PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

DIPLOMOVÁ PRÁCE

Nové příklady pro výuku PROLOGu

2014

Tomáš Glír

Anotace

Práce představuje programovací jazyk PROLOG (převážně pro implementaci SWI-Prolog) a její součástí je sbírka řešených příkladů pro jeho výuku. Text práce popisuje nejprve historický vývoj logického programování. Poté se věnuje teoretickým základům a popisu struktury PROLOGu. Na závěr je popsána implementace SWI-Prologu a několik reálných aplikací v různých implementacích z různých období. Praktická část se věnuje typickému užití jazyka, jeho vlastnostem a teoretickým omezením. Sbírka obsahuje klasické obměněné úlohy, netypické a nově vymyšlené příklady různé složitosti.

Děkuji vedoucímu diplomové práce RNDr. Miroslavu Kolaříkovi, Ph.D., za cenné rady a věcné připomínky při tvorbě této práce. Dále děkuji své rodině za podporu během celé doby studia.

Obsah

Seznam obrázků

[1] Obrázek je převzat z webu Wikipedie
[2] Obrázek je převzat z webu http://www.visual-prolog.com

1. Úvod

Jazyk PROLOG patří již od 70. let do seznamu nejběžnějších programovacích jazyků. Stejně jako LISP, který reprezentuje principy funkcionálního programování, nebo jazyky Smalltalk či C++ reprezentující větev objektového paradigmatu, je PROLOG typický zástupce jazyků logického programování.

Na PROLOG se často nahlíží jako na jazyk pouze pro akademické účely. Tudíž by se mohlo na první pohled zdát, že se toto paradigma pro praktické účely příliš neuchytilo. Nemusíme se bát přiznat, že se tento jazyk opravdu v praxi masově rozhodně nepoužívá. Dokonce se velmi často vůbec neužívá na celé projekty, ale jen na jejich vybrané části. Studentům informatiky se tedy logicky nabízí otázka proč a zda vůbec se tímto jazykem a logickým paradigmatem zabývat. I kdybychom se s logickým paradigmatem nikdy nesetkali, stále můžeme programovat kvalitní aplikace. Rozhodně ale není na škodu seznámit se i s tímto způsobem programování a hlavně přemýšlení.

Síla PROLOGu a logického programování totiž nespočívá jen v tomto samotném jazyku, ale v přístupu k programování a řešení daných problémů určitým způsobem, který je do jisté míry odlišný oproti jiným jazykům. Proto si můžeme ušetřit spoustu starostí a práce, pokud tuto znalost máme. Pro některé problémy byl jazyk PROLOGu přímo navržen a pro jiné problémy se jeho použití samo nabízí. Jak jsem již zmínil, nehodí se vždy a pro každý problém. Pokud ho ale užijeme správně, můžeme naše aplikace programovat lépe, efektivně a hlavně dané problémy lépe analyzovat z deklarativního hlediska, a tím pádem i lépe pochopit a naprogramovat v jakémkoliv jiném jazyku.

Nejspíš se v praxi s PROLOGem setká málokdo. To ale nebrání tomu, psát si alespoň malou část programů v PROLOGu. Tato práce se pokusí ukázat, v čem přesně je jeho síla, jaké jsou jeho typické vlastnosti a jak je správně použít. Že je možné PROLOG použít i v reálných komerčních aplikacích a že se v nich (ačkoliv se to mnohdy ani neví) skutečně používá.

Cílem této diplomové práce je vytvořit sbírku zajímavých příkladů s popisem pro výuku PROLOGu. Uvést základní i pokročilejší techniky a seznámit se s logickým programováním v PROLOGu obecně. Představit jazyk PROLOG, jeho silné a slabé stránky, teoretická omezení a nejběžnější případy jeho využití.

2. Historie logického programování

Počátek logického programování sahá do roku 1958. V tomto roce byl navržen abstraktní počítačový program Advice taker od tvůrce Lispu Johna McCarthy. V Advice takeru sloužila nejspíš poprvé matematická logika pro deklarativní reprezentaci jazyka a pro logické vyjádření vztahů mezi objekty. Pro řešení problémů – „problem-solver“, sloužil dokazovač vět – „theorem-prover“. Advice taker byl navržen pro řešení problémů tak, že dokázal manipulovat s větami zapsanými pomocí běžných formálních jazyků. Motivací k tomu bylo to, že některé základní verbální úvahy jsou relativně primitivní a mohou být zpracovány strojově. Pomocí dedukce z uvedených základních vztahů pak lze simulovat odvození dalších plynoucích faktů. V článku z roku 1958 jsou uvedeny základní vlastnosti, které by měl systém simulující lidskou inteligenci mít, a následně je prezentován popis příkladu. V letech 1960–1970 následoval vývoj dalších automatických odvozovacích systémů. [11]

Příklad programu v Advice taker[1]

```
1. at(I, desk) -> can(go(desk, car, walking))
2. at(I, car) -> can(go(home airport, drivin))
3. did(go(desk, car, walking)) -> at(I, car)
4. did(go(home, airport, driving)) -> at(I, airport)
5. canachult(at(I, desk), go(desk, car, walking), at(I, car))
6. canachult(at(I, car), go(home, airport, driving), at(I, airport))
7. canachult(at(I, desk),  prog(go(desk, car, walking),
   go(home, airport, driving)) -> at(I, airport)
8. do(go(desk, car, walking))
```

Přímým předchůdcem jazyka PROLOG byly Q–systémy, jejichž autorem je, stejně jako PROLOGu, Alain Colmerauer. Q–systémy na základě gramatických pravidel zpracovávají přirozený jazyk. Používají datovou strukturu Q–graf – orientovaný acyklický graf s jedním vstupním a jedním výstupním uzlem, kde je každý vrchol označený uspořádaný strom. Vstupní věta je obvykle lineární Q-graf, kde je každý vrchol jedno slovo (strom je redukován na jeden uzel s popisem tohoto slova). Po analýze věty má každý vrchol v grafu možný strom analýzy svého slova. Q-Systém se skládá ze sekvence Q–úprav, kde má každá úprava množinu Q–pravidel ve formě: $\langle matched\ path \rangle == \langle added\ path \rangle [\langle condition \rangle]$. Q–úpravy pracují tak, že prvně aplikují svá pravidla pomoci „instantizace“ (jednosměrné unifikace), a poté přidávají nové cesty do současného Q–grafu (přidáním vrcholů a jejich stromů vznikají nové cesty).

Rozšířením základní myšlenky Q–systémů vedlo ke vzniku PROLOGu. Konkrétně nahrazení instantizace unifikací tím, že bylo umožněno mít na pravé

[1] Příklad je převzat z textu Programs with Common Sense od J. McCarthy

straně pravidla nové proměnné a parametricky popsané stromy byly nahrazeny logickými termy. [29]

Jazyk PROLOG vznikl v roce 1972 ve Francii na univerzitě v Luminy, Marseille. Byl vytvořen autory Alain Colmerauer a Philippe Roussel. Název „PROLOG“ je zkratka z „PROgrammation en LOGique“ (francouzsky „programování v logice“). Tento jazyk byl navržen jako prostředek pro odvozování logických důsledků formulí predikátové logiky. A. Colmerauer pracoval na zpracování přirozeného jazyka. Logiku používal pro reprezentaci sémantiky a rezoluci pro získávaní odpovědí na dotazy.

V roce 1974 vznikl z tohoto návrhu na základě procedurální implementace Hornových klauzulí teoretický model PROLOGu od britského logika Roberta Kowalskiho.

V roce 1977 byla pro počítače PDP–10 od společnosti DEC vytvořena nová implementace tohoto jazyka od D. Warrena, která dala základ dnešní syntaxi jazyka PROLOG, a začaly vznikat první překladače PROLOGu.

Obrázek 1. Počítač PDP-10 [2]

V roce 1983 D. Warren navrhnul abstraktní stroj pro zpracování PROLOGu, Warren Abstract Machine (WAM). Návrh se později stal standardem pro překladače PROLOGu. WAM se skládá ze sady instrukcí a návrhu architektury paměti (haldy, lokálního zásobníku a paměti pro rušení vazeb proměnných při backtrackingu). Kód PROLOGu se kompiloval do nižšího WAM kódu pro zrychlení celého programu.

V roce 1981 byl jazyk PROLOGu vybrán pro japonský projekt počítačů 5. generace jako základ pro logický procesor 5. generace počítačů měla za cíl masivně

zvýšit počet CPU (oproti 4. generaci, která se zaměřila na zvýšení počtu logických obvodů v jednom CPU). Cíl byt vytvořit architekturu nového superpočítače pro vědecké výpočty, který by poskytoval koordinované paralelní nebo distribuované výpočty nad obsáhlými znalostními databázemi oproti dřívějším běžným souborovým systémům. Logický jazyk měl poskytovat přístup k datům z databáze a tato data také popisovat a definovat. Později měl tento počítač sloužit jako základ pro další vývoj v oblastech umělé inteligence.

Tento projekt trval zhruba 10 let a stál přibližně 400 miliónů dolarů. Tato paralelní architektura počítačů byla nakonec daleko pomalejší, dražší a složitější než běžný méně specializovaný hardware (např. Intel x86). Navíc nastal problém, který logický programovací jazyk zvolit jako most mezi paralelní výpočetní architekturou a užitím logiky jako znalostní báze a řešení problém pro aplikace umělé inteligence. Žádný se nakonec neukázal jako ideální a každý měl nějaké limitující podmínky.

Společně s jazykem LISP je dnes PROLOG nejpoužívanější jazyk pro symbolické výpočty, pro oblast umělé inteligence či v počítačové lingvistice nebo jako prostředek pro reprezentaci a zpracování znalostí. [1], [3]

2.1. Úvod do logického programování

Logické programování v běžném PROLOGu je založeno na predikátové logice 1. řádu. Konkrétně se omezuje na Hornovy klauzule. Existují i odlišné implementace založené na jiných základech. Běh programu představuje užití dokazovacích technik na zadané klauzule. Základními využívanými přístupy jsou unifikace, rekurze a backtracking. Jako odvozovací technika se užívá technika rezoluce. Program definuje vše, co můžeme vypočítat a odvodit. Logické programování tedy užívá matematické logiky jako prostředku pro programování.
Řešení problému se dělí mezi programátora (ručí za správnost programu v logické formě) a interpret (jako dokazovač), který odpovídá za tvorbu a efektivnost řešení.

2.2. Jiné typy logik a programovacích jazyků

Predikátová logika 1. řádu není jediná logika, která se v teorii logického programování užívá. Mnoho interpretů (λ Prolog, HiProlog, SWI-Prolog) PROLOGu dnes běžně implementuje logiku vyššího řádu. Pro větší komfort při programování užívá tzv. programování s omezujícími podmínkami (Constraint logic programming) nebo konkurenční programování pro paralelní a distribuované výpočty (vše v SWI-Prologu). Dalšími typy jsou například lineární nebo induktivní logická programování. Některé implementace PROLOGu jsou dokonce rozšířeny o objektovou nástavbu (Oblog, Logtalk) nebo je v nich zavedena kontrola typování. Přítomnost grafiky či jiných rozšíření je už samozřejmě úplně mimo téma logického programování. Platí, že v této práci se převážně zabývám logickým programováním z klasické predikátové logiky 1. řádu. [27]
Další logické jazyky odvozené z PROLOGu jsou např. Gödel, Mercury nebo Datalog. [4], [19], [21]

2.3. Prolog k PROLOGu

Převážná většina programovacích jazyků dnes užívá procedurální a nebo objektové paradigma. Popisuje se v nich, jak se má daná úloha vyřešit. PROLOG je programovací jazyk čtvrté generace a je typický zástupce logických jazyků. Patří mezi deklarativní jazyky, ve kterých se popisuje cíl výpočtu (co chceme spočítat) a přesný postup výpočtu (jak to chceme spočítat) přenecháváme systému (nějakému interpretu jazyka pro jeho konkrétní implementaci). V logickém programu, složeného z faktů a klauzulí, se snažíme abstraktně vyjádřit vztahy mezi objekty a případné další vztahy plynoucí z programu odvodit. Zodpovězení dotazu probíhá postupným ověřováním splnitelnosti jednotlivých cílů.

Během procesu vyhodnocení cíle (dotazu) dochází v případě úspěchu k nastavení proměnných (substituci a unifikaci) na takové hodnoty, které umožní jeho splnitelnost. Je-li více možností, jak dosáhnout splnění cíle, PROLOG je schopen je všechny zjistit pomocí postupu nazývaného zpětné prohledávání (backtracking). Neúspěch při vyhodnocení cíle neznamená, že PROLOG dokázal jeho neplatnost, ale pouze to, že nebyl schopen ze své databáze dokázat jeho platnost.

Interprety PROLOGu jsou většinou tvořeny tzv. čistým (pure) PROLOGem, který dodržuje pravidla logiky, a procedurálním nebo jiným rozšířením, které poskytuje programátorovi větší komfort. Na interpret PROLOGu se nahlíží jako na vyhodnocovací prostředí, které je tvořeno množinou relací mezi objekty. V tomto prostředí se pak pomocí odvozovacích mechanismů (rezoluce) zodpovídají uživatelské dotazy a určuje jejich platnost nebo neplatnost.

