ISTRATECH

JavaCro'23

Java i Prolog:
Ima neka (polu)tajna veza

Zlatko Sirotić, univ.spec.inf.

ISTRA TECH d.o.o., Pula

ISTRA TECH d.o.o. & me

- ISTRA TECH je novo ime (od 2015.) poduzeća **Istra informatički inženjering**, osnovanog 1990. godine.
- Radim na informatičkim poslovima od 1984. godine.
- Oracle softverske alate (baza, Designer CASE, Forms 4GL, Reports, Java) koristim oko 25 godina.
- Objavljivao sam stručne radove na kongresima / konferencijama HrOUG, JavaCro, CASE, KOM, "Hotelska kuća", te u časopisima "Mreža", "InfoTrend" i "UT".
- Neka moja programska rješenja objavljivana su na web stranicama firmi Oracle i Quest.
- Vanjski sam suradnik na Fakultetu informatike Pula (FIPU) od početka rada (2011./2012.).

ISTRATECH

HrOUG / JavaCro & me

JavaCro'23

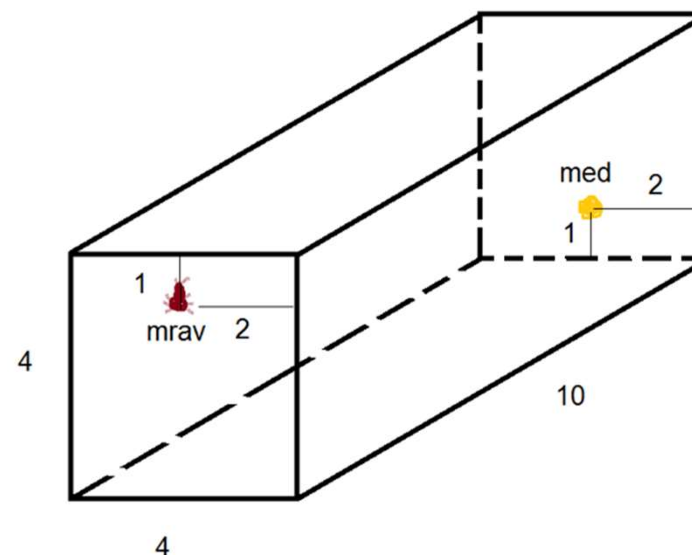
- Prvi put predavač na HrOUG 2002.
  Sudjelovao sam 18 puta (ne uključujući ovu godinu)
  i održao 24 predavanja.

- Prvi put predavač na JavaCro 2014.
  Sudjelovao sam 4 puta (ne uključujući ovu godinu)
  i održao 4 predavanja.

- 2014. Postoji li samo jedna "ispravna" arhitektura
  web poslovnih aplikacija?
  2015. Java paralelizacija
  2016. Java paralelizacija II – Streams
  2022. Kovarijanca i sedam OOPL-a

Mrav i med na prizmi (i valjku)

- Na HrOUG 2015. prvi put sam spomenuo Mrav i med na prizmi (verzija 0).



- Na HrOUG 2019. sam najavio prezentaciju verzije 1.

- Na stranici http://www.istratech.hr/mrav-i-med-na-prizmi-i-valjku-verzija-2/ nalazi se najnovija verzija Mrav i med na prizmi - verzija 2

Teme

- The Java® Virtual Machine Specification i Prolog

- Umjetna inteligencija i Prolog
- Matematička logika
- Veza matematičke logike i jezika Prolog
- Prolog primjeri, jednostavni
- Prolog program (naš) za simboličko deriviranje
  (pisan prije 30ak godina)

- Vrlo kratak pogled u
  The Java® Virtual Machine Specification, SE 21 Edition
  poglavlje
  4.10 Verification of class Files

The Java® Virtual Machine Specification

- Poznato je da su kreator Java jezika James Gosling i njegovi suradnici Bill Joy i Guy Steele, napisali i prvo izdanje **The Java® Language Specification.**
- U sljedećim izdanjima pridružili su se i drugi autori. Popis autora u Java SE 21 Edition je: James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman.
- Osim specifikacije Java jezika, postoji i **The Java® Virtual Machine Specification,** koju su prvobitno napisali Tim Lindholm i Frank Yellin.
- Popis autora u Java SE 21 Edition je: Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, Daniel Smith.

JVM Specification i Prolog

- Prolog se ne spominje u prvom izdanju (1996.) i drugom izdanju (1999.), ali u **Third Edition** (DRAFT **2009-05-12**), pod Notes on the Third Edition, Gilad Bracha napisao je:
  … The **third edition** of The Java™ Virtual Machine Specification incorporates all the changes that have been made to the specification through **Java SE 6**.
  … It was Eva Rose who, in her masters thesis, first proposed a radical revision of JVM bytecode verification.
  … Sheng Liang implemented the Java ME CLDC verifier.
  I was responsible for specifying the verifier, and Antero Taivalsaari led the overall specification of Java ME CLDC.
  …Wei Tao, together with Frank Yellin, Tim Lindholm and myself, **implemented the Prolog verifier that formed the basis for the specification in both Java ME and Java SE** ...
- Poglavlje **4. The class File Format**, sadrži (i) Prolog kod, konkretno u potpoglavlju **4.10 Verification of class Files.**

ISTRATECH

JavaCro'23

JVM Specification i Prolog

- **"Službena" treća edicija je napisana kasnije (2013., a ne 2009.), kao Java SE 7 Edition**, a u Preface piše (i):
- **The Java SE 6 platform in 2006 saw no changes to the Java programming language but an entirely new approach to bytecode verification in the JVM.**
  … Sheng Liang implemented the Java ME CLDC verifier. Antero Taivalsaari led the overall specification of Java ME CLDC and Gilad Bracha was responsible for specifying the verifier … Wei Tao, together with Frank Yellin, Tim Lindholm, and Gilad Bracha, implemented the Prolog verifier that formed the basis for the specification in both Java ME and Java SE. Wei then implemented the specification "for real" in the HotSpot JVM. Later, Mingyao Yang improved the design and specification, and implemented the final version that shipped in the Reference Implementation of Java SE 6.

