių kūrėjai turi tiksliai žinoti, ką atlieka kalbos konstrukcijos
 - Kad būtų įmanomi teisingumo įrodymai
 - Kad būtų įmanomi kompiliatorių generatoriai
 - Kad projektuotojai galėtų aptikti dviprasmybes ir nesuderinamumus
- Scheme

Veiklos semantika

- Aprašo programos prasmę, vykdyma jos sakinius mašinoje arba modeliuodama. Mašinos būsenos pokyčiai (atmintis, registratoriai) apibrėžia sakinių prasmę
- Norint naudoti veiklos semantiką aukšto lygio kalbai, reikalinga virtuali mašina
- Esam atlikę patys tokius bandymus – o ką gausim? Naudojome veiklos semantiką.

Problemos

- Pokyčiai per maži ir labai gausūs
- Kompiuterio atmintis yra labai didelė ir sudėtinga
- Dalyvauja ir kiti įrenginiai
- Realūs kompiuteriai ir kalbos nenaudojami
- Kuriamos tarpinės kalbos ir interpretatoriai

Lygiai

- Natūrali VS – visos programos
- Struktūrinė VS – atskirų sakinių

Grynasis interpretatorius

- Aparatinis grynasnis interpretatorius būtų per brangus
- Programinis grynasnis interpretatorius irgi turi problemą
 - Atskiro kompiuterio detalios charakteristikos trukdys suprasti veiksmus
 - Toks semantikos apibrėžimas būtų priklausomas nuo mašinos

Pavyzdys

- Sakinys
 - for (expr1; expr2; expr3)
 - ...
- Prasmė:
 - expr1;
 - loop: if expr2 == 0 goto out
 - ...
 - expr3;
 - goto loop
 - out:

Kompiuterio modeliavimas

- Geresnė alternatyva
- Procesas:
 - Sukurti transliatorių, kuris išeities kodą transliuoja į idealizuoto kompiuterio mašininį kodą
 - Sukurti idealizuoto kompiuterio modeliavimo programą

Veiklos semantikos taikymas

- Mokymas:
 - Kalbos vadovėliai ir mokomosios knygos
 - Programavimo kalbų mokymas
- Semantikos įvertinimas:
 - Gera, jei naudojama neformaliai (kalbos vadovėliai ir t.t.)
 - Labai sudėtinga, jei naudojama formaliai (VDL)
 - Buvo panaudota PL/I kalbos semantikos aprašymui
 - Veiklos semantikos aprašymas remiasi kitos žemesnio lygio kalbos panaudojimu, bet ne matematika

Žymėjimo semantika

Ypatybės

- Remiasi rekursine funkcijų teorija
- Sukūrė Scott ir Strachey (1970)
- Plačiausiai žinomas
- Tiksliausias semantikos aprašymo būdas

Apibrėžimas

- Žymėjimo specifikacijos apibrėžimas kalbai:
 - Kiekvienai kalbos esybei apibrėžti matematinį objektą
 - Apibrėžti funkciją, kuri susieja kalbos esybių atvejus su atitinkamais matematinių objektų atvejais
 - Privalumas – su matematiniais objektais galime atlikti tik apibrėžtus veiksmus
 - Sunkumas – sukurti objektus ir funkcijas
- Iš sintaksinės srities (funkcijos argumentai) žymi į semantinę sritį (funkcijos reikšmę)

Panašumai ir skirtumai kartu

- Veiklos semantika. Pokyčiai idealiame kompiuteryje
- Veiklos semantika išreiškia kalbos struktūras paprastesnėje kalboje (ir skirtumas čia pat)
- Žymėjimo semantika tam naudoja matematinius objektus
- Veiklos semantika naudoja **mašinos būsenas**. Būsenų pokyčius apsprendžia **algoritmai**
- Žymėjimo semantika naudoja **programos būseną**. Būsenų pokyčius apibrėžia **matematinės funkcijos**

Akivaizdus skirtumas

- Žymėjimo semantika nemodeliuoja pažingsniui programų skaičiavimo proceso, skirtingai nuo veiklos semantikos

Dešimtainiai skaičiai

```
<dec_num> → '0' | '1' | '2' | '3' | '4' | '5' |
              '6' | '7' | '8' | '9' | 
              <dec_num> ('0' | '1' | '2' | '3' |
                           '4' | '5' | '6' | '7' |
                           '8' | '9')
```

$$M_{dec}('0') = 0, \quad M_{dec}('1') = 1, \dots, \quad M_{dec}('9') = 9$$

$$M_{dec}(<dec_num> '0') = 10 * M_{dec}(<dec_num>)$$

$$M_{dec}(<dec_num> '1') = 10 * M_{dec}(<dec_num>) + 1$$

...

$$M_{dec}(<dec_num> '9') = 10 * M_{dec}(<dec_num>) + 9$$

Programos būsena

- Programos būsena – tai visų kintamuju reikšmės

$$s = \{ \langle i_1, v_1 \rangle, \langle i_2, v_2 \rangle, \dots, \langle i_n, v_n \rangle \}$$

- Tegul **VARMAP** bus funkcija, kuri duotam kintamojo vardui ir būsenai, gražina esamą kintamojo reikšmę

$$\text{VARMAP}(i_j, s) = v_j \ (\text{undef})$$

- Žymi būsenas į būsenas, bet ne į reikšmes, kaip išraiškos

Išraiškos

- Susieti su reikšmėmis
- Z – sveikujų skaičių aibė, vienintelė klaida – kintamasis turi neapibrėžtą reikšmę
- Susieti išraiškas su $Z \cup \{\text{klaida}\}$
- Išraiškos gali būti dešimtainiai skaičiai, kintamieji ar dvejetainės išraiškos, turinčios aritmetinį operatorių (+, *), apjungiantį du operandus, kurių kiekvienas gali būti išraiška.

Kalbos BNF

$\langle \text{expr} \rangle \rightarrow \langle \text{dec_num} \rangle | \langle \text{var} \rangle | \langle \text{binary_expr} \rangle$

$\langle \text{binary_expr} \rangle \rightarrow \langle \text{left_expr} \rangle \langle \text{operator} \rangle \langle \text{right_expr} \rangle$

$\langle \text{left_expr} \rangle \rightarrow \langle \text{dec_num} \rangle | \langle \text{var} \rangle$

$\langle \text{right_expr} \rangle \rightarrow \langle \text{dec_num} \rangle | \langle \text{var} \rangle$

$\langle \text{operator} \rangle \rightarrow + | *$

Išraiškų aprašas

```
Me(<expr>, s) Δ= case <expr> of
  <dec_num> =>
    Mdec(<dec_num>, s)
  <var> =>
    if VARMAP(<var>, s) == undef
      then klaida
      else VARMAP(<var>, s)
  <binary_expr> =>
    if (Me(<binary_expr>.<left_expr>, s) == undef
        OR Me(<binary_expr>.<right_expr>, s) ==
          undef)
      then klaida
    else
      if (<binary_expr>.<operator> == '+' then
          Me(<binary_expr>.<left_expr>, s) +
          Me(<binary_expr>.<right_expr>, s)
        else Me(<binary_expr>.<left_expr>, s) *
          Me(<binary_expr>.<right_expr>, s)
```

Priskyrimo sakiniai

- Susieti būsenos aibes su būsenos aibės $U \{klaida\}$

$M_a(x := E, s) \Delta = if \ M_e(E, s) == kлаida$
 then kлаida
 else $s' =$
 $\{<i_1, v_1'>, <i_2, v_2'>, \dots, <i_n, v_n'>\},$
 where for $j = 1, 2, \dots, n,$
 if $i_j == x$
 then $v_j' = M_e(E, s)$
 else $v_j' = VARMAP(i_j, s)$

Loginiai ciklai

- Susieti būsenos aibes su būsenos aibės $U \{klaida\}$

```
 $M_1(\text{while } B \text{ do } L, s) \Delta= \begin{cases} \text{if } M_b(B, s) == \text{undef} \\ \quad \text{then } \text{klaida} \\ \quad \text{else if } M_b(B, s) == \text{false} \\ \quad \quad \text{then } s \\ \quad \quad \text{else if } M_{s1}(L, s) == \text{klaida} \\ \quad \quad \quad \text{then } \text{klaida} \\ \quad \quad \quad \text{else } M_1(\text{while } B \text{ do } L, M_{s1}(L, s)) \end{cases}$ 
```

Ciklų prasmė

- Ciklo prasmė – tai programos kintamuju reikšmės, įvykdžius ciklą, jei nebuvo klaidų
- Iš ties, ciklas iš iteracijos yra pakeičiamas rekursija, kai rekursijos valdymą nusako kitos rekursinės būsenos žymėjimo funkcijos
 - Rekursija, lyginant su iteracija, yra daug lengviau aprašoma matematiškai

Įvertinimas

- Gali būti naudojama programų teisingumo įrodymui
- Suteikia formalų kelią mąstymui apie programas
- Gali padėti kalbos projektavimui
- Gali būti naudojama kompiliatorių generavimo sistemose
- Dėl sudėtingumo mažai naudos kalbos vartotojams

Aksiomų semantika

Apibrėžimas

- Remiasi formaliaja logika (predikatų skaičiavimai)
- Apibrėžia, ką būtų galima įrodyti apie programą (ar programa teisinga)
- Abstrakčiausias būdas
- Pradinis tikslas: formalus programų verifikavimas
- Aksiomos ir išvedimo taisyklės yra apibrėžtos kiekvienam kalbos sakinio tipui
- Prieš ir po kiekvieno sakinio rašomos loginės išraiškos, apribojimai programos kintamiesiems
- Loginės išraiškos vadinamos *teiginiais (assertions)*
- Sakinių prasmė apibrėžiama jų efektais į teiginius

Teiginiai

- Teiginys prieš sakinį (išankstinė sąlyga - **precondition**) nustato ryšius ir apribojimus tarp kintamųjų, kurie yra reikšmingi šiam vykdymo taškui
- Teiginys po sakinio yra po sąlyga (galutinė sąlyga – **postcondition**)
- Silpniausia išankstinė sąlyga yra mažiausias apribojimas, kuris užtikrins galutinės sąlygos įmanomumą
- Išankstinės sąlygos skaičiuojamos iš galutinių.

Aksiomų semantikos forma

- Prieš, po forma: $\{P\}$ **sakinys** $\{Q\}$
- Pavyzdys
 - $a = b + 1 \quad \{a > 1\}$
 - Galima išankstinė sąlyga: $\{b > 10\}$
 - Silpniausioji išankstinė sąlyga: $\{b > 0\}$ (mažiausiai ribojanti sąlyga)
- Išankstinė ir galutinė sąlygos pilnai apibrėžia sakinio prasmę

Programos įrodymo procesas

- Rezultato charakteristikos – visos programos galutinė sąlyga
- Visos programos norima sąlyga yra norimas rezultatas
 - Eikime atgal per programą atgal link pirmojo sakinio. Jei pirmojo sakinio išankstinė sąlyga yra tokia pati, kaip nurodyta programos specifikacijoje, tai programa teisinga

Aksiomos

- Aksioma – išvedimo taisyklė be viršutinės dalies, nes ji visada tiesa
- Priskyrimo sakinio aksioma
 $x = E \{Q\}$ (galutinė sąlyga)
tuomet išankstinė sąlyga $P = Q_{x \rightarrow E}$
 $\{Q_{x \rightarrow E}\} x = E \{Q\}$
- Išvedimo (**inference**) taisyklė:

$$\frac{\{P\} S \{Q\}, P' \Rightarrow P, Q \Rightarrow Q'}{\{P'\} S \{Q'\}}$$

- Galutinė sąlyga gali būti silpninama, išankstinė – stiprinama

Aksiomos 2

- Išvedimo taisyklė sekai S1; S2

{P1} S1 {P2}

{P2} S2 {P3}

$$\frac{\{P1\} S1 \{P2\}, \{P2\} S2 \{P3\}}{\{P1\} S1; S2 \{P3\}}$$

Išvedimo taisyklė ciklui

- Išvedimo taisyklė loginiam ciklui, tikrinančiam sąlygą prieš ciklą

{P} while B do S end {Q}

$$\frac{(I \text{ and } B) \text{ S } \{I\}}{\{I\} \text{ while } B \text{ do } S \{I \text{ and } (\text{not } B)\}}$$

čia I – ciklo invariantas (indukcijos hipotezė)

Ciklo invariantas

- Indukcija
 - $\{P\}$ while B do S end $\{Q\}$
- I privalo tenkinti tokias sąlygas:
 - $P \Rightarrow I$ -- ciklo invariantas turi galiouti pradžioje
 - $\{I\} B \{I\}$ – Loginės sąlygos skaičiavimas privalo nepakeisti I galiojimo
 - $\{I \text{ and } B\} S \{I\}$ -- I nepasikeičia, įvykdžius ciklą
 - $(I \text{ and } (\text{not } B)) \Rightarrow Q$ -- Jei I true ir B false, Q galioja
 - Ciklas baigia – gali būti sunku įrodyti

Ciklo invariantas - 2

- Ciklo invariantas yra silpnesnė ciklo galutinės sąlygos versija ir yra išankstinė sąlyga taip pat.
- Ciklo invariantas privalo būto pakankamai silpnas, kad galiotų prieš ciklo vykdymą, bet apjungtas su ciklo pabaigos sąlyga, privalo būti pakankamai stiprus, kad užtikrintų ciklo galutinės sąlygos galiojimą

Įvertinimas

- Sukurti aksiomas ir išvedimo taisykles visiems kalbos sakiniams yra sunku.
- Tai gera priemonė teisingumo įrodymams, puikus karkasas programų analizei, bet nelabai naudinga kalbos vartotojams ir kompiliatorių kūrėjams

Santrauka

- BNF ir bekontekstės gramatikos yra ekvivalentiškos metakalbos
- Atributų gramatika gali aprašyti kalbos sintakse ir semantiką
- Pirminiai semantikos aprašymo metodai:
 - Veiklos, žymėjimo, aksiomų

P175B124

Leksinė ir sintaksinė analizė

***To understand a program you
must become both the machine
and the program.***

A. Perlis

Praėjusios temos klausimai

1. Kokius žinote kalbos gramatikų aprašymo lygmenis?
2. Kada buvo sukurta BNF (Backus-Naur forma) ir koks buvo jos tikslas?
3. Kas sudaro bekontekstę gramatiką?
4. Kada bekontekstė gramatika yra dviprasmiška?
5. Kas yra atributų gramatika?
6. Kokius atributų tipus žinote?
7. Kaip kalbos sakinių prasmę apibrėžia veiklos semantika?
8. Kaip kalbos sakinių prasmę apibrėžia žymėjimo semantika?
9. Kokie yra panašumai ir skirtumai tarp veiklos ir žymėjimo semantikų?
10. Pristatykite būdą, kurį naudoja aksiomų semantika programų teisingumo įrodymui
11. Kokiam tikslui turi vertę aksiomų semantika?

Temos klausimai

- Leksinė analizė
- Sintaksės analizė
 - Problema
 - Rekursinis nusileidžiantis gramatinis nagrinėjimas (**recursive-descent parser**)
 - Iš apačios į viršų gramatinis nagrinėjimas (**bottom-up parser**)

Sintaksės analizė

- Kalbos įgyvendinimo sistemos – kompiliatoriai – privalo analizuoti pradinj kodą – sintaksės analizė (SA)
- SA – tai kompiliatoriaus širdis
- SA remiasi gramatika
- Kita – programos listingo formavimas, programas sudėtingumas, konfigūravimo failo turinys
- Beveik visi SA naudoja kalbos sintaksės BNF aprašą
- Kompiliatorių nesimokome

BNF privalumai sintakseje

- Aiškus ir glaustas sintaksės aprašas tiek žmonėms, tiek programoms
- Analizatorius gali būti tiesiogiai bazuojamas BNF
- Analizatorius, kurie bazuojami BNF, lengva prižiūrėti dėl jų modulišumo

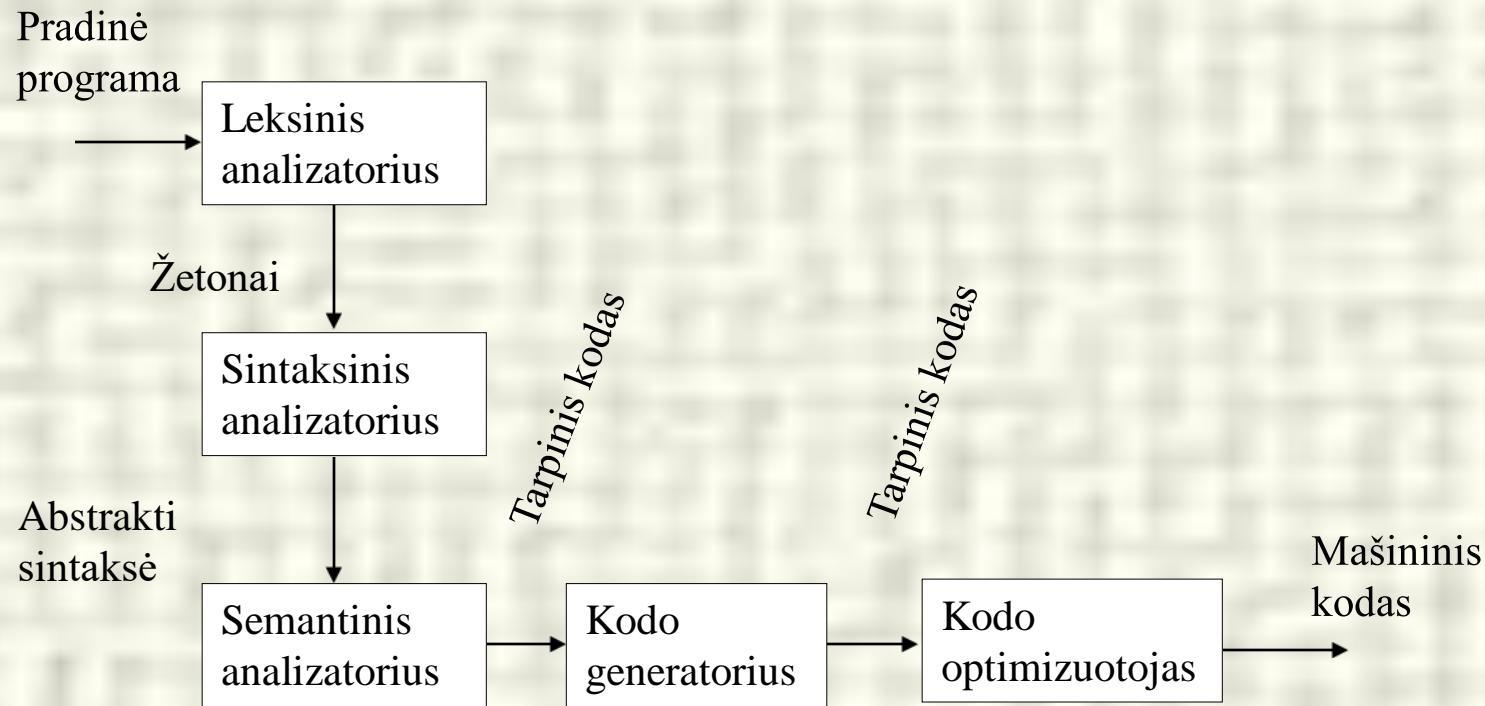
Sintaksės analizės dalys

- *Leksinė analizė:*
 - Žemo lygio dalis (**small-scale**) – *leksikos analizatorius* kartais vadinamas skeneriu. Matematiškai, baigtinis automatas, bazuojamas reguliaria gramatika
 - Mažos kalbos konstrukcijos, tokios kaip vardai ir skaitmeniniai literalai
- *Sintaksinė analizė:*
 - Aukšto lygio dalis (**large-scale**) – *sintaksinis analizatorius (parser)*. Matematiškai, postūmio automatas, bazuojamas bekontekste gramatika
 - Didelės kalbos konstrukcijos, tokios kaip išraiškos, sakiniai, programiniai vienetai

Atskyrimo priežastys

- *Paprastumas* – leksinei analizei naudojami mažiau sudėtingi metodai. Jų atskyrimas supaprastina leksikos ir sintaksės analizatorius
- *Efektyvumas* – atskyrimas leidžia optimizuoti leksinį analizatoriu
- *Mobilumas* – leksinio analizatoriaus dalys gali būti nepernešamos, bet sintaksės analizatorius visados pernešamas

Kompiliavimo procesas



Leksinė analizė

- Tikslas: transformuoti programos atvaizdavimą
- Įvestis: spausdinami ASCII simboliai
- Išvestis: žetonai
- Atmeta: tuščius tarpus, komentarus
- Apibrėžimas: žetonas yra logiškai susietų simbolių seka, atvaizduojanti vieną kalbos elementą

Sintaksinė analizė

- Remiasi BNF/EBNF gramatika
- Įvedimas: žetonai – leksinė analizė – tai sintaksinės analizės dalis
- Rezultatas: abstraktus sintaksės medis (gramatinis medis)
- Abstrakti sintaksė: gramatinis medis su išdėstymu, dauguma neterminalų atmesti

Semantinė analizė

- Patikrinti, kad visi kintamieji deklaruoti
- Atlikti tipų tikrinimą
- Išterpti numanomus konversijos operatorius
(padaryti juos išreikštus)

Kodo optimizavimas

- Ivertinti išraiškas kompliačijos metu
- Perdėlioti kodą, kad pagerintų atminties išnaudojimą
- Pašalinti bendras mažesnes išraiškų dalis
- Pašalinti nereikalingą kodą

Kodo generavimas

- Rezultatas: mašininis kodas
- Komandų parinkimas
- Registru valdymas
- "Peephole" optimizavimas

Leksikos analizē

Leksikos analizatorius

- Tikslas: transformuoti programos atvaizdavimą
- Pattern matcher – rinkinių atitikimas (reguliarios išraiškos).
- Sintaksinės analizės front end.
- Surenka simbolius į logines grupes ir priskiria vidinius kodus pagal grupių struktūrą
- Loginės grupės vadinamos leksemomis ir šiuų grupių vidiniai kodai yra vadinami žetonais.