Můžeme se setkat s různými dialekty jazyka PROLOG. Dnes je ale nejpoužívanější (a téměř jediný uznávaný) standard ISO Prolog, který vychází ze staršího standardu Edinburgh Prolog. Tento standard detailně popisuje syntaxi a sémantiku jazyka a jeho vlastnosti. Standard ISO Prologu nalezneme na webu `http://www.deransart.fr/prolog/docs.html`.

Velká výhoda jazyka PROLOG je jeho snadná a úsporná syntaxe (původně byl navržen pro lingvisty, ne programátory). Programy v něm bývají obvykle velice krátké a přehledné. Snadno se modifikují nebo rozšiřují a výsledné zdrojové soubory zabírají většinou maximálně několik desítek kB. Jako nevýhodu je třeba zmínit rychlost jazyka, především u numerických úloh, pro které nebyl vůbec určen. Dnes problém rychlosti u interpretovaných jazyků není tak podstatný, jako dříve. Rychlost PROLOGu a kompilovaných jazyků se většinou vůbec neporovnává, ale přesto se snažíme psát programy efektivně. Při psaní programu v PROLOGu jde většinou o deklarativní vyřešení daného problému a není potřeba maximální rychlosti řešení. Pokud bychom psali nějakou aplikaci kriticky závislou na rychlosti, zvolíme samozřejmě jiný jazyk. [1], [2]

3. Teoretické základy logického programování

Logikou a logickým programováním se zabývají autoři J. W. Lloyd, U. Nilsson, L. Sterling a E. Shapiro z jejichž knih jsem v této kapitole především čerpali. [5], [6], [7]

Logický program je z pohledu logiky množina axiomů a/nebo pravidel, které definují určitou relaci, tj. vztah, mezi objekty – reálnými nebo i abstraktními věcmi z našeho světa. Na výpočet logického programu se díváme jako na dedukci nějakých následků, které z logického programu plynou a lze je určitým způsobem z programu odvodit. Program tedy definuje tuto množinu následků.

3.1. Predikátová logika 1. řádu

Predikátová logika 1. řádu je formální odvozovací systém používaný k popisu matematických teorií a vět. Predikátová logika formalizuje vzájemné vztahy mezi objekty. Budou zde uvedeny základní pojmy.

Klauzule je formule ve tvaru $\forall x_1 \ldots \forall x_k (L_1 \vee \ldots \vee L_m)$, kde L_i jsou literály – dále nedělitelné prvky (výrokové proměnné nebo jejich negace), a $x_i, i = 1, \ldots, k$ všechny proměnné vyskytující se v $L_1 \vee \ldots \vee L_m$.

Term je v predikátové logice definován induktivně:

- každá proměnná je term,
- každý konstantní symbol c jazyka L je term,
- F je n-ární funkční symbol jazyka L a $t_1, \ldots, t_n$ jsou termy, pak $F(t_1, \ldots, t_n)$ je term,
- jazykem zde rozumíme pevně zvolenou množinu symbolů, jazyk L predikátové logiky je popsán dále.

Atomická formule jazyka L je výraz ve tvaru $P(t_1, \ldots, t_n)$, kde P je n-ární predikátový symbol jazyka L a $t_1, \ldots, t_n$ jsou termy. Formule se nazývá **otevřená**, neobsahuje-li žádný kvantifikátor, a **uzavřená**, je-li každá proměnná v ní obsažená kvantifikována. **Podformulí formule** φ je každá formule, která je částí formule φ.

Proměnná x je **vázána** ve formuli φ jestliže existuje podformule ψ formule φ ve tvaru $(\forall x)\psi(x)$. Proměnná x je **volná** ve formuli φ, jestliže x má výskyt v nějaké podformuli ψ formule φ takové, že ψ není podformule žádné formule tvaru $(\forall x)(\chi(x))$.

Term t je **substituovatelný** za proměnnou x do formule φ, jestliže x není volná v žádné podformuli tvaru $(\forall y)\psi$, kde proměnná y má výskyt v termu t. Pokud náš term t obsahuje proměnnou y, která je v místě substituce vázaná,

musí tam být i x vázaná. Je-li x proměnná, t term a φ formule, $\varphi(x/t)$ značí formuli, která vznikne nahrazením termu t za každý volný výskyt proměnné x ve φ.

Jazyk L predikátové logiky je určen:

- proměnnými $x, y, z, \ldots$
- relačními symboly $P, Q, R, \ldots$ a jejich aritou σ, v jazyku musí existovat alespoň jeden relační symbol,
- funkčními symboly $F, G, H, \ldots$ a jejich aritou σ, pro aritu 0 nazýváme symbol f **konstantou**,
- symboly logických spojek negace a implikace,
- symbolem pro univerzální kvantifikátor,
- pomocnými symboly – závorky, čárka.

Jazyk L je jednoznačně určen relačními a funkčními symboly a jejich aritou (počtem argumentů). **Typ jazyka** nazýváme trojici $\langle R, F, \sigma \rangle$.

Jazyk s rovností nazýváme jazyk, pokud se mezi relačními symboly vyskytuje symbol $\approx$. V predikátové logice užíváme rovněž symbolů konjunkce ($\wedge$), disjunkce ($\vee$), ekvivalence ($\leftrightarrow$) a $(\exists x)\varphi$ jako zkrácení za $\neg(\forall x)\neg\varphi$. Konjunkce je pravdivá tehdy, jsou-li pravdivé oba spojované výroky, jinak je nepravdivá. Slovně vyjadřuje spojení *a současně*. Disjunkce je pravdivá tehdy, je-li pravdivý alespoň jeden ze spojovaných výroků, jinak je nepravdivá. Slovně vyjadřuj spojení *nebo*. Ekvivalence je pravdivá tehdy, když jsou oba výroky pravdivé nebo nepravdivé, jinak je nepravdivá. Slovně vyjadřuje spojení *právě tehdy, když*.

Predikátová formule reprezentuje nějaké matematické tvrzení v jisté formální teorii. Formule jazyka L typu $\langle R, F, \sigma \rangle$ jsou definovány následovně:

- $r \in R$ je relační symbol s aritou $\sigma = n$ a $t_1, \ldots, t_n$ jsou termy, pak $r(t_1, \ldots, t_n)$ je formule,
- pokud φ je formule a x proměnná, pak $\neg\varphi$ a $(\forall x)\varphi$ jsou formule,
- když φ a ψ jsou formule, pak $\varphi \to \psi$ je formule.

3.1.1. Pravdivostní ohodnocení a struktura pro jazyk

Nyní bude definováno pravdivostní ohodnocení pro predikátovou logiku.

Struktura pro jazyk $\mathcal{M}$ typu $\langle R, F, \sigma \rangle$ je $\mathcal{M} = \langle M, R^M, F^M \rangle$, kde:

- $R^M = \{r^M \subseteq M^n \mid r \in R, \sigma(r) = n\}$,

- $F^M = \{f^M : M^n \to M \mid f \in F, \sigma(f) = n\}$,
- M je neprázdná množina.

Tato struktura definuje systém relací, např. porovnání prvků, a funkcí, např. aritmetické operace, na nějaké množině M, např. na nějaké číselné množině. **M-ohodnocení** je zobrazení v přiřazující každé proměnné x prvek $v(x) \in M$.

Hodnotu $\|t\|_{M,v}$ termu t v M při zobrazení v definujeme:

- $v(x)$ je-li t proměnná x,
- $f^M(\|t_1\|_{M,v}, \ldots, \|t_k\|_{M,v})$ je-li t tvaru $f(t_1, \ldots, t_k)$.

Pravdivostní hodnota $\|\varphi\|_{M,v}$ formule φ při M-ohodnocení v je definována:

- $\|r(t_1, \ldots, t_n)\|_{M,v} = 1$ je-li $\langle \|t_1\|_{M,v}, \ldots, \|t_k\|_{M,v} \rangle \in r^M$, jinak 0,
- $\|\neg\varphi\|_{M,v} = 1$ pro $\|\varphi\|_{M,v} = 0$ a naopak pro $\|\neg\varphi\|_{M,v} = 0$,
- $\|\varphi \to \psi\|_{M,v} = 1$ pro $\|\varphi\|_{M,v} = 0$ nebo $\|\psi\|_{M,v} = 1$, jinak 0,
- $\|(\forall x)\varphi\|_{M,v} = 1$ pokud pro každé $v' =_x v$ je $\|\varphi\|_{M,v'} = 1$, jinak 0.

Formule φ je **pravdivá** ($\|\varphi\|_{M,v} = 1$) nebo **nepravdivá** ($\|\varphi\|_{M,v} = 0$) ve struktuře M při ohodnocení v. Formule φ je **tautologie ve struktuře M** jestliže $\|\varphi\|_{M,v} = 1$ pro každé M-ohodnocení v.

Formální teorie jazyka L typu $\langle R, F, \sigma \rangle$ je každá množina T formulí jazyka L. Neformálně řečeno, teorie nám značí nějakou množinu předpokladů – formulí. **Model teorie** T je nějaká struktura M jazyka typu $\langle R, F, \sigma \rangle$.

3.1.2. Axiomy, odvozování a důkaz

Axiomy predikátové logiky jsou následující formule:

- $\varphi \to (\psi \to \varphi)$,
- $(\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi))$,
- $(\neg\psi \to \neg\varphi) \to (\varphi \to \psi)$,
- $(\forall x)\varphi \to \varphi(x/t)$ je-li t substituovatelné za x,
- $(\forall x)(\varphi \to \psi) \to (\varphi \to (\forall x)\psi)$.

Odvozovací pravidla pro predikátovou logikou jsou pravidla:

- modus ponens – z $\varphi, \varphi \to \psi$ odvoď ψ,
- pravidlo generalizace – z φ odvoď $(\forall x)\varphi$.

Důkaz formule φ z množiny T formulí je libovolná posloupnost $\varphi_1, \ldots, \varphi_n$, kde platí, že $\varphi_n = \varphi$ a každá φ_i pro $i \leq n$ je:

- axiomem,
- formulí z T,
- nebo plyne z předchozích formulí pomocí modus ponens nebo pravidla generalizace.

Formule je **dokazatelná z T**, existuje-li důkaz této formule z T.

3.2. Logika vyššího řádu a další typy logik

Kromě predikátové logiky 1. řádu, tzv. klasické logiky, existují další druhy logik. Některé jsem uvedl v úvodu. Z pohledu PROLOGu je zajímavá hlavně **predikátová logika vyššího řádu**. Ačkoliv jazyk PROLOG na ní ve svém základu není založen, stručně se o ní nyní zmíním z toho důvodu, že logika vyšších řádů se dnes jako rozšíření běžně v SWI-Prologu nebo Ciao Prologu užívá.

Logika vyššího řádu se odlišují tím, že povoluje predikáty uvnitř predikátů a kvantifikování predikátů a funkcí. U predikátové logiky 1. řádu jsou predikáty spojeny s teorií množin. V případě logik vyšších řádů bývá na predikáty nahlíženo jako na množiny množin. Predikát vyššího řádu je predikát, který má jeden nebo víc jiných predikátů jako argumenty. Pro lepší představu lze uvést funkce vyšších řádů z funkcionálního programování, které mají podobné vlastnosti. [17], [18]

Typičtí zástupci predikátů vyšších řádů z SWI-Prologu jsou například predikáty `setof/3`, `bagof/3` nebo `findall/3`, které nejčastěji slouží pro agregaci všech možných výsledků. Nebo predikát `call/1` sloužící pro vyvolání zadaného cíle. Podobně jako ve funkcionálním programování lze nalézt predikáty `map/3`, `apply/2` nebo `foldr/4` sloužící např. pro úpravu seznamu. Tyto predikáty jsou také nazývány jako *meta-predikáty*, protože to jsou predikáty pracující s jinými predikáty. Příklad jejich použití můžeme najít v přiložené sbírce příkladů. [16]

Mezi další neklasické typy logik patří fuzzy logika, modální a temporální logika. Existuje i implementace interpretu Fuzzy Prolog. Dále se těmito uvedenými logikami nebudu zabývat, protože se v logických jazycích užívají velmi zřídka a není to obsahem mé práce.

4. Logika z pohledu jazyka PROLOG

V této sekci uvedu souvislosti mezi logikou a jazykem PROLOG. Opět budu čerpat od autorů J. W. Lloyd, U. Nilsson, L. Sterling a E. Shapiro, kteří tuto problematiku také popisují. [5], [6], [7]

4.1. Termy

Term je základní struktura v logickém programu. Termy jazyka L jsou definovány induktivně. Proměnné a konstanty jsou termy.
Složené termy (struktury) jsou také termy. Složený term se skládá z funktoru a posloupnosti jednoho nebo více argumentů, které jsou termy. Příklady složených termů jsou např. `one(1), son(john, bill), test(X), tree(tree(nil, nil, a), b, c, list(d, list(e, nil)))`.
Funktor je tedy určen svým jménem f a svou aritou n – počtem argumentů. Zápis má formát $f(t_1, t_2, \ldots, t_n)$.