JVM Specification, Tim Lindholm

- Na Amazon.com, u opisu knjige
  The Java Virtual Machine Specification,
  Java SE 7 Edition (Java Series), 3rd Edition,
  Tim Lindholm, Frank Yellin, Gilad Bracha i Alex Buckley,
  pod About the Author(s), piše (i):

  Tim Lindholm is a former Distinguished Engineer at Sun Microsystems. He was a contributor to the Java programming language and the senior architect of the Java Virtual Machine, later working on Java for mobile devices. **Prior to Sun, he worked on virtual machines and runtime systems for Prolog at Argonne National Laboratory and Quintus.**
  He holds a B.A. in Mathematics from Carleton College.

Tim Lindholm i Oracle v. Google

- https://www.theverge.com/2012/4/19/2959503/google-engineer-lindholm
  Google engineer Lindholm: 'I had little involvement in Android'

  As anticipated, today's schedule in the copyright phase of the Oracle v. Google infringement case included testimony from Tim Lindholm. **Lindholm joined Google as a software engineer in 2005 and has gained some notoriety as the author of the 2010 email allegedly attempting to convince Android chief Andy Rubin that officially licensing Java was the best solution for Android:**
  "What we've actually been asked to do (by Larry and Sergei) is to investigate what technical alternatives exist to Java for Android and Chrome. We've been over a bunch of these, and think they all suck. **We conclude that we need to negotiate a license for Java under the terms we need."**

Umjetna inteligencija i Prolog

- Primjena "umjetne inteligencije" (engl. skraćenica AI) u informatici ima dugu povijest, još od 50-ih godina 20. stoljeća.
- **Lisp** (LISt Processor) je **funkcijski programski jezik**, koji se intenzivno koristio u AI (uglavnom u SAD), a specificiran je 1958. Od jezika koji se i danas intenzivno koriste, samo Fortran je stariji (1954.-57.).
- Izvan SAD-a se u AI području uglavnom koristio **logički jezik Prolog** (PROgrammming in LOGic), specificiran 1970.
- Temelj za Prolog je (matematička) **logika prvog reda** (first-order logic; **logika sudova** je njen podskup), iako podržava i neke predikate drugog reda (npr. setof i bugof). Visoko je deklarativan (u tom smislu usporediv sa SQL-om).
- IBM-ov AI računalni sustav (hardver i softver) **Watson** pobijedio je 2011. godine ljudske prvake u kvizu "Jeopardy!". Softver je pisan većinom u jezicima C++, Java i Prolog.

Umjetna inteligencija (AI)

- Često se kaže da je **strojno učenje** (Machine Learning - ML) podskup umjetne inteligencije (Artificial Intelligence - AI).
- Naime, područja AI često se klasificiraju ovako:
  - Knowledge representation and reasoning,
  - Natural language processing,
  - **Machine Learning,**
  - Planning,
  - Multi-agent systems,
  - Computer vision,
  - Robotics,
  - Philosophical aspects.
- Pritom je **duboko učenje** (Deep Learning - DL) podskup ML-a koji koristi posebnu vrstu umjetnih neuralnih mreža (Artificial Neural Networks - ANN), - **duboke neuralne mreže** (DNN).

Umjetna inteligencija (AI)

- No duboko učenje se može promatrati i kao jedan pristup umjetnoj inteligenciji, a drugi je **Good Old-Fashioned AI** (GOFAI), kako prikazuje Skansi (2018).
- Napomena: po nekima, GOFAI se odnosi samo na ograničenu vrstu **simboličke umjetne inteligencije**, naime na agente koji se temelje na pravilima ili logičke agente.