Leksinės analizės procesas

- Duomenys: spausdinami ASCII simboliai
- Rezultatas: žetonai
- Atmeta: tarpus, komentarus
- Paprastai žetonai yra įvardintos sveikojo tipo konstantos

Žetonai ir leksemos

Sakinys

result = **oldsum** – **value**;

Žetonas leksema

IDENT **result**

ASSIGN_OP =

IDENT **oldsum**

SUB_OP –

IDENT **value**

SEMICOLON ;

Veiksmai

- Iprastai, leksikos analizatorius – tai funkcija, kurią kviečia parseris, kai jam reikia naujos leksemos.
- Įterpia vartotojo apibrėžtus vardus į simbolių lentelę
- Aptinka sintaksines klaidas leksemose ir praneša vartotojui

Įgyvendinimo būdai

- Trys įgyvendinimo būdai:
 - Parašyti formalų žetonų aprašą ir naudoti PĮ, kuri automatiškai sudaro leksikos analizatorių šiam aprašui. lex Unix'e. Formali kalba artima reguliariosioms išraiškoms
 - Suprojektuoti būsenų diagramą, kuri aprašo žetonus ir parašyti programą, kuri įgyvendina šią būsenų diagramą
 - Suprojektuoti būsenų diagramą, kuri aprašo žetonus ir ranka suprojektuoti lentelinį šios diagramos įgyvendinimą

Būsenų diagramos projektavimas

- Būsenų diagrama – tai nukreiptas (**directed**) grafas
 - Triviali būsenų diagrama turėtų išeities kalbos kiekvieno simbolio kiekvienos būsenos perėjimą (**arc**)
- Daugelyje atvejų perėjimai gali būti apjungti, norint supaprastinti būsenų diagramą
 - Kai atpažįstamas kintamojo vardas, visos didžiosios ir mažosios raidės yra ekvivalentiškos
 - Kai atpažįstamas skaitmeninis literalas, visi skaitmenys yra ekvivalentiški

Lentelė

- Rezervuoti žodžiai ir kintamuju vardai gali būti atpažinti kartu
 - Naudokite lentelę, norint nuspresti, ar galimas kintamojo vardas yra iš tiesų rezervuotas žodis

Baigtinis automatas

- Būsenų diagramos atvaizduoja baigtinį automatą
- Baigtinis automatas gali atpažinti reguliarią kalbą
- Leksikos analizatorius yra baigtinis automatas
- Žetonai – tai reguliari kalba

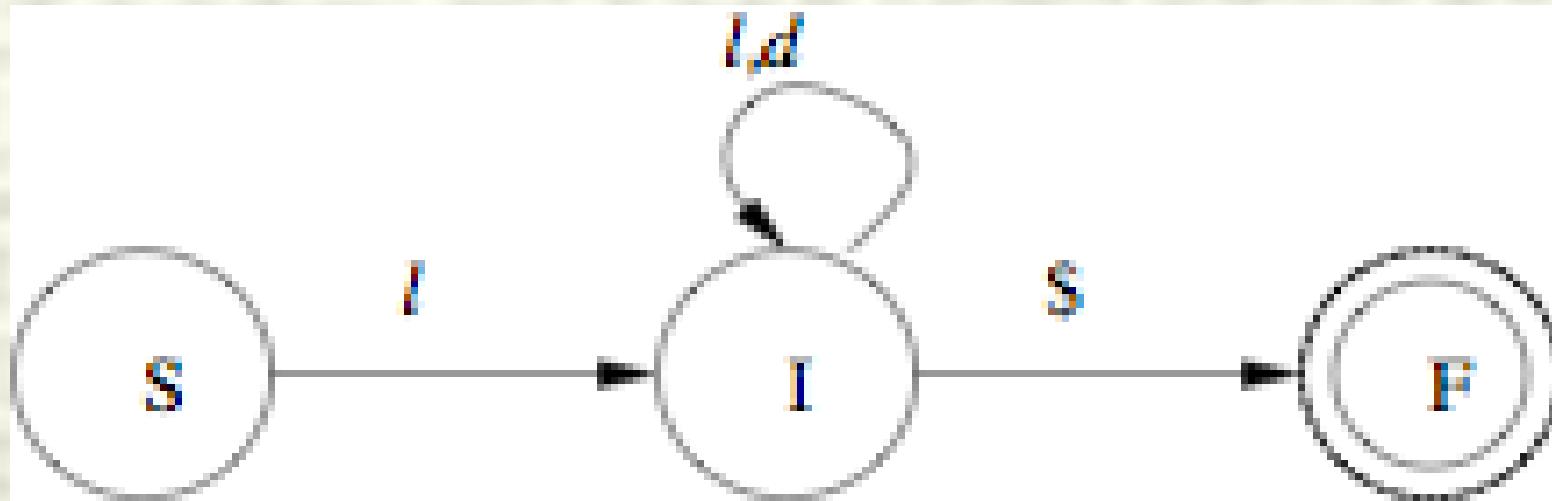
Baigtinių būsenų automatas

1. Būsenų aibė atvaizduojama grafo mazgais
2. Įvedimo alfabetas papildomas unikaliu simboliu, žyminčiu įvedimo pabaigą
3. Būsenos perėjimo funkcija atvaizduojama nukreipta kraštine iš vieno mazgo į kitą ir pažymima vienu ar daugiau alfabeto simbolių ir veiksmų.
4. Unikali starto būsena
5. Viena ar daugiau galutinių būsenų (būsenų be išėjimo kraštinių)

Deterministinis automatas

- Baigtinių būsenų automatas yra deterministinis, jei kiekvienai būsenai ir kiekvienam įėjimo simboliui yra tik viena išėjimo kraštinė iš būsenos, pažymėtos šiuo simboliu – nėra dviprasmiškumo. Nedeterministinis – kai kelios briaunos, pažymėtos tuo pačiu simboliu iš tos pačios viršūnės.

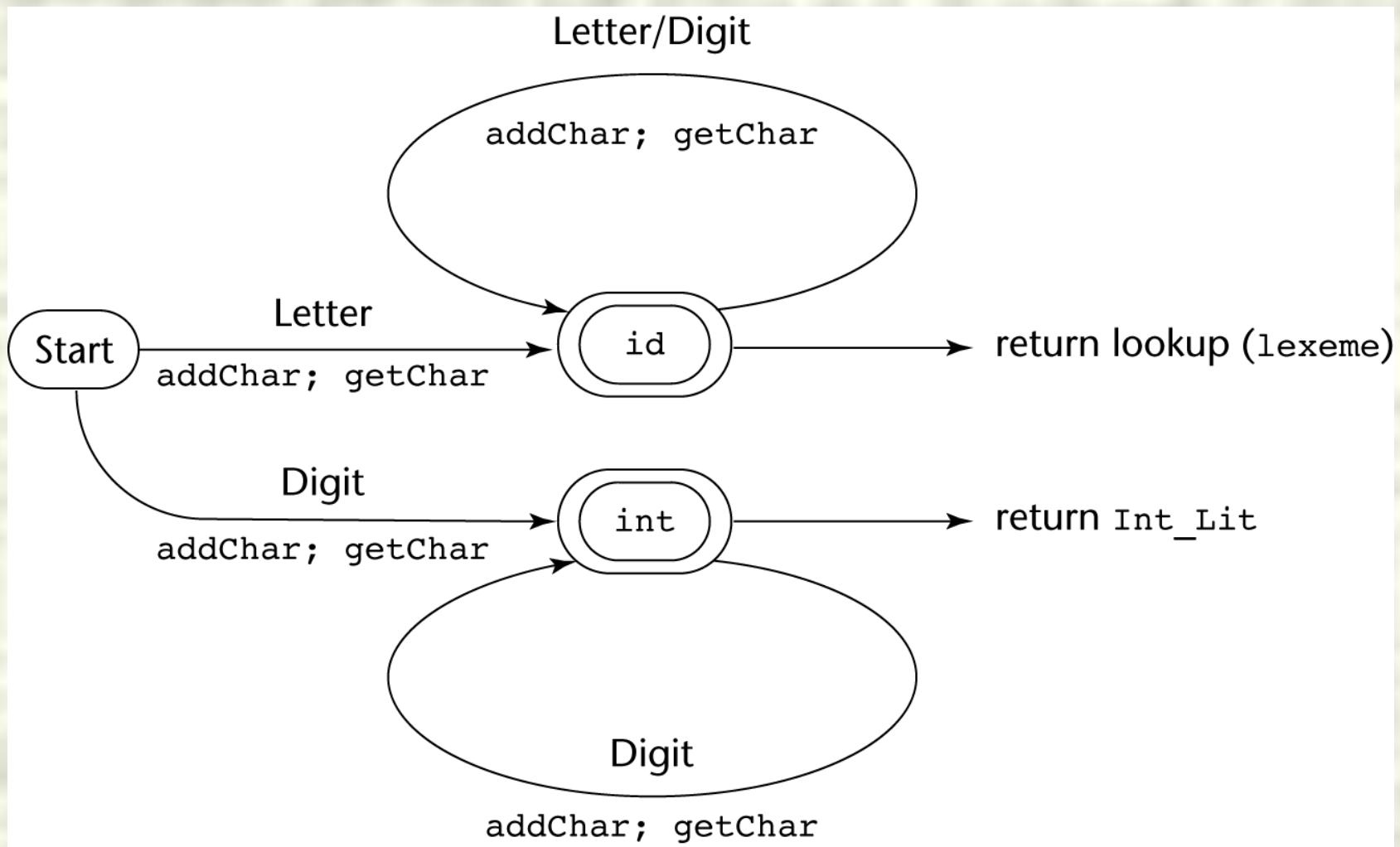
Vardų automatas



Naudingos paprogramės

- **getChar** – įveda naują simbolį iš įvesties, patalpina jį į **nextChar**, nustato jo klasę ir patalpina jo klasę į **charClass**
- **addChar** – simbolį iš **nextChar** talpina į vietą, kur kaupiama leksema
- **lookup** – nustato, ar sukaupta leksemoje eilutė yra rezervuotas žodis

Būsenų diagramma



Simbolių lentelė

- Leksikos analizatorius yra atsakingas už pradinj simbolių lentelės sudarymą. Ji naudojama kaip DB vardų saugojimui
- Simbolių lentelės esybės saugo vartotojo apibrėžtus vardus ir jų atributus
- Atributus prideda kitos kompiliatoriaus dalys

Sintaksinė analizė

Sintaksės problema

- Parsing problem
- Tikslas – atpažinti šaltinio struktūrą
- Skirtingi įgyvendinimo tikslai:
 - Surasti visas sintaksines klaidas,
 - Sukurti gramatinį medį
- Kiekvienai klaidai išduoda atitinkamą diagnostinį pranešimą. Atstato savo būseną, kad surastų kuo daugiau klaidų
- Duomenys: žetonai
- Rezultatas: gramatinis medis

Žymėjimai

- Terminaliniai simboliai – mažosios raidės alfabeto pradžioje (a, b, ...)
- Neterminaliniai simboliai – didžiosios raidės alfabeto pradžioje (A, B, ...)
- Terminaliniai arba neterminaliniai simboliai – didžiosios raidės alfabeto pabaigoje (W, X, Y, Z)
- Terminalų eilutės – mažosios raidės alfabeto pabaigoje (w, x, y, z)
- Mišrios eilutės – mažosios graikų abécélės raidės (α , β , γ , δ)

Gramatikos analizatorių rūšys

- Dvi rūšys
 - *Iš viršaus žemyn* – sukuria gramatinį medį, pradedant nuo šaknies
 - Kairiojo išvedimo tvarka
 - *Iš apačios į viršų* – sukuria gramatinį medį, pradedant nuo lapų
 - Tvarka, priešinga dešiniajam išvedimui
- Naudingi analizatoriai žiūri tik vienu žetonu į priekį

Iš viršaus žemyn

- Duotai sakinio formai $xA\alpha$, parseris privalo parinkti teisingą A-taisyklę, kad gautų kitą sakinio formą kairiausiaame išvedime, naudodamas tik pirmajį žetoną, gautą pagal A
- x – terminalų eilutė, A – neterminalas, α - mišri eilutė. A yra kairiausias neterminalas, ir jis privalo būti išplėstas. Tinkama taisyklė privalo būti parinkta.
- Įvairūs būdai. Bendriausias – palyginti naują įvedimo žetoną su pirmuoju simboliu, kurį gali sugeneruoti taisyklė

352 išvedimas

<Integer> → <Integer> <Digit>
→ <Integer> <Digit> <Digit>
→ <Digit> <Digit> <Digit>
→ 3 <Digit> <Digit>
→ 3 5 <Digit>
→ 3 5 2

- Tai kairinis išvedimas.

Iš viršaus žemyn algoritmai

- Rekursinės nusileidimo – koduotas BNF įgyvendinimas
- Alternatyva – įgyvendinimas lentele
- Abu yra LL :
 - Pirmoji L – įvedimo skaitymas iš kairės į dešinę (**left-to-right scan**)
 - Antroji L – kairiausiasis id išvedimas generuojamas

Parserių sudėtingumas

- Parseriai, kurie dirba su nedviprasmiška gramatika, yra sudėtingi ir neefektyvūs ($O(n^3)$, čia n įvedimo eilutės ilgis)
- Kompiliatoriai naudoja parserius, kurie dirba tik su nedviprasmiškos gramatikos poaibiu ir laikas yra tiesinis ($O(n)$)

Rekursinis nusileidimo parseris

- Kiekvienam gramatikos neterminalui yra paprogramė, kuri išnagrinėja sakinius, kuriuos gali sugeneruoti šis neterminalas
- Paprastai paprogramės yra rekursinės. Rekursija atspindi programavimo kalbų prigimtį, kai viena į kitą talpinamos įvairių rūšių struktūros (ciklo sakiniuose – kiti ciklo sakiniai, skliaustuose – kiti skliaustai).
- EBNF labai tinka rekursiniams nusileidimo parseriui, nes ji minimizuoja neterminalų skaičių

Paprastų išraiškų gramatika

```
<expr> → <term> { (+ | -) <term>}  
<term> → <factor> { (* | /)  
    <factor>}  
<factor> → id | int_constant | (  
    <expr> )
```

Rekursinis nusileidimas

- Sakykim, turim leksikos analizatorių `lex`, kuris naujo žetono kodą įrašo į `nextToken`
- Kodavimo procesas, kai yra tik viena RHS:
 - Kiekvieną RHS terminalinį simbolį palyginkite su nauju žetonu. Jei jie atitinka, teskite, priešingu atveju – klaida
 - Kiekvienam RHS neterminalo simboliui kvieskite jo atitinkamą paprogramę

Pirma gramatikos taisyklė

```
/* <expr> → <term> { (+ | -) <term>} */  
  
void expr() {  
    printf("Enter <expr>\n");  
    /* Analizuojame pirmąjį termą */  
    term();  
    /* Kol + ar -, kvesti lex naujam žetonui ir apdoroti  
     * kitą termą */  
  
    while (nextToken == ADD_OP ||  
          nextToken == SUB_OP) {  
        lex();  
        term();  
    }  
    printf("Exit <expr>\n");  
}
```

Paprogramė

- Ši paprogramė neaptinka klaidų
- Paprogramė paprasta, nes tik viena RHS
- Susitarimas: kiekviena nagrinėjimo paprogramė palieka naują žetoną **nextToken**

Daugiau nei vienas RHS

- Jei neterminalas turi daugiau, nei vieną RHS, reikia nustatyti, kurią RHS nagrinėti
 - Tinkama RHS parenkama, remiantis nauju jėjimo žetonu
 - Naujas žetonas lyginamas su pirmuoju žetonu, kurį gali sugeneruoti kiekviena RHS, kol bus surastas sutapimas
 - Jei sutapimo nėra, sintaksinė klaida

Kita gramatikos taisyklė

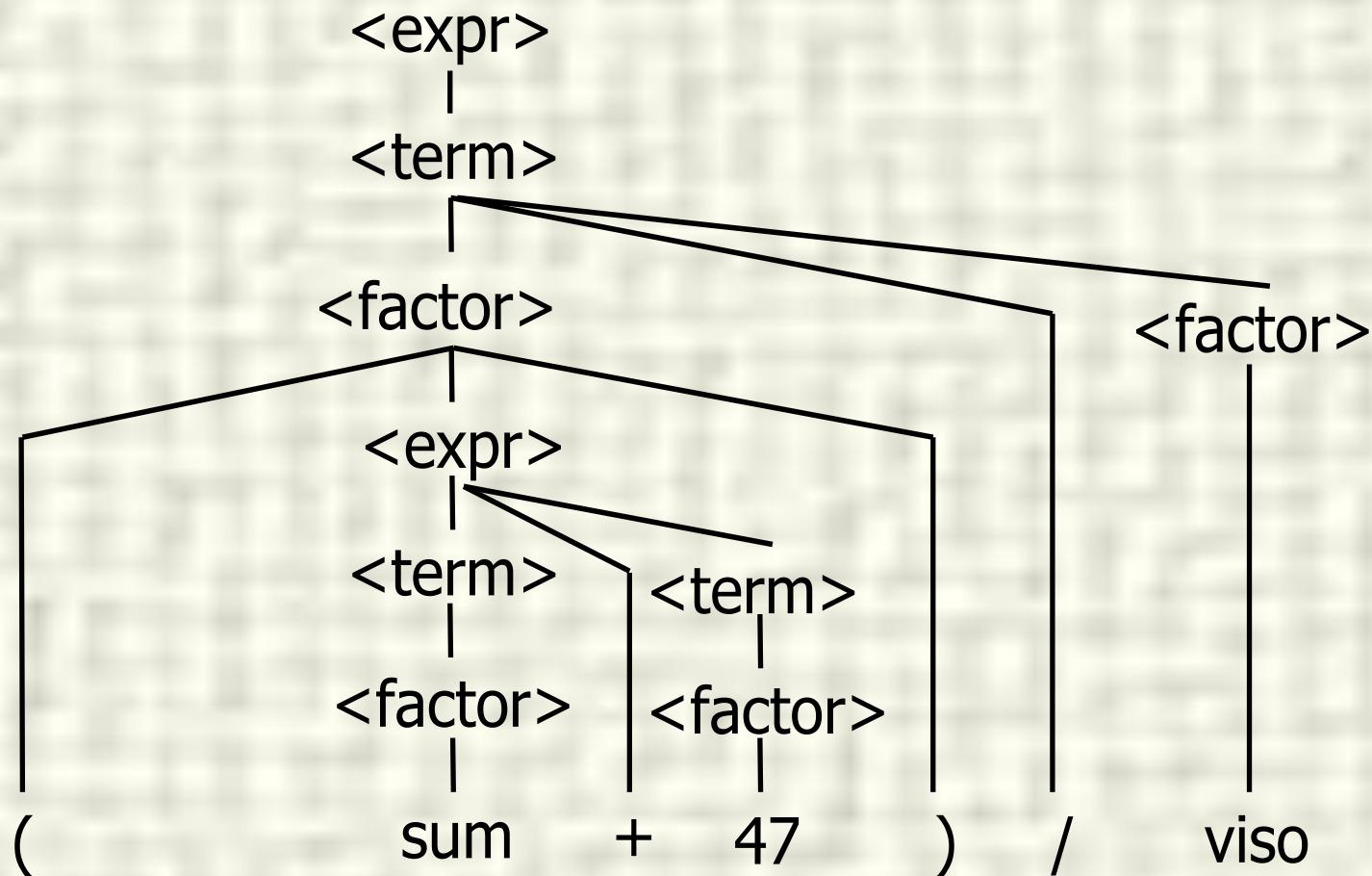
```
/* <term> -> <factor> { (* | /) <factor>} */

void term() {
    printf("Enter <term>\n");
    factor(); /* Nagrinēti pirmajā nari */
    /* Kol naujas žetonas * ar /, imti kitā žetonā ir
       nagrinēti kitā nari */
    while (nextToken == MULT_OP || nextToken == DIV_OP)
    {
        lex();
        factor();
    }
    printf("Exit <term>\n");
}
```

Paskutinė gramatikos taisyklė

```
/* <factor> → id | int_constant | ( <expr> ) */
void factor() {
    /* Nustatyti, kuris RHS */
    if (nextToken == ID_CODE || nextToken == INT_CODE)
        lex();
    /* Jei RHS (<expr>) – kvieсти lex, kad praeitų kairiųjų
       skliaustą, kvieсти expr, ir tikrinti dešiniųjų skliaustų */
    else if (nextToken == LP_CODE) {
        lex();
        expr();
        if (nextToken == RP_CODE)
            lex();
        else
            error();
    }
    else error(); /* Nė vienas RHS neatitinka */
}
```

Gramatinis medis (sum+47)/viso



LL gramatikos charakteristikos

LL gramatikos problema

- Kairiosios rekursijos problema
 - Jei gramatika turi tiesioginę ar netiesioginę kairiąją rekursiją, ji negali būti iš viršaus žemyn parserio baze
 - $A \rightarrow A + B$
 - Tiesioginė kairioji rekursija

Problemos sprendimas

- Gramatiką galima modifikuoti, kad neliktu tiesioginės kairiosios rekursijos

Kiekvienam neterminalui A,

1. Sugrupuokite A-taisykles kaip $A \rightarrow Aa_1 \mid \dots \mid Aa_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ čia nė viena iš β 's neprasideda A
2. Pakeiskite pradines A-taisykles:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow a_1 A' \mid a_2 A' \mid \dots \mid a_m A' \mid \epsilon$$

ϵ – tuščia eilutė, ištrinymo taisyklė

Pavyzdys

- Turime

$\langle E \rangle \rightarrow E + T \mid T$

$\langle T \rangle \rightarrow T * F \mid F$

$\langle F \rangle \rightarrow (E) \mid id$

$\alpha_1 = + T$ ir $\beta = T$

$\langle E \rangle \rightarrow T E'$

$\langle E' \rangle \rightarrow + T E' \mid \epsilon$

$\alpha_1 = * F$ ir $\beta = F$

$\langle T \rangle \rightarrow F T'$

$\langle T' \rangle \rightarrow * F T' \mid \epsilon$

Porinis nesusijungimas

- Pairwise disjointness test
- Antroji gramatikos charakteristika, kuri trukdo iš viršaus žemyn nagrinėjimą, yra porinio nesusijungimo nebuvismas
 - Neįmanoma nustatyti teisingą RHS, pažiūrėjus vieną žetoną į priekj
 - Testas – suskaičiuoti aibę pagal pateiktą RHS neterminala: $\text{FIRST}(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\}$
(If $\alpha \Rightarrow^* \varepsilon$, ε yra $\text{FIRST}(\alpha)$)

\Rightarrow^* žymi 0 ar daugiau išvedimo žingsnių

Porinio nesusijungimo testas

- Kiekvieno gramatikos neterminalo A, kuris turi daugiau nei vieną RHS, kiekvienai porai taisyklių $A \rightarrow \alpha_i$ ir $A \rightarrow \alpha_j$ turi galioti:
 $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$

- Pirmasis simbolis privalo būti unikalus
- Pavyzdžiai:

$A \rightarrow aB \mid bAb \mid Bb; B \rightarrow cB \mid d$

$A \rightarrow aB \mid BAb; B \rightarrow aB \mid b$

Kairysis veiksnys

- Kairysis veiksnys (left factoring) gali išspresti problemą

Pakeiskite

**<variable> → identifier | identifier
[<expression>]**

taip

**<variable> → identifier <new>
<new> → ε | [<expression>]**

arba taip

<variable> → identifier [[<expression>]]

Iš apačios į viršų gramatinė analizė

Pavyzdys

- Turime

$\langle E \rangle \rightarrow E + T \mid T$

$\langle T \rangle \rightarrow T * F \mid F$

$\langle F \rangle \rightarrow (E) \mid id$

$E \Rightarrow \underline{E + T}$
 $\Rightarrow E + \underline{T * F}$
 $\Rightarrow E + T * \underline{id}$
 $\Rightarrow E + \underline{F} * id$
 $\Rightarrow E + \underline{id} * id$
 $\Rightarrow \underline{T} + id * id$
 $\Rightarrow \underline{F} + id * id$
 $\Rightarrow \underline{id} + id * id$

Problema

- Duotai dešinei sakinio formai α reikia nustatyti, kokia turi būti α sumažinta eilutė, kuri yra taisyklės kairė pusė, pažymėta LHS, norint gauti ankstesnę sakinio formą dešiniajame išvedime
- Atvirkščias dešiniajam išvedimui
- Bendriausi algoritmai – LR šeima

Iš apačios į viršų: pavyzdys

- $S \rightarrow aAc$
- $A \rightarrow aA | b$
- $S \Rightarrow aAc \Rightarrow aaAc \Rightarrow aabc$
- Pradedam nuo aabc

Dešinysis 352 išvedimas

$\langle \text{Integer} \rangle \Rightarrow \underline{\langle \text{Integer} \rangle \langle \text{Digit} \rangle}$
 $\Rightarrow \langle \text{Integer} \rangle \underline{2}$
 $\Rightarrow \underline{\langle \text{Integer} \rangle \langle \text{Digit} \rangle} \underline{2}$
 $\Rightarrow \langle \text{Integer} \rangle \underline{5} \underline{2}$
 $\Rightarrow \underline{\langle \text{Digit} \rangle} \underline{5} \underline{2}$
 $\Rightarrow \underline{\underline{3}} \underline{5} \underline{2}$

Apibrėžimai

- β - dešinės sakinio formos (**right sentential form**) $\gamma = \alpha\beta w$ rankenėlė (**handle**)
tada ir tik tada, kai $S =>^*_{rm} \alpha Aw =>_{rm} \alpha\beta w$
- β - dešinės sakinio formos γ *frazė*
tada ir tik tada, kai $S =>^* \gamma = \alpha_1 A \alpha_2 =>^+ \alpha_1 \beta \alpha_2$
- β - dešinės sakinio formos γ *paprasta frazė* (**simple phrase**)
tada ir tik tada, kai $S =>^* \gamma = \alpha_1 A \alpha_2 => \alpha_1 \beta \alpha_2$