4.1.1. Proměnné

Proměnné jsou termy a zastupují v programu nějakou entitu. Jsou zapsány velkým písmenem nebo podtržítkem a navazujeme na ně nějaké hodnoty. Jsou sdílené pouze v rámci jedné klauzule, nikoliv v rámci více klauzulí.
Uzavřený term je term, kde se nevyskytují proměnné. Například `color(blue)` nebo `day(monday, mon)`.
Otevřený term je term, kde se vyskytuje jedna nebo více proměnných. Například `color(X)` nebo `day(Y, mon)`.
Anonymní proměnná je speciální proměnná, která se nesdílí v rámci pravidel (má ve všech atomických formulích právě jeden výskyt) a nelze se na ní odkazovat. Je značena symbolem podtržítko.

4.2. Definitní program

Logický program lze intuitivně chápat jako soubor faktů a pravidel. Nazývá se definitním programem. Nastává otázka, jak správně formálně logický program popsat. Jak ověřit, že je definován korektně a vyjadřuje skutečně to, co má. Logický program je tedy nějaká teorie obsahující axiomy.

Definitní program se skládá z konečné množiny pravidel a/nebo faktů. Nejjednodušší program lze zapsat ve formě konečné množiny faktů. **Jazyk logického programu** L je tedy určen:

- množinou funkčních symbolů s jejich aritami,
- množinou relačních symbolů s jejich aritami.

Cíl G, kvantifikovaný $\exists$, je **logickým důsledkem programu** P, pokud je v programu P klauzule $A \leftarrow B_1, \ldots, B_n, n \geq 0$, kde $B_1, \ldots, B_n$ jsou logické důsledky programu P a A je instance cíle G. G musí být odvozen z programu P v konečném počtu kroků aplikace pravidla rezoluce.

Význam definitního programu $M(P)$ je množina uzavřených cílů odvozených z programu P. Program P je, neformálně, **správný** k významu $M(P)$, pokud význam programu P je podmnožina $M(P)$. Správný program tedy neříká nic, co jsme předem nezamýšleli. Program P je **správný a úplný** k zamýšlenému významu M, když $M = M(P)$. Cíl dotazu je pravdivý vzhledem k zamýšlenému významu, pokud je jeho členem. Např. cíl je pravdivý, pokud se jméno predikátu a konstanty symbolů objevují v programu – lze je v programu nalézt a tím pádem z programu vyplývají. Nyní detailněji popíšu model M a Herbrandovskou interpretaci.

4.3. Herbrandovská interpretace

Herbrandovská interpretace je interpretace, ve které je všem konstantám a funkčním symbolům v programu přiřazen jednodušší význam, než v termové (univerzální) algebře. Každá konstanta je interpretována sama sebou. Každý funkční symbol je interpretován jako aplikace funkce na její argumenty, kde každý argument je term. Relační symboly značí podmnožinu relevantní herbrandovy báze, tedy které uzavřené atomy jsou v interpretaci pravdivé. To umožňuje interpretovat symboly v množině klauzulí čistě syntaktickým způsobem. Pokud je klauzule S nesplnitelná, pak je tu konečná nesplnitelná množina uzavřených instancí z herbrandova univerza. Protože je tato množina konečná, jejich nesplnitelnost může být ověřeno v konečném čase.

Herbrandovo univerzum U_L **programu** P pro jazyk L_P predikátové logiky 1. řádu je množina všech uzavřených termů.
Herbrandova báze B_P **programu** P je množina všech uzavřených atomických formulí jazyka L_P.
Pro definitní program $p(X) \leftarrow q(f(X), g(X))$ a $r(Y) \leftarrow$, skládající se z funkčních symbolů f, g a relačních symbolů p, q, r, je herbrandovo univerzum $a, f(a), g(a), f(f(a)), f(g(a)), g(f(a)), g(g(a)), \ldots$, a herbrandova báze $p(a), q(a, a), r(a), p(f(a)), p(q(a)), q(a, f(a)), q(f(a), a), \ldots$.
Herbrandova struktura programu P je každá podmnožina herbrandovy báze, $M \subseteq B_p$. Tato struktura vymezuje, které uzavřené atomické formule považujeme za pravdivé. **Pravdivost** klauzule $A \leftarrow B_1, \ldots, B_n$ v herbrandově struktuře M pro program P značíme jako $M \models A \leftarrow B_1, \ldots, B_n$, pokud pro každou substituci θ, kde $\{A\theta, B_1\theta, \ldots, B_n\theta\} \subseteq B_P$ platí, že pokud $\{B_1\theta, \ldots, B_n\theta\} \subseteq M$, pak $A\theta \in M$.
Herbrandův model programu P je herbrandova struktura M pro tento program, pokud pro každou klauzuli $A \leftarrow B_1, \ldots, B_n$, z programu P platí $M \models A \leftarrow B_1, \ldots, B_n$. Tedy M je modelem P pokud je každá klauzule z P prav-

divá v M. Herbrandova báze programu P je triviálně jeho modelem. $Mod(P)$ značíme množinu všech herbrandovských modelů P.
Nejmenší herbrandův model programu P nazýváme herbrandovu strukturu $M_P \subseteq B_p$ definovanou jako $M_p = \bigcap_{i \in I} M_i$, kde M_i je systém všech herbrandových modelů programu P.

4.4. Definitní klauzule

Definitními klauzulemi v definitním programu rozumíme všechna fakta a pravidla. Z pohledu logického programu jsou definitní klauzule uzavřené, univerzálně kvantifikované formule ve tvaru: $(\forall X_1) \ldots (\forall X_m)((A_1 \wedge \ldots \wedge A_n) \rightarrow A_0)$.

Pravidla jsou výroky ve tvaru $A \leftarrow B_1, B_2, \ldots, B_n$, kde $n \geq 0$. Cíl A je **hlava** pravidla, konjunkce cílů (atomických formulí) $B_1, \ldots, B_n$ je **tělo** pravidla. Procedurálně se na pravidla můžeme dívat takto: „Abychom zodpověděli dotaz A, musíme zodpovědět konjunkci dotazů $B_1, \ldots, B_n$“. Z pohledu logiky dotaz znamená: „A platí, pokud platí $B_1, \ldots, B_n$“ a $\leftarrow$ značí logickou implikaci ve výrazu $A \leftarrow B_1, \ldots, B_n$. Pravidla nám umožňují definovat nové složitější vztahy v programu pomocí již existujících jednodušších vztahů.
Není překvapivé, že pravidla mají opět blízký vztah k logice a to konkrétně k odvozovacímu pravidlu **modus ponens**: Z $\varphi, \varphi \Rightarrow \psi$ odvoď ψ. Toto pravidlo říká, že z pravidla $R = (A \leftarrow B_1, B_2, \ldots, B_n)$ a faktů $B_1, B_2, \ldots B_n$ může být A odvozeno, pokud je $A \leftarrow B_1, B_2, \ldots, B_n$ instancí R. Ve slovním vyjádření to znamená: „Jestliže A platí a z A vyplývá B platí, pak i B platí.“

Fakt je speciální případ pravidla ve formě $A.$, když $n = 0$. Někdy se fakta nazývají **atomické formule**. Z pohledu logiky není důležité, zda je nějaký vztah v programu dán faktem nebo pravidlem a vyskytuje se v programu přímo ve formě faktu nebo je odvozen pomocí pravidla.

Definitní cíl, neboli dotaz, je klauzule ve formě $\leftarrow B_1, \ldots, B_n$. Její hlava je tedy prázdná. Každý prvek dotazu B_i je nazýván subcíl dotazu. Cíle mají existenční význam, např: „Existují dané atomy s vlastností car?“ pro dotaz `car(X)`.

Redukce cíle G v programu P je nahrazení G tělem instance klauzule která je v programu P a jejíž hlava je identická k cíli G. Redukci užívá interpret PROLOGu při zodpovězení dotazu a její aplikace bude uvedena dále.
Můžeme ji popsat slovně takto: V programu nalezneme daný fakt nebo pravidlo podle hlavy zadaného dotazu. Vyhledání faktu nebo pravidla se provádí v programu shora dolů podle názvu termu – funktoru a podle jeho arity. Pokud není podle hlavy dotazu nalezen žádný odpovídající výsledek, dotaz neuspěje. Pokud se jedná o fakt, který se v programu nachází, dotaz je úspěšně splněn, protože fakt nemá tělo (nejedná se o pravidlo) a již neprodukuje další subcíle. Pokud se jedná o pravidlo a hlava dotazu se unifikuje s některým pravidlem v programu, přejde se na řešení dalších subcílů v těle vyhledaného pravidla, které se opět postupně snažíme z programu odvodit výše uvedeným postupem. Pokud se to u všech podaří, celý dotaz je splněn.

Prázdná klauzule je klauzule s prázdným předpokladem i závěrem a značí se znakem □. Má význam sporu.

Definitní klauzule (fakta, pravidla a cíle) jsou **Hornovy klauzule**. Skupině klauzulí se stejným jménem a aritou se říká také **predikáty**. Na fakta a pravidla v této skupině se lze dívat jako na možné alternativy při výpočtu.

4.5. Unifikace a substituce

Unifikace a substituce patří do základního výpočetního procesu při plnění dotazu v logickém programu. Pro zodpovězení otevřeného dotazu z programu se hledají klauzule, které jsou instancemi dotazu. Řešení je reprezentováno substitucí – na proměnné jsou navázány hodnoty. Odpověď na dotaz je `no` (`false`), pokud v programu není nalezen vhodný fakt/pravidlo. Odpověď `yes` (`true`) obdržíme, pokud se jedná o uzavřený dotaz a je v programu nalezen.

4.5.1. Substituce

Substituce Θ je konečná (i prázdná) množina párů $X_i = t_i$, kde X_i je proměnná a t_i je term. Dále $X_i \neq X_j$ pro $\forall i \neq j$ a X_i se nevyskytuje v t_j pro žádné i a j.
Jednoduchý příklad substituce je např. {`X=mon`}. Výsledek aplikace substituce {`X=mon`} na term `day(monday, X)` je term `day(moday, mon)`. Tento fakt `day(monday, mon)` tedy značí název dne v týdnu a jeho zkratku. Substituce tedy může být aplikována na termy.

Aplikací substituce se rozumí souběžné nahrazení proměnných nějakými termy. Výsledek aplikování substituce Θ na term A se značí ΘA a obdržíme ho nahrazením t každého výskytu X v A, pro každý pár $X = t$ v Θ. A je **instance** B, když existuje taková substituce Θ, že $A = B\Theta$.
Cíl `day(monday, mon)` je instance `day(monday, X)`. Podobně je třeba `day(monday, mon)` instancí `day(X, Y)` pro substituci {`X=monday,Y=mon`}. Substituce samozřejmě nemusí vždy existovat nebo může mít více výsledků. Pro dotaz `father(john, X)` mohou existovat řešení {`X=ann`}, {`X=pete`} značící, že John je otec dvou dětí Ann a Pete.
C je **společná instance** A a B, pokud v programu existuje instance A a B. Tedy, jsou tu substituce Θ_1 a Θ_2, že $C = A\Theta_1$ je identické k $B\Theta_2$. Například `car(bmw, black, Y)` a `car(bmw, X, Y)` mají společnou instanci `car(bmw, black, 1996)` po aplikování substitucí {`X=black,Y=1996`}.
Term s je **obecnější** než term t, když t je instancí s, ale s není instancí t.
Term t je **varianta** termu s, pokud s je instancí t a t je instancí s. Například `member(X, tree(X, Right, Left))` a `member(Y, tree(Y, Z, Left))`.

Z pohledu logiky jsou proměnné v dotazech kvantifikovány existenčním kvantifikátorem $\exists$. Předchozí dotaz `day(monday, X)` má po přidání kvantifikátoru význam: „Existuje tu takové X, že *monday* je název pro zkratku X?“

To lze obecně u dotazů zapsat jako dotaz $p(T_1, T_2, \ldots, T_n)$, který obsahuje proměnné $X_1, X_2, \ldots, X_k$ s významem: "Existují tu takové $X1, X2, \ldots, X_k$ pro $p(T_1, T_2, \ldots, T_n)$?"

Generalizací se nazývá existenční dotaz P, který je logickým důsledkem své instance, $P\Theta$, pro nějakou substituci Θ. Fakt `day(monday, mon)` implikuje, že tu existuje X, že pro `day(monday, X)` je {`X=mon`}.

Při unifikaci a substituci může někdy nastat problém, že proměnné vyskytující se v klauzulích jsou stejné. Tyto proměnné je ale potřeba v různých klauzulích nějak odlišit. Proto se užívá technika **přejmenování proměnných**, aby se žádná z nich nevyskytovala jinde. Pro klauzule $A \leftarrow B_1, \ldots, B_n$ a $C \leftarrow D_1, \ldots, D_m$ se jednoduše užije substituce θ ve tvaru $\{X_1/Y_1, \ldots, X_k/Y_k\}$, kde jsou X_i proměnné vyskytující se v klauzuli s hlavou A a Y_i jsou po dvou různé proměnné nevyskytující se v klauzuli s hlavou B. Tato nová varianta klauzule již nebude mít společné proměnné s druhou klauzulí.