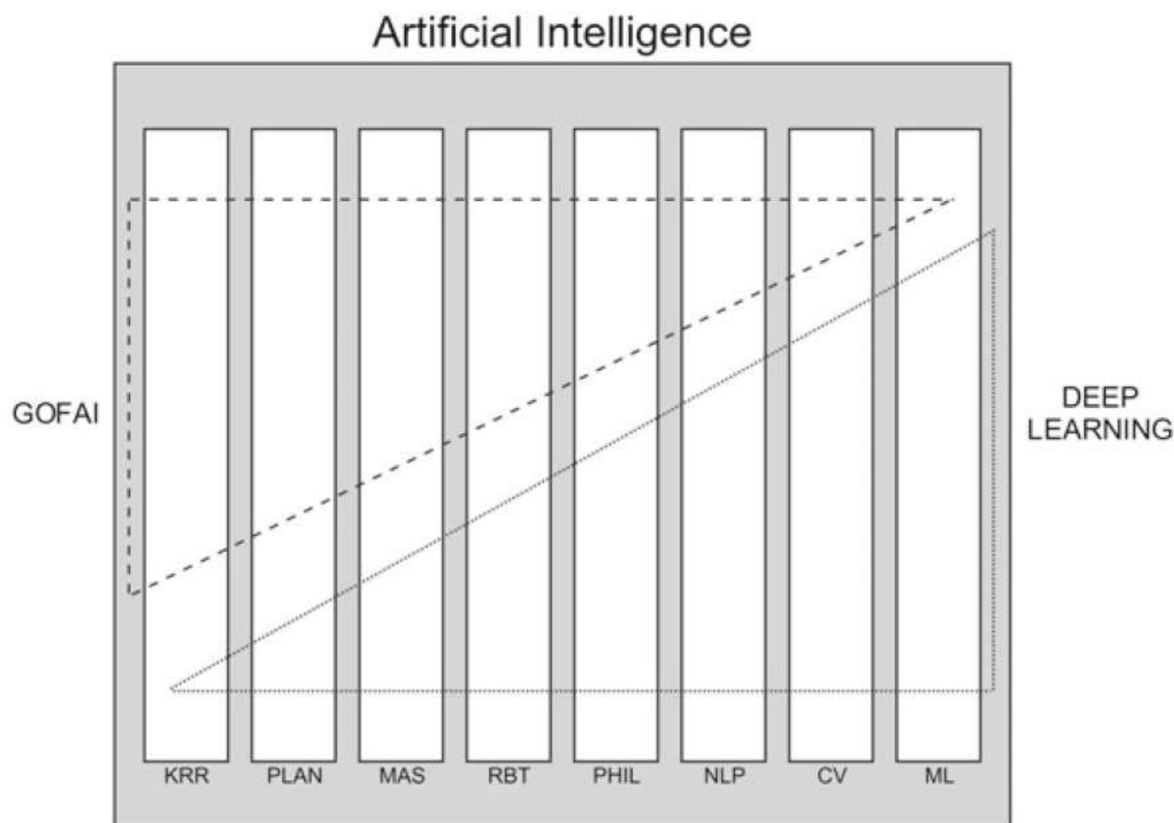


Fig. 1.1 Vertical and horizontal components of AI

Moja alternativna prezentacija

- **Paralelizacija i sedam OOPL-a**
  - kako je OpenAI translatirao moj JavaCro 2015 program u C++, Eiffel, C#, Scalu, Kotlin, Swift
- Na JavaCro 2015 održao sam prezentaciju "Java paralelizacija", na kojoj sam prikazao i dvije (svoje) verzije Java paralelnog programa programa za brojanje prostih brojeva - pomoću Java 5/6 Executora i Java 7 ForkJoin. Sljedeće godine prikazao sam "Java paralelizacija 2 - Streams", varijantu rješenja pomoću Java 8 Streams.
- Probao sam da ChatGPT translatira Java Executor program u C++, Eiffel, C#, Scalu, Kotlin, Swift.
- Za neke jezike je napravio potpuno dobar kod (čak je i poboljšao dio programa), za druge je kod trebalo samo malo popraviti, a za treće je bilo neuspješno.

Nezavisan razvoj aksiomatike i logike – do Fregea

AKSIOMATIKA

Euklid (4. st. Pr.Kr.)
V
V
Dedekind (1888.)
(tzv. Peanova aritmetika)

LOGIKA

Aristotel (4. st. Pr.Kr.)
V
V
Boole (1847.)
(matematika logike)

G.Frege (1879.)
(logika matematike)
kreirao logiku 1.reda (osnova za Prolog)

Odnos između (semantičke) valjanosti i (sintaktičke) dokazivosti (izvodljivosti) kod logičkih sustava

- Ako se u logičkom sustavu na temelju valjanosti (svake) tvrdnje može pokazati njena dokazivost (izvodljivost iz aksioma), onda je takav sustav **(semantički) potpun**:

  Ako iz ⊨ **A** (A je valjana) slijedi ⊢ **A** (A je dokaziva), logički sustav je **(semantički) potpun**.
  Napomena: operatori ⊨ i ⊢ su **metalogički**, a ne logički.

- Ako se u logičkom sustavu mogu dokazati samo valjane tvrdnje, onda je takav sustav **pouzdan** (korektan):

  Ako iz ⊢ **A** (A je dokaziva) slijedi ⊨ **A** (A je valjana), logički sustav je **pouzdan**.

Usporedba osobina logike sudova, logike 1. reda i aritmetike

Logički / deduktivni sustav	**Pouzdan / konzistentan**	**Potpun**	**Odlučiv**
Logika sudova	DA (pouzdan)	DA (semantički) (Post, 1921.)	DA
Logika prvog reda	DA (pouzdan)	DA (semantički) (Gödel, 1929.)	**NE** Churchov teorem (Church, Turing, 1936.) = rješenje za Hilbertov Entscheidungs-problem (1928.)
Aritmetika	DA (konzistentan) (Gentzen, 1936., nefinitnim metodama); **NE može se dokazati unutar nje same** (Gödel, 1931.)	**NE** (sintaktički) (Gödel, 1931.)	**NE** Churchov teorem

Logika sudova (Propositional Logic)

- **Operatori (veznici)** logike sudova (**račun/algebra sudova**):

negacija (negation) ¬

disjunkcija (disjunction) ∨ konjunkcija (conjunction) ∧

implikacija (implication) → ekvivalencija (equivalence) ↔

ekskluzivno ili (exclusive or) ⊕ nili (nor) ↓ ni (nand) ↑

- Primjer formule logike sudova:
  $(p \to q) \leftrightarrow (\neg q \to \neg p)$

- Napomena: mala slova (npr. p, q) označavaju **atomarne (atomic) sudove** ili atome.

- Jedna interpretacija (u prirodnom jeziku):
  ako **pada kiša** (p), tada **trava je mokra** (q)
  ekvivalentno je
  ako **trava nije mokra** (¬q), tada **ne pada kiša** (¬p).

Logika sudova (Propositional Logic)

- Prikaz nekih logički ekvivalentnih formula. Operator ≡ je **metalogički** operator, tj. predstavlja metalogičku ekvivalenciju, dok je ↔ logički operator:

A ↔ B ≡ (A → B) ∧ (B → A)

A ⊕ B ≡ ¬(A → B) ∨ ¬(B → A)

A → B ≡ ¬A ∨ B

A → B ≡ ¬(A ∧ ¬B)

A ∨ B ≡ ¬(¬A ∧ ¬B)

A ∧ B ≡ ¬(¬A ∨ ¬B)