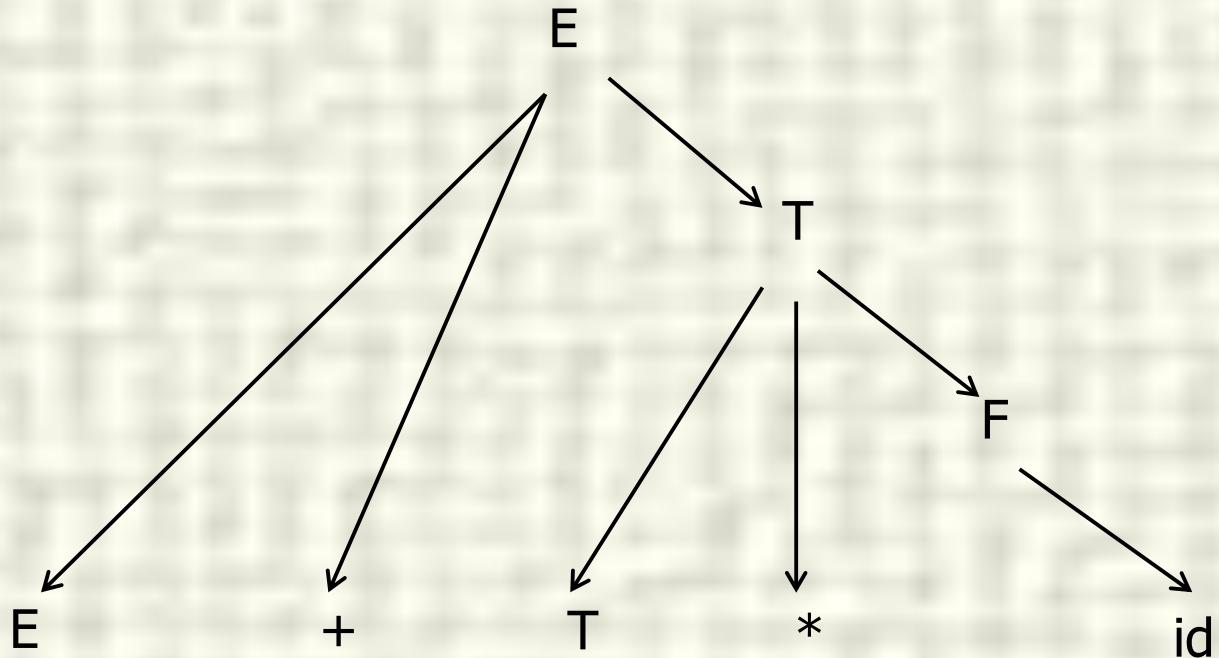
Paaiškinimas

- Frazė yra tam tikro gramatinio medžio visų lapų eilutė
- Paprasta frazė yra tiesiog frazė, kuri gauta, panaudojant tik vieną išvedimo žingsnį iš šaknies neterminalinio mazgo

Gramatika

- $E \rightarrow E + T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow (E) \mid id$

Gramatinis medis $E + T * id$



Gaunamas sakiny s $E + T * id$. 3 vidiniai mazgai – 3 fraz \dot{e} , bet tik viena paprasta fraz \dot{e} (id – ranken \dot{e} l \dot{e})

Rankenėlės

- Dešinės sakinio formos rankenėlė yra jos kairiausioji paprasta frazė
- Duotam gramatiniam medžiui lengva surasti rankenėlę
- Gramatinis nagrinėjimas – tai rankenelių suradimas

Poslinkio ir sumažinimo algoritmai

- Iš apačios į viršų – postūmio-sumažinimo algoritmai (**shift-reduce**)
- Sudėtinė dalis – dëklas
- Sumažinimas – tai veiksmas, pakeičiant parserio dëklo viršuje rankenėlę atitinkama LHS
- Poslinkis – tai veiksmas, perkeliant kitą žetona į parserio dëklo viršų
- Kiekvienas parseris yra stumk žemyn automatas (**pushdown automaton**)

LR parserių privalumai

- Sukūrė Donald Knuth, 1965. Kanoninis LR
- Programa ir lentelė
- Jie gali dirbti su visų programavimo kalbų gramatikomis.
- Jie dirba platesnėje gramatikų klasėje, nei kiti iš apačios į viršų algoritmai, bet lygiai taip pat efektyvūs, kaip ir kiti algoritmai
- Aptinka klaidas taip greitai, kaip tik įmanoma
- LR gramatikų klasė yra gramatikų, nagrinėjamų LL parseriais, super aibė
- Sunkumas – lentelės sudarymas

LR parseriai

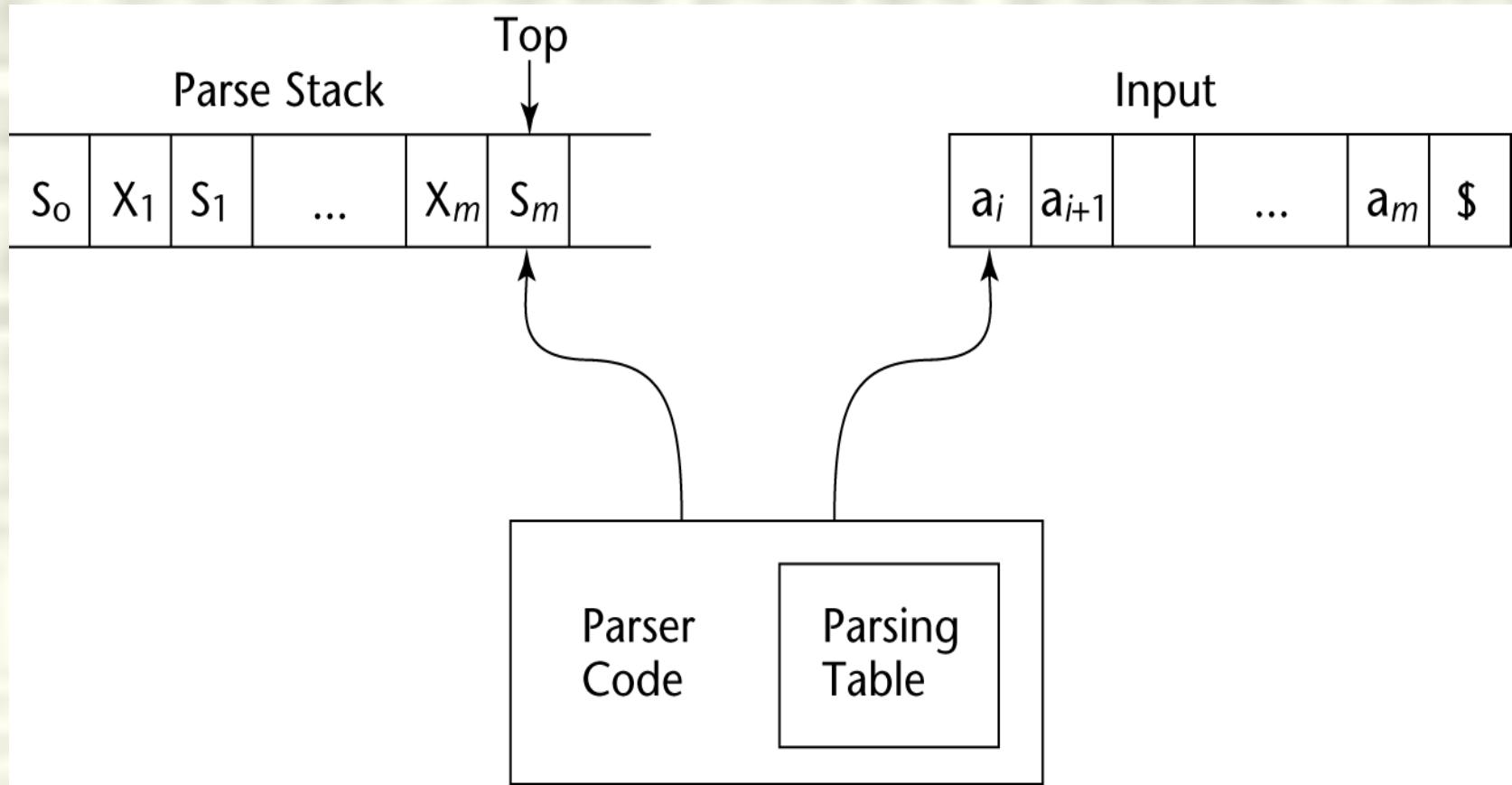
- LR parserius būtina sukonstruoti, naudojant priemonę
- Knuth'o įžvalga: iš apačios į viršų parseris gali naudoti visą parserio istoriją, norėdamas padaryti sprendimą
 - Nedaug parserio situacijų, kurios gali atsitikti, jas galima saugoti parserio dėkle

LR konfigūracija

- LR konfigūracija saugo LR parserio būseną

$(S_0X_1S_1X_2S_2\dots X_mS_m, a_ia_i+1\dots a_n\$)$

LR parserio struktūra



Lenteliniai parseriai

- LR parseriai yra lenteliniai su dviem lentelėmis: ACTION ir GOTO
 - ACTION lentelė nurodo parserio veiksmą, kai duota parserio būsena ir naujas žetonas
 - Eilutės yra būsenų vardai; stulpeliai yra terminalai
 - GOTO lentelė nurodo, kurią būseną padëti į parserio déklo viršų, kai atliktas sumažinimo veiksmas
 - Eilutės yra būsenų vardai; stulpeliai yra neterminalai

Parserio veiksmai – 1

- Pradinė konfigūracija: $(S_0, a_1 \dots a_n \$)$
- Parserio veiksmai:
 - Jei $\text{ACTION}[S_m, a_i] = \text{Shift } S$, nauja konfigūracija:
 $(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m a_i S, a_{i+1} \dots a_n \$)$
 - Jei $\text{ACTION}[S_m, a_i] = \text{Reduce } A \rightarrow \beta$ ir $S = \text{GOTO}[S_{m-r}, A]$, čia $r = \beta$ ilgis, nauja konfigūracija:
 $(S_0 X_1 S_1 X_2 S_2 \dots X_{m-r} S_{m-r} A S, a_i a_{i+1} \dots a_n \$)$

Parserio veiksmai – 1a

- Poslinkiui, naujas įvedimo simbolis talpinamas į dėklą kartu su būsenos simboliu, kuris yra poslinkio operacijos dalis iš ACTION lentelės
- Sumažinimui, rankenėlė šalinama iš dėklo karto su jos būsenos simboliais. Stumiamas taisyklės LHS, stumiamas būsenos simbolis iš GOTO lentelės, naudojant būsenos simbolį, esantį žemiau naujosios LHS dėkle ir naujos taisyklės LHS kaip eilutę ir stulpelis į GOTO lentelę

Parserio veiksmai - 2

- If $\text{ACTION}[S_m, a_i] = \text{Accept}$, nagrinėjimas yra baigtas ir klaidų nerasta.
 - If $\text{ACTION}[S_m, a_i] = \text{Error}$, parseris kviečia klaidų apdorojimo procedūrą.
1. $E \rightarrow E + T$
 2. $E \rightarrow T$
 3. $T \rightarrow T * F$
 4. $T \rightarrow F$
 5. $F \rightarrow (E) \mid \text{id}$
 6. $F \rightarrow \text{id}$

LR parserio lentelė aritmetikai

State	Action							Goto		
	id	+	*	()	\$	E	T	F	
0	S5		S4				1	2	3	
1		S6				accept				
2		R2	S7		R2	R2				
3		R4	R4		R4	R4				
4	S5			S4			8	2	3	
5		R6	R6		R6	R6				
6	S5			S4				9	3	
7	S5			S4					10	
8		S6			S11					
9		R1	S7		R1	R1				
10		R3	R3		R3	R3				
11		R5	R5		R5	R5				

Parserio lentelė

- Parserio lentelė gali būti sugeneruota duotai gramatikai, naudojant priemonę **yacc** arba **bison**

Santrauka

- Sintaksinė analizė skirtoma į dvi dalis: leksikos analizatorius ir sintaksės analizatorius
- Leksikos analizatorius skaido programą į mažas dalis ir formuoja žetonus
- Sintaksės analizatorius aptinka klaidas ir sukuria gramatinį medį
- Rekursinis nusileidimo parseris yra LL parseris, kuris remiasi EBNF
- Iš apačios į viršų parserių užduotis: surasti esamos sakinio formos poaibį
- Poslinkio ir sumažinimo LR parserių šeima yra bendriausias iš apačios į viršų parserių įgyvendinimo būdas

P175B124

Funkcinės programavimo kalbos

Sometimes, the elegant implementation is a function. Not a method. Not a class. Not a framework. Just a function.

—John Carmack

Praėjusios paskaitos kartojimas

1. Kokios yra 3 priežastys, kodėl sintaksės analizatoriai remiasi BNF?
2. Paaiškinkite 3 priežastis, kodėl leksinė analizė yra atskirta nuo sintaksinės analizės.
3. Kas yra leksinis analizatorius?
4. Išvardinkite leksinio analizatoriaus sudarymo būdus.
5. Kas yra būsenų diagrama leksiniame analizeriuje?
6. Kokie yra du skirtinių sintaksinės analizės tikslai?
7. Paaiškinkite dvi gramatikos charakteristikas, kurios neleidžia gramatikai būti iš viršaus žemyn gramatinio analizatoriaus baze.
8. Paaiškinkite, kodėl kompiiliatoriuose naudojami analizės algoritmai, kurie dirba tik su gramatikų poaibiu.
9. Pristatykite tris LR gramatinių analizatorių privalumus.
10. Koks yra ryšys tarp LR ir LL gramatinių analizatorių?

Temos klausimai

- Matematinės funkcijos.
- Pagrindai
- LISP
- Scheme
- COMMON LISP
- ML
- Haskell
- F#
- Funkcinių programavimo kalbų (FPK) taikymai
- Liepiamujų ir funkcių kalbų palyginimas

FPK bazė

- Liepiamujų kalbų projektas tiesiogiai remiasi
von Neumann architektūra
 - Pirmasis rūpestis yra efektyvumas, bet ne tinkamumas programavimui
- Funkcinių kalbų projektas remiasi matematinėmis funkcijomis
 - Solidi teorinė bazė, kuri artimesnė vartotojui, bet neturi nieko bendro su architektūra

FPK privalumai

- FPK savybės leidžia tinkamai vertinti iššūkius, kurios sukelia pramonės poslinkis link daugiaoprocesorinės architektūros
- FPK žinojimas gali stipriai pagerinti kodo rašymą kitose srityse. Jūsų kode bus daugiau nuorodų skaidrumo.
- “Mock roles, not objects” (TDD, paper, 2004) – keisti mąstymą nuo pokyčių apie objektų būsenas į jų bendravimą su kitais objektais.
- Kai kas gali tvirtinti, kad funkcinis programavimas ir objektinė orientacija – tai tik vienas kito atspindys, kompiuterinė yin ir yang forma
- John Backus paskaita 1978 m. (lengviau skaitomos, patikimesnės, teisingesnės). Funkcinė kalba FP.

Liepiamųjų trūkumas

- Tipizuotos funkcinės kalbos: ML, Haskell, OCaml, F#
- Liepiamosios per kintamuosius turi būseną. Didelė programa – problema. FPK – kintamųjų, būsenos nėra
- Lisp prasidėjo kaip funkcinė, vėliau įgijo liepiamųjų bruožų – padidėjo efektyvumas
- Tipizuotos funkcinės kalbos: ML, Haskell, OCaml, F# praplėtė taikymo sritis

Matematinės funkcijos

Apibrėžimas

- Matematinė funkcija – tai žymėjimas (**mapping**) iš vienos aibės narių, vadinamų srities (**domain**) aibe, į kitą aibę, vadinamą rėžio (**range**) aibe
- Žymėjimas išreiškiamas išraiška ar lentele
- Pirmoji charakteristika – jvertinimo seką valdo rekursija ir sąlyginės išraiškos, bet ne priskyrimai ir iteracija
- Antra – nėra šalutinių efektų, kurie susiję su kintamaisiais, modeliuojančiais vietas atmintyje. Visada gausime tą pačią reikšmę tiems patiemis duomenims.
- Liepiamajame – kintamieji išreiškia funkcijos būseną ir reikšmė gali priklausyti nuo globalių kintamujų

Lambda išraiška

- Lambda išraiška nurodo parametrus ir funkcijos žymėjimą tokia forma (Alonzo Church, 1941)

$\lambda(x) \ x * x * x$

Pvz.: $\text{cube } (x) \equiv x * x * x$

- Lambda išraiškos aprašo bevardes (**nameless**) funkcijas. (**Lambda calculus**)
- Lambda išraiška taikoma parametrams, talpinant parametrus į išraišką

pvz.: $(\lambda(x) \ x * x * x)(2)$

gaunam 8

Aukštesnės eilės funkcija

- Aukštesnės eilės (**higher-order**) funkcija ar *funkcinė forma* (*functional form*) – tai forma, kurioje parametrai yra funkcijos arba rezultatas yra funkcija, arba abu
- Viena bendra funkcių formų rūšis yra funkcių kompozicija

Funkcijų kompozicija

- Funkcinė forma ima 2 funkcijas kaip parametrus, kurios rezultatas yra funkcija, sudaryta iš pirmosios funkcijos, taikomos antrosios funkcijos rezultatui

Forma: $h \equiv f \circ g$

kuri reiškia $h(x) \equiv f(g(x))$

Dėl $f(x) \equiv x + 2$ ir $g(x) \equiv 3 * x$,

$h \equiv f \circ g$ sukuria $(3 * x) + 2$

Taikyti visiems

Funkcinė forma, kurios parametras vienintelė funkcija, sukurianti reikšmių sąrašą, gaunamą, taikant funkciją kiekvienam sąrašo elementui

Žymima: α

Tegul $h(x) \equiv x * x$

$\alpha(h, (2, 3, 4))$ sukuria (4, 9, 16)

Yra daugiau funkinių formų

Pagrindai

Procesas

- FPK tikslas yra atkartoti kaip galima geriau matematines funkcijas
- FPK bazinis skaičiavimo procesas yra visiškai skirtinas nuo liepiamosios kalbos
 - Liepiamojoje kalboje atliekamos operacijos ir rezultatas saugomas kintamuosiuose vėlesniams naudojimui
 - Kintamujų valdymas yra pastovus rūpestis ir sudėtingumo šaltinis liepiamajame programavime
- FPK kintamieji nėra būtini, kaip ir matematikoje

Ypatybės

- **Grynoji FPK** nenaudoja priskyrimų ir kintamųjų
- Be kintamųjų neįmanomos iteracinių konstrukcijos, nes jas valdo kintamieji
- Kartojimą turi apibrėžti rekursija
- Be kintamųjų nėra būsenos

Koncepcijos

- *Nuorodų skaidrumas (referential transparency)* – išraiška yra skaidri pagal nuorodas, jei ji gali būti pakeista jos reikšme ir programos elgsena nepasikeis
- Rekursija lėtina vykdymą
- *Galinė (tail) rekursija* – rekursinių funkcijų rašymas, kad jos automatiškai galėtų būti konvertuojamos į iteracines (rekursinis kreipinys yra paskutinė išraiška funkcijos kamiene)

Liepiamujų apribojimai

- *Funkcijos* riboja tipus kintamujų, kurių reikšmės gali būti grąžintos iš funkcijos.
- Riboja funkcių formų rūšis
- Negali grąžinti funkcijų
- Šalutiniai efektai

Skaičiuoklės

- Funkcinės kalbos labai tinka specifikacijų, kurios gali būti vykdomos, rašymui
- Skaičiuoklėje lentelės reikšmė apibrėžiama per kitų lastelių reikšmes – dėmesys tam, kas turi būti suskaičiuota, bet ne kaip.
- Nekreipiame dėmesio, kuria tvarka turi būti suskaičiuotos lastelių reikšmės. Pasitikime.
- Nesakome skaičiuoklei, kad išskirtų atminties
- Laštelėje reikšmę apibrėžiame išraiška, bet ne komandų seka.

SQL

- Kita beveik funkcinė kalba yra SQL
- SQL rašoma išraiška, kuri nurodo, ką reikia suskaičiuoti, bet ne kaip. Atlikimo tvarka taip pat nenurodoma
- SQL įgyvendinimai dažnai atlieka optimizavimus dėl išraiškų tvarkos įvertinimo
- Nėra priskyrimo

QuickSort Haskell

- Skliaustai naudojami grupavimui, bet ne funkcijoms žymėti. Funkcija rašoma $f\ x$, bet ne $f(x)$. Gali būti užrašyta $(f\ x)$

```
quicksort :: Ord a => [a] -> [a]
```

```
quicksort [] = []
```

```
quicksort (p:xs) = (quicksort lesser) ++ [p] ++  
(quicksort greater)
```

where

```
lesser = filter (< p) xs
```

```
greater = filter (>= p) xs
```

Sintaksė

- Skliaustai kairėje žymi funkcijos argumentų rinkinius
- Du filtrų kreipiniai naudoja sukurtus predikatus, pirmasis iš jų atrenka elementus xs , mažesnius už centrinį elementą p
- Funkcijas apibrėžiame kaip rinkinių atitikimo sakinius: rinkinys $(p:xs)$ žymi netuščią sąrašą su pradžios elementu p ir uodega xs
- Pirmasis sakinys yra funkcijos signatūra, sakanti, kad ji transformuoja elementų sąrašą nesvarbaus tipo į to paties tipo elementų sąrašą, tipas yra tipų klasės Ord atstovas

Aiškumas ir trumpumas

- Funkcinės programos yra glaustesnės, trumpesnės 2-10 kartų, negu atitinkamos liepiamosios
- Rikiavimas gali būti užrašytas trumpiau, naudojant sąrašų operacijas:

```
qsort (p:xs) = qsort [x | x <- xs, x < p] ++ [p]
++ qsort [x | x <- xs, x >= p]
```

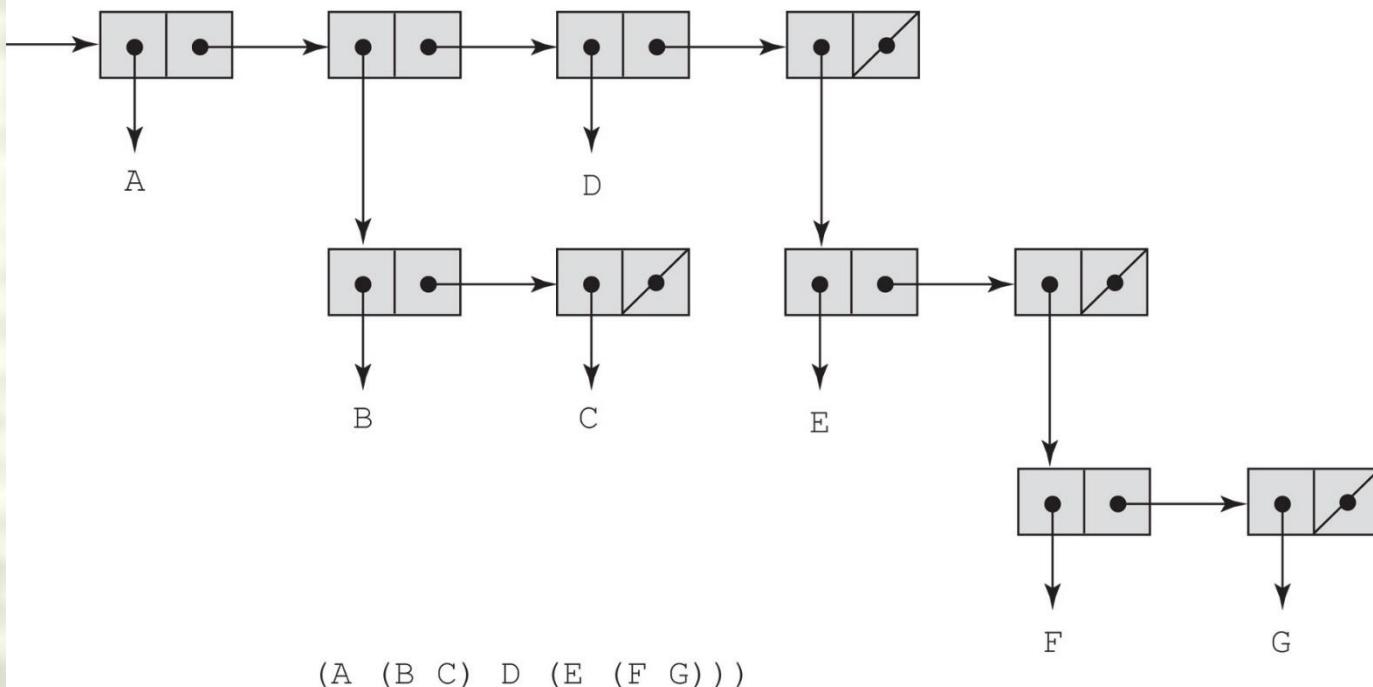
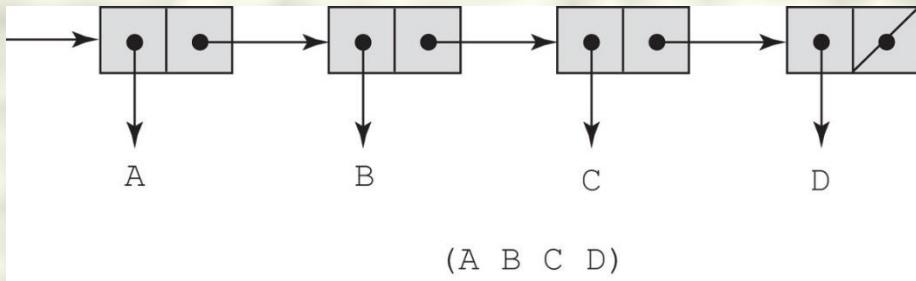
- Pirmoji išraiškos dalis reiškia, kad x, paimtam iš xs, tokiam, kad $x < p$, sudarykite sąrašą ir kvieskite rekursiškai qsort. Panašiai, antroji išraiškos dalis.