4.5.2. Unifikace

Interpret PROLOGu (nebo jiného logického jazyka) musí správné substituce pro dané uzavřené instance správně spočítat. To je možné a slouží k tomu unifikační algoritmus, který určuje správné instance termů. Unifikace je základem v logickém odvozování.

Unifikátor dvou termů je substituce, která dělá termy identické. Říkáme, že jsou termy **unifikované**. Unifikace má blízký vztah k společným instancím. Unifikátor určuje společné instance a naopak tyto instance určují unifikátor.

Např. `append([1,2,3],[4],List)` a `append([X|R], Y, [X|Z])` jsou unifikované substitucí $X = 1, R = [2, 3], Y = [4], List = [1|Z]$, společná instance určená unifikující substitucí je `append([1,2,3], [4], [1|Z])`.

Nejobecnější unifikátor (mgu) dvou termů je unifikátor, kde je společná instance nejobecnější. Je možné dokázat, že pokud můžeme dva termy unifikovat, všechny mgu jsou ekvivalentní. Unifikační algoritmus tedy spočítá nejobecnější unifikátor dvou termů, pokud existuje.

Unifikační algoritmus počítá nejobecnější unifikátor dvou termů T a S. Užívá zásobníku pro ukládání srovnání (rovností), které je potřeba vyřešit, a aktuální substituce Θ. Výstup algoritmu je nejobecnější unifikátor termů T a S nebo selhání. Algoritmus vyzvedává rovnosti ze zásobníků a končí, pokud je zásobník prázdný nebo selže zpracování rovností.

Mohou nastat následující situace:

- S a T jsou identické konstanty nebo proměnné, rovnost je správná a výpočet pokračuje vyzvednutím další rovnosti ze zásobníků,
- S je proměnná a T je term, který neobsahuje S. Zásobník je prohledán pro všechny výskyty S, které jsou nahrazeny T. Všechny výskyty S v Θ jsou

rovněž nahrazeny T. Poté je substituce $S = T$ přidána do Θ. Je důležité, aby se S nevyskytovala v T. Tento test je známý jako **occurs check**.

- T je proměnná a S je term, nastane symetrická situace výše,
- S a T jsou složené termy se stejnými jmény funktorů a stejnou aritou $f(S_1, \ldots, S_n), f(T_1, \ldots, T_n)$. Rovnosti jsou přidány na zásobník. Pro unifikaci termů se musí každý pár argumentů zároveň unifikovat. Toho se dosáhne vložením n rovností $S_i = T_i$ na zásobník.
- v jiném případě je nahlášen neúspěch algoritmu,
- Pokud je zásobník vyprázdněn, termy se unifikovaly a unifikátor bude nalezen jako Θ.

```
Vstup: Termy T a S
Výstup: mgu termů T a S nebo selhání
Algoritmus:
Inicializace substituce Theta na prázdnou,
zásobník obsahuje rovnost T = S, failure = false.

while(stack != empty && failure = false)
pop X=Y from the stack;
    case:
     X je proměnná nevyskytující se v Y:
       substituuj Y za X na zásobníku a v Theta;
       add X = Y do Theta;
     Y je proměnná nevyskytující se v X:
       substituuj X za Y na zásobníku a v Theta;
       add Y = X do Theta;
     X a Y jsou identické konstanty nebo proměnné:
       continue;
     X je f(X1, ..., Xn) a Y je f(Y1, ..., Yn) pro nějaký funktor f a n>0:
       push Xt = Yt, i = 1..n na zásobník;
     default:
       failure = true;

if(failure)
return failure;
  else
return Theta;
```

Occurs check neboli „zabránění výskytu“ je část algoritmu syntaktické unifikace, zabraňující zacyklení při nekonečné substituci výrazů. Pro výraz

$X = f(X)$ tu není v herbrandově univerzu žádná konečná instance těchto termů, jedná se tzv. o cyklický term. Obecně tento případ nastane při unifikaci proměnné V a struktury S, pokud struktura S tuto proměnnou V v sobě obsahuje.
Různé implementace PROLOGu tuto podmínku většinou z důvodu efektivity nekontrolují a tak může dojít k zmíněnému zacyklení nebo k vytvoření cyklu v datových strukturách. Při vynechání této kontroly se nám změní časová složitost algoritmu unifikace termu T a S z $O(size(T) + size(S))$ na $O(min(size(T), size(S)))$. Obecně ale pro jednoduchou unifikaci (proměnná – term) velmi často platí jen $O(1)$.
Implementace SWI-Prolog má volitelnou možnost nastavení kontroly této podmínky. Ta pochází z ISO Prologu podle vestavěného predikátu `unify_with_occurs_check/2`.

4.6. Rezoluce

Rezoluce je efektivní metoda zkoumání logické pravdivosti, zavedena Alanem Robinsonem. Jedná se o metodu automatického dokazování tvrzení (lze jí užít pro strojové dokazování), která je úplná dokazovací metoda (nalezne spor v libovolné sporné množině klauzulí). Rezoluce se užívá v logických programovacích jazycích – tudíž i v PROLOGu, protože důkaz logického vyplývání (pomocí axiomů a odvozovacích pravidel – přímý důkaz) může být zvláště v predikátové logice značně obtížný. Rezoluční metoda proto nepracuje s pravdivostními modely, ale důkaz se provádí syntakticky. Oproti automatickému dokazování ve výrokové logice musíme predikátovou formuli převést na prenexní tvar a eliminovat z ní kvantifikátory. Rezoluční dokazování je dále složitější o metodu unifikace. Platnost dané formule se dokazuje sporem v konečné množině klauzulí, tj. dokazuje se, že jsou výchozí axiomy nebo formule ve sporu s negací dokazované formule. Důkaz sporem je dokončen odvozením prázdné klauzule. Chceme-li dokázat formuli X, do množiny formulí se jednoduše přidá $\neg X$, protože je-li formule X tautologie, pak je formule $\neg X$ kontradikce a naopak. Užívá se zde tří principů:

- vyvracení – formule X vyplývá z předpokladů $A_1, \ldots, A_n$ právě tehdy, je-li formule $A_1 \wedge \ldots \wedge A_n \wedge \neg X$ kontradikce. Problém důkazu dané formule se převede na problém důkazu nesplnitelnosti negace této formule.

- rezoluční odvozovací pravidlo – jsou-li splnitelné klauzule $(A_1 \vee \ldots \vee A_m \vee L)\Theta$, $(B_1 \vee \ldots \vee B_n \vee \neg L)\sigma$, pak je splnitelná také klauzule $(A_1 \vee \ldots \vee A_m \vee B_1 \vee \ldots \vee B_n)\Theta$,

- Robinsonův rezoluční princip – umožňuje vyvodit spor z nesplnitelné formule a tak dokázat její nesplnitelnost a tím dokázat platnost původní.

Pro formuli X, která je nesplnitelná, rezoluční metoda v konečném čase zastaví. V opačném případě metoda nemusí nikdy skončit. Proto pokud chceme rozhodnout, zda je formule X pravdivá, použije se rezoluční metodu na formuli $\neg X$ a zjišťuje se, zda je nesplnitelná. Pokud ano, rezoluce vrátí kladnou odpověď, jinak nemusí nikdy skončit. Obecně, chceme-li dokázat $A_1, \ldots, A_n \models B$, použije se metodu na formuli $A_1 \wedge \ldots \wedge A_n \wedge \neg B$, protože pokud je tato formule nesplnitelná, pak je předchozí formule tautologie a vztah vyplývání platí.
Tvar pravidla základní rezoluce pro predikátovou logiku je

$$\frac{(p \vee A) \wedge (\neg q \vee B)}{(A \vee B)\Theta},$$

kde je Θ substituce unifikující p a q, a kde se $(A \vee B)\Theta$ nazývá resolventou.
Předchozí výrazy lze zapsat jako $\frac{C_1 \wedge C_2}{R}$ a to jako $C_1 \wedge C_2 \rightarrow R$. Resolventa R tedy není ekvivalentní původní konjunkci klauzulí C_1 a C_2. Proto, pokud je resolventa nesplnitelná, je nesplnitelná i původní konjunkce. To znamená, že pokud se pomocí rezoluce povede odvodit v množině klauzulí M prázdnou klauzuli, pak teorie

M nemůže mít žádný model, a je sporná. Nebo naopak, je-li teorie M sporná, pak v ní lze pomocí rezoluce odvodit prázdnou klauzuli, což je kontradikce.

Rezoluční metodu lze aplikovat na predikátové formule, které jsou v tzv. **Skolemově normální formě (SNF)** nebo, v případě výrokové logiky, v konjunktivní normální formě.
SNF je metoda pro odstranění existenčních kvantifikátorů z predikátových formulí. Formule predikátové logiky je v SNF, pokud je v tzv. prenexní normální formě (PNF). Má pro formuli X tvar $Q_1x_1Q_2x_2 \ldots Q_nx_nB$, kde Q_i je všeobecný nebo existenční kvantifikátor a x_i jsou navzájem různé proměnné, B je formule tvořená pouze z elementárních výrokových symbolů $\neg, \wedge$ a $\vee$. Výraz Q_ix_i se nazývá prefixem formule X. Ne každá formule v PNF je v také v SNF a to v tom případě, obsahuje-li nějaký existenční kvantifikátor. Ke každé formuli X predikátové logiky existuje formule X' v PNF (a SKF), která je splnitelná, právě když je formule X splnitelná. Výsledná formule tedy nemusí být nutně ekvivalentní původní formuli.
Nejjednodušší příklad skolemizace je např. pro výraz $\exists x\ P(x)$, který se změní na $P(c)$, kde c je nová konstanta (ještě se nikde ve formuli nevyskytuje). Obecně je skolemizace provedena nahrazením každé existenčně kvantifikované proměnné y termem $f(x_1, \ldots, x_n)$, kde je f nový funkční symbol. Zde se f nazývá Skolemova funkce a term Skolemův term. Např. pro výraz $\forall x\ (g(x) \vee \exists y R(x, y))$ dostaneme výraz $\forall x\ (g(x) \vee R(x, f(x)))$. Zde $f(x)$ je Skolemův term a f Skolemova funkce, která mapuje x na y.

Postup rezoluční metody pro predikátovou logiku:

- převod všech formulí do SNF,
- přidání negovaného závěru S do množiny předpokladů,
- opakuj dokud se dojde ke sporu nebo nelze nic nového odvodit:
  - vyber 2 klauzule,
  - proveď rezoluci s užitím dané unifikace,
  - pokud je resolventa prázdná klauzule, nalezli jsme spor (S plyne z předpokladů),
  - pokud ne, přidej resolventu do předpokladů.

4.6.1. SLD rezoluce

Rezolučních metod, kromě základní rezoluční metody, je celá řada. Pro jazyk PROLOG nás zajímá SLD (Selective Linear Definite clause) rezoluční metoda. Daný typ rezoluce (např. LI-rezoluce, LD-rezoluce) obecně dává nějaké omezující podmínky na rodičovské klauzule nebo resolvované klauzule a udává způsob

odvození další resolventy.
Lineární rezoluce je obecně rezoluce, kde lze rezoluční důkaz zapsat jako lineární sekvenci klauzulí $C_1, C_2, \ldots, C_e$, kde je klauzule C_1 vstupní klauzule. Každá další klauzule C_{i+1} je resolventou nějaké předchozí rodičovské klauzule C_i. Pokud je poslední klauzule C_e prázdná klauzule, dojde k refutaci. Nyní uvedu základní typy lineárních rezolucí, až k SLD rezoluci.

Lineární rezoluce je rezoluce ve tvaru posloupnosti dvojic $\langle C_0, B_0\rangle, \ldots, \langle C_n, B_n\rangle$, kde $C = C_{n+1}$, S je vstupní klauzule, C dokazovaná formule a B_i boční klauzule.

1. C_0 a každé B_i jsou z S nebo nějaké C_j, kde $j < i$,
2. každé $C_{i+1}, i \leq n$ je resolventou C_i a B_i.

Lineární vstupní rezoluce (LI) množiny $S = P \cup G, \langle G_0, C_0\rangle, \ldots, \langle G_n, C_n\rangle$ je lineární vyvrácení S, která začíná klauzulí G, boční klauzule jsou jen klauzule z P. G_{i+1} je resolventa G_i a C_i.

Lineární vstupní rezoluce pro uspořádané klauzule (LD) je rezoluční vyvrácení $P \cup G$ ve tvaru posloupnosti $\langle G_0, C_0\rangle, \ldots, \langle G_n, C_n\rangle$ uspořádaných klauzulí, že $G_0 = G, G_{n+1} = \square, C_i \in T$ a každé $G_i, 1 \leq i \leq n$ je resolventou uspořádaných klauzulí G_{i-1} a C_{i-1}.