A ∨ B ≡ ¬A → B

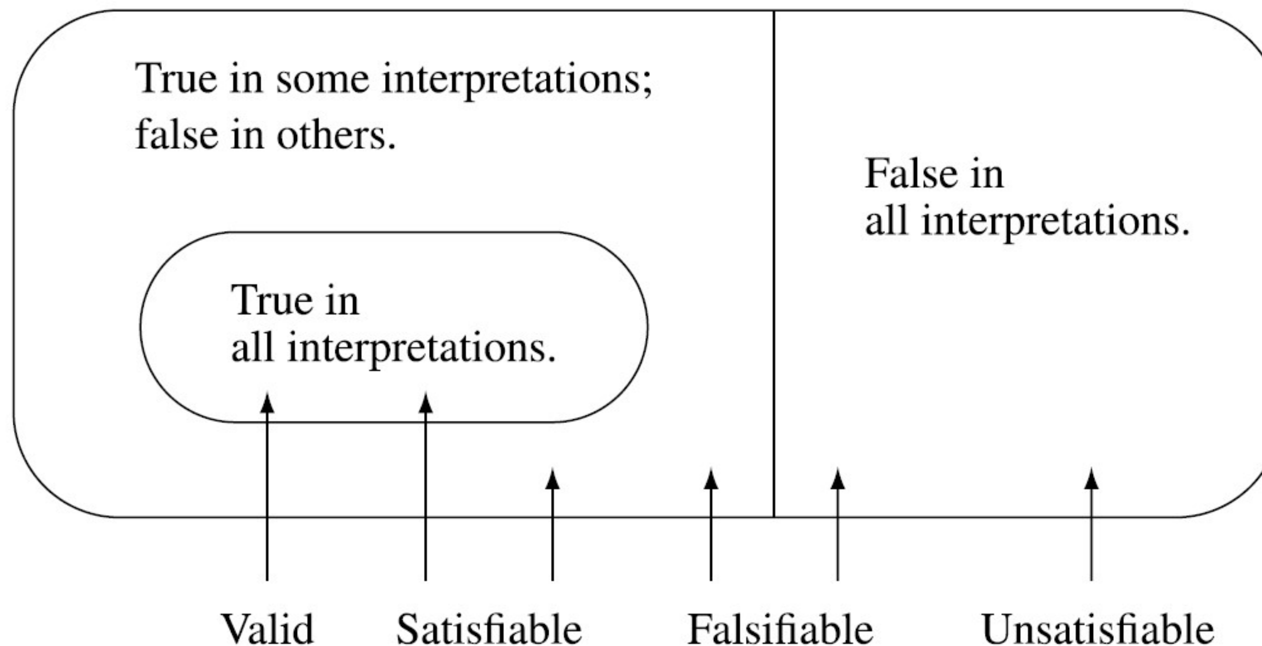
A ∧ B ≡ ¬(A → ¬B)

- Napomena: velika slova (npr. A, B) predstavljaju **logičke varijable**, koje označavaju bilo koju logičku formulu.
- Iz prethodnog se vidi da nisu nužni svi operatori. **Dovoljna su dva: ¬ i jedan od ∨ ili ∧. Pa čak i samo jedan, ↓ ili ↑.**

Logika sudova (Propositional Logic)

- Sa **semantičkog** stajališta, formula može biti:

- **zadovoljiva** (satisfiable): istinita u barem jednoj interpretaciji
- **valjana** (valid): istinita u svim interpretacijama (**tautologija**); označava se s $\models$ **A**
- **nezadovoljiva** (unsatisfiable): lažna u svim interpretacijama (što znači da je njena negacija valjana!)
- **oboriva** (falsifiable): lažna u barem jednoj interpretaciji.



Logika sudova (Propositional Logic)

- Logika sudova (za razliku od složenije logike prvog reda) ima **proceduru odlučivanja** (engl. decision procedure), tj. algoritam koji u konačnom vremenu vraća odgovor da li je formula valjana (ili da li je zadovoljiva).
- Jedna procedura odlučivanja je izrada **tablice istinitosti (semantička tablica)**.
- Sljedeća tablica istinitosti pokazuje da je (prije prikazana) formula logike sudova $(p \to q) \leftrightarrow (\neg q \to \neg p)$ valjana:

p	q	$(p$	$\to$	$q)$	$\leftrightarrow$	$(\neg$	q	$\to$	$\neg$	$p)$
T	T	T	T	T	T	F	T	T	F	T
T	F	T	F	F	T	T	F	F	F	T
F	T	F	T	T	T	F	T	T	T	F
F	F	F	T	F	T	T	F	T	T	F

Logika prvog reda (First-Order Logic)

- Logika prvog reda (**logika predikata, ili račun predikata**) je nadskup logike sudova, i dodaje joj sljedeće:
  - **predikate**: n-arni predikat se interpretira kao n-arna relacija nad određenom domenom
  - **varijable i konstante** (iz određene domene), koje čine argumente predikata
  - **univerzalni kvantifikator** (universal quantifier) ∀ (čita se "za svaki")
  - **egzistencijalni kvantifikator** (existential quantifier) ∃ (čita se "postoji").
- **LPR s funkcijama** dodaje još **funkcije**, tako da argumenti u predikatima mogu biti i funkcije.
- Kod LPR, **kvantifikatori se primjenjuju samo na varijable**, a ne npr. na predikate (to je logika drugog reda).

Logika prvog reda (First-Order Logic)

- Primjer jednostavne formule (ovaj primjer nema funkcije):
  ∀x p(a, x) čita se "za svaki x vrijedi p(a, x)"
  (x je varijabla, p je dvomjesni predikat, a je konstanta).
- Jedna interpretacija, kod koje je formula istinita:
  I1 = (N , {≤}, {0}) - domena je N, tj. skup prirodnih brojeva; predikat *p* se zamjenjuje relacijom ≤, konstanta *a* s nulom, pa za svaki x element od N **vrijedi** da 0 ≤ x.
- Jedna interpretacija, kod koje formula nije istinita:
  I1 = (Z , {≤}, {0}) - domena je Z, tj. skup cijelih brojeva (negativni cijeli brojevi, nula i pozitivni cijeli brojevi); predikat *p* se zamjenjuje relacijom ≤, konstanta *a* s nulom, pa **ne vrijedi** za svaki x element od Z da 0 ≤ x
  (ne vrijedi za negativne brojeve).
- **LPR nema proceduru odlučivanja, ali je poluodlučiva** (procedura odlučivanja može ne završiti).