LISP

Duomenų tipai

- *Duomenų objektų tipai*: pradžioje buvo tik atomai ir sąrašai
- *Sąrašų forma*: rinkinys su skliaustais iš sąrašų ir atomų
pvz.: (A B (C D) E)
- Iš pradžių LISP buvo kalba be tipų
- LISP sąrašai viduje saugomi kaip vienkrypčiai susietieji sąrašai su dviem rodyklėmis

Sqrašai



Interpretavimas

- Lambda žymėjimas yra naudojamas nurodyti funkcijoms ir jų apibrėžimams. Funkcijų taikymai ir duomenys turi tą pačią formą
pvz.: jei sąrašas (A B C) interpretuoojamas kaip duomenys, jis yra paprastas atomų A, B ir C sąrašas.
Jei jis interpretuoojamas kaip funkcijos taikymas, jis reiškia, kad funkcija vardu A yra taikoma dviem parametroms B ir C
- Pirmasis LISP interpretatorius pasirodė tik kaip šio žymėjimo skaičiavimo galimybių universalumo demonstravimas
- EVAL funkcija kitų funkcijų vertinimui
- Dinaminis akiratis (scoping) – netycia

Scheme

Kilme

- 1975 m. LISP dialektas, suprojektuotas būti švaresnis, modernesnis ir paprastesnis, nei esami kiti LISP dialektai
- Naudoja tik statinį akiratį, betipė kalba
- Funkcijos yra pirmos klasės esybės (**entity**)
 - Jos gali būti išraiškų reikšmės ir sąrašų elementai
 - Jos gali būti priskirtos kintamiesiems ir perduotos kaip parametrai

The Scheme Interpreter

- In interactive mode, the Scheme interpreter is an infinite read-evaluate-print loop (REPL)
 - This form of interpreter is also used by Python and Ruby
- Expressions are interpreted by the function EVAL
- Literals evaluate to themselves

Vykdomas

- Parametru išraiškos skaičiuojami be nustatytos eilės
- Parametru reikšmės yra įstatomos į funkcijos kamieną
- Vykdomas funkcijos kamienas
- Funkcijos kamieno paskutinės išraiškos reikšmė yra funkcijos reikšmė
- Komentarai rašomi už kabliataškio

Primityvios funkcijos

- Aritmetinės: +, -, *, /, ABS, SQRT, REMAINDER, MIN, MAX
pvz.: (+ 5 2) rezultatas 7
- QUOTE — ima vieną parametrą; grąžina parametrą be skaičiavimo
 - QUOTE yra reikalingas, nes Scheme interpretatorius vardu EVAL visada įvertina parametrus pagal funkciją, prieš taikydamas juos funkcijai. QUOTE naudojamas norint išvengti parametru skaičiavimo, kada tai nėra tinkama
 - QUOTE gali būti trumpinamas, naudojant apostrofą
'(A B) ekvivalentu (QUOTE (A B))

Lambda funkcija

- Lambda išraiškos
 - Forma remiasi λ žymėjimu

pvz.: (LAMBDA (x) (* x x))
- Lambda išraiškos gali būti taikomos – tai bevardės funkcijos

pvz.: ((LAMBDA (x) (* x x)) 7)
x vadinamas susietu (**bound**) kintamuju

Speciali funkcija: DEFINE

Dvi formos:

1. Susieti vardą su išraiškos reikšme

pvz.: (DEFINE pi 3.141593)

panaudojimo pavyzdys: (DEFINE two_pi (* 2 pi))

2. Susieti lambda išraišką su vardu

pvz.: (DEFINE (square x) (* x x))

panaudojimo pavyzdys: (square 5)

- DEFINE skaičiavimo procesas yra skirtinas! Pirmasis parametras niekada neskaičiuojamas. Antrasis parametras skaičiuojamas ir susiejamas su pirmuoju

Išvedimo funkcijos

- Dažniausiai nereikalingos, nes aukščiausio lygmens funkcijos rezultatas visuomet parodomas
- Scheme turi PRINTF, kuri panaši į printf iš C kalbos
- Įvedimas, išvedimas ne FPK dalis

Skaitmeninės tikrinimo funkcijos

- Numeric predicate
- $\#^T$ yra tiesa ir $\#^F$ yra melas
- $=, \neq, >, <, \geq, \leq$
- EVEN?, ODD?, ZERO?, NEGATIVE?
- NOT funkcija invertuoja loginės išraiškos logiką

Sąlygos tikrinimas: IF

- Control flow
- Išrinkimas – speciali forma, IF

(IF predicate then_exp else_exp)

pvz.:

```
(IF (<> count 0)
    (/ sum count)
    0)
```

Sąlygos tikrinimas: COND

- Daugybinis išrinkimas – speciali forma, COND bendra forma:
$$(\text{COND}$$

$$(\textit{predicate } \textit{expr} \text{ } \{ \textit{expr} \})$$

$$(\textit{predicate } \textit{expr} \text{ } \{ \textit{expr} \})$$

$$\dots$$

$$(\textit{predicate } \textit{expr} \text{ } \{ \textit{expr} \})$$

$$(\text{ELSE } \textit{expr} \text{ } \{ \textit{expr} \}))$$
- Gražina pirmos poros, kurio predikato reikšmė yra tiesa, paskutinės išraiškos reikšmę

COND pavyzdys

```
(DEFINE (compare x y)
  (COND
    ((> x y) "x is greater than y")
    ((< x y) "y is greater than x")
    (ELSE "x and y are equal"))
  )
)
```

COND pavyzdys 2

```
(DEFINE (leap? year)
  (COND
    ((ZERO? (MODULO year 400)) #T)
    ((ZERO? (MODULO year 100)) #F)
    (ELSE (ZERO? (MODULO year 4))))
))
```

Sąrašo funkcijos CAR ir CDR

- CAR ima sąrašo parametru; grąžina pirmą sąrašo elementą

pvz.: (CAR ' (A B C)) grąžina A

(CAR ' ((A B) C D)) grąžina (A B)

- CDR ima sąrašo parametru; grąžina sąrašą be pirmo elemento

pvz.: (CDR ' (A B C)) grąžina (B C)

(CDR ' ((A B) C D)) grąžina (C D)

Sąrašo funkcijos CONS ir LIST

- CONS turi du parametrus, pirmasis iš jų gali būti sąrašas arba atomas, o antrasis - sąrašas; grąžina naują sąrašą, kur pirmasis parametras tampa pirmuoju sąrašo elementu, o antrasis parametras – likusi sąrašo dalis
- LIST ima bet kokį parametru kiekį ir grąžina sąrašą, kur parametrai tampa elementais

Sąrašo funkcijos

(CAR '((A B) C D)) returns (A B)

(CAR 'A) is an error

(CDR '((A B) C D)) returns (C D)

(CDR 'A) is an error

(CDR '(A)) returns ()

(CONS '() '(A B)) returns (() A B)

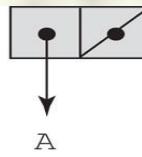
(CONS '(A B) '(C D)) returns ((A B) C D)

(CONS 'A 'B) returns (A . B) (du atomai)

Sąrašo funkcijos grafiškai

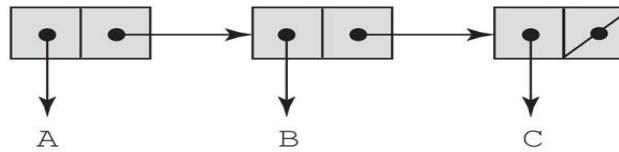
(CONS 'A '())

A



(CONS 'A '(B C))

A



B

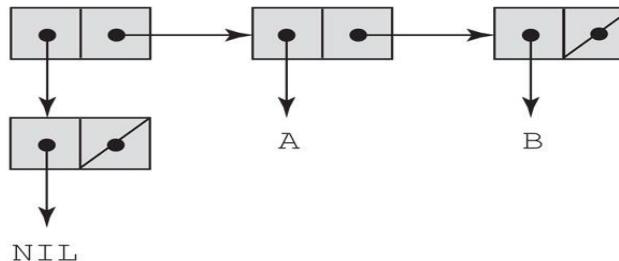
C

(CONS '() '(A B))

(() A B)

A

B



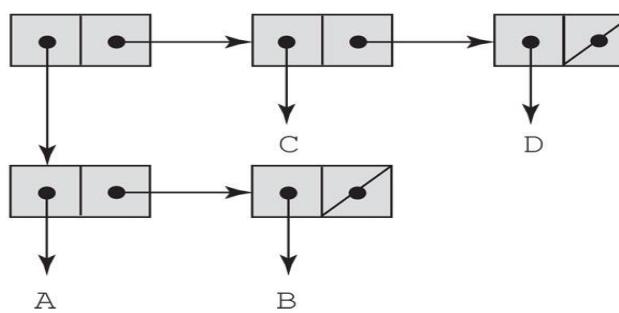
NIL

(CONS '(A B) '(C D))

((A B) C D)

C

D



A

B

Predikato funkcija EQ?

- EQ? Turi du simbolinius parametrus; ji grąžina $\#T$ jei abu parametrai yra atomai ir tie patys; kitu atveju - $\#F$

pvz.: $(\text{EQ? } 'A 'A)$ grąžina $\#T$

$(\text{EQ? } 'A 'B)$ grąžina $\#F$

- Jei EQ? iškviečiama su sąrašo parametrais, rezultatas nepatikimas

- Taip pat EQ? neveikia su skaitmeniniais atomais

$(\text{EQ? } 'A ' (A B))$ yields $\#F$

$(\text{EQ? } ' (A B) ' (A B))$ yields $\#T \text{ or } \#F$

$(\text{EQ? } 3.4 (+ 3 0.4))$ yields $\#T \text{ or } \#F$

Predikato funkcija EQV?

- EQV? Kaip ir EQ?, tačiau dirba su simboliniais ir skaitmeniniaisiais atomais; lygina reikšmes, bet ne rodykles

(EQV? 3 3) bus #T

(EQV? 'A 3) bus #F

(EQV? 3.4 (+ 3 0.4)) bus #T

(EQV? 3.0 3) bus #F (realusis ir sveikasis yra skirtinti)

Predikatai LIST? ir NULL?

- LIST? ima vieną parametą; grąžina #T , jei parametras yra sąrašas; kitaip - #F
- NULL? ima vieną parametą; grąžina #T , jei parametras yra tuščias sąrašas; kitaip - #F
 - NULL? grąžina #T , jei parametras yra ()

Funkcijų pavyzdžiai 1

- `member` imai atomą ir paprastą sąrašą; grąžina `#T`, jei atomas yra sąraše; `#F` kitu atveju

```
DEFINE (member atm lis)
```

```
(COND
```

```
((NULL? lis) #F)
```

```
((EQ? atm (CAR lis)) #T)
```

```
((ELSE (member atm (CDR lis))))
```

```
))
```

Funkcijų pavyzdžiai 2

- `equalsimp` ima du sąrašus kaip parametrus; grąžina `#T`, jei du sąrašai sutampa; `#F` kitaip

```
(DEFINE (equalsimp lis1 lis2)
  (COND
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) #F)
    ((EQ? (CAR lis1) (CAR lis2))
     (equalsimp(CDR lis1)(CDR
lis2)))
    (ELSE #F)
  )))
```

Funkcijų pavyzdžiai 3

- `equal` ima du bendrus sąrašus kaip parametrus; grąžina `#T`, jei du sąrašai sutampa; `#F` kitaip
- ```
(DEFINE (equal lis1 lis2)
 (COND
 ((NOT (LIST? lis1))(EQ? lis1 lis2))
 ((NOT (LIST? lis2)) #F)
 ((NULL? lis1) (NULL? lis2)))
 ((NULL? lis2) #F)
 ((equal (CAR lis1) (CAR lis2))
 (equal (CDR lis1) (CDR lis2))))
 (ELSE #F)
))
```

# Funkcijų pavyzdžiai 4

---

- `append` imai du sąrašus kaip parametrus; grąžina prie pirmojo sąrašo pridėtą antrajį sąrašą

```
(DEFINE (append lis1 lis2)
 (COND
 ((NULL? lis1) lis2)
 (ELSE (CONS (CAR lis1)
 (append (CDR lis1)
 lis2))))
))
```

# Funkcija LET

---

- Išraiškų dalių išskyrimui. Bendra forma:  
(LET (  
    (name\_1 expression\_1)  
    (name\_2 expression\_2)  
    ...  
    (name\_n expression\_n))  
    kamienas  
)  
• Įvertinti visas išraiškas, tuomet susieti reikšmes su vardais; įvertinti kamieną

# Pavyzdys LET

---

```
(DEFINE (quadratic_roots a b c)
 (LET (
 (root_part_over_2a
 (/ (SQRT (- (* b b) (* 4 a c))))(* 2 a)))
 (minus_b_over_2a (/ (- 0 b) (* 2 a))))
 (DISPLAY (+ minus_b_over_2a root_part_over_2a))
 (NEWLINE)
 (DISPLAY (- minus_b_over_2a root_part_over_2a)))
))
```

# Galinė rekursija

---

- Apibrėžimas: funkcija yra *galinė rekursija*, jei jos rekursinis kreipinys yra paskutinė funkcijos operacija
- Galinės rekursijos funkcija gali būti automatiškai konvertuota kompiliatoriumi, kad naudotų iteraciją, kuri yra žymiai greitesnė
- Scheme kalbos apibrėžimas reikalauja, kad visos galinės rekursijos funkcijos būtų pakeistos iteracinėmis

# Galinė rekursija: pavyzdys

---

- Funkcijos perrašymas, kad būtų su galine rekursija

Originali: (DEFINE (factorial n)

(IF (= n 0)

1

(\* n (factorial (- n 1))))

))

Galinė rekursija: (DEFINE (facthelp n factpart)

(IF (= n 0)

factpart

facthelp((- n 1) (\* n factpart)))

))

(DEFINE (factorial n)

(facthelp n 1))

# Funkcinės formos

---

- Kompozicija
  - Ankstesnieji pavyzdžiai ją naudojo
  - (CDR (CDR ' (A B C))) – ką grąžina ?
- Taikoma visiems – viena forma Scheme yra mapcar

- Taiko duotą funkciją visiems duoto sąrašo elementams;

```
(DEFINE (mapcar fun lis)
 (COND
 ((NULL? lis) ())
 (ELSE (CONS (fun (CAR lis))
 (mapcar fun (CDR lis)))))))
)
```

# Funkcijos, kuriančios kodą

---

- Scheme galima apibrėžti funkciją, kuri kuria Scheme kodą ir reikalauja jo interpretavimo
- Tai yra įmanoma, nes interpretatorių vartotojas gali iškvesti funkcija EVAL

# Skaičių sąrašo pridėjimas

---

```
((DEFINE (adder lis)
 (COND
 ((NULL? lis) 0)
 (ELSE (EVAL (CONS '+ lis))))
))
```

- Parametras – pridedamas skaičių sąrašas; adder įterpia + operatorių ir įvertina gautą sąrašą
  - Naudoja CONS atomo + įterpimui į skaičių sąrašą
  - + turi būti su apostrofuu, kad neįvyktų skaičiavimas
  - Perduodamas naujas sąrašas EVAL skaičiavimui

# **Kitos funkcinės kalbos**

# Common LISP

---

- Įvairių populiairių LISP dialektų daugumos savybių kombinacija 1980 m. pradžioje
- Didelė ir sudėtinga kalba – Scheme priešingybė
- Savybės:
  - įrašai
  - masyvai
  - kompleksiniai skaičiai
  - eilutės
  - galingos I/O galimybės
  - paketai su prieigos valdymu
  - iteraciniai valdymo sakiniai

**ML**

# Apibrėžimas

---

- Statinio akiračio funkcinė kalba (1997) su sintakse artimesne Pascal, nei LISP
- Naudoja tipų paskelbimą, bet taiko ir tipų išvedimą dėl nenurodyto tipo kintamuju
- Stipri tipų kontrolė (Scheme yra pagrindinai be tipų) ir neturi numatytojos tipų konversijos
- Turi išimčių valdymą ir modulio sąvoką ADT įgyvendinimui
- Turi sąrašus ir operacijas su sąrašais

# Funkcijos

---

- Funkcijos paskelbimo forma:

*fun varda (parametrai) = kamienas;*

pVz.: *fun cube (x : int) = x \* x \* x;*

- Gr̄žimo reikšmei gali būti nustatytas tipas

*fun cube (x) : int = x \* x \* x;*

- Jei tipas nenurodytas, tai būtų *int* (skaitmeninėms reikšmėms)

- Vartotojo apibréžtos užklotos funkcijos nėra galimos, jei norētume *cube* funkcijos su realiaisiais parametrais, reiktų funkcijai suteikti kitą vardą

# Išrinkimas ir rinkiniai

---

- Išrinkimas
  - if *išraiška* then *then\_išraiška*
  - else *else\_išraiška*

čia pirmoji išraiška turi būti loginė
- Rinkinių atitikimas (*pattern matching*)  
naudojamas, kad suteiktų galimybę funkcijai veikti su skirtinomis parametru formomis
- fun fact(0) = 1  
| fact(1) = 1  
| fact(n : int) : int =  
    n \* fact(n - 1)

# Sąrašai

---

- Sąrašai

Bevardės konstantos vardijamos laužtiniuose skliaustuose

[3, 5, 7]

[] tuščias sąrašas

Scheme CONS dvejetainis infix operatorius ::

4 :: [3, 5, 7], kurio rezultatas [4, 3, 5, 7]

Scheme CAR – vienetinis operatorius hd

Scheme CDR – vienetinis operatorius tl

```
fun length([]) = 0
| length(h :: t) = 1 + length(t);
```

```
fun append([], lis2) = lis2
| append(h :: t, lis2) = h :: append(t, lis2);
```

- Dar turi išvardijimo tipus, masyvus, kortežus (**tuples**)

# Susiejimas

---

- **val** sakinys susieja (**bound**)vardą su reikšme (panašiai kaip DEFINE Scheme)

```
val distance = time * speed;
```

- kaip ir DEFINE atveju, val nėra tas pats, kaip priskyrimas liepiamomoje kalboje

- **val** sakiniai dažnai naudojami **let** konstrukcijoje

**let**

```
 val radius = 2.7
```

```
 val pi = 3.14159
```

**in**

```
 pi * radius * radius
```

**end;**

# Įtaka

---

- Kalbos tyrinėtojai
- Haskell
- Caml at INRIA (Prancūzijos institutas)
- OCaml – objektinė
- F# Microsoft – OCaml pagrindas
- F# – pirmos klasės kalba – bendrauja su kitomis .NET platformos kalbomis

# Haskell

# Apibrėžimas

---

- Panaši į ML (sintaksė, statinis akiratis, stipriai tipizuota, tipų išvedimas, rinkinių atitikimas) (1999)
- Skiriiasi nuo ML (ir daugumos kitų funkinių kalbų) tuo, kad tai grynoji funkcinė kalba (nėra kintamuju, priskyrimo sakinių ir jokių šalutinių efektų)

Sintaksės skirtumai nuo ML

fact 0 = 1

fact n = n \* fact (n - 1)

fib 0 = 1

fib 1 = 1

fib (n + 2) = fib (n + 1) + fib n

# Skirtumai nuo ML

---

1. Nenaudojamas rezervuotas žodis fun pažymėti funkcijoms
2. Skliaustai nenaudojami formaliuju parametru atskyrimui
3. Alternatyvūs funkcijų aprašai turi tokią pat išvaizdą
4. Funkcijos gali būti polimorfinės (tinka suderinamiems tipams)

# Funkcijų apibrėžimas

---

- Pridėtos apsaugos (**guard**)

```
fact n
| n == 0 = 1
| n > 0 = n * fact(n - 1)
```

```
sub n
| n < 10 = 0
| n > 100 = 2
| otherwise = 1
```

```
square x = x * x
```

- Veikia bet kuriam skaitmeniniam x tipui

# Sąrašai

---

- Sąrašo užrašymas: elementai laužtiniuose skliaustuose  
pvz.: `directions = ["north", "south", "east", "west"]`
- Ilgis: #  
pvz.: `#directions` yra 4
- Aritmetinės sekos su .. operatoriumi  
pvz.: `[2, 4..10]` yra `[2, 4, 6, 8, 10]`
- Apjungimas ++  
pvz.: `[1, 3] ++ [5, 7]` gausime `[1, 3, 5, 7]`
- CONS, CAR, CDR per dvitaškio operatorių (kaip ir Prolog)  
pvz.: `1 : [3, 5, 7]` gausime `[1, 3, 5, 7]`

# Faktorialas

---

```
product [] = 1
```

```
product (a:x) = a * product x
```

```
fact n = product [1..n]
```

# Sąrašų samprata

---

- Aibės užrašymo forma
- 20 pirmųjų sveikujų kvadratų sąrašas: [ n \* n | n  $\leftarrow$  [1..20] ]
- Duoto parametro visi daugikliai:

```
factors n = [i | i \leftarrow [1..n ÷ v^2],
n mod`i == 0]
```

rodo, kad funkcija naudojama kaip  
dvejetainis operatorius

# QuickSort

---

```
sort [] = []
sort (a:x) =
 sort [b | b < x; b <= a]
 ++ [a] ++
 sort [b | b < x; b > a]
```

Iliustruoja glauustumą

# Užvėlintas įvertinimas

---

- Kalba yra *griežta*, jei ji reikalauja, jog visi faktiniai parametrai būtų pilnai įvertinti
- Kalba nėra griežta *griežta*, jei ji neturi šio griežto reikalavimo
- Negriežtos kalbos yra efektyvesnės ir turi daugiau galimybių – *begaliniai sąrašai*
- Užvėlintas įvertinimas (*lazy evaluation*) – skaičiuoja tik tas reikšmės, kurios reikalingos
- Sveikieji skaičiai  
positives = [0..]
- Ar 16 yra kvadratas  
squares = [n \* n | n ← [0..]]  
member squares 16

# Kvadrato nario nustatymas

---

- Nario nustatymo funkcija gali būti parašyta:

```
member [] b = False
```

```
member (a:x) b=(a == b) || member x b
```

- Tačiau ši funkcija veiks tik tuomet, jei parametras yra tikslus kvadratas, jei ne, generacija be pabaigos
- Teisingesnė versija:

```
member2 (m:x) n
```

```
| m < n = member2 x n
```

```
| m == n = True
```

```
| otherwise = False
```

# F#

# Apibrėžimas

---

- Remiasi OCaml, kuris bazuojamas ML ir Haskell
- Pagrindinai funkcinė kalba, bet su liepiamaisiais bruožais ir OP
- Turi pilną aplinką, didelę biblioteką ir bendrauja su kitomis .NET kalbomis
- Turi kortežus (**tuples**), sąrašus, unijas, įrašus, kintančius (**mutable**) ir nekintančius (**immutable**) masyvus
- Turi bendrines sekas, kurių reikšmes gali sukurti generatoriai ir iteracijos

# Sekos

---

**let** x = seq {1..4};; – interaktyvus interpretatorius

- Sekos reikšmių generavimas yra užvėlintas

**let** y = seq {0..10000000};;

Nustato y į [0; 1; 2; 3;...]