Lineární vstupní rezoluce se selekčním pravidlem (SLD) je rezoluce, kde je selekční pravidlo libovolná funkce, která vybere literál z každé uspořádané cílové klauzule. SLD rezoluce je úplná (umí klauzuli vyvrátit, pokud je to možné), pro Hornovy klauzule. Hornovy klauzule jsou klauzule ve formě disjunkce literálů, které obsahují nejvýše jeden pozitivní literál.
Pro množinu klauzulí P a cílovou klauzuli $G = \neg A_1(x), \ldots, \neg A_n(x)$ se dokazuje nesplnitelnost $P \cup G$, tedy $P \wedge \forall x\ (\neg A_1(x) \vee \ldots \vee \neg A_n(x))$. To je ekvivalentní $P \models \exists x\ (A_1(x) \wedge \ldots \wedge A_n(x))$. Zadáním cíle G se chce zjistit, zda existují nějaké objekty, které na základě P splňují formuli $A_1(x) \wedge \ldots \wedge A_n(x)$. Po aplikaci všech mgu, které postupně použijeme při SLD odvozování na jednotlivé proměnné x, získáme konkrétní případy zmíněných objektů splňující danou formuli. V í-tém kroku se resolvuje na literálu $R(G_i) = A_i$.

Odvození definitního cíle G' z cíle $G =\leftarrow G_1, \ldots, G_m$ a definitní klauzule $C = C_0 \leftarrow C_1, \ldots, C_n$ pomocí substituce θ nastane, pokud existuje $i \in 1, \ldots, m$ a platí:

- G_i je unifikovatelná s C_0,
- substituce θ je $mgu(G_i, C_0)$,
- $G' =\leftarrow G_1\theta, \ldots, G_{i-1}\theta, C_1\theta, \ldots, C_n\theta, G_{i+1}\theta, \ldots, G_m\theta$.

SLD derivace pro definitní program P a definitní cíl G se tedy skládá z posloupnosti definitních cílů $G = G_0, G_1, \ldots$, posloupnosti klauzulí $C_1, C_2, \ldots$ a posloupnosti nejobecnějších unifikátorů $\theta_1, \theta_2, \ldots$, tak, že každá G_{i+1} vznikne z G_i a C_{i+1} použitím θ_{i+1}.
Derivace může být:

- nekonečná – derivace není důkazem spornosti, protože každý cíl G_i je neprázdný,
- konečná neuspívající – derivaci již nelze prodloužit dalším krokem odvození a cíl $G_n \neq \square$,
- konečná uspívající (refutace) – derivace je důkazem spornosti, posledním krokem odvození je $G_n = \square$.

4.6.2. SLD stromy

SLD rezoluce je často zobrazovaná v tzv. SLD derivačním stromu. SLD strom pro program P a cíl G je strom, kde v kořenu je cíl G. Libovolný uzel G' je uzel, kde jeho bezprostřední následník je výsledkem rezoluce nejlevějšího literálu G'. SLD strom tedy graficky reprezentuje celý výpočet nebo jeho nějakou část. Výpočetní strom může být už u jednoduchých pravidel značně rozsáhlý. Proto se pro znázornění výpočtu užívá spíše při výuce, než při praktickém psaní příkladů. Úspěšné cesty v SLD stromu jsou ty, které končí $\square$, ostatní jsou neúspěšné. Mechanismus PROLOGu prochází (nebo přesněji postupně vytváří, jak probíhá výpočet a odvozování) SLD strom do hloubky zleva doprava a hledá první úspěšnou cestu, poté další při zadání středníku – provedení backtrackingu. Případně projde celý strom, pokud nedojde k zacyklení v nekonečné větvi, a prohlásí, že daný cíl není logickým důkazem programu – odpověď No.

SLD strom pro $P \cup G$ zachycuje všechny možné větve výpočtu. Kořen stromu je ohodnocen definitním cílem G. Hrany ve stromu jsou ohodnoceny programovými klauzulemi. Jednotlivé uzly ve stromu reprezentují stavy výpočtu. Cesta od kořene k listu reprezentuje danou SLD derivaci. Podle stavu listu je derivace nekonečná, konečná neuspívající nebo refutace. Pořadí prohledávání je dané jednoznačným očíslováním programových klauzulí shora dolů a jednoznačným očíslováním subcílů v cíli G zleva doprava. Při hledání do hloubky, které užívají překladače PROLOGu, je výpočet efektivní s lineární paměťovou složitostí, ale výpočet nemusí být nalezen, pokud se vydá po nekonečné větvi. Při hledání do šířky je vždy nalezen prázdný cíl (pokud nějaký existuje), protože se strom prohledává po jednotlivých patrech, ale exponenciálně roste paměťová složitost. Proto se často v reálných programech užívá jakéhosi kompromisu mezi těmito dvěma základními způsoby a užívají se následující přístupy. Dvě tzv. neinformované metody a jedna informovaná.

- Neinformované metody – nemají při prohledávání k dispozici žádné další znalosti o prohledávaném prostoru, které by umožnily rychlejší průchod k cíli. Prochází systematicky všechny uzly v nějakém pořadí, dokud nenajdou řešení.

- Informované metody – při prohledávání mají navíc nějakou znalost o prohledávaném prostoru a aktuální části řešení. Pomocí nějaké heuristické metody se snaží odhadnout, jak daleko se aktuální řešení od cílového nalézá, a podle toho případně změnit prostor, který prohledávají.

Mezi neinformované metody patří prohledávání do hloubky s omezením a iterativní prohledávání do hloubky. Mezi informované například A* algoritmus.
Prohledávání do hloubky s omezením je modifikace běžného prohledávání do hloubky, ke kterému je navíc přidáno číslo. To reprezentuje maximální limit hloubky při prohledávání. Pokud je tento limit překročen, provede se návrat se do rodičovského uzlu a již se nepokračuje v dalším hledání v potomcích aktuálního uzlu, i když by to bylo dále možné. Tímto se dá vyhnou zacyklení výpočtu v nějaké nekonečné větvi výpočetního stromu. Vystavujeme se tím ale nebezpečí, že nějaké platné řešení ztratíme, takže je potřeba limit hledání adekvátně zvolit k danému problému.
Iterativní prohledávání do hloubky je spojení metod prohledávání do hloubky a šířky. Jedná se o modifikaci předchozí metody, kdy se v jednotlivých iteracích prohledává daný strom s tím, že se jednotlivé iterace postupně od nuly (kořen) až do nějakého limitu zvětšují. Metoda postupně vrací řešení jako při prohledávání do hloubky s omezením po jednotlivých patrech stromu. Pro limit 0 leží řešení v kořenu stromu, pro limit 1 leží řešení v potomcích kořene v pořadí zleva-doprava (běžný průchod do hloubky), a tak dále až do limitu.
Algoritmus A* je algoritmus pro vyhledávání optimální cesty v grafu. V principu se jedná o Dijkstrův algoritmus, který navíc užívá nějaké heuristiky pro aktuální řešení, které reprezentuje aktuální uzel výpočtu. Z počátečního uzlu se snažíme najít nejkratší nebo nejlevnější cestu do cílového uzlu.

5. Popis jazyka PROLOG a jeho vlastností

Popisem jazyka PROLOG a jeho vlastností se zabývají autoři I. Bratko, W. Clocksin a M. Bramer, z jejichž knih jsem v této kapitole čerpal. Užívám i informací z referenčního manuálu SWI-Prologu. Nebudu zde uvádět kompletní popis jazyka a suplovat tak referenční manuál, ale uvedu základy společné pro většinu implementací PROLOGu, především ale pro SWI-Prolog. Některé se ovšem mohou lišit dle daného interpretu. [1], [2], [3], [23]

5.1. Datové typy a proměnné

Jediný datový typ v PROLOGu je term. Žádná typová kontrola ani zadání typu při zapsání termu neprobíhá. (některé nové implementace mají ale typování zavedeno) Termy můžeme rozdělit na:

- struktury,
- jednoduché termy:
  - proměnné,
  - konstanty:
    - čísla,
    - atomy.

Obecná BNF syntaxe PROLOGu[3]

```
<program> ::= <clause list> <query> | <query>
<clause list> ::= <clause> | <clause list> <clause>
<clause> ::= <predicate> . | <predicate> :- <predicate list>.
<predicate list> ::= <predicate> | <predicate list> , <predicate>
<predicate> ::= <atom> | <atom> ( <term list> )
<term list> ::= <term> | <term list> , <term>
<term> ::= <numeral> | <atom> | <variable> | <structure>
<structure> ::= <atom> ( <term list> )
<query> ::= ?- <predicate list>.
<atom> ::= <small atom> | ' <string> '
<small atom> ::= <lowercase letter> | <small atom> <character>
<variable> ::= <uppercase letter> | <variable> <character>
<lowercase letter> ::= a | b | c | ... | x | y | z
<uppercase letter> ::= A | B | C | ... | X | Y | Z | _
<numeral> ::= <digit> | <numeral> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

[3]Syntaxe je převzata z webu http://cseweb.ucsd.edu

```
<character> ::= <lowercase letter> | <uppercase letter> |
                <digit> | <special>
<special> ::= + | - | * | / | \ | ^ | ~ | : | . | ? |  | # | $ | &
<string> ::= <character> | <string> <character>
```

5.1.1. Čísla a atomy

Původní specifikace PROLOGu podporovala pouze celá čísla. Pozdější implementace přidávají práci s reálnými i racionálními čísly. Běžně se užívají externí aritmetické knihovny, např. GNU GMP. Aritmetické predikáty jsou vestavěné predikáty, které pracují s aritmetickými výrazy. Výraz je buď běžné číslo nebo funkce. Funkce jsou termy, které jsou vyhodnocovány aritmetickými predikáty. Argumenty funkcí jsou opět výrazy.
Atomy jako nedělitelné termy lze rozdělit do tří skupin. Na atomy skládající se ze speciálních znaků (př. interpunkce, matematické symboly). Atomy jako řetězce znaků začínající malým písmenem, které obsahují pouze písmena, číslice a podtržítko. A atomy uzavřené v apostrofech pro složitější řetězce složené z různých znaků. Každý atom může být reprezentován i jeho ASCII nebo Unicode hodnotou. PROLOG také umožňuje pracovat se znaky jako s binárními proudy.

5.1.2. Proměnné

Proměnné v logickém programu mají odlišný význam než v běžném např. procedurálním programování. Neslouží přímo pro uložení nějaké hodnoty do paměti, ale pro vyjádření nějaké entity, reprezentaci vztahu k jiným entitám a její reprezentaci v logickém programu. Vyskytují se v predikátech, kde popisují vztahy mezi objekty, nebo v dotazech, kde reprezentují hledaný objekt. Jejich názvy začínají velkým písmenem nebo podtržítkem, pokud se jedná o anonymní proměnnou. Rozsah platnosti proměnné je pouze v jedné klauzuli, tedy od začátku hlavy až po poslední predikát v těle pravidla. Hodnotu proměnné získáme pomocí unifikace a po jejím přiřazení se již dále nemění, pokud pomocí backtrackingu není požádáno o alternativní hodnotu.

Proměnné lze rozdělit na:

- volné – hodnota zatím není známa a hledáme ji,
- vázané – její hodnotu již z průběhu výpočtu známe,
- anonymní – značí se znakem podtržítko, její hodnota není důležitá a ve výsledku se nezobrazuje.

5.1.3. Struktury

Struktury jsou opět termy a slouží nám pro reprezentaci složitějších útvarů v programu mezi více (zanořenými) objekty. Skládají se z funktorů (názvů) a argumentů. Počet argumentů udává aritu. Speciálním případem struktur jsou řetězce a seznamy. Stejně jako fakta a pravidla se struktury odlišují svým názvem a poté aritou. Operátor `:-` pro definici pravidla je také struktura zapsaná v infixovém tvaru. Struktury jsou často znázorněny ve formě stromu, ať už se jedná například o aritmetický výraz nebo strukturu popisující slova ve větě.

5.2. Fakta, pravidla a dotazy

Fakta jsou základní klauzule, které v programu vypovídají o vlastnostech objektu (fakta s aritou 1) nebo vztazích mezi objekty (fakta s aritou 2 a více).
Pravidla jsou klauzule, které pomocí implikace umožňují z programu odvozovat nová fakta. Pravidla se zapisují ve tvaru: `hlava :- tělo`. Hlava reprezentuje odvozovaný fakt a tělo podmínky, kdy je fakt pravdivý. Tělo může obsahovat jeden nebo více cílů. Odvozením pravdivosti těla je odvozena i pravdivost hlavy.
Dotazy spouští výpočet programu a slouží k ověření platnosti nějakého tvrzení, které plyne nebo neplyne z našeho logického programu. Pokud je výpočet úspěšný a byl zadán dotaz s volnou proměnnou, pomocí unifikace je na proměnnou hodnota navázána a interpret nalezne jednu nebo více odpovědí. Dotazy za sebou lze v příkazové řádce řetězit do složitějších dotazů, podobně jako za sebou lze psát několik klauzulí v těle pravidla.

S fakty a pravidly lze pracovat i dynamicky. To znamená přidávat je a odebírat je z logického programu za jeho běhu. K tomu se užívají predikáty `assert/1` pro vkládání a `retract/1` pro odebírání. Existují i další varianty těchto predikátů pro vkládání termů na začátek nebo na konec databáze faktů (např. `assertz/1` nebo `retracta/1`). Každý takový dynamický predikát musí být v SWI-Prologu předem označen pomocí predikátu `dynamic/1`, jinak je považován za statický a není ho možné za běhu měnit. Pokud je potřeba pracovat přímo s hlavou a tělem nějakého pravidla, lze užít predikát `clause/2` ve formě `clause(Head, Body)`, kde proměnné Head odpovídá hlava první klauzule, která se s ní unifikuje. Následně se unifikuje tělo klauzule s proměnou Body.
Dynamická práce s klauzulemi není doporučována a měli bychom ji užívat zřídka. Může vést k těžko odhalitelným chybám. Vkládání nových klauzulí je doporučováno jen v případě, kdy nově vložená klauzule již z logického programu plyne. Mazání klauzulí je doporučováno, pokud jsou v logickém programu redundantní. V těchto případech není ovlivněn význam logického programu, ale je vylepšena jeho efektivnost.

5.3. Seznamy

Seznamy jsou také struktury a tím pádem i termy. Jsou definovány induktivně. Jejich definice je odvozena od seznamů v jazyce LISP. Prázdný seznam (atom), je seznam a značí se `[]`. Neprázdný seznam je definován jako tečkový pár `.(Hlava, Tělo)`, kde hlava je prvním prvkem seznamu a tělo zbytek seznamu. K definici seznamu slouží binární funktor `.` (tečka). Tento zápis se v logickém programu užívá spíše výjimečně a prvky seznamu jsou zapisovány do hranatých závorek oddělenými čárkami (např. `[a, b, c, d]`). U seznamů se často užívá další binární operátor a to `|` (roura), který slouží k přístupu k hlavě a zbytku těla seznamu (např. `[H1, H2|Rest]`). Seznamy mohou být samozřejmě různě zanořené do sebe (např. `[[a, b], [c], [d, [e, f]]]`). Zpracování seznamů lze považovat za jeden ze základů PROLOGu. Proto každý interpret PROLOGu většinou obsahuje mnoho vestavěných predikátů pro práci se seznamy. Z těch základních lze uvést např. `member/2`, `append/3`, `length/2`, `select/3` nebo `delete/3`.

5.4. Řetězce a znaky

Řetězce jsou posloupnosti znaků v uvozovkách nebo apostrofech a vyjadřují nějaký zapsaný text. Jako znaková sada se v PROLOGu používá opět dle implementace Unicode nebo ASCII, takže jednotlivé zapsané znaky jsou ekvivalentní svému číselném vyjádření. Pokud řetězec zapíšeme jako `X='test'`, dostaneme odpověď v textovém vyjádření `X=test`. Pokud ale zapíšeme `X="test"`, dostaneme odpověď v ASCII hodnotách `X=[116, 101, 115, 116]`. Zpracování jazyka byl jeden z úkolů PROLOGu již od začátku, proto stejně jako u seznamů v interpretu najdeme spoustu vestavěných predikátů např. pro převod jednotlivých hodnot, určení typu znaku (číslo, znak, bílý znak) a podobně.

5.5. Základní operátory

Operátory jsou z pohledu PROLOGu opět struktury, většinou zapsané v infixovém zápisu. Např. výraz `-(A,B)` se běžně zapisuje jako `A-B`. Operátory proto nepřináší nic nového, ale přispívají k lepší čitelnosti kódu. Většina operátorů je z důvodu efektivity implementována jako vestavě predikáty. V PROLOGu vystupují běžné aritmetické operátory, operátory porovnání s/bez přiřazením a s/bez vyhodnocením, operátor vyčíslení, operátor negace, operátory pro řízení běhu programu a samozřejmě operátory pro definici pravidla, dotazu a logické spojky reprezentující konjunkci (čárka) a disjunkci (středník).
PROLOG užívá dva logické nulární predikáty (nejsou to tedy přímo operátory, ale uvedu je zde) pro reprezentaci pravdy a nepravdy, `true` a `false`.
V PROLOGu je možnost definice vlastních operátorů v infixovém, prefixovém i postfixovém tvaru, pomocí výrazu: `:-op(priorita, tvar, jméno)`. Všechny

operátory je možné i předefinovat, kromě operátoru čárka. Prázdný seznam [] není možné definovat jako operátor. Priorita jednoduchého termu je 0.
Následující kód ukazuje všechny základní operátory v SWI-Prologu. První sloupec ukazuje prioritu operátoru (0-1200), druhý ukazuje pozici funktoru f a jeho argumentů x a případně y, třetí jméno (zápis) operátoru.[4]