ISTRATECH JavaCro'23

Preneksna konjunktivna normalna forma (PCNF)

- Formula u preneksnoj konjunktivnoj normalnoj formi (prenex conjunctive normal form, PCNF) ima oblik:
  $Q_1\ x_1\ \ldots Q_n\ x_n\ M$
  gdje su Q_i kvantifikatori (∀ ili ∃),
  a M je matrica bez kvantifikatora,
  i M se nalazi u CNF formi,
  tj. M ima oblik konjunkcije disjunkcija, npr. (A ∨ B) ∧ (C ∨ D).

- Primjer formule u PCNF:
  ∀y ∀z ([p(f(y)) ∨ ¬p(g(z)) ∨ q(z)]
  ∧ [¬q(z)   ∨ ¬p(g(z)) ∨ q(y)])

- **Svaka formula logike prvog reda može se pretvoriti u logički ekvivalentnu formulu u PCNF obliku.**

Skolemizacija (po logičaru Skolemu)

- Skolemizacija je preoblikovanje formule iz PCNF oblika **u oblik bez egzistencijalnih kvantifikatora** (uvođenjem tzv. Skolemovih funkcija).
- **Skolemizacijom se ne dobiva logički ekvivalentna formula, ali su polazna i Skolemizirana formula isto zadovoljive.**
- Npr. formula koja je u PCNF obliku:
  $\exists x \exists y \forall z((p(x) \vee \neg p(y) \vee q(z)) \wedge (\neg q(x) \vee \neg p(y) \vee q(z)))$
  Skolemizira se u oblik:
  $\forall z((p(a) \vee \neg p(b) \vee q(z)) \wedge (\neg q(a) \vee \neg p(b) \vee q(z)))$
  gdje su a i b Skolemove funkcije (u konkretnom slučaju su to konstante), koje (Skolemove funkcije) odgovaraju egzistencijalno kvantificiranim varijablama x i y.
- Dodatno se može eliminirati univerzalni kvantifikator i matrica napisati u (logički ekvivalentnom) **klauzalnom obliku**:
  $\{ \{p(a), \neg p(b), q(z)\}, \{\neg q(a), \neg p(b), q(z)\} \}$

Rezolucija

- Algoritam (proceduru, metodu) rezolucije (resolution) kreirao je John Alan **Robinson, 1965.** (na temelju onoga što su napravili Davis i Putnam, 1960.).
- Algoritam rezolucije je **u logici sudova** pouzdan (sound) i kompletan (complete), te je on i procedura odlučivanja za nezadovoljivost formule (u klauzalnom obliku).
- **U logici prvog reda, algoritam rezolucije je i dalje pouzdan i potpun**, **ali nije procedura odlučivanja (nego poluodlučivanja), jer algoritam može ne završiti.**
- **Algoritam opće rezolucije** (general resolution) se temelji na dva "podalgoritma":
  - **temeljnoj rezoluciji** (ground resolution), koja se primjenjuje nad **temeljnim klauzulama** logike prvog reda, kao da su to sudovi logike sudova
  - **unifikaciji** (unification).

Rezolucija

- Kod **temeljne rezolucije** se iz dvije temeljne klauzule **eliminiraju (međusobno) suprotne klauzule** (clashing clauses) i dobiva se jedna temeljna klauzula – **rezolventa**.
  Primjer:
  {q(f(b)), r(a, f(b))} i
  {p(a), ¬q(f(b)), r(f(a), b)} daje
  {r(a, f(b)), p(a), r(f(a), b)}
- **Unifikacijom** se "skoro suprotne klauzule" pokušavaju učiniti ("pravim") suprotnim klauzulama, primjenom **supstitucije.**
  Primjer: **p(f(x), g(y))** i **¬p(f(f(a)), g(z))**
  nakon primjene **supstitucije {x ← f(a), y ← z}**
  postaju ("prave") suprotne klauzule
  p(f(f(a)), g(z))
  ¬p(f(f(a)), g(z))
  koje se mogu međusobno poništiti.

Rezolucija

- Još jedan primjer opće rezolucije.
- Dvije temeljne klauzule:
  {p(f(x), g(y)), q(x, y)} i
  {¬p(f(f(a)), g(z)), q(f(a), z)}
  imaju "skoro suprotne klauzule (označene zelenom bojom).
- One se supstitucijom **{x ← f(a), y ← z}**
  pretvaraju u ("prave") suprotne klauzule, pa se nad temeljnim klauzulama može primijeniti temeljna rezolucija.
- Time se dobije temeljna klauzula-rezolventa:
  {q(f(a), z), q(f(a), z)}
  ili jednostavnije
  {q(f(a), z)}

Logičko programiranje

- Rezolucija je izvorno kreirana kao metoda (algoritam) za automatsko dokazivanje teorema (automatic theorem proving).
- Kasnije se otkrilo da reducirana varijanta rezolucije (npr. SLD rezolucija) može poslužiti za programiranje. Taj pristup zove se **logičko programiranje**.
- Program se izražava kao skup logičkih klauzula. Upit je klauzula koja se dodaje tom skupu.
- **Tehnički gledano, upit predstavlja negaciju tvrdnje koju želimo dokazati. Ako upit ne uspije, ta tvrdnja je dokazana (dokazivanje opovrgavanjem).**
- **"Usput" se dobiju (na temelju unifikacije) i rezultati programa - to čini (praktičnu) razliku između dokazivanja teorema i logičkog programiranja.**

Hornove klauzule (Horn clauses)