- Numatytasis žingsnis yra 1, bet gali būti bet koks

**let** seq1 = seq {1..2..7}

Nustato seq1 į [1; 3; 5; 7]

- Iteratoriai – neužvėlinti sąrašams ir masyvams

**let** cubes = seq {for i in 1..4 -> (i, i \* i \* i)};;

Nustato kubus į [(1, 1); (2, 8); (3, 27); (4, 64)]

# Funkcijos

---

- Jei su vardu, apibrėžiamos su `let`; Jei lambda išraiškos, apibrėžiamos su `fun`

(`fun` a b -> a / b)

- Nėra skirtumo tarp vardo, apibrėžto su `let` ir funkcijos be parametru
- Funkcijos galiojimo sritij nustato postūmis (`indentation`)

```
let f =
 let pi = 3.14159;
 let twoPi = 2.0 * pi;
```

# Funkcijos 2

---

- Jei funkcija rekursinė, jos apibrėžimas turi turėti `rec`
- Vardai funkcijose gali būti iš naujo apibrėžti, taip baigiant galiojimą prieš tai buvusio vardo

```
let x4 x =
 let x = x * x
 let x = x * x
 x; ;
```

Pirmasis `let` funkcijos kamiene sukuria naują `x`; versiją, užsibaigia paramетro galiojimas; Antrasis `let` kamiene kuria kitą `x`, užbaigdamas prieš tai buvusį `x`

# Funkciniai operatoriai

---

## – Srautas (`|>`) (pipeline)

- Dvejetainis operatorius, kuris siunčia savo kairiojo operando reikšmę į paskutinį kreipinio parametą (dešinysis operandas)

```
let myNums = [1; 2; 3; 4; 5]
let evenTimesFive = myNums
 |> List.filter (fun n -> n % 2 = 0)
 |> List.map (fun n -> 5 * n)
```

Grąžinama reikšmė [10; 20]

# Funkciniai operatoriai 2

---

## – Kompozicija (>>)

- Sukuria funkciją, kuri taiko kairijį operandą duotam parametru i ir gautą rezultatą perduoda dešiniajam operandui
- Išraiška  $(f >> g) \times$  ekvivalenti matematinei išraiškai  $g(f(x))$

## – Skrudintos (*curried*) funkcijos – grąžinamas rezultatas yra funkcija

```
let add a b = a + b;;
```

```
let add5 = add 5;;
```

# Įvertinimas

---

- Sukurta ankstesniųjų FPK pagrindu
- Palaiko visas šiuolaikines programavimo metodologijas
- Pirmoji funkcinė kalba, suprojektuota bendravimui su kitomis plačiai naudojamomis kalbomis
- Turi gerą aplinką ir programų biblioteką

# **Praktinē vertē**

# Taikymai

---

- LISP naudojamas dirbtiniams intelektui
  - Žinių atvaizdavimas
  - Apsimokymas
  - Natūralios kalbos apdorojimas
  - Kalbos ir vaizdinių modeliavimas
- Scheme naudoja pradiniam programavimo mokymui kai kuriuose universitetuose

# Praktinės programos 1

---

- Software AG, stambi Vokietijos PĮ kompanija, pardavinėja ekspertinę sistemą Natural Expert, sukurtą funkcinėje kalboje. Vartotojai gerai atsiliepia apie sistemą. Ji susieta su DB.
- Ericsson sukūrė naują funkcinę kalbą Erlang, skirtą telefonams. Ericsson savo kalba džiaugiasi dėl programų trumpumo ir greito kūrimo.
- Amoco perrašė Miranda kalba didelę dalį programos, skirtos naftos rezervuarų modeliavimui. Programa buvo daug trumpesnė, atrado klaidų savo ankstesnėje programoje, galiausiai, ją pavertė į C++ programą

# Praktinės programos 2

---

- Durham universiteto mokslininkai panaudojo Miranda, vėliau Haskell, septynių metų projektui LOLITA, 30 000 kodo eilučių natūralios kalbos apdorojimui.
- Query yra užklausų kalba O2 objektinėje DBVS. O2Query – objektinė užklausų kalba ir funkcinė kalba.
- ICAD Inc pardavinėja CAD sistemą, skirtą mechanikos ir aeronautikos inžinieriams. Sistema naudoja funkcinę kalbą ir užvėlintą įvertinimą, kad neperskaičiuoti projekto dalių, kurios šiuo metu nematomos ekrane. Tai žymiai pagerino greitį.
- Glasgow Haskell kompiliatorius parašytas Haskell: apie 100,000 kodo eilučių.
- Pugs, geriausias perl6 įgyvendinimas, parašytas Haskell

# Kitos kalbos

---

- Funkcinių bruožai pasirodo liepiamosiose kalbose
  - Anoniminės funkcijos (lambda išraiškos)
    - JavaScript: funkcijos apibrėžime vardas nebūtinas
    - C#: `i => (i % 2) == 0` (grąžins true ar false, priklausomai nuo lygišumo)
    - Python: `lambda a, b : 2 * a - b`

# Python palaikymas

---

- Python turi aukštesnės eilės funkcijas filter ir map (dažnai naudoja lambda išraiškas kaip jų pirmuosius parametrus)

```
map(lambda x : x ** 3, [2, 4, 6, 8])
```

Returns [8, 64, 216, 512]

- Python turi dalinj funkcijos taikymą

```
from operator import add
add5 = partial(add, 5)
```

(pirmoji eilutė importuoja add kaip funkciją)

**Naudojimas:** add5(15) – rezultatas?

# Ruby palaikymas

---

- Ruby blokai
  - Tai veiksmingos paprogramės, kurios perduodamos metodams. Metodas tampa aukštesnės eilės paprogramė.
  - Blokas gali būti konvertuotas į paprogramės objektą su `lambda`

```
times = lambda {|a, b| a * b}
```

Naudojimas: `x = times.(3, 4)` (**priskiria** `x 12`)

- Times gali būti skrudintas

```
times5 = times.curry.(5)
```

Naudojimas: `x5 = times5.(3)` (**priskiria** `x5 15`)

# Palyginimas

---

- Liepiamosios kalbos:
  - Efektyvus vykdymas
  - Sudėtinga semantika
  - Sudėtinga sintaksė
  - Lygiagretumą programuotojas projektuoja
- Funkcinės kalbos:
  - Paprasta semantika
  - Paprasta sintaksė
  - Neefektyvus vykdymas
  - Programas galima automatiškai paversti lygiagrečiomis

# Santrauka

---

- Funkcinio programavimo kalbos naudoja funkcijas, sąlygines išraiškas, rekursiją ir funkcinės formas, kad valdytų programos vykdymą vietoj liepiamujų savybių tokiu kaip kintamieji ir priskyrimai
- LISP pradėjo kaip funkcinė kalba, bet vėliau įjungė liepiamasių savybes
- Scheme yra paprastas LISP dialektas, kuris naudoja statinį akiratį išimtinai
- COMMON LISP yra didelė LISP kalba
- ML yra statinio akiračio ir stipriai tipizuota funkcinė kalba, kuri turi tipų numatymą, išimčių valdymą, duomenų tipų įvairovę ir ADT
- Haskell yra užvėlinto įvertinimo funkcinė kalba, turinti begalinius sąrašus ir aibių išraiškas
- Glynosios funkcinės kalbos turi pranašumą prieš liepiamasių alternatyvas, bet jų žemesnis efektyvumas trukdo jų paplitimui esamoje kompiuterių architektūroje

**P175B124**

**Vardai, galiojimo sritys ir  
saistymai**

***The first step toward wisdom is  
calling things by their right  
names***

***Chinese Proverb***

# Funkcinio programavimo klausimai

---

1. Kas yra FPK pagrindas?
2. Kas yra aukštesnės eilės funkcija?
3. Ką užrašo lambda išraiška?
4. Išvardinkite tipizuotas funkcines kalbas.
5. Koks funkinių programavimo kalbų tikslas?
6. Ar Scheme kalba turi valdymo kontrolės sakinius. Jei taip, išvardinkite
7. Kas yra galinė rekursija?
8. Kodėl yra svarbu, kad rekursija būtų galinė?
9. Ką reiškia atidėtas įvertinimas?
10. Kokia funkinių programavimo kalbų savybė sukludė jų plačiam paplitimui?
11. Kokia funkinių kalbų charakteristika užtikrina, kad jų semantika yra paprastesnė, nei liepiamujų kalbų?

# Temos klausimai

---

- Vardai
- Kintamieji
- Saistymo (**binding**) koncepcija
- Galiojimo sritis (**scope**)
- Galiojimo sritis ir gyvavimo laikas (**lifetime**)
- Gyvavimo aplinka (**referencing environment**)
- Vardinės konstantos

# Kintamasis

---

- Liepiamosios kalbos ir von Neumann architektūra
- Pirminiai komponentai: atmintis, procesorius
- Kintamasis – atminties įastelės abstrakcija
- Abstrakcijų charakteristikos artimos įastelių charakteristikoms (sveikasis)
- Abstrakcijos yra labai toli nuo atminties organizavimo

# Kintamųjų atributai

---

- (sextuple of attributes)
- Tipas – svarbiausias
- Vardas – vienas iš svarbesniųjų
- Adresas
- Reikšmė
- Gyvavimo laikas
- Galiojimo sritis

# Vardai

# Pirminiai projektavimo iššūkiai

---

- Ar didžiosios ir mažosios raidės yra skirtingos (**case sensitive**)?
- Ar specialūs žodžiai yra rezervuoti žodžiai ar raktiniai žodžiai?

# Ilgis

---

- Negali būti per trumpas
- Pavyzdžiai:
  - FORTRAN 95: maksimalus – 31
  - C99: neapribotas, bet tik pirmieji 63 reikšmingi;  
Taip pat išoriniai vardai apriboti 31
  - C#, Ada, Java: be apribojimų ir visi reikšmingi
  - C++: be apribojimų, tačiau kompiliatoriai dažnai apriboja (vardai saugomi lentelėje)

# Vardo forma

---

- Raidė, po kurios raidės, skaitmenys, pabraukimo simboliai
- Pabraukimo simbolis buvo populiarus 1970 – 1980
- C šeimoje buvo pakeistas kupranugario (*camel*) notacija (kiekvienas tolimesnis žodis pradedamas didžiaja raide)
- Tai stiliaus, bet ne kalbos problema

# Specialūs simboliai

---

- PHP: visi kintamieji privalo prasidėti dolerio ženklu
- Perl: visi kintamieji privalo prasidėti specialiais simboliais, kurie nustato kintamojo tipą (\$, @, %)
- Ruby: kintamieji, kurie prasideda @, yra objektų kintamieji; kintamieji, kurie prasideda @@, yra klasės kintamieji
- Fortran iki 90 versijos vardai galėjo turėti tarpus

# Didžiosios mažosios

---

- Case sensitive
- Trūkumas: skaitomumas (panašūs vardai yra skirtini)

  - C šeimos kalbose didžiosios ir mažosios yra skirtinos
  - Fortran iki 90 buvo tik didžiosios
  - Haskell, mažoji raidė (kintamujų vardai), didžioji raidė (konstruktorių vardai)

- Dar blogiau Java ir C# kalbose, kur iš anksto apibrėžti vardai yra mišrūs. Rašomumo problema. (FileNotFoundException)

# Specialūs žodžiai

---

- Didina skaitomumą; naudojama išskirti sakinius
- *Raktinis žodis* – tai žodis, kuris yra specialus tik tam tikrame kontekste (Fortran);
  - `Real Integer` (*Real duomenų tipas, todėl Real - raktinis žodis*)
  - `Integer = 3.4` (*Integer – kintamasis*)
- *Rezervuotas žodis* - specialus žodis, kuris negali būti naudojamas kaip vartotojo apibrėžtas vardas
- Problema su rezervuotais žodžiais: daug susidūrimų (COBOL turi 300 rezervuotų žodžių!)
- Iš anksto apibrėžti (*predefined*) vardai: bibliotekų paprogramės. Jie turi prasmę, bet vartotojas jiems gali suteikti savo prasmę

# Kintamųjų vardas ir adresas

---

- Vardas – ne visi kintamieji juos turi
- Adresas – atminties adresas, su kuriuo jis susietas
  - Kintamasis gali turėti skirtingus adresus skirtingais laiko momentais vykdymo metu
  - Kintamasis gali turėti skirtingus adresus skirtingose programos vietose
  - Jei du kintamujų vardai gali būti naudojami prieigos pačios atminties vietas, jie vadinami antrininkais arba sinonimais
  - Antrininkus ([aliases](#)) sukuria rodyklės, nuorodos, C ir C++ unijos
- Antrininkai kenkia skaitomumui, patikimumui

# Tipas ir reikšmė

---

- *Tipas* apibrėžia kintamojo reikšmių aibę ir galimas operacijas su šio tipo reikšmėmis
- *Reikšmė* – atminties vietas, su kuria susietas kintamasis, turinys
  - **l-reikšmė** – tai kintamojo adresas
  - **r-reikšmė** – tai kintamojo reikšmė

# Abstrakti atminties ląstelė

---

- *Abstrakti atminties ląstelė* – fizinė ląstelė ar ląstelių rinkinys, susietas su kintamuoju
- Fizinė ląstelė yra adresuoojamas dydis, dažniausiai baito dydžio, 8 bitai
- Per mažas dydis programos kintamiesiems
- Norint pasiekti **r-reikšmę**, **1-reikšmę** turi būti apibrėžta pirmiau

# **Saistymas**

# Saistymo koncepcija

---

- Terminas *saistymas* yra susiejimas tarp savybės (reikšmė) ir esybės (kintamasis).
- *Saistymo laikas* – tai laikas, kada susiejimas įvyksta.

# Saistymo laikai

---

- **Kalbos projektavimo laikas** – susieja operatoriaus simbolj su operacija
- **Kalbos įgyvendinimo laikas** – susieja realujį skaičių su jo atvaizdavimu
- **Kompiliavimo laikas** – susieja kintamajį su tipu
- **Sujungimo laikas** – atskiras kompiliavimas. Ryšių redaktorius sudaro bendrą modulių schemą ir išsprendžia tarpmodulines nuorodas
- **Įkrovimo laikas** -- susieja C ar C++ statinius kintamuosius su atminties ląstele
- **Vykdymo** – susieja ne statinius kintamuosius su atminties ląstelėmis

# Statinis ir dinaminis

---

- Saistymas yra *statinis*, jei jis įvyksta prieš vykdymo laiką ir lieka nepakitus per visą programos vykdymą.
- Saistymas yra *dinaminis*, jei jis įvyksta vykdymo metu ar gali pasikeisti programos vykdymo metu

# Tipo saistymas

---

- Prieš naudojant kintamąjį, jis turi būti susietas su tipu.
- Kaip tipas nurodytas?
- Kada saistymas įvyksta?
- Jei statinis, tipas gali būti paskelbtas išreikštai arba numanomai

# Statinis tipo saistymas

---

- *Išreikštasis paskelbimas* – tai programos sakinys, kuriами skelbiami kintamujų tipai
- *Numanomas deklaravimas* yra kintamujų tipų paskelbimo numatytais mechanizmas (pagal pirmajį kintamojo pasirodymą programe). Tipo išvedimo ([inference](#)) mechanizmas
- Dauguma naujesnių kalbų (pradedant 1965) reikalauja išreikšto paskelbimo, bet Perl, JavaScript, Ruby, ML.

# Numanomas paskelbimas

---

- Fortran, BASIC, Perl, Ruby, JavaScript ir PHP naudoja numanomą paskelbimą (Fortran turi abu)
- Fortran:
  - I, J, K, L, M, N – sveikieji
  - Direktyva Implicit none
- Perl:
  - \$ - skaliaras
  - @ - masyvas
  - % - asociatyvus masyvas
- Ivertinimas:
  - Privalomumas: rašomumas
  - Trūkumas: patikimumas (mažiau rūpesčių su Perl)

# Tipo išvedimas

---

- Kai kurios kalbos naudoja tipo išvedimą, kad nustatyti kintamuju tipus pagal kontekstą:
  - C# - kintamasis gali būti deklaruotas su **var** ir pradine reikšme. Pradinė reikšmė nustato tipą
  - Visual BASIC 9.0+, ML, Miranda, Haskell, F#, ir Go naudoja tipų išvedimą. Pasirodymo kontekstas apsprendžia kintamojo tipą, nebūtinai priskyrimo sakinys

# Dinaminis tipo saistymas

---

- JavaScript, Python, Ruby, PHP – grynieji interpretatoriai ir C# (ribotai)
- Priskyrimo sakinys (JavaScript)

```
list = [2, 4.33, 6, 8];
```

```
list = 17.3;
```

- Privalumas: lankstumas – viena paprogramė visiems tipams
- Trūkumai:
  - Didelė kaina (dinaminis tipų tikrinimas ir interpretavimas)
  - Mažiau – patikimumo: kompiliatorius nelabai gali aptikti tipų klaidas

# Statinio tipo problemos

---

- Dėl tipų patikrinimo turi problemų ir statinio saistymo kalbos, kurios atlieka tipų konversiją.
- Tai Fortran, C, C++
- Parametruose su nuoroda C++ visuomet vietoj realiojo tipo galima pateikti sveikajį tipą.

# Tipo išvedimas ML kalboje

---

- fun circumf(r) = 3.14159 \* r \* r;
  - Išves realuji
- fun times10(x) = 10 \* r;
  - Išves sveikaji
- fun square(x) = r \* r;
  - Išves skaitmeninj (vadinasi, sveikaji)
- fun squarer(x) : real = r \* r;
  - realusis

# Atminties paskyrimas

---

- Atminties paskyrimas
  - Paskyrimas ([allocation](#))
  - Grąžinimas ([deallocation](#))
- Kintamojo gyvavimo laikas – tai laikas, kai jam yra priskirta tam tikra atminties ląstelė (jis gauna ją, ji paimama iš kintamojo)

# **Kintamųjų kategorijos pagal gyvavimo laiką**

# Keturios kategorijos

---

- Statinis
- Dėklo dinaminis
- Išreikštas kupetos dinaminis
- Numanomas kupetos dinaminis

# Statiniai kintamieji

---

- Statinis – susiejamas su atminties ląstele iki vykdymo ir lieka susietas su pačia ląstele per visą vykdymą (C ir C++ `static` kintamieji)
  - Privalumai: efektyvumas (tiesioginis adresavimas), paprogramių istorijos palaikymas
  - Trūkumas: lankstumo trūkumas (nėra rekursijos), atmintimi negalima pasidalinti
- `static` kintamieji C++ klasėje nėra tas pats, kas `static` kintamieji už klasės ribų. Maišatis

# Déklo dinaminiai

---

- Déklo dinaminiai – atminties įastelės paskiriamos, kai vykdymo metu sutinkamas paskelbimo sakėnas
- Paskelbimo sakėnas gali būti bet kur, bet saistymas su atmintimi gali būti atliekamas metodo pradžioje
- Jei skaliaras, visi atributai, išskyrus adresą, susiejami statiškai
  - Lokalūs kintamieji C funkcijoje ir Java metoduose
- Privalumai: leidžia rekursiją; taupo atmintį
- Trūkumai:
  - Priskyrimo ir grąžinimo išlaidos
  - Paprogramės negali prisiminti praeities
  - Neefektyvios nuorodos – netiesioginis adresavimas

# Išreikšti kupetos dinaminiai

---

- *Išreikšti kupetos dinaminiai* – priskiriami ir gražinami pagal išreikštas direktyvas, kurias nurodo programuotojas.  
Bevardžiai kintamieji
- Pasiekiami tik per rodykles ir nuorodas – dinaminiai objektai C++ (new ir delete), visi Java objektai, C# objektai gali būti ir dėkle, bet yra ir rodyklės (unsafe)
- Privalumas: dinaminis atminties valdymas
- Trūkumai:
  - Naudojimo sunkumai
  - Kreipinio į kintamąjį kaina
  - Atminties valdymo įgyvendinimo sunkumas

# Numanomi kupetos dinaminiai

---

- *Numanomas kupetos dinaminis* – atminties paskyrimą ir gražinimą sukelia priskyrimo sakinys
  - APL kintamieji; eilutės ir masyvai Perl, JavaScript ir PHP
  - Visi atributai priskiriami dinamiškai
- Privalumas: lankstumas – bendrinis kodas
- Trūkumai:
  - Neefektyvu, nes visi atributai yra dinaminiai
  - Klaidų aptikimo praradimas

# **Kintamųjų atributas – galiojimo sritis**

# Galiojimo sritis

---

- Kintamojo galiojimo sritis – tai sakiniai, kuriuose jis matomas
- Kintamasis yra matomas sakinyje, jei į kintamąjį galima kreiptis
- Kalbos galiojimo srities taisyklės nustato, kaip vardai susiejami su kintamaisiais
- Programos vieneto nelokalūs kintamieji yra tie, kurie tame matomi, bet nedeklaruoti

# Statinė galiojimo sritis

---

- Įvedė Algol 60 – pagal poziciją programos tekste.
- Statiškai nustatoma iki programos vykdymo
- Norint susieti vardą su kintamuju, reikia surasti paskelbimo sakinį
- Du statinių galiojimo sričių kalbų tipai:
  - Paprogramės gali būti sunertos (**nested**)
  - Paprogramės negali būti sunertos
- Nesunertų paprogramų atveju, statines galiojimo sritis formuoja tik paprogramės, o sunertas galiojimo sritis formuoja klasės ir blokai

# Sunertos paprogramės

---

- Kai kurios kalbos leidžia talpinti paprogramę į paprogramę, tuomet sukuriami viena į kitą įdėtos statinės galiojimo sritys (Ada, JavaScript, Fortran 2003, Common Lisp, F#, Python)
- *Kintamojo atributų paieškos procesas:* ieškoti paskelbimų, pirmiausiai lokaliai, tuomet didinti apimančias galiojimo sritis, kol bus surastas duotam vardui
- Apimančios statinės galiojimo sritys vadinamos statiniais protėviais; artimiausia statinė galiojimo sritis vadinama statiniu tévu

# Kintamojo paslėpimas

---

- Kintamieji gali būti paslėpti, turint kintamajį tuo pačiu vardu arčiau.
- Ada leidžia pasiekti paslėptus kintamuosius
  - E.g., `unit.name`

# JavaScript kodas

---

```
function big() {
 function sub1 () {
 var x = 7;
 sub2();
 }
 function sub2() {
 var y = x;
 }
 var x = 3;
 sub1();
}
```



Kokia x reikšmė?