```
1200 xfx -->, :-
1200 fx  :-, ?-
1150 fx  dynamic, discontiguous, initialization, meta_predicate,
         module_transparent, multifile, thread_local, volatile
1100 xfy ;, |
1050 xfy ->, *->
1000 xfy ,
990  xfx :=
900  fy  \+
900  fx  ~
700  xfx <, =, =.., =@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=,
         \=, \==, is, >:<, :<
600  xfy :
500  yfx +, -, /\, \/, xor
500  fx  ?
400  yfx *, /, //, rdiv, <<, >>, mod, rem
200  xfx **
200  xfy ^
200  fy  +, -, \
```

5.6. Operátory pro řízení běhu

Průběh výpočtu lze do jisté míry ovlivňovat třemi operátory pro řízení běhu (výpočtu) programu. Jsou to nulární operátory `fail`, `true` a `!` (řez). Dva operátory je možné užít dohromady v technice *cut-fail* nebo *fail-true*, pokud je třeba vypsat všechny možné alternativy nějakého faktu.
Operátor `fail` vždy selže. Lze ho jednoduše implementovat jako `fail :- x=y.`
Operátor `true` naopak vždy uspěje.
Operátor řezu slouží pro kontrolu procedurálního chování logického programu. Tento operátor nepatří do základní definice jazyka. Umožňuje ale výpočet programu zefektivnit, a proto ho většina nových implementací obsahuje. Řez redukuje prohledávaný prostor možných řešení tím, že dynamicky ořezává SLD-strom. Umožňuje vynechat ve výpočtu ta řešení, u kterých víme, že je nebudeme potřebovat a nebo by v našem výpočtu neměla smysl, byla nesprávná nebo pokud

[4]Operátory jsou převzaty z webu www.swi-prolog.org

víme, že se v dané sekci SLD-stromu již žádná řešení nevyskytují. Řez může tedy ovlivňovat efektivitu programu a výpočtu nebo může mít přímo vliv na obdržená řešení. Řez funguje následovně:

- při dopředném výpočtu je řez splněn a pokračuje se dalším cílem v těle klauzule,
- při zpětném chodu je řez aktivován,
- řez ořezává všechny klauzule pod aktuální klauzulí ve které se nachází řez,
- řez ořezává všechna alternativní řešení nacházející se v těle klauzule vlevo od řezu $(B_1, \ldots, B_k)$, ale neovlivňuje řešení nacházející se vpravo $(B_{k+2}, \ldots, B_n)$, pro klauzuli $A \leftarrow B_1, \ldots, B_k, !, B_{k+2}, \ldots, B_n$

Řez je často užíván pro vyjádření vzájemně rozdílných řešení. V těle klauzule je proveden nějaký, třeba aritmetický test. Následně je aplikován řez a pokračuje se v dalším výpočtu těle klauzule. Rozdílné řešení je díky řezu zanedbáno a program se ním nemusí zbytečně zabývat.
Rozlišují se dva druhy řezů a to zelený a červený řez. Zelený řez neovlivňuje význam programu. Po přidání nebo odebrání řezu neztratíme nebo neobdržíme po výpočtu žádné výsledky. Výpočet bude pouze efektivnější. Při přidání červeného řezu naopak některé výsledky ztratíme a tím pádem má přímo vliv na význam programu. Červený řez často užíváme v případě, kdy požadujeme nějaké první nalezené řešení a další nás už nezajímají. Po odebrání tohoto typu řezu ale logický program nemusí vždy fungovat správně, protože se při přidání červeného řezu změní množina cílů, které lze z programu dokázat. Proto se často doporučuje tomuto řezu spíše vyhýbat a raději v klauzuli zvolit nějakou omezující podmínku. Obecně je dobré napsat logický program první bez řezů, poté na vhodných místech doplnit zelené řezy, a nakonec pokud je to opravdu nutné, červené.

5.7. Negace

Negace se značí predikátem `not(X)` nebo nověji `\+(X)`. Oproti klasické negaci v logice však `not(X)` neznamená, že je výraz `X` nepravdivý. Definitní program zaznamenává platná fakta a obecně z něj nijak neplyne negace nějakého faktu. Jediný způsob, jak PROLOG může říct, že je daný výrok nepravdivý, je pokusit se to dokázat z daných faktů a pravidel. Pokud tento pokus selže, je výrok nepravdivý. To se nazývá jako **negace jako selhání**.
Problém při tomto způsobu dokazování je ten, že PROLOG nemá k dispozici nějaká fakta nebo pravidla a tak není schopný tento závěr dokázat, i když je pro člověka na první pohled tento závěr špatný (příklad na obrázku níže). Nepravdivost je tedy jen relativní k daným faktům a pravidlům, které PROLOG právě

```
% zeleny rez - efektivita programu
sign(N, positive) :-
        % N kladne, netreba zjistovat zaporne nebo nulu
        N>0, !.
sign(N, negative) :-
        % N zaporne, netreba zjistovat nulu
        N<0, !.
% zde jiz rez nema vyznam
sign(N, zero) :-
        N=:=0.

% cerveny rez - ztraceni vysledku
member(X, [X|_]) :-
        % prvek X nalezen na zacatku seznamu
        % jiz se nehleda dalsi vyskyt i kdyz tam muze byt
        !.
member(X, [_|List]) :-
        % na zacatku seznamu neni prvek X
        % prohledavame rekurzivne zbytek seznamu
        member(X, List).
```

Obrázek 2. Zelený a červený řez

užívá. Tento problém je znám jako **předpoklad uzavřeného světa**. Detailní popis negace z teoretického hlediska najdeme například v knize Foundations of Logic Programming od J. W. Lloyd, v kapitole 3. [5]

Implementace v interpretu PROLOGu je možná dvěma způsoby:

1. `not(X):-call(X),!,fail.`
2. `not(X).`