- Hornove klauzule su posebna vrsta logičkih klauzula, oblika:
  $A \leftarrow B_1, \ldots, B_n \equiv A \leftarrow B_1 \wedge \ldots \wedge B_n \equiv A \vee \neg B_1 \vee \ldots \vee \neg B_n$
  tj. imaju **najviše jedan pozitivan literal** (ovdje je to A).
- Kako se vidi, kod logičkog programiranja preferira se korištenje **operatora reverzne implikacije** ←, umjesto →.
- **Definitne klauzule** imaju **točno jedan** pozitivan literal.
- Literal A je **glava** (head), literali B_i čine **tijelo** (body) klauzule.
- Koristi se i sljedeća terminologija:
  - **činjenica** je Hornova klauzula bez tijela: $A \equiv A \leftarrow True$
  - **cilj ili upit** je klauzula bez glave: $\leftarrow B_1, \ldots, B_n$
  - **pravilo zaključivanja** ima oboje: $A \leftarrow B_1, \ldots, B_n$
- Operator ← ima deklarativnu i proceduralnu interpretaciju:
  deklarativno: A je istina ako su istina B_1 i ... i B_n
  proceduralno: da bi izvršio zadatak A, izvrši B_1 i ... i B_n

SLD rezolucija

- **Selective Linear Definite clause resolution** (linearna rezolucija za jezik definitnih klauzula s funkcijom izbora) je osnovna metoda zaključivanja u logičkom programiranju.
- Kao i opća metoda rezolucije, i ona je pouzdana (sound) i kompletna za pobijanje (refutation complete), **ali samo ako se primjenjuje na Hornove klauzule**.
- SLD rezoluciju čini sekvenca koraka rezolucije između klauzule-cilja (upita) i programske klauzule.
- SLD rezoluciju određuju pravilo za biranje literala u upitu - **pravilo računanja** (computation rule), i pravilo za biranje određene programske klazule - **pravilo pretraživanja** (search rule).
- **SLD rezolucija je jako ovisna o odabranim pravilima računanja i (naročito) pretraživanja**. Čak i kada postoji korektan odgovor, SLD rezolucija može ne završiti, ili završiti bez nalaženja odgovora (ovisno o tim pravilima).

Primjer logičkog programa u obliku Hornovih klauzula i SLD rezolucije

```
 1. ancestor(x, y) ← parent(x, y)
 2. ancestor(x, y) ← parent(x, z), ancestor(z, y)

 3. parent(bob, allen)
 4. parent(catherine, allen)
 5. parent(dave, bob)
 6. parent(ellen, bob)
 7. parent(fred, dave)
 8. parent(harry, george)
 9. parent(ida, george)
10. parent(joe, harry)
```

Primjer logičkog programa u obliku Hornovih klauzula i SLD rezolucije

- Pogledajmo primjer zaključivanja nad prethodnim programom (pod rednim brojem 11. je polazni cilj):

```
11. ← ancestor(y, bob), ancestor(bob, z)
12. ← parent(y, bob), ancestor(bob, z)     1,11 {y←dave}
13. ← ancestor(bob, z)                     5,12
14. ← parent(bob, z)                       1,13 {z←allen}
15. <prazna klauzula>                      3,14
```

- **Nađen je jedan točan odgovor (od više): y=dave i z=allen.**
- U ovom slučaju su primijenjena ova **pravila**:
  - **računanja**: uvjeti u upitu biraju se **s lijeva na desno**
  - **pretraživanja**: klauzule se pretražuju **od vrha prema dnu**.
- Da su primijenjena druga pravila, moglo se desiti da **program nikad ne završi, ili ne nađe točan odgovor.**

Prolog program (ekvivalentan prethodnom)

- **Program** je skup klauzula oblika **glava :- tijelo.** Koristi se **:-** umjesto ← i obavezna je točka na kraju.
- **Klauzula-pravilo** (rule clause) ima oba dijela, **klauzula-činjenica** (fact) ima samo glavu, **klauzula-cilj ili upit** (goal) ima samo tijelo.
- **Varijable** se u Prologu pišu velikim početnim slovima.
- **Proceduru** čine klauzule s istom glavom (ovdje su dvije):

```
ancestor(X, Y) :- parent(X, Y). % X je predak od Y
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
parent(bob, allen). % bob je allenov roditelj
parent(catherine, allen).
parent(dave, bob).
parent(ellen, bob).
parent(fred, dave).
parent(harry, george).
parent(ida, george).
parent(joe, harry).
```

Prolog program (ekvivalentan prethodnom logičkom programu)

- Primjeri postavljanja upita (klauzula-ciljeva):

```
?- parent(bob, allen). % da li je bob alenov roditelj?
true.
?- ancestor(allen, bob). % da li je allen bobov predak?
false.

?- parent(X, allen). % tko je allenov roditelj?
X = bob ; % znak ; mi kucamo - tražimo sljedeći odgovor
X = catherine. % točka - sustav zna da više nema odgovora

?- ancestor(ellen, X). % Kome je ellen predak?
X = bob ;
X = allen ; % sustav još ne zna da nema više odgovora
false.
```

Način rada Prolog programa

- Prolog sustav koristi SLD rezoluciju (SLDNF varijantu).
- **Pravilo računanja** (computation rule):
  uvjeti u upitu biraju se s lijeva na desno.
- **Pravilo pretraživanja** (search rule):
  klauzule se pretražuju od vrha prema dnu procedure.
- Primjer: upit ?- ancestor(Y, bob), ancestor(bob, Z).
  će uspjeti i vratiti supstituciju Y = dave, Z = allen.
- Prolog sustav će to raditi ovako:

```
:- ancestor(Y, bob), ancestor(bob, Z).
:- parent(Y, bob), ancestor(bob, Z).{Y <- dave}
:- ancestor(bob, Z).
:- parent(bob, Z). {Z <-allen}
```