# Blokai

---

- Atskirose kodo sekcijose, kur pradžioje skelbiami kintamieji, vadinamos blokais
- Statinės galiojimo sritys programos vienetų viduje – prasidėjo nuo ALGOL 60
- Statinės galiojimo sritys, kaip ir paprogramėse, sukuriamas blokuose

# Blokų pavyzdys

---

- C:

```
void sub() {
 int count;
 while (...) {
 int count;
 count++;
 ...
 }
 ...
}
```

- Lokalūs kintamieji sunertuose blokuose tais pačiais vardais C ir C++, bet ne Java ir C# - perdaug klaidų

# LET konstrukcija

---

- Dauguma funkcinių kalbų turi let konstrukcijos variantą
- let konstrukcija turi 2 dalis:
  - Pirmoji dalis susieja vardus su reikšmėmis
  - Antroji dalis naudoja vardus, apibrėžtus pirmojoje dalyje
- Scheme:

```
(LET (
 (vardas1 išraiška1)
 ...
 (vardasn išraiškan))
```

# LET konstrukcija 2

---

- ML:

**let**

val vardas<sub>1</sub> = išraiška<sub>1</sub>

...

val vardas<sub>n</sub> = išraiška<sub>n</sub>

**in**

išraiška

**end;**

- F#:

- Pirmoji dalis: **let** kairė\_pusė = išraiška
- (kairė\_pusė yra arba vardas, arba kortežas)
- Visa kita yra antroji dalis

# Simbolių lentelė

---

- Simbolių lentelė – tai struktūra, kurioje kompiliatorius saugo vardą ir jo saistymus
- Sakykim, kad kiekvienas vardas unikalus savo galiojimo srityje
- Duomenų struktūra gali būti žodynas, kur raktas yra vardas

# Paieškos procesas

---

1. Kai tik įeinama į galiojimo sritį, naujas žodynas į dėklą
2. Kai tik išeinama iš galiojimo srities, žodyną šalinti iš dėklo
3. Kiekvienam deklaruotam vardui, generuoti atitinkamą saistymą ir talpinti porą vardas-saistymas į dėklo viršūnę
4. Duotas vardas, ieškoti žodyne, pradedant dėklo viršūne
  - a) Jei surastas, grąžinti saistymą
  - b) Jei ne, pakartoti paiešką tolimesniame žodyne, kuris yra dėkle
  - c) Jei nerastas nei viename, klaida

# Deklaravimo tvarka

---

- C, C++, Java ir C# paskelbimo sakinys gali būti ten, kur gali būti sakinys
  - C, C++ ir Java lokalių kintamuju galiojimo sritis yra nuo paskelbimo pradžios iki bloko pabaigos
  - C# bet kurio deklaruoto bloko viduje kintamojo galiojimo sritis yra visas blokas, nepriklausomai nuo deklaravimo vietas
    - Tačiau, prieš naudojimą kintamasis vis tiek turi būti paskelbtas

# Globali galiojimo sritis

---

- C, C++, PHP ir Python leidžia apibrėžti funkcijų paskelbimus faile
  - Šios kalbos leidžia skelbti kintamuosius funkcijų paskelbimų išorėje
- C ir C++ turi paskelbimus (vien tik atributai) (**declaration**) ir apibrėžimus (atributai ir atmintis) (**definition** ir skiria atmintį)
  - `extern int sum`
  - `::x`

# Globali sritis PHP

---

- Programos įterpiamos į XHTML žymėjimo dokumentus, įvairiuose fragmentuose, tiek sakiniai, tiek funkcijų apibrėžimai
- Kintamojo galiojimo sritis, (numanomai) paskelbtas funkcijoje, yra jai lokalus
- Kintamojo galiojimo sritis, netiesiogiai deklaruotas už funkcijos, galioja iki programos pabaigos, bet praleidžia tarpines funkcijas
  - Globalūs kintamieji gali būti pasiekiami funkcijoje per `$GLOBALS` masyvą tuo pačiu kintamojo vardu arba paskelbiant jį `global`

# Globali sritis Python

---

- Nėra išreikšto deklaravimo
- Globalus kintamasis yra pasiekiamas funkcijoje, bet gali būti priskirtas tik, jei funkcijoje buvo paskelbtas `global`
- Funkcijos gali būti įdėtos vieną į kitą. Kintamieji pasiekiami per statinio akiračio taisykles, bet privalo būti deklaruoti per `nonlocal`

# Statinės srities įvertinimas

---

- Veikia gerai daugelyje situacijų
- Problemų:
  - Daugelyje atvejų, per daug pasiekiamas
  - Kai programa vystoma, pradinė struktūra sugriaunama ir lokalūs kintamieji tampa globaliai; paprogramės irgi traukia link globalių
- Inkapsuliacija

# Dinaminė galiojimo sritis

---

- Remiasi programos vienetų kvietimo seką, ne jų tekstuiniu išdėstymu
- Nuorodos į kintamuosius yra susiejamos į paskelbimus, ieškant atgal per paprogramių kvietimo grandinę, kuri atvedė iki esamo taško
- Naudojo ankstyvasis Lisp, APL, Snobol, Perl.

# Galiojimo srities pavyzdys

Big

- X paskelbimas
- Sub1
  - X paskelbimas
  - ...

call Sub2

...

Sub2

...

- kreipinys į X -

...

...

call Sub2

...

Big kviečia Sub1  
Sub1 kviečia Sub2  
Sub2 naudoja X

# Palyginimas

---

- Statinis akiratis
  - Kreipinys į X yra Big X
- Dinaminis akiratis
  - Kreipinys į X yra Sub1 X

# Įvertinimas

---

- *Įtaka didelė:*
  - Nelokalių kintamujų statiskai atributų neįmanoma nustatyti
  - Gali kreiptis ne visada į tą patį kintamąjį
- *Trūkumai:*
  1. Kai paprogramė vykdoma, jos kintamieji yra matomi visiems paprogramės kreipiniams. Vykdoma anksčiau nebaigusių paprogramų aplinkoje. Mažiau patikimumo.
  2. Neįmanomas statinis nelokalių kintamujų tipų tikrinimas
  3. Blogas skaitomumas – statiskai neįmanoma nustatyti kintamojo tipo
  4. Ilgesnis išrinkimo laikas
- *Privalumai:*
  1. Nereikia parametru, viskas matoma

# Sritis ir gyvavimo laikas

---

- Gyvavimo sritis ir gyvavimo laikas ([lifetime](#)) yra artimai susijusios, bet skirtingos koncepcijos
- Erdvinė ir laikinė koncepcijos
- Kintamojo gyvavimo laikas – tai laiko intervalas, kada jam buvo išskirtas atminties blokas
- Atvejai, kai sritis ir gyvavimo laikas nesusiję:
  - static kintamasis C ar C++ funkcijose
  - Kintamasis deklaruotas funkcijoje, funkcija kviečia kitą funkciją

# Sakinio gyvavimo aplinka

---

- Sakinio gyvavimo aplinka (**referencing environment**) yra vardų, kurie matomi tame sakinyje, rinkinys
- Statinės galiojimo srities kalbose lokalūs kintamieji ir visi kiti apimančios galiojimo srities kintamieji
- Paprogramė yra aktyvi, jei ji pradėjo vykdymą, bet dar nepabaigė
- Dinaminės galiojimo srities kalbose visi lokalūs kintamieji ir visi matomi visų aktyvių paprogramiu kintamieji

# Įvardintos konstantos

---

- Įvardinta konstanta – tai kintamasis, kuris susiejamas su reikšme atminties priskyrimo metu
- Privalumai: skaitomumas ir lengvas modifikavimas, patikimumas
- Naudojamas parametrizuoti programas
- Reikšmių susiejimas su įvardintomis konstantomis gali būti statinis arba dinaminis

# Įvardintos konstantos kalbose

---

- FORTRAN 95: pastoviosios išraiškos – statinis susietumas
- Ada, C++ ir Java: bet kurio tipo išraiškos – dinaminis susietumas
- C# turi dvi rūšis `readonly` ir `const`
  - `const` – reikšmės susiejamos kompiliavimo metu
  - `readonly` – reikšmės susiejamos dinamiškai

# Santrauka

---

- Didžiosios mažosios raidės varduose ir specialūs žodžiai – vardų projektavimo pasirinkimai
- Kintamuosius charakterizuoja 6 atributai: vardas, adresas, reikšmė, tipas, gyvavimo laikas, galiojimo sritis
- Saistymas yra atributų asociacija su programos esybėmis
- Skaliarinių kintamujų kategorijos: statiniai, dėklo dinaminiai, išreikšti kupetos dinaminiai, netiesioginiai kupetos dinaminiai
- Stiprūs tipai reiškia visų tipų klaidų aptikimą

**P175B124**

**Duomenų tipai**

***Types are the leaven of computer  
programming;  
they make it digestible.***

***Robin Milner***

# Vardų klausimai

---

1. Ką vadiname rezervuotu žodžiu?
2. Ką vadiname raktiniu žodžiu?
3. Kas yra kintamojo  $f$ -reikšmė? Kas yra kintamojo  $r$ -reikšmė?
4. Kokius atributus turi kintamasis?
5. Kokie yra statinių kintamujų privalumai ir trūkumai?
6. Kokios yra skaliarinių kintamujų kategorijos pagal gyvavimo laiką?
7. Apibrėžkite gyvavimo laiką ir galiojimo sritį. Koks yra skirtumas tarp jų?
8. Kokie yra statinės galiojimo srities privalumai ir trūkumai?
9. Kokie yra dinaminės galiojimo srities privalumai ir trūkumai?
10. Kas yra sakinio gyvavimo aplinka?
11. Kokie yra projektavimo iššūkiai dėl vardų?

# Temos klausimai

---

- Primityvūs tipai
- Eilutės
- Vartotojo apibrėžti išvardijamieji  
**(enumeration)**
- Masyvai
- Asociatyvūs masyvai
- Įrašai

# Tipų istorija

---

- Duomenų tipas apibrėžia duomenų rinkinį ir iš anksto apibrėžtas operacijas su šiais duomenimis
- Kaip duomenų tipai atspindi realybę?
- Mažai duomenų tipų Fortran, daugiau – COBOL
- Daug duomenų tipų PL/I
- Lanksti duomenų struktūra **Algol 68** – svarbiausia pažanga. **Vartotojas apibrėžia.**
- Vartotojo apibrėžti duomenų tipai – pagerintas skaitomumas, keičiamumas
- Abstraktūs duomenų tipai – sąsaja, atskirta nuo įgyvendinimo

# Tipų panaudojimas

---

1. Klaidų aptikimas
2. Pagalba programos skaidymui į modulius – tipų tikrinimas tarp modulių
3. Dokumentavimas – dokumentuoja informaciją apie duomenis, leidžia nuspėti programos elgseną
  - Suprasti kalbos semantiką – suprasti jos tipų sistemą
  - Struktūruoti dažniausiai naudojami yra masyvai ir įrašai. Padidėjės asociatyvių masyvų naudojimas

# Kintamojo atributai

---

- Deskriptorius yra kintamojo atributų rinkinys
- Deskriptoriai statiniams atributams (tik kompiliatorius) ir dinaminiams atributams (vykdymo sistema)
- Deskriptoriai naudojami atminčiai išskirti
- Projektavimo pasirinkimas visiems duomenų tipams:
  - **kokios operacijos ir kaip jas apibrėžti**

# **Primityvūs duomenų tipai**

# Apibrežimas

---

- Beveik visos programavimo kalbos turi primityvius duomenų tipus
- Primityvūs duomenų tipai – tie, kurie nėra apibrėžti kitų tipų terminais
- Kai kurie primityvūs duomenų tipai yra tiesiog aparatūros atspindys
- Kiti reikalauja tik mažo ne aparatūros palaikymo jų įgyvendinimui (Python long integer tipas, L skaičiaus gale)
- Neigiamų skaičių įgyvendinimai:
  - ženklo užrašas, (ženklas ir absoliuti reikšmė, netinka aritmetikai)
  - dviejų papildymas – loginė inversija teigiamo plius vieno
  - ir vieneto papildymas – du nulio atvaizdavimai

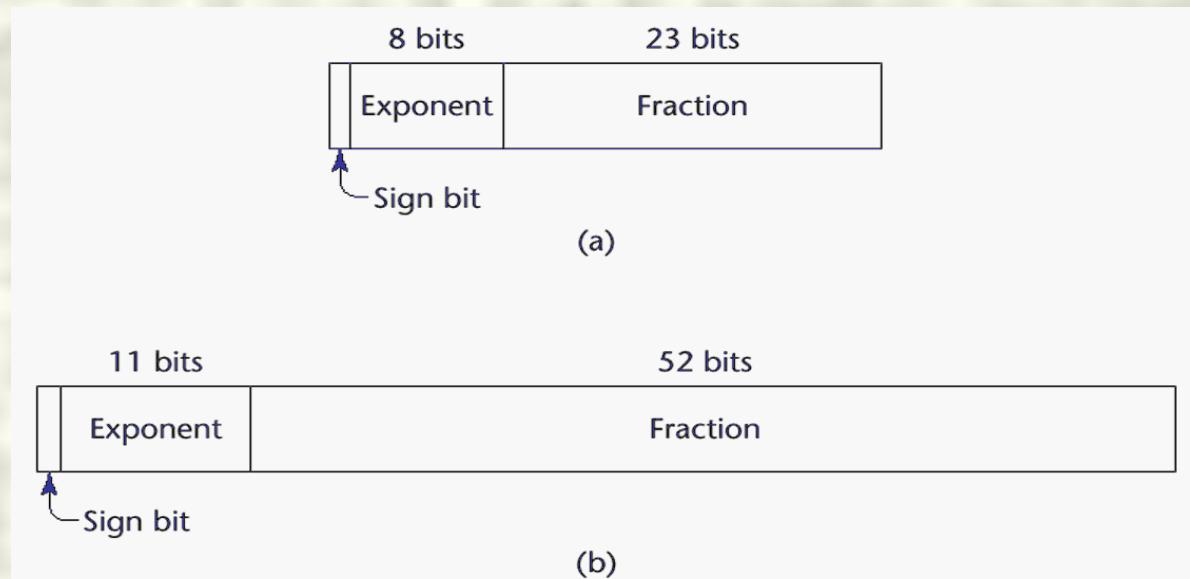
# Sveikasis

---

- Iki 8 sveikujų duomenų tipų
- Beveik tiksliai atspindi aparatūrą, todėl susiejimas trivialus
- Java su ženklu sveikieji tipai: `byte`, `short`, `int`, `long`. C++, C# turi be ženklo
- Sveikasis tipas turi fiksuotą reikšmių skaičių dėl fiksuoto atminties dydžio paskyrimo
- Išimtys:
  - Haskell sveikasis (`Int`) atvaizduoja neapribotus sveikuosius
  - Python neriboto ilgio sveikojo tipo reikšmės, F# turi taip pat

# Slankaus taško

- Modeliuoja realiuosius skaičius, bet tik apytiksliai
- Dauguma kalbų turi 2 slankaus taško tipus (float ir double)
- Dažniausiai atitinka aparatūrą, bet ne visada
- IEEE Floating-Point Standard 754



# Kompleksiniai

---

- Kai kurios kalbos turi kompleksinių skaičių tipą (Fortran ir Python)
- Kiekviena reikšmė sudaryta iš dviejų dalių: realiosios ir menamosios
- Literalo forma (Python):  
 $(7 + 3j)$ , čia 7 yra realioji dalis, o 3 – menamoji dalis

# Dešimtainis

---

- Verslo skaičiavimams (pinigai)
  - COBOL
  - C#
  - F#
- Saugo fiksuotą dešimtainių skaitmenų kiekį koduotoje formoje (**binary coded decimal**), panašiai kaip eilučių simboliai
- *Privalumas:* tikslumas
- *Trūkumai:* apribota eilė, netaupo atminties – mažiausiai 4 bitai kiekvienam skaitmeniui
- Koks yra skirtumas nuo dvejetainio saugojimo?
- 6 skaitmenys – 24 bitai, dvejetainiame 20 bitų

# Loginis

---

- Visų paprasčiausias
- Tik dvi reikšmės „true“ ir „false“
- Įvedė Algol 60
- Išimtis C89, but ne C99. Skaitmeninės naudojamos kaip loginės, bet ne Java ir C#.
- Gali būti įgyvendintas bitais, bet dažniausiai baitais
  - Privalumas: skaitomumas

# Simbolinis

---

- Saugomas kaip skaitmeniniai kodai
- Dažniausiai naudojama koduotė: ASCII
- Alternatyva 16 bitų koduotė: Unicode (UCS-2)
  - Turi simbolius iš daugelio kalbų
  - Java pirmoji panaudojo
  - C#, F#, Python, Perl and JavaScript taip pat palaiko Unicode
- 32 bitų Unicode (UCS-4, UTF-32), paskelbtas 2000
  - Palaiko Fortran, pradedant 2003
- Python neturi tipo vienam simboliui (eilutė 1 ilgio)

# Eilutės tipai

# Simbolių eilutės

---

- Reikšmės – simbolių sekos
- Pažymimos išvedimo reikšmės, daug įvedimo, išvedimo
- Projektavimo pasirinkimai:
  - Ar tai primityvus tipas ar specialus masyvo atvejis?
  - Ar eilutės ilgis turėtų būti statinis ar dinaminis?

# Tipinės eilučių operacijos

---

- Priskyrimas – skirtinių ilgiai
- Palyginimas ( $=$ ,  $>$ , ... ) – skirtinių ilgiai
- Apjungimas (Catenation)
- Eilutės dalies paémimas (substring reference) – slice
- Rinkinių atitikimas (pattern matching)  
kalboje ar bibliotekoje

# Eilutės kalbose

---

- C
  - Neprimityvus
  - naudoja **char** masyvus su spec. simboliu gale ir funkcijų biblioteką. Nesaugo nuo perpildymo – problemos
- C++
  - Neprimityvus, bet **string** klasė yra
- Java
  - Primityvus per **String** klasę, **StringBuffer** klasę, taip pat (**mutable**). C# ir Ruby
- Python – primityvus tipas, pastoviosios eilutės
- F#. Eilutės yra klasė, pastoviosios

# Rinkinių atitikimas

---

- Perl, JavaScript, Ruby ir PHP
  - Reguliarios išraiškos. Išsivystė iš UNIX eilutinio redaktoriaus ed.
  - /[A-Za-z][A-Za-z\d]+/
- C++, Java, Python, C# ir F# bibliotekose

# Ilgio nustatymas

---

- Statinis: COBOL, Java String klasė, Python, Ruby's String klasė, C#, F#.
- *Apribotas dinaminis ilgis*: C
  - Specialus simbolis rodo pabaigą
- *Dinaminis* (be apribojimų): Perl, JavaScript, C++

# Eilutės tipo įvertinimas

---

- Padeda rašomumui
- Negalima pateisinti tokio tipo trūkumo
- Primityvus tipas su statiniu ilgiu nedaug kainuoja
- Pattern matching and catenation – turėtų būti
- Dinaminis ilgis puiku, bet ar vertos tokios išlaidos

# Įgyvendinimas

---

- Statinis ilgis: kompiliavimo laiko deskriptorius
- Ribotas dinaminis ilgis: gali reikti vykdymo laiko deskriptoriaus ilgiui, bet ne C
- Dinaminis ilgis: reikia vykdymo laiko deskriptoriaus; priskyrimas ir grąžinimas yra didžiausia įgyvendinimo problema
- Deskriptoriai dažniausiai saugomi simbolių lentelėje

# Deskriptoriai

|               |
|---------------|
| Static string |
| Length        |
| Address       |

Kompiliavimo laiko  
deskriptorius  
statinėms eilutėms

|                        |
|------------------------|
| Limited dynamic string |
| Maximum length         |
| Current length         |
| Address                |

Vykdymo laiko  
deskriptorius  
ribotoms  
dinaminėms eilutėms

# Dinaminio įgyvendinimas

---

3 būdai:

- Susietasis sąrašas
  - trūkumas – papildoma atmintis ryšiams, sudėtingos eilučių operacijos
- Rodyklių masyvas į atskirus simbolius
  - Trūkumas – extra atmintis, bet greitesnis apdorojimas
- Eilutė gretimose atminties laštelėse
  - Problema, kai eilutė auga.

# Išvardijimo tipas

# Apibrėžimas

---

- Visos galimos reikšmės, kurios yra vardinės konstantos, pateikiamos apibrėžime
- Tai susietų konstantų grupavimo būdas
- C#  
`enum days {mon, tue, wed, thu, fri, sat, sun};`
- Dažniausiai priskiriamos iš eilės einančios sveikosios reikšmės 0, 1, ...

# Projektavimo pasirinkimai

---

- Ar išvardijimo konstanta gali pasirodyti keliuose tipuose. Jei taip, kaip atskirti, kuriam priklauso?
  - Ar išvardijimo reikšmės verčiamos į sveikajį tipą?
  - Bet kuris kitas tipas verčiamas į išvardijimo tipą?
- 
- Šios problemas susijusios su tipų tikrinimu

# Projektai

---

- C ir Pascal – pirmosios kalbos, kurios įtraukė
- Nėra Perl, Javascript, PHP, Ruby, Lua
- Python nuo 3.4.0 (2014 kovo 16 d.) jau turi
- C++
  - enum colors {red, blue, green, yellow, white}
  - colors myColor = blue, yourColor = green;
  - Gali pasirodyti tik viename rinkinyje
  - Jos verčiamos į sveikaji, jei to reikalauja aplinka  
(myColor++ ;)
  - Sveikojo priskirti negalima, nebent išreikštas tipų keitimas
- C# neverčiamos į sveikaji, visa kita tas pats

# Išvardijimo įvertinimas

---

- Padeda skaitomumui – spalvos nereikia įvardinti skaičiumi
- Padeda patikimumui, nes kompiuteris gali patikrinti:
  - Operacijas – spalvos nesudedamos
  - Išvardijimo kintamasis negali įgyti reikšmės, kurios nėra jo tipe
  - negalima priskirti sveikosios reikšmės iš ne konstantų intervalo (negalioja C)
  - F#, C# ir Java 5.0 geriau palaiko išvardijimą, nei C++ ir C, nes išvardijimo tipo kintamieji neverčiami į sveikajį tipą;

# Poaibių tipai

---

- Ištisas tipo poaibis

- 12..18 – tai sveikojo tipo poaibis

- Ada, VHDL

```
type Days is (mon, tue, wed, thu, fri, sat, sun);
subtype Weekdays is Days range mon..fri;
subtype Index is Integer range 1..100;
```

```
Day1: Days;
```

```
Day2: Weekdays;
```

```
Day2 := Day1;
```

- Naudojamas ciklų indeksams

# Poaibių įvertinimas

---

- Skaitomumas
  - Skaitančiajam yra aišku, kad kintamieji gali įgyti tik poaibio reikšmes
- Patikimumas
  - Netinkamų reikšmių priskyrimas aptinkamas
- Nėra šiuolaikinėse, išskyrus Ada95

# Įgyvendinimas

---

- Išvardijimo tipai įgyvendinami kaip sveikieji
- Poaibių tipai įgyvendinami kaip téviniai tipai su apribojimais dėl priskyrimu
- Padidina kodo dalį, bet galima optimizuoti

# Masyvų tipai

# Masyvas

---

- Masyvas – tai homogeninių elementų junginys, kuriame kiekvienas elementas įvardijamas jo pozicija, pradedant nuo pirmojo elemento.
- Atskirą elementą pasiekiame, naudodami indeksą (**subscript**)
- C šeimos kalbose masyvo elementai yra to paties tipo. C# turi **ArrayList**, bet jis beveik nenaudojamas
- JavaScript, Python, Ruby kintamieji – betipės nuorodos į objektus. Masyvo elementai gali rodyti į skirtingo tipo objektus

# Projektavimo pasirinkimai

---

- Kokius tipus naudoti indeksams?
- Ar tikrinti indekso ribas?
- Kada indeksų ribos susiejamos?
- Kada paskiriama atminties?
- Maksimalus indeksų kiekis?
- Ar masyvai gali būti inicializuojami?
- Ar yra masyvų skiltys (**slices**)?

# Indeksai

---

- *Indeksavimas* – tai perėjimas nuo indeksų prie elementų (**finite mapping**)  
**masyvas (indeksas) → elementas**
  - Šioje išraiškoje yra 2 skirtingi tipai
- Indekso sintakse
  - Vieni skliaustai ar keli skliaustai – masyvų masyvas
  - FORTRAN, Ada naudoja apvalius skliaustus
    - Ada naudoja apvalius skliaustus specialiai, norėdama parodyti panašumą tarp masyvų nuorodų ir kreipinių į funkcijas, nes abu yra atvaizdžiai – mažina skaitomumą
  - Dauguma kalbų naudoja laužtinius skliaustus

# Indeksų tipas

---

- FORTRAN, C, Java: tik sveikieji
- Neįprasta Perl: masyvas @list, nuoroda - \$list[1]
  - Galima kreiptis su neigiamu indeksu
- Indekso ribų tikrinimas
  - C, C++, Perl ir Fortran netikrina indekso ribų
  - Java, ML, C#, Python tikrina indekso ribas

# Masyvų kategorijos

---

- Apatinė masyvo dydžio riba yra numanoma, dažniausiai
- 4 kategorijos priklausomai nuo saistymo su indeksais, saistymo su atmintimi ir iš kur skiriama atmintis
- Statinis masyvas, fiksuotas dėklo dinaminis masyvas, fiksuotas kupetos dinaminis masyvas, kupetos dinaminis masyvas
- Pirmos 3 kategorijos negali pakeisti dydžio

# Indeksų susiejimas

---

- *Statiniai*: masyvo ribos yra statiskai susiejamos ir atminties paskiriama prieš vykdymą
  - Privalumas: efektyvumas (nėra dinaminio priskyrimo)
- *Fiksotas déklo dinaminis*: masyvo ribos yra statiskai susiejamos, bet atminties paskiriama vykdymo metu
  - Privalumas: erdvės efektyvumas

# Indeksų susiejimas 2

---

- *Fiksotas kupetas dinaminis:* indeksų ribos susiejamos dinamiškai ir atminties išskiriama dinamiškai vykdymo metu
  - Privalumas: lankstumas – masyvo dydžio nereikia žinoti iki masyvo naudojimo
- *Kupetas dinaminis:* susiejimas indeksų ir atminties dinaminis bei gali keistis daug kartų
  - Privalumas: lankstumas – masyvai gali augti ar trauktis vykdymo metu

# Įgyvendinimas

---

- C ir C++ masyvai, kurie turi **static** modifikatorių, yra statiniai
- C ir C++ masyvai be **static** modifikatoriaus yra fiksuoti dėklo dinaminiai
- C ir C++ turi fiksuotus kippetos dinaminius masyvus (Java visi tokie masyvai)
- C# turi papildomą klasę **ArrayList**, kuri suteikia kippetos dinaminius. Java taip pat turi, **get**, **set** metodai turi būti naudojami. Bendrinė **List** C# - kippetos dinaminiai
- Perl (**push**, **unshift**, priskyrimai indeksu), JavaScript taip pat, Python ir Ruby turi kippetos dinaminius masyvus
- JavaScript elementai nebūtinai iš eilės einantys

# Įgyvendinimas 2

---

- Python, Ruby ir Lua masyvai auga tik pridedant elementus, ar naudojant apjungimą
- Ruby, Lua turi neigiamus indeksus, Python – ne.
- Kreipinys į nesantį elementą Python – vykdymo klaida, Ruby ir Lua – néra klaidos (reikšmė **nil**)

# Pradinės reikšmės

---

- C šeimos kalbos
  - `int list [] = {1, 3, 5, 7, 32}`
- C ir C++ kalbos
  - `char *names [] = {"Birus", "Takus", "Smulkus"};`
- Ada
  - `List : array (1..5) of Integer :=  
(1 => 15, 3 => 30, others => 0);`
- Python
  - **Sarašai**  
`list = [x ** 2 for x in range(14) if x % 3 == 0]`  
Kokios reikšmės bus patalpintos į list?