```
man(john).
man(adam).
woman(sue).
woman(eve).
married(adam, eve).
% vrátí adam a eve
married(X) :- married(X, _).
married(X) :- married(_, X).
% už nevrátí john a sue, PROLOG to nemá z čeho odvodit
not_married(X) :- \+married(X).
% úprava, zachytíme všechna fakta co potřebujeme
human(X) :- man(X); woman(X).
% vrátí john a sue
not_married2(X) :- human(X), not_married(X).
% obecně lze těžko a ne vždy zachytit všechna fakta pro to,
% aby PROLOG mohl vždy správně negaci odvodit
% př. položíme dotaz not_married2(ivan). -> false
```

Obrázek 3. Negace v PROLOGu

5.8. Práce se soubory, vstup a výstup

Tyto vestavěné predikáty jsou součástí každé implementace PROLOGu a umožňují výpis na konzoli nebo do souboru či načtení dat ze souboru nebo proudu. Se samotnou logikou nemají již nic společného, ale zpříjemňují nám práci v interpretu. Podobně jako u funkcionálního programování vykonávají nějaký vedlejší efekt mimo samotný logický výpočet. Po tom, co vykonají nějaký vedlejší efekt, je jejich pravdivost vždy true a není u nich umožněna alternativní volba výpočtu. Základní predikáty pro výpis a vstup jsou `write/1` a `read/1`. Predikát `write(X)` vypíše term `X` na zvolený výstup. Predikát `read(X)` načte term ze zvoleného vstupu a pokusí se ho unifikovat s `X`. Pro práci s externími soubory užíváme predikáty `see/1` a `tell/1`. Dále najdeme podle dané implementace různé predikáty pro formátovaný výstup a další, v SWI-Prologu například `writef/2` nebo `formatf/2`. Neméně důležitý je vestavěný predikát `consult/1`, který nám ze zvoleného souboru načte všechna fakta a pravidla a vloží do databáze našeho programu.

5.9. Vlastnosti jazyka

Mezi základní vlastnosti jazyka (nebo chceme-li vlastnosti užívané při výpočtu) patří rekurze, průchod do hloubky s backtrackingem a reverzibilita řešení. Do základních vlastností lze také zařadit reprezentaci dat v seznamech nebo složených strukturách a práci se symbolickými výrazy. Metody unifikace, substituce a rezoluce užívané při výpočtu, jsem popsal v předchozí teoretické části.

5.9.1. Rekurze

Rekurze je v PROLOGu jediná možnost, jak reprezentovat cykly (a mimo jiné nám tím zajišťuje sílu jazyka ekvivalentní Turingovu stroji). Umožňuje z logického programu odvodit složitější vztahy mezi termy. Ty by jinak nemohly být reprezentovány jen jako jednotlivá fakta. Jejich definice pomocí nerekurzivního pravidla by byla zbytečně složitá nebo by nebyla dostatečně obecná. Na rekurzivní pravidla je často nahlíženo jako na generalizaci množiny nerekurzivních pravidel. Součástí rekurzivního pravidla je limitní podmínka pro zastavení rekurze, která se uvádí jako první klauzule. Další klauzule je samotný předpis rekurze. Proto je u rekurze důležitý zápis pořadí jednotlivých pravidel, aby byl program efektivní nebo aby jeho výpočet neskončil v nekonečně větvi (např. při užití levé rekurze) derivačního stromu.
Rekurzivní pravidla se často týkají tranzitivního uzávěru. Tranzitivní uzávěr R' je binární relace R, kde $R' = \cap_M[(R \subseteq M) \wedge (\forall a, b, c)(([a, b] \in M \wedge [b, c] \in M) \Rightarrow ([a, c,] \in M))]$. Mezi fakty může existovat tranzitivní relace, která vyjadřuje přenos vztahu mezi prvky – $\forall x, y, z((R(x, y) \wedge R(y, z)) \rightarrow R(x, z))$. Tuto relaci

najdeme např. na příkladu uzlů v grafu (soused mého souseda je můj soused), které jsou nějak propojené a hledáme cestu, nebo na známém příkladu hledání předků (předek mého předka je také můj předek).

Rekurze vznikne jednoduše tak, že v těle pravidla opět zavolá klauzuli se stejným názvem, jako má hlava pravidla – přímá rekurze. Nebo se v těle zavolá pravidlo B, které opět volá první pravidlo A původní klauzuli – nepřímá rekurze. O lineární rekurzi mluvíme v tom případě, pokud se v těle pravidla provádí jen jedno rekurzivní volání. V opačném případě mluvíme o stromové rekurzi. U běžného lineárního rekurzivního volání probíhají fáze navíjení a odvíjení. Časová i paměťová složitost je lineární. Paměťovou složitost lze vylepšit koncově rekurzivním voláním s akumulátorem. Koncově rekurzivní volání je volání, na které není aplikována žádná jiná operace (např. u faktoriálu násobení $n * fac(n-1)$). Akumulátor je tvořen např. jednou proměnnou jako argument klauzule, která nám reprezentuje již spočítaný průběžný výsledek.

Lze najít mnoho příkladů užití rekurze. Jeden ze základních je rekurzivní prohledávání seznamu (nebo jiné složitější struktury), což je jedna z nejčastějších úloh v PROLOGu. Prohledává se od prvního prvku (hlava seznamu) a pokud ten nesplňuje zadanou podmínku, prohledává se rekurzivně zbytek prvků (tělo seznamu) až po prázdný seznam, který nám tvoří limitní podmínku hledání. Obecně takto fungují všechny rekurzivní postupy při zpracování seznamu (např. zjištění délky seznamu, zda se jedná o seznam, součet prvků a podobně). Obdobně se pracuje se dvěma nebo více seznamy, kdy opět pracujeme s hlavami seznamů a rekurzivně zpracováváme jejich těla.

Další užití rekurze je v rekurzivních datových typech. Nejznámější takový typ je binární strom definovaný jako `tree(Element, Left, Right)`, kde *Element* je prvek v daném uzlu, $Left$ levý následovník (podstrom) aktuálního uzlu, *Right* pravý. Limitní podmínka zde tvoří prázdný strom ve formátu `tree(void)`. Strom může mít například tvar `tree(a,tree(b,void,void),tree(c,void,void))`. Rekurze zde spočívá v tom, že následovník uzlu – nějaký podstrom, je také strom. Podobně jako při prohledávání seznamu zde prohledáme aktuální uzel a pokud prvek nebyl nalezen, rekurzivně pokračujeme v hledání v levém a pravém podstromu.

K rekurzi lze užít vestavěný predikát `repeat`, který vždy uspěje a provádí tak nekonečný cyklus.

5.9.2. DFS a backtracking

Průchod do hloubky (DFS) a backtracking jsou běžné algoritmy pro nalezení všech řešení v určitém problému. Řešení se zde buduje postupně a případně se navracíme zpět k dalšímu kandidátovi, pokud nám předešlý nevyhovuje, a nebo je řešení jen částečně správné a nelze již dál ve výpočtu pokračovat. V tom případě se výpočet vrátí o několik kroků zpět a pokračuje se v hledání dalšího řešení od posledního platného kroku. PROLOG metodu backtrackingu užívá při

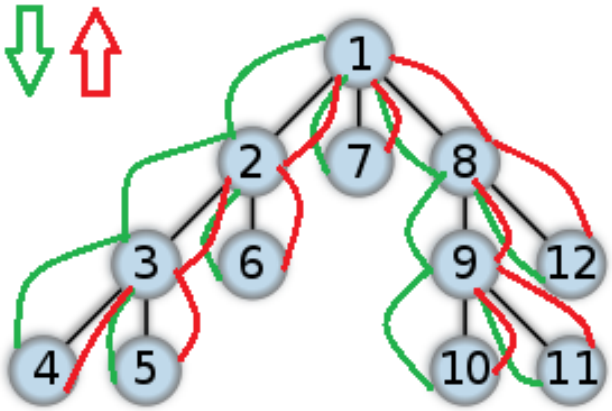
```
% klasický cyklus
my_repeat.
my_repeat :- my_repeat.

% rekurzivní prohledávání seznamu
member(X, [X|_]).
member(X, [_|List]) :-  member(X, List).

% rekurzivní hledání předků
parent(john, paul).
parent(paul, tom).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :-
        parent(X, Z),
        ancestor(Z, Y).

% prohledávání binárního stromu
bint_s(node(X, _, _), X). % v uzlu
bint_s(node(Y, L, _), X) :-
        % levý podstrom
        X < Y,
        bint_s(L, X).
bint_s(node(Y, _, R), X) :-
        % pravý podstrom
        X > Y,
        bint_s(R, X).
```

Obrázek 4. Ukázka rekurze

odvození všech řešení, které z programu plynou, pokud se nepožaduje jen první nalezené. Řešení dostáváme postupně z SLD stromu metodou průchodem do hloubky, kdy je postupně možné dostat všechny platné řešení, které odpovídají zadanému dotazu. Zelená čára v obrázku číslo 5 zobrazuje dopředný průchod ve stromu, červená zpětný návrat do předka uzlu. Uzly jsou procházeny v pořadí ve kterém jsou očíslovány. V každém uzlu SLD stromu se samozřejmě nemusí nalézat platné řešení.

5.9.3. Reverzibilita

Funkcionální programování může být považováno za podmnožinu logického programování. Funkce lze považovat za zvláštní případ relací. Díky relační povaze predikátů je možné je používat v obou směrech. Například predikát `length/2`,



Obrázek 5. DFS+Backtracking

který se používá pro určení délky seznamu, dokáže stejně dobře generovat kostru seznamu požadované délky. Když se nad tím zamyslíme, je to velice silná vlastnost, kterou neposkytuje žádné jiné paradigma. V PROLOGu nám pro obdržení několika výsledků stačí napsat třeba jen jedno pravidlo s danou hlavičkou, kde každý výsledek může reprezentovat určitá proměnná v hlavičce.

5.9.4. Reprezentace dat v seznamech, struktury

Při reprezentaci dat máme v PROLOGu na výběr ze dvou možností (pokud neuvažujeme to, že jsou data načítána ze souboru a ihned zpracována). Lze je reprezentovat jako jednoduchá fakta, případně složenou strukturu, nebo je reprezentovat různě zanořeným seznamem. Oba způsoby mají své výhody a nevýhody a často záleží jen na programátorovi, jak k danému problému přistoupí.
Většina implementací navíc obsahuje tři užitečné nástroje, a to operátor `=..` a vestavěné predikáty `functor/3` a `arg/3`. Predikát functor pomáhá analyzovat term (např. `functor(jmeno(petr, maly), N, A). N=a, A=2`), kde první argument je term, druhý vrací jeho název a třetí aritu. Predikát arg vrací n-tý argument termu. Zmíněný operátor `=..` převádí term na seznam a naopak (např. `jmeno(petr, maly) =.. List, List = [jmeno, petr, maly]`). Proto je možné podle potřeby seznamy převádět na struktury a naopak. Pro inspekci struktury nebo při zpracování dat lze užít predikáty `integer/1`, `atom/1`, `compound/1`, `var/1`, `ground/1` a jiné pro určení typu daného termu. Velice často se data reprezentují nějakým, třeba binárním, stromem.

```
% příklad struktury - tries
% slova ve stromu: to, te, tea, ted, ten | a | i, in, inn
tree(root, ' ',
      [tree(t, nil,
            [tree(o, to, []),
             tree(e, nil,
                  [tree(a, tea, []),
                   tree(d, ted, []),
                   tree(n, ten, [])])]),
       tree(a, a, []),
       tree(i, i,
                [tree(n, in,
                     [tree(n, inn, [])])])]).