- Ako želimo (označavamo sa ;) Prolog daje i ostale odgovore:

```
Y = ellen, Z = allen ;
Y = fred, Z = allen ;
false.
```

ISTRATECH

JavaCro'23

Prolog program može ne završiti (zapravo, završi sa: out of local stack)

- Pravilo računanja s lijeva na desno (i to naročito kad imamo lijevu rekurziju) i pravilo pretraživanja od vrha prema dnu, mogu uzrokovati nezavršavanje Prolog programa - kada ga prikažemo kao stablo, program tada ima beskonačne grane. To je svojstveno bilo kojoj SLD rezoluciji s takvim pravilima.
- Prolog sustav radi tako da pretražuje prvo u dubinu (depth-first), a ne prvo u širinu (breadth first), pa neće naći rješenje koje se nalazi desno od beskonačne grane (na stablu).
- Dodatno, Prolog program može neuspješno završiti zato što sustav (zbog performansi) izostavlja jednu od kontrola algoritma za SLD rezoluciju, kontrolu javljanja (occurs-check).
- Zato su Prolog programi vrlo osjetljivi na redoslijed klauzula i redoslijed uvjeta (unutar klauzule-pravila ili klauzule-cilja; klauzule-činjenice nisu problematične, jer nemaju uvjeta).

Prolog primjenjuje SLDNF varijantu SLD rezolucije

- Prolog primjenjuje **SLDNF** rezoluciju (Negation as Failure), kod koje se **negacija shvaća kao konačan neuspjeh zadovoljavanja cilja**.
- Negacija se označava pomoću operatora **not**.
- **NF ponekad može dati nelogične odgovore**:

```
lijepo_pjeva(X) :- not(kresti(X)), ptica(X).
ptica(svraka).
ptica(slavuj).
kresti(svraka).
```

- Logičan odgovor na pitanje da li slavuj lijepo pjeva:

```
?- lijepo_pjeva(slavuj).
true.
```

- Nelogičan odgovor na pitanje da li neka ptica lijepo pjeva:

```
?- lijepo_pjeva(X).
false.
```

ISTRATECH JavaCro'23

Prolog primjenjuje SLDNF varijantu SLD rezolucije

- Prolog radi na temelju tzv. **pretpostavke zatvorenog svijeta** (closed world assumption – CWA) :
  ako se nešto ne može izvesti iz baze znanja, to je lažno.
- Sljedeći primjer, u kojem je zamijenjen redoslijed uvjeta u klauzuli lijepo_pjeva, daje logične odgovore:

```
lijepo_pjeva(X) :- ptica(X), not(kresti(X)).
ptica(svraka).
ptica(slavuj).
kresti(svraka).
```

- Logičan odgovor na pitanje da li slavuj lijepo pjeva:

```
?- lijepo_pjeva(slavuj).
true.
```

- Logičan odgovor na pitanje da li neka ptica lijepo pjeva:

```
?- lijepo_pjeva(X).
X = slavuj.
```

Prolog program za (simboličko) deriviranje

```
% definira ^ kao infiksni operator (xfy), prioriteta 100
:- op(100, xfy, [^]).

% glavna procedura
deriviraj(F, X, R) :- d(F, X, R1), sredi(R1, R).

% ! označava rez: reže klauzule (iste procedure)
% ispod one u kojoj se nalazi, i utječe da se
% ciljevi lijevo od njega zadovolje samo jednom

% derivacija konstante
d(Const, X, 0) :- atomic(Const), !.

% derivacija varijable (X)
d(F, X, 1) :- F == X, !.
```

ISTRATECH

JavaCro'23

Prolog program za (simboličko) deriviranje

```
d(ln(F), X, R) :-
    d(F, X, R1),
    R = R1 / F, !.

d(sin(F), X, R) :-
    d(F, X, R1),
    R = cos(F) * R1, !.

d(cos(F), X, R) :-
    d(F, X, R1),
    R = -sin(F) * R1, !.

d(tan(F), X, R) :-
    R1 = sin(F) / cos(F),
    d(R1, X, R), !.
```

Prolog program za (simboličko) deriviranje

```
d(asin(F), X, R) :-
    d(F, X, R1),
    R = 1 / (1 - F ^ 2) ^ 0.5 * R1, !.

d(acos(F), X, R) :-
    d(F, X, R1),
    R = -1 / (1 - F ^ 2) ^ 0.5 * R1, !.

d(atan(F), X, R) :-
    d(F, X, R1),
    R = 1 / (1 + F ^ 2) * R1, !.
```

Prolog program za (simboličko) deriviranje

```
d(-F, X, R) :-
   d(F, X, R1),
   R = -R1, !.

d(F1 + F2, X, R) :-
   d(F1, X, R1),
   d(F2, X, R2),
   R = R1 + R2, !.

d(F1 - F2, X, R) :-
   d(F1, X, R1),
   d(F2, X, R2),
   R = R1 - R2, !.
```

Prolog program za (simboličko) deriviranje

```
d(F1 * F2, X, R) :-
   d(F1, X, R1),
   d(F2, X, R2),
   R = R1 * F2 + F1 * R2, !.

d(F1 / F2, X, R) :-
   d(F1, X, R1),
   d(F2, X, R2),
   R = (R1 * F2 - F1 * R2) / F2 ^ 2, !.

% derivacija potencije
d(F ^ S, X, R) :-
   atomic(S),
   d(F, X, R1),
   R = S * F ^ (S - 1) * R1, !.
```

Prolog program za (simboličko) deriviranje

```
% logaritamsko deriviranje
d(F1 ^ F2, X, R) :-
    d(F1, X, R1),
    d(F2, X, R2),
    R = F1 ^ F2 * (F2 / F1 * R1 + R2 * ln(F1)), !.
```

- Pokazuje se da je sređivanje deriviranog izraza skoro kompleksnije nego samo deriviranje!
- Zato ne navodimo (naš) programski kod za tu namjenu, koji uspijeva srediti neke izraze, ali moguće je da dođe ne samo do "ružnog", već i do netočnog "sređivanja".