# Operacijos

---

- C šeima neturi operacijų masyvams
- APL turi galingas masyvų apdorojimo operacijas
- Ada leidžia masyvų priskyrimą ir apjungimą
- Perl palaiko priskyrimą, bet ne palyginimą
- Python masyvai saugo nuorodas, gali būti nevienarūšiai. Masyvų priskyrimai, bet tai tik nuorodos pakeitimas. `+`, `in`, `is` (ar du kintamieji rodo tą patį objektą?) , `==` (visų atitinkamų objektų palyginimas nurodytuose objektuose, neribojant gylio).
- Ruby taip pat nuorodos ir leidžia masyvų apjungimą, `==`
- Fortran leidžia operacijas tarp atskirų masyvų elementų
  - Pavyzdžiu, `+` operatorius tarp dviejų masyvų sukurs trečią masyvą, kur bus atitinkamų elementų sumos
  - Bibliotekinės funkcijos matricoms

# Daugiamatiški masyvai

---

- Stačiakampus (**rectangular**) masyvas yra daugiamatis masyvas, kurio eilutės yra vienodo ilgio ir stulpeliai yra vienodo ilgio
- Dantyta (**jagged**) matrica turi eilutes su kintančiu elementų skaičiumi
  - Dažniausiai, masyvų masyvai
- C, C++, C# ir Java palaiko dantytus masyvus, indeksai atskirai [2][7]
- Fortran, Ada ir C# palaiko stačiakampius masyvus, indeksai kartu [2, 7]

# Skiltys

---

- Skiltis – tai masyvo poaibis. Tai tik kreipimosi į masyvą forma
- Skiltys yra naudingos kalbose, kurios turi masyvo operacijas
- Perl, `@vector[3..6] = @vector2[3, 5, 7]`

# Skilčių pavyzdžiai

---

- Fortran

```
Integer, Dimension (10) :: Vektorius
Integer, Dimension (3, 3) :: Mat
Integer, Dimension (3, 3, 3) :: Cube
```

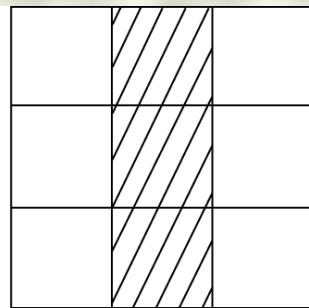
Vektorius (3:7) yra penkių elementų masyvas

- Ruby palaiko sluoksnius su slice metodu

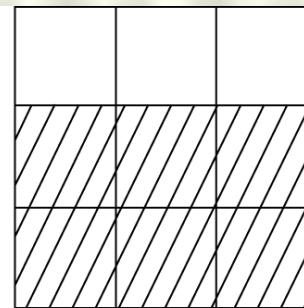
```
list.slice(2, 2) grąžina trečią ir ketvirtą list sąrašo elementus
```

```
List.slice(1..3) grąžina antrą, trečią ir ketvirtą list sąrašo elementus
```

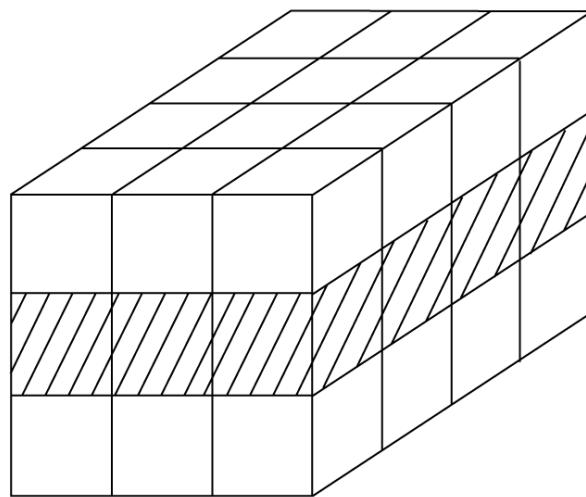
# Skiltys Fortran



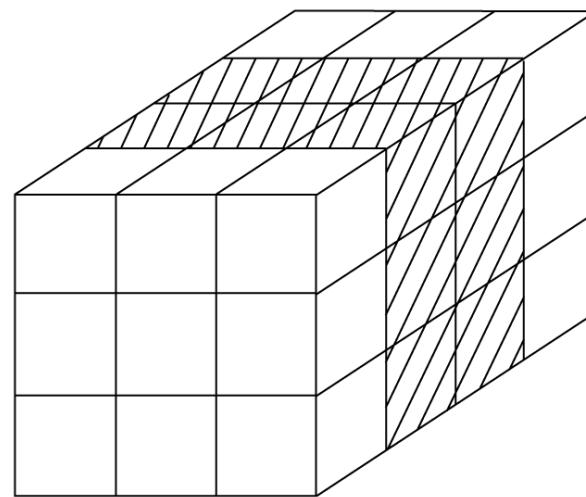
MAT (1:3, 2)



MAT (2:3, 1:3)



CUBE (2, 1:3, 1:4)



CUBE (1:3, 1:3, 2:3)

# Python

---

- Turi skiltis, taip pat skiltims galima nurodyti žingsnį. Skilties pradžios ir/ar pabaigos indeksus galima praleisti (tuomet imama nuo pradžios ir/ar iki pabaigos). Jei žingsnis -1 – sąrašas apverčiamas. Pvz.:

- ```
>>> a = range(10)
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[2:5]
[2, 3, 4]
>>> a[2:8:2]
[2, 4, 6]
>>> a[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Įgyvendinimas

- Žymiai daugiau kompiliavimo laiko pastangų
- Prieigos funkcija atvaizduoja indekso išraiškas į adresą masyve
- Prieigos funkcija vienmačiam:
$$\text{adresas}(\text{list}[k]) = \text{adresas}(\text{list}[\text{žemiausioji riba}]) + ((k - \text{žemiausioji riba}) * \text{elemento dydis})$$

Daugiamočio prieiga

- Du bendri būdai:
 - Eilučių tvarka – daugumoje kalbu
• Vieta $(a[i, j]) = \text{adresas} [\text{row_lb}, \text{col_lb}] + (((i - \text{row_lb}) * n) + (j - \text{col_lb})) * \text{elemento dydis}$
 - Stulpelių tvarka – Fortran

	1	2	...	$j-1$	j	...	n
1							
2							
:							
$i-1$							
i					\otimes		
:							
m							

Kompiliavimo laiko deskriptoriai

Array
Element type
Index type
Index lower bound
Index upper bound
Address

Vienmatis masyvas

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
:
Index range n
Address

Daugiamatis masyvas

Asociatyvūs masyvai

- Asociatyvus masyvas – tai nesutvarkytas duomenų rinkinys, kuris indeksuojamas tokiu pat reikšmių, vadintamų raktais, kiekiu
 - Turi būti saugomi vartotojo apibrėžti raktai
- Projektavimo pasirinkimai:
 - Kokia raktų forma?
 - Masyvo dydis bus statinis ar dinaminis?
- Perl, Python, Ruby ir Lua
- Bibliotekose Java, C++, C#

Asociatyvūs masyvai Perl

- Vardai prasideda (**hashes**)% ; Literalai apjungiami skliaustais

```
%Sav_sk = ("Mon" => 77, "Tue" => 79, "Wed"  
=> 65, ...);
```

- Indeksuojama naudojant riestinius skliaustus

```
$Sav_sk{ "Wed" } = 83;
```

- Elementus galima šalinti komanda **delete**

```
delete $Sav_sk{ "Tue" };
```

- Visas išvalomas

```
$Sav_sk = () ; if (exists $Sav_sk{ "Wed" })
```

- **each** operatorius ciklui, **keys** , **values** operatoriai

Asociatyvūs masyvai kitur

- Python, vadinami žodynais, reikšmės – nuorodos į objektus. Apribojimai objektui – būti *hashable*, t.y. turėti realizuotą `__hash__()` metodą, mutable (`list()` ar `dict()`) objektais raktas būti negali
- Ruby, kaip ir Python, raktas gali būti bet koks objektas, ne tik eilutė
- PHP masyvas yra surikiuotas atvaizdis (žodynas). PHP masyvai gali turėti sveikuosius ir eilučių raktus tuo pat metu; neskirsto tarp indeksuotų ir asociatyvių masyvų
- Lua lentelė – asociatyvus masyvas, kur raktas ir reikšmė gali būti bet kurio tipo ([] – masyvui, taškas – įrašams)
- Patogu paieškai, idealu, kai duomenys suporuoti

Asociatyvių įgyvendinimas

- Perl gera organizacija: greitas išrinkimas ir greitas augimas
 - 32 bitų hash reikšmė skaičiuojama kiekvienam elementui ir saugoma su elementu. Pradžioje naudojama mažai šios reikšmės bitų
 - Kai reikia didinti masyvą, hash funkcijos keisti nereikia, naudojama daugiau bitų
- PHP masyvai talpinami į atmintį, naudojant hash funkciją, tačiau visi elementai susieti jų sukūrimo eilės tvarka. Galima naudoti current ir next funkcijas

Įrašų tipai

Įrašas

- Įrašas – tai galimai heterogeninių duomenų junginys, kuriame atskiri elementai atpažįstami vardais
- Įrašai nuo COBOL laikų yra daugumoje programavimo kalbų
- Įrašai ir heterogeniniai masyvai – tas pats?
- Struktūros C++ ir C# naudojamos duomenų apjungimui
- Projektavimo pasirinkimai:
 - Kokia kreipinio į lauką forma?
 - Ar galimi elipsiniai ([elliptical](#)) kreipiniai?

Įrašai COBOL

- COBOL naudoja lygmens numerius, norėdamas parodyti įrašų gylį; kitos kalbos naudoja rekursinį aprašą

01 EMP-REC.

 02 EMP-NAME .

 05 FIRST PIC X(20) .

 05 MID PIC X(10) .

 05 LAST PIC X(20) .

 02 HOURLY-RATE PIC 99V99 .

Kreipiniai į įrašus

- Kreipiniai į įrašų laukus
 1. COBOL
Lauko_vardas OF įrašo_vardas_1 OF ... OF įrašo_vardas_n
 2. Kitos kalbos (taškas)
įrašo_vardas_1. įrašo_vardas_2. ...
 įrašo_vardas_n.lauko_vardas
- Pilni kreipiniai (**fully qualified reference**) turi turėti visų įrašų vardus
- Elipsiniai kreipiniai leidžia praleisti įrašų vardus, jei nėra dviprasmiškumo

Operacijos su įrašais

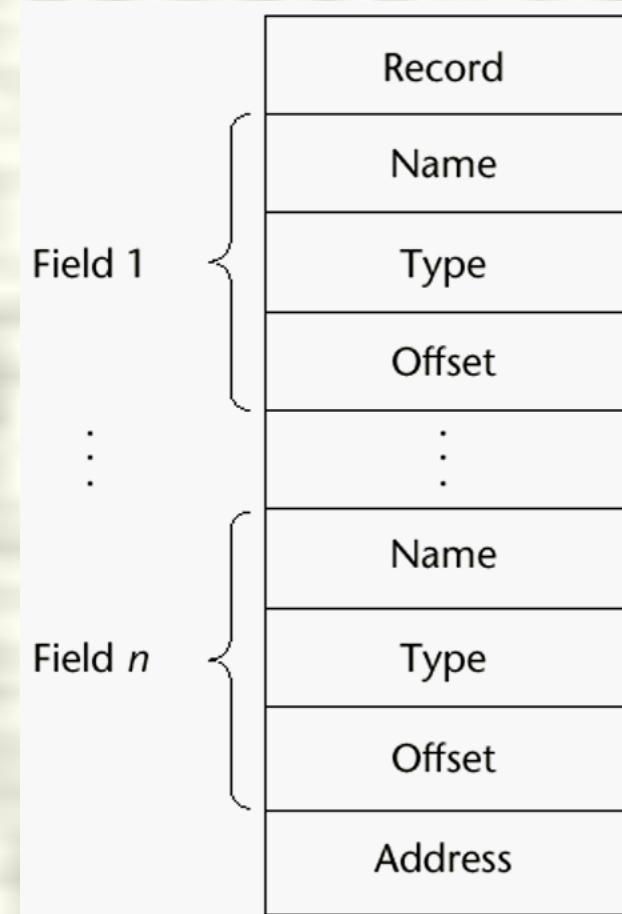
- Priskyrimas, jei tipai sutampa
- COBOL turi MOVE CORRESPONDING
 - Kopijuojant šaltinio įrašo lauką į atitinkamą taikinio lauką

Palyginimas su masyvais

- Įrašai naudojami, kai reikšmės yra skirtingų tipų
- Masyvo elementų išrinkimas daug lėtesnis, nes adresas dinaminis
- Įrašams irgi galėtų būti dinaminiai indeksai, bet nebūtų galimas tipų tikrinimas ir būtų daug lėčiau

Įrašo įgyvendinimas

Adreso poslinkis yra susiejamas su kiekvienu lauku



Unija

- Unija – tai tipas, kai kintamieji gali saugoti skirtingo tipo reikšmes skirtingais laiko momentais toje pačioje atminties vietoje
- Projektavimo pasirinkimai
 - Ar reikia tikrinti tipą (tikrinimas dinaminis)?
 - Ar unijas galima dėti į įrašus?

Laisvosios ir ne (unijos)

- **Discriminated**
- Fortran, C ir C++ nėra tipų tikrinimo, todėl unijos vadinamos laisvosiomis
- Tipų tikrinimas reikalauja, kad kiekvienas unijos tipas turėtų indikatorių, vadinamą diskriminantu
 - Palaiko Ada, įvedė Algol 68

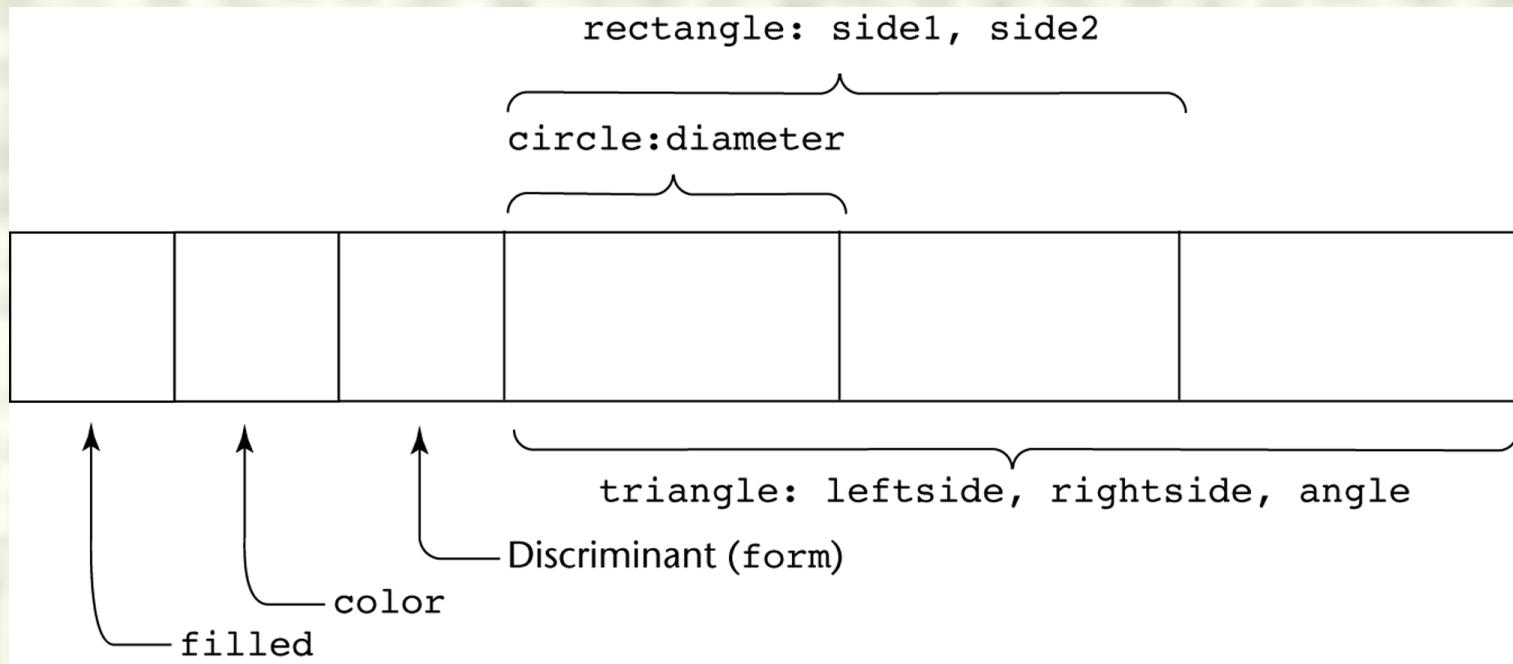
Ar kodas turi problemą?

```
union flextype {  
    int intEl;  
    float floatEl;  
};  
  
union flextype eill;  
float x;  
eill.intEl = 27;  
x = eill.floatEl;
```

Ada unija

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
    Filled: Boolean;
    Color: Colors;
    case Form is
        when Circle => Diameter: Float;
        when Triangle =>
            Leftside, Rightside: Integer;
            Angle: Float;
        when Rectangle => Side1, Side2: Integer;
    end case;
end record;
```

Ada unijos iliustracija



Trijų kintamujų unija

Ada unijos

Fig1 : Figure – neapribota, bet tipą galima pakeisti tik priskyrimo sakinio metu

Fig2 : Figure (Form => Triangle) – apribota

if (Fig1.Diameter > 3.0) ... – vykdymo metu tikrinama, ar tai Circle

F# unijos

```
type intReal =  
    | IntValue of int  
    | RealValue of float;;
```

```
let ir1 = IntValue 17;;  
let ir2 = RealValue 3.4;;
```

Unijų įvertinimas

- Laisvosios unijos (Fortran, C ir C++) yra nesaugios
 - Nėra tipų tikrinimo
- Java ir C# neturi unijos
 - Saugumo atspindys
- Ada, ML, Haskell ir F# unijos yra saugios

Santrauka

- Kalbos duomenų tipai stipriai paveikia kalbos stilių ir naudingumą
- Dauguma liepiamujų kalbų turi tokius primityvius tipus: skaitmeninis, simbolinis ir loginis
- Vartotojo išvardijami ir tipų poaibiai yra patogūs, kelia kalbos skaitomumą ir patikimumą
- Eilutės realizuojamos nevienodai kalbose
- Masyvus ir įrašus turi dauguma kalbų
- Unijos yra ne visose kalbose.

P175B124

Išraiškos ir priskyrimo sakiniai

Stay the hell out of other people's code!

Russ Olsen, author of Eloquent Ruby and Clojure developer

Tipų skyriaus klausimai

1. Kokie projektavimo pasirinkimai dėl eilučių?
2. Kokie yra projektavimo pasirinkimai dėl masyvų?
3. Kas yra heterogeninis masyvas? Kokiose kalbose jis yra įmanomas?
4. Ką vadiname dantytu masyvu?
5. Kas yra prieigos funkcija masyvui?
6. Kas yra asociatyvus masyvas ir kokie jo privalumai?
7. Kokios asociatyvaus masyvo ypatybės PHP kalboje?
8. Kokie yra daugiamaco masyvo saugojimo būdai atmintyje?
Pateikite kalbu atstovus.
9. Kuo pasižymi masyvo inicializavimas Ada kalboje?
10. Iš kokias kategorijas skirstomi masyvai pagal atmintį ir indeksų saistymą? Kiekvienai kategorijai pateikite po kalbos atstovą.

Kortežai

Kortežų tipai

- Kortežas yra duomenų tipas, panašus į įrašą, išskyrus, kad elementai neįvardinti
- Python, ML ir F# leidžia grąžinti daugybę reikšmių iš funkcijos
 - Python
 - Artimai susiję su sąrašais, bet nekeičiami, galima konvertuoti į masyvą ir atgal
 - Kortežo literalas

myTuple = (3, 5.8, 'apple')

nuoroda su indeksu [] skliaustuose (prasideda 1)

Apjungimas + ir naikinimas del. Yra ir daugiau.

Kortežai ML

```
val myTuple = (3, 5.8, 'apple');
```

Turi būti bent 2 elementai

- Prieiga:

- `#1 (myTuple)` - pirmas elementas

- Naujas kortežo tipas gali būti apibrėžtas

```
type intReal = int * real;
```

Kortežai F#

```
let tup = (3, 5, 7)
```

Jei 2 elementai, funkcijos fst ir snd

```
let a, b, c = tup
```

Priskiria kortežą kortežo rinkiniui (a, b, c)
tam atvejui, kai daugiau elementų

Kortežus turi ir C++. Tipas tuple.

Dažnai naudojamas kaip masyvo elementas

Sąrašų tipai

Sarašai

- LISP ir Scheme sarašai išskiriami skliaustais be kablelių
 $(A \ B \ C \ D)$ ir $(A \ (B \ C) \ D)$
- Duomenys ir kodas turi tą pačią formą
duomenys: $(A \ B \ C)$
kodas $(A \ B \ C)$ – funkcija A su parametrais B ir C
- Pagelbėti interpretatoriui
 $'(A \ B \ C)$ – duomenys

Sąrašai Python

- Sąrašų duomenų tipas – tai ir masyvas
- Skirtingai nuo Scheme, Common LISP, ML ir F#, Python sąrašai kintami
- Elementai bet kokio tipo tam pačiam sąraše
- Sąrašo sukūrimas su priskyrimu

```
myList = [3, 5.8, "grape"]
```

Python sąrašai

- Kreipinys su indeksais, pradedam nuo 0

`x = myList[1]` priskiria `x` 5.8

- Galima pašalinti `del`

`del myList[1]`

- Sąrašo sudarymas – remiasi aibės apibrėžimu

`[x * x for x in range(7) if x % 3 == 0]`

`range(7)` sukuria `[0, 1, 2, 3, 4, 5, 6]`

Sukonstruotas: `[0, 9, 36]`

Sarašų generatoriai

- Haskell

```
[n * n | n <- [1..10]]
```

Kamienas | patikslīintojas

- F#

```
let myArray = [|for i in 1 .. 5 -> (i * i)|];;
```

Rodyklių tipai

Rodyklės

- Rodyklės tipo reikšmės – adresai ir nil
- Dvi skirtinges paskirtys
 - Netiesioginis adresavimas
 - Dinaminės atminties valdymas
- Kupetos (**heap**) dinaminiai kintamieji – anoniminiai kintamieji
- Rodyklės – ne struktūriniai kintamieji, skiriasi ir nuo skaliaru
- Nuorodos ir reikšmės kintamieji
- Priskyrimas ir reikšmės paėmimas (**dereferencing**) – dvi pagrindinės operacijos

Projektavimo iššūkiai

- Kokia yra rodyklės galiojimo sritis ir gyvavimo laikas?
- Koks yra kupetos dinaminio kintamojo gyvavimo laikas?
- Ar rodyklės turi būti apribotos tipu, į kurio reikšmę rodo?
- Ar rodykles naudoti dinaminės atminties valdymui, netiesioginiam adresavimui ar abiems?
- Ar kalba turėtų palaikyti rodyklių tipus, nuorodų tipus, ar abu?