JVM Specification i Prolog

- Prolog kod nalazi se, od prve edicije u kojoj se pojavio (tj. od The Java® Virtual Machine Specification, Java SE 7 Edition), pa do najnovije edicije (Java SE 21 Edition), unutar poglavlja **4. The class File Format** (koje ima oko 300 stranica, od ukupno oko 600).

- Preciznije, nalazi se u potpoglavlju:
  4.10 Verification of class Files (oko 165 stranica),
  koje (potpoglavlje) se sastoji od

  4.10.1 Verification by Type Checking (oko 155 stranica) i
  4.10.2 Verification by Type Inference.

4.10 Verification of class Files

- Even though a compiler for the Java programming language must only produce class files that satisfy all the static and structural constraints in the previous sections, the Java Virtual Machine **has no guarantee that any file it is asked to load was generated by that compiler or is properly formed**.
- Applications such as web browsers do not download source code, which they then compile; these applications download already-compiled class files. The browser needs to determine whether the class file was produced by a trustworthy compiler or by an adversary attempting to exploit the JVM.
- Because of these potential problems, the Java Virtual Machine needs to verify for itself that the desired constraints are satisfied by the class files it attempts to incorporate. **A Java Virtual Machine implementation verifies that each class file satisfies the necessary constraints at linking time** (§5.4).

4.10 Verification of class Files

- **Link-time verification enhances the performance of the run-time interpreter.** Expensive checks that would otherwise have to be performed to verify constraints at run time for each interpreted instruction can be eliminated. The Java Virtual Machine can assume that these checks have already been performed.

- **There are two strategies that Java Virtual Machine implementations may use for verification.**
- **Verification by type checking** must be used to verify class files whose version number is greater than or equal to 50.0.
- **Verification by type inference** must be supported by all Java Virtual Machine implementations, except those conforming to the Java ME CLDC and Java Card profiles, in order to verify class files whose version number is less than 50.0. **(SE 5)**

ISTRATECH JavaCro'23

4.10.1 Verification by Type Checking

- A class is type safe if all its methods are type safe, and it does not subclass a final class.

```
classIsTypeSafe(Class) :-
  classClassName(Class, Name),
  classDefiningLoader(Class, L),
  superclassChain(Name, L, Chain),
  Chain \= [],
  classSuperClassName(Class, SuperclassName),
  loadedClass(SuperclassName, L, Superclass),
  classIsNotFinal(Superclass),
  classMethods(Class, Methods),
  checklist(methodIsTypeSafe(Class), Methods).
classIsTypeSafe(Class) :-
  classClassName(Class, 'java/lang/Object'),
  classDefiningLoader(Class, L),
  isBootstrapLoader(L),
  classMethods(Class, Methods),
  checklist(methodIsTypeSafe(Class), Methods).
```

ISTRATECH JavaCro'23

4.10.1.1 - 4.10.1.9

- 4.10.1.1 Accessors for Java Virtual Machine Artifacts
- 4.10.1.2 Verification Type System
- 4.10.1.3 Instruction Representation
- 4.10.1.4 Stack Map Frames and Type Transitions
- 4.10.1.5 Type Checking Abstract and Native Methods
- 4.10.1.6 Type Checking Methods with Code
- 4.10.1.7 Type Checking Load and Store Instructions
- 4.10.1.8 Type Checking for protected Members
- 4.10.1.9 Type Checking Instructions

ISTRATECH JavaCro'23

4.10.1.6 Type Checking Methods with Code

- Non-abstract, non-native methods are type correct if they have code and the code is type correct.

```
methodIsTypeSafe(Class, Method) :-
  doesNotOverrideFinalMethod(Class, Method),
  methodAccessFlags(Method, AccessFlags),
  methodAttributes(Method, Attributes),
  notMember(native, AccessFlags),
  notMember(abstract, AccessFlags),
  member(attribute('Code', _), Attributes),
  methodWithCodeIsTypeSafe(Class, Method).
```

Zaključak

- Sada je "in" **multiparadigmatsko programiranje**, tj. istovremeno korištenje više programskih jezika koji pripadaju različitim paradigmama (objektnoj, funkcijskoj, logičkoj ...), ili korištenje programskih jezika koji pokrivaju više paradigmi.
- Najčešće se koristi mješavina objektnih i funkcijskih paradigmi. **No logička paradigma je i dalje zanimljiva**, naročito za određene klase programskih problema.
- Često se kaže da je strojno učenje podskup umjetne inteligencije (AI). No strojno učenje (ML), posebno duboko učenje (DL) kao podskup ML, može se promatrati i kao drugi pristup umjetnoj inteligenciji – jedan pristup je logički ili Good Old-Fashioned AI (GOFAI) pristup, a drugi je DL, koji je danas u velikom zamahu. **Ali, ne treba odbaciti niti logički (GOFAI) pristup, za koji je Prolog vrlo značajan.**
- **Kako smo vidjeli, Prolog je poslužio i za JVM specifikaciju - Verification by Type Checking.**

ISTRATECH

JavaCro'23

Literatura (dio)

- Ben-Ari, M. (2012): Mathematical Logic for Computer Science (3. izdanje), Springer
- Čubrilo, M. (1989): Matematička logika za ekspertne sisteme, Informator, Zagreb
- Nilsson, U., Maluszynski J. (2000): Logic, programming and Prolog (2. izdanje), John Wiley & Sons (the book may be downloaded … for personal use …)
- Oracle (2023): The Java® Virtual Machine Specification, Java SE 21 Edition
- Radovan, M. (1987): Programiranje u PROLOGu, Informator, Zagreb
- Skansi, S. (2018): Introduction to Deep Learning, Springer
- SWI Prolog Reference Manual, version 8.4.2, February 2022, www.swi-prolog.org