Rodyklių problemos 1

- Pirmoji kalba su rodyklėmis – PL/I
- Kabančios rodyklės (**dangling pointer**)
 - Rodyklė rodo į dinaminį kintamąjį, kurio atmintis buvo grąžinta
 - Sukuriamas naujas kupetas dinaminis kintamasis ir rodyklė p1 nukreipiama į jį
 - p2 priskiriama p1 reikšmė
 - Kintamojo, į kurį rodė p1, atmintis grąžinama. Jei p1 reikšmė nebuvo pakeista, abi p1 ir p2 bus kabančios
 - Išreikštas atminties grąžinimas kelia problemų

Rodyklių problemas 2

- Prarastas (**lost**) dinaminis kintamasis
 - Dinaminis kintamasis dar turi atmintį, bet jo pasiekti jau neįmanoma (**garbage** – šiukšlės)
 - Rodyklei p_1 priskiriamas naujas sukurtas dinaminis kintamasis
 - Vėliau rodyklei p_1 priskiriamas kitas naujai sukurtas dinaminis kintamasis, negrąžinus senojo atminties
 - Dinaminių kintamųjų praradimo procesas vadinamas atminties nutekėjimu (**memory leakage**)

Rodyklės C ir C++

- Labai lankstu, bet reikia naudoti atsargiai
- Gali rodyti į bet kurį kintamąjį, nesvarbu kada paskirta atminties
- Naudojama dinaminiam atminties valdymui ir adresavimui
- Galima rodyklių aritmetika
- Tiesioginis reikšmių ir adresų paėmimas
- Nebūtina nurodyti tipą (**void ***). **void *** gali rodyti į bet kurį tipą ir tipas gali būti patikrintas, bet negalima paimti reikšmės

Nuorodos tipai

- Nuorodos kintamasis yra panašus į rodyklės tipą, išskyrus vieną svarbų skirtumą, rodyklė rodo į adresą atmintyje, o nuoroda rodo į reikšmę
- C++ turi specialų rodyklės tipą, vadinamą *nuorodos tipu*, kuris pirmiausiai naudojamas formaliesiems parametrams
 - Privalumai perdavimo pagal reikšmę ir pagal nuorodą
- Perduodant parametrus su rodykle, kodas mažiau saugus ir sunkiau skaitomas
- Java išplečia C++ nuorodos kintamuosius ir jie pilnai pakeičia rodykles
- C# turi Java nuorodas ir C++ rodykles
- Smalltalk, Python, Ruby, Lua kintamieji – nuorodos

Rodyklių įvertinimas

- Kabančios rodyklės ir kabantys objektai – dinaminio atminties valdymo problema
- Rodyklės kaip ir goto, išplečia įastelių, kurias galima pasiekti, kiekj
- Rodyklės ar nuorodos yra reikalingos dinaminėms duomenų struktūroms – be jų apsieiti negalima
- Rodyklės reikalingos prietaisų tvarkyklių rašymui, specialūs absolutiniai adresai privalo būti pasiekti
- Hoare (1973): “**Their introduction into high-level languages has been a step backward from which we may never recover**”

Rodyklių įgyvendinimas

- Rodyklės ir nuorodos naudojami atminties valdymui
- Didieji kompiuteriai naudoja vieną reikšmę
- Intel mikroprocesoriai naudoja segmentą ir poslinkį. 16 bitų poros.

Kabančių rodyklių sprendimas

- *Antkapis (tombstone)*: papildoma kippetos laštelė, kuri yra rodyklė į dinaminį kintamąjį
 - Esamas rodyklės kintamasis rodo tik į antkapį
 - Kai kippetos kintamojo gražinama atmintis, antkapis lieka, bet reikšmė nulis
 - Eikvoja erdvę ir laiką, beveik nenaudojamas
- *Spynos-ir-raktai*: rodyklės reikšmės atvaizduojamos kaip pora (raktas, adresas)
 - Rodyklės reikšmės atvaizduojamos poromis (raktas, adresas). Spyna saugoma prie dinaminio kintamojo.
 - Kai priskiriama dinaminio kintamojo atmintis, sukuriama spynos reikšmė ir talpinama į spynos ir rako vietas
 - Kai gražinama dinaminio kintamojo atmintis, spynos reikšmė sunaikinama

Kupetos valdymas

- Labai sudėtingas vykdymo laiko procesas
- Vienodo dydžio lastelės ir kintamo dydžio lastelės
- Lisp: kiekviena lastelė jau turi rodykle
- Du būdai šiukšlių surinkimui
 - Nuorodų skaitliukai (nekantrus būdas – **eager**): atstatymas palaipsninis
 - Žymęti-šluoti (**mark-sweep**) (atidėtas būdas – **lazy**): atstatymas, kai trūksta atminties

Nuorodų skaitliukas

- Palaiko kiekvienai įstalelei, į kurią rodo kelios rodyklės, skaitliuką:
 - *Trūkumai*:
 - reikia erdvės, ypač, jei įstelės mažos, praradimas
 - Laiko dėl skaitliuko skaičiavimo – atidėtas perskaičiavimas
 - problemos įstelėms, sujungtoms žiedu
 - *Privalumas*: Neužima vykdymo laiko stipriai, dirba bendrame programos procese.

Žymėti ir šluoti

- Vykdymo sistema ir gražina atmintį pagal poreikį, tuomet žymėjimo ir šlavimo procesas prasideda
 1. Kiekviena kupetas įastelė turi papildomą bitą, kurį naudoja surinkimo algoritmas. Kiekviena įastelė iš pradžių priskirta šiukslėms
 2. Visų rodyklių naudotos įastelės pažymimos kaip ne šiukslės – sunkiausias
 3. Visos šiukslių įastelės gražinamos į laisvų įastelių sąrašą
 - Trūkumas: pradiniam variante vykdomas per retai. Kai vykdomas, sukeldavo ilgus užlaikymus. Šiuolaikiniai išvengia, nes vykdomi dažniau

Kintamo dydžio ląstelės

- Visi sunkumai, kurie būna vieno dydžio ląstelėms
- Naudoja dauguma programavimo kalbų
- Jei žymėti-šluoti naudojama, papildomos problemas
 - Pradinis visų ląstelių indikatorių nustatymas kupetoje yra sudėtingas. Skirtingi dydžiai – skenavimas yra problema. Dydis priekyje.
 - Žymėjimo procesas netrivialus, kaip eiti tollyn grandine, jei nėra apibrėžta rodyklė į kitą ląstelę – jei nėra vietas. Galima pridėti po rodyklę
 - Laisvos atminties sąrašo palaikymas – dar viena išlaidų sritis. Pradžioje gali būti viena ląstelė – visa atmintis. Skaidymas – blokai gali pasidaryti per maži, suliejimas, rikiavimas. Ieškoti tinkamo dydžio bloko.

Tipų tikrinimas

- Apibendrinti operatorių ir operandų koncepciją, ištraukiant paprogrames ir priskyrimus
- *Tipų tikrinimas* – tai veikla, tikrinant operatoriaus operandų tipų suderinamumą
- Suderinamas tipas – tai tipas, kuris yra teisėtas operatoriui arba pagal kalbos taisykles gali būti numanomai konvertuotas į teisętą tipą (**coercion**)
- Tipo klaida – tai operatoriaus panaudojimas netinkamo tipo operandui

Tipų tikrinimas 2

- Jei visi tipų susiejimai yra statiniai, beveik visi tipų tikrinimai gali būti statiniai
- Jei tipų susiejimai yra dinaminiai, visi tipų tikrinimai privalo būti dinaminiai (PHP, JavaScript)
- Programavimo kalba turi stiprius tipus, jei gali būti aptiktos visos tipų klaidos
- Stiprių tipų privalumas – leidžia aptikti visus netinkamus kintamujų panaudojimus, kurie pasireiškia tipo klaida

Stiprių tipų kalbos

- FORTRAN nėra: parametrai, EQUIVALENCE
- C ir C++ nėra: parametru tipų tikrinimas gali būti išvengtas; unijų tipai netikrinami
- Java ir C# yra (**type coercion**) beveik stiprios.
- ML ir F# - stiprios (**no coercion**)
- Tipų konvertavimo taisyklės stipriai paveikia stiprius tipus – gali susilpninti

Vardų tipų ekvivalencija

- Tipų suderinamumas, tipų ekvivalentiškumas
- *Vardų tipų ekvivalentiškumas* reiškia, kad du kintamieji turi ekvivalentiškus tipus, jei jie deklaruoti tuo pačiu tipu
- Lengva įgyvendinti, bet ribojantis:
 - Sveikujų poaibiai nėra ekvivalentiški su sveikaisiais tipais
 - Masyvai ar vartotojo apibrėžti tipai, šių tipų kintamieji perduodami per parametrus. Tipai turi būti apibrėžti tik kartą globaliai
 - Visi tipai turi turėti vardus. Galimi anoniminiai tipai, tuomet kompiliatorius turi suteikti vardus

Struktūrinė tipų ekvivalencija

- *Struktūrinė tipų ekvivalencija* reiškia, kad du kintamieji turi ekvivalentiškus tipus, jei jų tipų struktūros yra identiškos
- Lanksčiau, bet sunku įgyvendinti, nelengva palyginti tipų struktūrą.

Tipų ekvivalencija

- Panagrinėkime dviejų struktūrinių tipų problemą:
 - Ar du įrašų tipai ekvivalentiški, jei struktūriškai tie patys, bet naudoja skirtingus laukų vardus?
 - Ar du masyvų tipai ekvivalentiški, jei jie tokie patys, išskyrus indeksus ([1..10] ir [0..9])?
 - Ar du išvardijimo tipai ekvivalentiški, jei jų komponentai įvardinti skirtingai?
 - Su struktūrine ekvivalencija negalima atskirti tarp tos pačios struktūros tipų (skirtingi greičio matavimo vienetai, abu realūs)

Teorija ir duomenų tipai

- Tipų teorija naudoja matematikos, logikos, kompiuterių mokslo ir filosofijos žinias
- Dvi tipų teorijos šakos kompiuterių moksle
 - Praktinė – komercinių kalbų duomenų tipai
 - Abstrakti – lambda skaičiavimų tipai
- Tipų sistema – tai tipų aibė ir taisyklės, kurios juos valdo programose

Teorija ir duomenų tipai 2

- Tipų sistemos formalus modelis – tai tipų aibė ir funkcijų rinkinys, kuris apibrėžia tipų taisykles
 - Atributų gramatika arba tipų žemėlapis ir funkcijų rinkinys gali būti tipų taisyklių apibrėžimu
 - Baigtiniai saistymai (**finite mapping**) – modeliuoja masyvus ir funkcijas
 - De Carto sandaugos (**Cartesian product**) – modeliuoja įrašus ir kortežus
 - Aibių unijos – modeliuoja uniju tipus
 - Poaibiai – modeliuoja tipų poaibius

Išraiškos

Temos klausimai

- Aritmetinės išraiškos
- Užkloti operatoriai
- Tipų konversija
- Santykio ir loginės išraiškos
- Sutrumpintas įvertinimas (**short-circuit evaluation**)
- Priskyrimo sakiniai
- Mišrus (**mixed-mode**) priskyrimas

Įvadas

- Išraiškos yra pagrindinė priemonė skaičiavimų užrašymui programavimo kalboje
- Norint suprasti išraiškų įvertinimą, svarbu žinoti operatorių ir operandų įvertinimo eilę

Realizavimo problemas

- Operatorių ir operandų įvertinimo tvarka dažnai neskelbiama. Igyvendinimo pasirinkimas – skirtini rezultatai
- Tipų neatitikimas
- Tipų priverstinė konversija
- Sutrumpintas įvertinimas
- Liepiamosiose kalbose dominuoja priskyrimo sakiniai

Aritmetinės išraiškos

Aritmetinės išraiškos

- Aritmetinis įvertinimas buvo vienas iš svarbiausiųjų pirmųjų programavimo kalbų motyvų
- Aritmetinę išraišką sudaro operatoriai, operandai, skliaustai ir kreipiniai į funkcijas
- Operatoriai gali būti vienetiniai (**unary**), dvejetainiai (**binary**) ir trejetaudiniai (**ternary**)
- Dvejetainiai – infiksinis, prefiksinis (Perl) operatoriai

Projektavimo pasirinkimai

- Operatorių viršenybės (**precedence**) taisyklės
- Operatorių asociatyvumo taisyklės
- Operandų įvertinimo eilė
- Operandų įvertinimo šalutiniai efektai
- Operatorių užklojimas
- Skirtingi tipai išraiškose

Operatorių viršenybės taisyklės

- Išraiškos operatorių viršenybės taisyklės nustato eilę, kuria „gretimi“ skirtinges viršenybės operatoriai yra įvertinami
- Tipiniai viršenybės lygmenys
 - skliaustai
 - vienetiniai operatoriai (**identity**) (+ beprasmis),
 - Java ir C# konvertuoja iš short ir byte į int (neišreikštai)
 - ** (Fortran, Ruby, Ada)
 - *, /
 - +, -

Asociatyvumo taisyklės

- Išraiškos operatorių asociatyvumo taisyklės nustato eilę, kuria „gretimi“ vienodos viršenybės operatoriai yra įvertinami
- Tipinės asociatyvumo taisyklės
 - Iš kairės į dešinę, išskyrus **, kuris iš dešinės į kairę
 - Kai kada vienetiniai operatoriai iš dešinės į kairę (FORTRAN)
- APL yra skirtina; visi operatoriai turi vienodą viršenybę ir iš dešinės į kairę
- Viršenybės ir asociatyvumo taisykles galima pakeisti skliaustais

Asociatyvumas

- $a + -b + c - d$
- $a ** b ** c$ (Ada negalima, neasociatyvi operacija)
- $(a ** b) ** c$
- Kai kurios operacijos matematiškai asociatyvios
- Operacijos su realaisiais skaičiais nėra asociatyvios
- Jei asociatyvi, kompiliatorius gali pertvarkyti įvertinimo eilę dėl optimizavimo
- Sveikujų sudėtis gali būti neasociatyvi
 - $a + b + c + d$, a, c – labai didelė teigiamą, b, d – labai didelė neigiamą

Ruby išraiškos

- Visi aritmetiniai, santykio ir priskyrimo operatoriai, o taip pat masyvo indeksavimas, poslinkis ir bitinės logikos operatoriai yra įgyvendinti kaip metodai
 - Šio reiškinio išdava – visus operatorius gali užkloti taikomosios programos

Sąlyginės išraiškos

- Sąlyginė išraiška (**conditional expression**)
 - C šeimos kalbos, Perl, JavaScript, Ruby
 - Pavyzdys:

```
vidurkis = (kiekis == 0) ? 0 : suma / kiekis
```

- Skaičiuojama lyg būtų parašyta

```
if (kiekis == 0)
    vidurkis = 0
else
    vidurkis = suma / kiekis
```

Operandų įvertinimo tvarka

1. Kintamieji: atneša reikšmes iš atminties
2. Konstantos: kartais atneša iš atminties; kartais konstanta yra mašininės kalbos komandoje
3. Išraiškos su skliaustais: pirmenybinis įvertinimas
4. Įdomiausia, kai operandas – tai kreipinys į funkciją

Šalutinis efektas

- *Funkcijos šalutinis efektas:* kai funkcija keičia dviejų krypčių parametrą arba ne lokalų kintamajį
- Problema su funkcijos šalutiniais efektais:

- Kai funkcija išraiškoje keičia kitą išraiškos operandą

a = 10;

/* sakykim, kad ji keičia parametrą */

b = a + funkcija(&a);

Šalutinio efekto sprendimai

1. Kalba neturėtų leisti funkcijų šalutinio efekto

Nebūtų dviejų krypčių parametru funkcijoje

Nebūtų nelokalių nuorodų

Privalumas: veikia!

Trūkumas: nelankstumas

2. Fiksuota operandų įvertinimo seka

Trūkumas: riboja kompiliatorių optimizavimą

Java reikalauja, kad operandai turėtų būti įvertinami iš kairės į dešinę

Nuorodų skaidrumas

- Referential transparency
- Jei bet kurios dvi programos išraiškos, kurios įgyja tą pačią reikšmę, gali būti pakeistos viena iš jų, ir programos rezultatai nepakis
- Pavyzdys
 - `Re1 = (fun(a) + b) / (fun(a) - c);`
 - `temp = fun(a);`
 - `Re2 = (temp + b) / (temp - c)`
 - Jei šalutiniai efektai, nėra nuorodų skaidrumo
- Lengviau suprasti semantiką
- Atitinka matematiką

Operatorių užklojimas

- Kai operatorius naudojamas daugiau nei vienam tikslui
- Kai kas jau įprasta (+ for `int` and `float`)
- & (bitinis ir adreso) dviem skirtingiems tikslams C++
- -(minusas) – vienetinis ar dvejetainis (vienas operandas pamirštas)
- Kai kas kelia rūpesčių (* C ir C++)
 - Kompiliatorius negali aptikti klaidų (operando nebuvoimas turėtų būti aptinkama klaida)
 - Blogesnis skaitomumas

Operatorių užklojimas 2

- C++ ir C# leidžia vartotojui užkloti operatorius
- Pagalba skaitomumui, kai naudojama prasmingai
 - $A * B + C * D$ geriau, nei
 - MatrixAdd(MatrixMult(A, B), MatrixMult(C, D))
- Galimos problemas:
 - Vartotojai gali apibrėžti kokią nors nesąmonę
 - Skaitomumas gali nukentėti, net kai užklotas operatorius turi prasme

Tipų konversija

Tipų konversija

- Siaurinanti (**narrowing**) konversija – kai konvertuojama į tipą, kuris negali turėti visų pradinio tipo reikšmių (float į int)
- Platinanti (**widening**) konversija – kai konvertuojama į tipą, kuris gali bent aproksimuoti visas pradinio tipo reikšmes (int į float)

Mišrus būdas

- Mišraus tipo išraiška turi skirtingų tipų operandus
- Ne aritmetiniams operatoriams tarp skirtingų tipų
- **Coercion** – numanoma tipų konversija.
Kompiliatorius pats atlieka
- Priverstinio tipų keitimo trūkumai:
 - Sumažina kompiliatoriaus galimybes aptikti tipų klaidas
- Daugumoje kalbų visi skaitmeniniai tipai yra keičiami išraiškose, naudojant platinančią konversiją
- Ada nėra tipų konversijų išraiškose

Išreikšta tipų konversija

- Cast
- Pavyzdžiai
 - C: `(int)angle`
 - C++ `static_cast <int> angle`
 - F#: `float (sum)`

Pastaba. ML ir F# sintaksė labai panaši į funkcijos kvietimo sintakę

Klaidos išraiškose

- Priežastys
 - Prigimtiniai aritmetikos apribojimai (dalyba iš nulio)
 - Kompiuterio aritmetikos apribojimai (perpildymas) ([overflow](#), [underflow](#))
- Išimtys

Santykio ir loginės išraiškos

Santykio išraiškos

- Naudoja santykio (**relational**) operatorius ir įvairių tipų operandus
- Tampa kokia nors logine išraiška
- Naudojami operatorių simboliai įvairiose kalbose skiriasi šiek tiek (`!=`, `/=`, `~`, `.NE.`, `<>`, `#`)
- JavaScript ir PHP turi du papildomus santykio operatorius `==` ir `!=`

Panašūs į jų giminaičius `==` ir `!=`, tik nėra operandų tipų konversijos

`"7" == 7` yra tiesa

Loginės išraiškos

- Operandai ir rezultatas yra loginio (boolean) tipo
- Pavyzdžiai

FORTRAN 77	FORTRAN 90	C	Ada
.AND.	and	& &	and
.OR.	or		or
.NOT.	not	!	not xor

Santykio ir loginės išraiškos

- C89 neturi loginio tipo – ji naudoja **int** tipą, kai 0 žymi melą ir ne nulis žymi tiesą
- C išraiškų trūkumas: **a < b < c** yra galima išraiška, bet rezultatas ne tai, ko laukiame:
 - Įvertinamas kairysis operandas, rezultatas 0 arba 1
 - Gautas rezultatas lyginamas su trečiu operandu (**c**)
- Perl, Ruby turi **&&** ir **and**, **||** ir **or**. Žodinė turi žemesnį prioritetą, **&& > ||**, bet **and** ir **or** – vienodi
- Skaitomumui ir patikimumui reikalingas loginis tipas

Sutrumpintas įvertinimas

- Short circuit
- Išraiška, kai rezultatas gaunamas neįvertinant visų operandų
- Pavyzdys: $(13*a) * (b/13-1)$
Jei a yra 0, nėra prasmės įvertinti $(b/13-1)$
- Problema su nesutrumpintu įvertinimu

```
index = 0;  
while (index < length) && (LIST[index] != value)  
    index++;
```

 - Kai $index=length$, $LIST [index]$ sukels indeksavimo problemą (sakykim, kad $LIST$ turi $length$ elementų)

Sutrumpintas įvertinimas 2

- C, C++ ir Java: naudoja sutrumpintą įvertinimą loginiams operatoriams (`&&` ir `||`), bet taip pat turi bitinius operatorius, kurie sutrumpinto įvertinimo (`&` ir `|`) neatlieka
- Ruby, Perl, Python, ML ir F# visi loginiai operatoriai visada įvertinami sutrumpintai
- Sutrumpintas įvertinimas gali turėti šalutinio efekto problemų išraiškose (`a > b`) `||` (`b++ / 3`)

Priskyrimo sakiny

Priskyrimas

- Bendra sintakė
<kintamasis> <priskyrimas> <išraiška>
- Priskyrimo operatorius
 - = FORTRAN, BASIC ir C šeimos kalbose
 - := ALGOLs, Pascal, Ada
- = gali būti blogai, jei panaudoti jį palyginimui (todėl C šeimos kalbos naudoja == palyginimui)

Sąlyginis taikinys

- Perl

```
($flag ? $total : $subtotal) = 0
```

kuris yra ekvivalentus

```
if ($flag) {  
    $total = 0  
} else {  
    $subtotal = 0  
}
```

Sutrumpintas užrašas

- (compound assignment) Sutrumpintas metodas naudojamas priskyrimuose
- Pasirodė ALGOL; pritaikė C
- Pavyzdys

$a = a + b$

Rašomas kaip

$a += b$

Vienetinis operatorius

- Vienetiniai priskyrimo operatoriai C šeimos kalbose, Perl, JavaScript kombinuoja padidinimą ir mažinimą su priskyrimu
- Pavyzdžiai

sum = ++count (count **padidinamas**, priskiriamas sum)

sum = count++ (**priskiriamas** sum, count
padidinamas)

count++ (count **padidinamas**)

Priskyrimas kaip išraiška

- C, C++ ir Java priskyrimas sukuria rezultatą ir gali būti naudojamas išraiškoje
- Pavyzdys:

```
while ((ch = getchar()) != EOF) {...}
```

ch = getchar() įvykdomas; rezultatas (priskyrimas ch) yra naudojamas kaip sąlygos reikšmė while sakinyje
- Priskyrimas privalo būti su skliaustais
- Daugybė priskyrimo taikinių
- Šalutinis efektas – vykdymo tvarka
- if (x=y) in C, C++, bet ne Java, C# - leidžiama tik boolean

Sąrašų priskyrimai

- Perl ir Ruby turi sąrašinį priskyrimą

Pavyzdžiui,

```
($first, $second, $third) = (20, 30, 40);
```

- Sukeisti reikšmes

```
($first, $second) = ($second, $first);
```

- Python

```
a, b, c = (1, 2, 3)
```

```
a, b = b, a
```

Mišrus priskyrimas

- Priskyrimo sakiniai gali būti mišrūs
- Fortran, C ir C++ bet kokia skaitmeninė reikšmė gali būti priskirta bet kurio tipo skaitmeniniams kintamajam
- Python ir Ruby nėra mišrių priskyrimų, nes tipai susieti su objektais, bet ne su kintamaisiais
- Java ir C# tik platinantis priskyrimas yra įmanomas
- Ada nėra priskyrimo tipo keitimo

Santrauka

- Išraiškos
- Operatorių viršenybė ir asociatyvumas
- Operatorių užklojimas
- Mišrios išraiškos
- Įvairios priskyrimo formos