% zanořený seznam zamestnanec
zamestnanec([jmeno, 'Jan', 'Maly'],
            [adresa, [ulice, 'Kratka', 33],
                     [mesto, 66004, 'Olomouc']],
            [funkce, ucetni],
            [plat, 15000],
            [pracoviste, [budova, a3,
                                  [kancelar, 115]]]).
```

Obrázek 6. Ukázka reprezentace dat

6. Implementace PROLOGu

6.1. Interpret SWI-Prolog

Tento interpret je v současné době jedna z nejoblíbenějších implementací jazyka PROLOG, užívaná jak pro výuku logického programování, tak pro tvorbu reálných aplikací. Pochází od autora Jana Wielemakera, který vytvořil první verzi již v roce 1987. Jan Wielemaker je autorem několika desítek vědeckých článků, převážně o logickém programování a PROLOGu, a zabývá se např. i sémantickými weby. Je zároveň autorem grafického rozšíření XPCE. V současné době se na vývoji stále podílí spolu s Anjo Anjewierdenem a dalšími pracovníky z University of Amsterdam. Zkratka SWI znamená Sociaal-Wetenschappelijke Informatica (anglicky „Social Science Informatics“) a je to označení skupiny na univerzitě, kde zmínění autoři pracují.

SWI-Prolog byl napsán v programovacím jazyce C (standardu C99), jistá část je napsána v samotném PROLOGu. Jedná se o multiplatformní interpret, který podporuje 32bit i 64bit systémy Unix, Windows XP/Vista/7/8, Mac OS a linuxové systémy. Je distribuován pod LGPL licencí a je možné ho opatřit z webu swi-prolog.org. Kromě samotného interpretu v binární podobě (po instalaci cca 35MB v systém Windows, samotné jádro má velikost pouze 650KB) je možné

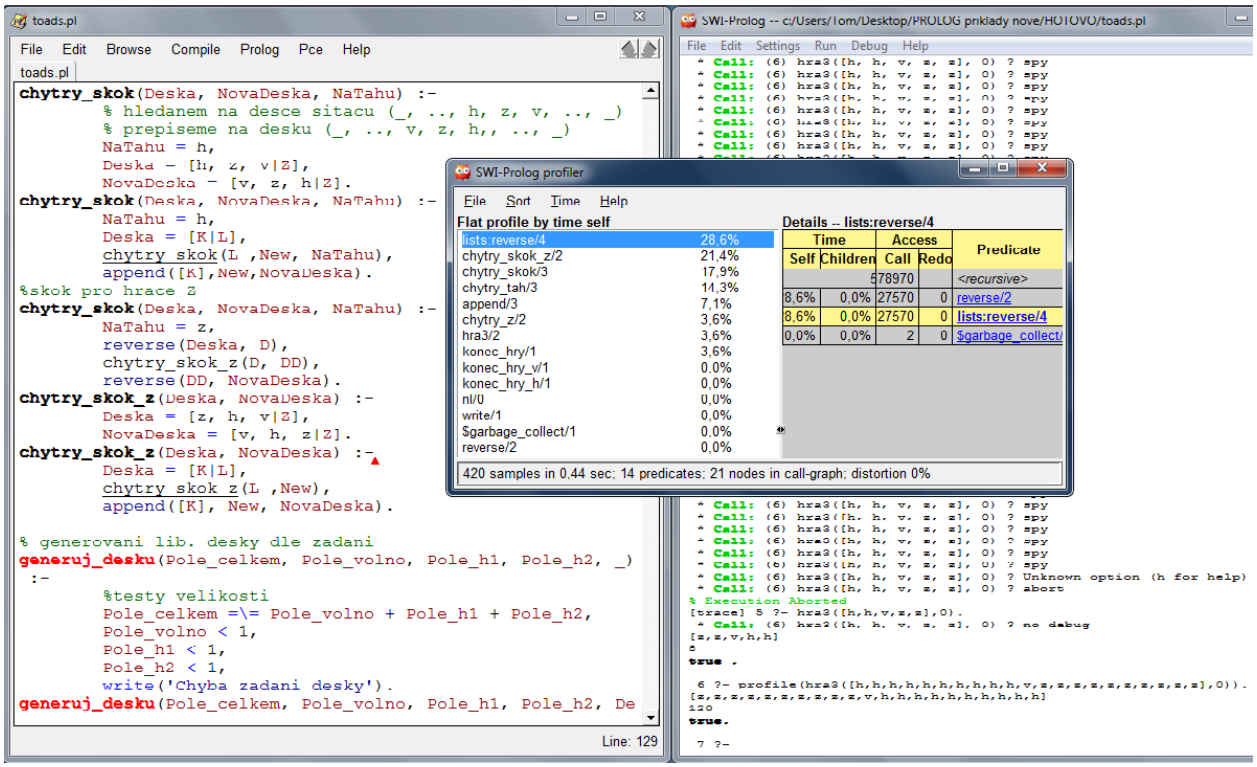
z webu stáhnout zdrojové kódy interpretu v jazyce C, kompletní dokumentaci ke zdrojovým kódům a SWI-Prologu, oblíbené starší verze a několik rozšiřujících balíčků. Interpret v současnosti nalezneme jen v anglickém jazyce. Jeho vývoj probíhá dodnes a poslední stabilní verze pro stažení (v době psaní této práce) má označení 6.4.1. SWI-Prolog ve své implementaci používá standard ISO Prolog 1, stejně jako Edinburgh Prolog a některé hlavní části Quintus a SICStus Prologu. Je z velké části kompatibilní s mnoha jinými implementacemi, především s Ciao, YAP a GNU Prologem. Rychlost interpretace kódu není špatná, ale mezi ostatními (hlavně placenými) implementacemi PROLOGu se považuje spíše za průměrnou. Interpret podle autorů nemá žádné limity pro velikost programu, délku atomů a aritu termů (referenční manuál sice uvádí omezení na hodnotu 1024, ale lze to změnit), což není u všech implementací běžné. Poskytuje indexaci predikátů a vyhledávání v hašovací tabulce, takže rychlost neklesá ani při velmi velkých programech. Při práci s aritmetikou se užívá GNU GMP knihovna, takže běžné matematické výpočty probíhají relativně rychle. Pro náročnější výpočty obsahuje rozhraní do programu MATLAB.

Kromě samotné implementace jazyka PROLOG obsahuje mnoho dalších vylepšení. Umožňuje běh programu ve více vláknech, obsahuje IDE, knihovny pro programování s omezujícími podmínkami (knihovny Constraint Handling Rules, clp(FD), clp(R,Q) a další). Má rozhraní do jazyků C, C++, Java (JPL), Python, C Sharp a ODBC systémů. Obsahuje knihovnu Web server pro např. generování HTML, HTTP autentizace nebo správu sezení. Podporuje parsování SGML/XML a RDF souborů. Obsahuje grafický debugger a vestavěný PceEmacs editor s podporou zvýraznění syntaxe a její kontroly. Nástroj PlUnit pro testování jednotek (unit testing). Umožňuje překlad kódu do .exe souborů pomocí externích programů.

Ačkoliv je SWI-Prolog určen především pro úlohy umělé inteligence, zpracování jazyka a podobně, obsahuje už od roku 1987 nástroj XPCE pro tvorbu GUI s běžnými komponentami (menu, tlačítka, popisky, záložky...). Nabízí i další externí možnosti grafiky. Samotné IDE prostředí SWI-Prologu umožňuje nastavení základních parametrů jako barvy a velikosti písma a základní nastavení XPCE GUI. [23], [30]

6.2. Vestavěné predikáty a knihovny SWI-Prologu, moduly

Implementace SWI-Prologu obsahuje mnoho vestavěných predikátů a knihoven. Tyto pomocné nástroje jsou součástí prostředí SWI-Prologu. Mnoho knihoven je přímo odvozeno z ISO Prologu. Některé jsou upravené přímo pro SWI-Prolog a, podle tvrzení autorů, daleko efektivnější, než ISO specifikace. Vestavěné predikáty je možné ihned užívat při spuštění interpretru, stejně tak jako většinu knihoven. Ovšem některé knihovny se musí předem načíst. To se provede snadno



Obrázek 7. Prostředí SWI-Prologu

příkazem `consult/1` nebo `use_module/1`. Stejně tak je možné načíst svůj vlastní zdrojový kód nebo svoji vytvořenou knihovnu, protože vestavěné predikáty a knihovny nejsou nic jiného, než napsané zdrojové kódy.

Mohli by nás napadnout dvě přirozené otázky, a to jaký je rozdíl mezi knihovnami a vestavěnými predikáty a proč je dobré je používat. Rozdíl mezi nimi není veliký a některé vestavěné predikáty jsou dokonce součástí běžných knihoven. Pokud se ale vestavěný predikát pokusíme přepsat (předefinovat v kódu naším vlastním, tj. bude mít stejné jméno a počet argumentů), nastane chyba. To u knihovny nenastane a v kódu jde používat stejné názvy predikátů, jako jsou definované v načtené knihovně. Poté je ovšem potřeba rozlišit jednotlivé části kódu pomocí modulů. Knihovna je tedy souhrn nějakých predikátů.

Moduly tvoří jmenný prostor pro nějaké predikáty a knihovny moduly obvykle obsahují. Jednotlivé moduly zapouzdřují nějakou množinu predikátů a prvně je definují v hlavičce názvem a počtem argumentů. Mohou importovat jiné moduly a tak explicitně vyjadřovat závislosti mezi nimi. Toto vše dobře umožňuje zpřehledňovat rozsáhlé projekty a měnit jednotlivé moduly. Moduly jsou definované přímo ve standardu ISO Prolog, bohužel je ale mnoho jiných verzí interpretů nepoužívá nebo nejsou navzájem kompatibilní. SWI-Prolog obsahuje dva speciální typy modulů. Systémový (system), který obsahuje všechny vestavěné predikáty. Uživatelský (user), který uživatel načítá běžně přes konzoli interpretu

nebo ze souboru a neobsahuje v sobě žádnou explicitní deklaraci cizího modulu. Uživatelský navíc automaticky importuje systémový modul. Ostatní běžné moduly automaticky importují predikáty z modulu user, takže mohou užívat všechny predikáty.

Knihovny SWI-Prologu obvykle začínají hlavičkou s komentářem autora (kdy byla knihovna vytvořena, kdo ji vytvořil, čeho se týká atd.). Poté následuje definice modulu ve formě : $-module('MyModule', [my\_predicate\_1/2, \ldots, my\_pr\_n/4])$. První argument je jméno modulu, poté následuje seznam jmen predikátů s jejich aritami, které v dané knihovně definujeme. Většinou je dobrým zvykem přidat pro každý predikát do komentáře definici argumentů a to, zda je vstupní (znak +), výstupní (−) nebo vstupně-výstupní (?). Samotná knihovna může užívat jiné knihovny nebo moduly. Ty jsou následně uvedené pod definici modulu, např. : $-use\_module(library(error))$. Následuje ostatní nastavení jako nastavení kontroly prostředí (environment control) pomocí tzv. vlajek (Prolog flags) a podobně. Nakonec následují samotné definici predikátů.
Rozdíl je i ve velikosti výsledné aplikace, kde při užití predikátů z knihovny užíváme jen ty, které skutečně používáme. Vestavěné predikáty se načítají pro danou implementaci vždy všechny.

Užívání vestavěných predikátů a knihoven má i další výhody. Programátorovi šetří námahu a čas, takže pokud danou knihovnu nebo predikát známe, lze ho použít a není třeba ho pracně vymýšlet. Navíc známé predikáty (např. zpracování seznamů) jsou napsány efektivním a prověřený způsobem. Většina vestavěných predikátů SWI-Prologu je napsána v jazyce C, takže jsou i velice rychlé. Mnoho vestavěných predikátů je i u různých implementací (především Quintus a SICStus Prolog) společných podle ISO standardu, proto lze snadno přejít z jedné na druhou či upravit cizí kód. Navíc bývá zdrojový kód knihoven běžně přístupný a dobře zdokumentovaný, takže není problém kód daného predikátu získat a případně upravit dle svých potřeb.
Nyní uvedu několik základních zajímavých knihoven:

- aggregate – obsahuje agregační predikáty a definuje agregační operace, např. `sum`, `min`, `max` a jiné,
- apply – definuje predikáty, které aplikují nějaký predikát na seznam, např. `maplist`, `foldl`,
- charsio – obsahuje predikáty pro převod termů, atomů a čísel na posloupnosti ASCII znaků,
- clpfd – knihovna pro logické programování s omezujícími podmínkami pro celá čísla, rozšiřuje běžnou aritmetiku SWI-Prologu a umožňuje definovat různé podmínky na řešení, např. $X\# > 3 \rightarrow X$ in $4 \ldots sup$,
- clpqr – obdoba knihovny clpfd pro práci s racionálními a reálnými čísly,

- csv – knihovna pro zpracování dat ve formě CSV souborů,
- lists – knihovna obsahující základní predikáty pro manipulaci se seznamy, např. `member`, `append`, `select`, `permutation` a mnoho dalších,
- ordsets – knihovna pro práci s uspořádanými množinami (seznamy),
- random – obsahuje predikáty pro generování náhodných čísel,
- readutil – knihovna s predikáty pro čtení souborů, jednotlivých řádků a proudů,
- XPCE – knihovna pro vytváření GUI aplikací,
- semweb/rdf_db a http/http_open – knihovny pro práci s web. rozhraním a sémantickými weby.

Detailní popis všech knihoven a vestavěných predikátů najdeme v referenčním manuálu `http://www.swi-prolog.org/pldoc/refman/`. [9], [23]

6.3. Další implementace PROLOGu

SWI-Prolog není jediný interpret PROLOGu. Nyní se zmíním i o ostatních implementacích a uvedu jejich základní vlastnosti a rozdíly mezi jednotlivými interprety. Všechny implementace jsou samozřejmě založeny na predikátové logice 1. řádu a Hornových klauzulích. Teoretický základ je totožný, ale ne všechny nutně dodržují všechny zažité standardy nebo naopak ve své implementaci poskytují mnoho jiných rozšíření.

Interprety lze rozdělit několika způsoby. Nejzákladnější dělení se nabízí podle platformy, na které interpret můžeme spustit. Převážná většina nabízených interpretů je pod GNU a LGPL licencí nebo zdarma pro nekomerční využití. Interprety jsou většinou multiplatformní, převážně pro unixové systémy a systémy MS-Windows, případně pro systémy OS X (či dříve MAC OS X). Ale lze najít i specifické implementace pro JVM (Java virtual machine) nebo pro webové prohlížeče.

Další dělení je podle samotné syntaxe a vnitřního mechanismu PROLOGu. Drtivá většina dnešních interpretů užívá syntaxi ISO Prolog a Edinburgh Prolog. Proto by se mohlo na první pohled zdát, že kódy pro různé verze budou z velké části přenosné. Pokud se vyhneme užívání netypických a přidaných rozšíření v různých implementací a budeme se držet standardu, skutečně tomu tak z velké části bude. Bohužel převážná většina verzí nedodržuje standard ISO Prolog úplně do písmene, takže se drobným rozdílům většinou nevyhneme. To u syntaxe není až tak závažný problém, ale můžeme najít rozdíly i v sémantice. Takže dva stejné vestavěné predikáty se stejnými argumenty mohou v různých verzích vykonávat rozdílné výpočty, než očekáváme. Naštěstí se jedná spíše o výjimky a tudíž se s tímto problémem nesetkáme příliš často. Některé implementace jako např. Visual Prolog se vydaly svou vlastní cestou a rozdíly zde jsou jak v syntaxi, tak sémantice, na první pohled značné.

Další dělení se týká různých rozšíření a vlastností, případně optimalizací interpretů (např. webové podpory, poskytovaného grafického rozhraní, přítomnost debuggeru, možnost kompilovat kód, implementace vláken atd.). Velké rozdíly jsou i v rychlosti jednotlivých interpretů. Tato problematika je ovšem mimo rámec této práce a proto uvedu dva odkazy, kde si zájemci mohou něco o testování a srovnání rychlostí jednotlivých verzí přečíst:
http://www.cs.kuleuven.be/publicaties/rapporten/cw/cw312.pdf
http://www.stups.uni-duesseldorf.de/prob/index.php5

Sluší se uvést, že i když je PROLOG většinou považován za pomalý jazyk, dnešní implementace umí být přes mnoho optimalizací velice rychlé, umožňují kompilaci kódu a vkládání do ostatních programovacích jazyků. Takže jsou interprety PROLOGu schopny rychle vyhodnotit program skládající se ze statisíců řádků kódu během pár sekund. Samozřejmě nemohou konkurovat jazykům typu C, ale pro běžné aplikace je jejich rychlost dostačující. Nyní krátce popíšu tři vybrané zajímavé implementace.

Visual Prolog dnes patří k nejlepším placeným interpretům PROLOGu. I když je pouze pro systém MS-Windows a jeho syntaxe je značně odlišná, nabízí spoustu zajímavých rozšíření. Jeho jazyk je sice založen na Hornových klauzulích, unifikaci a metodě srovnávání vzorů (pattern matching). Je ale plně orientován na objektové programování a silně typovaný. Plně podporuje přímé volání WIN32 API funkcí a propojení s C/C++ a vícevláknové zpracování. Jeho vývojové prostředí umožňuje tvorbu GUI a obsahuje běžně používané komponenty i grafický debugger. Umožňuje rovněž tvorbu .exe i .dll souborů. Obsahuje také knihovny pro práci s webem, databázemi a mnoho dalšího pro tvorbu i velkých komerčních projektů. Pokud chceme programovat pouze pod MS-Windows velké projekty netradičním způsobem – pomocí logického paradigma, je to správná volba. Najdeme ho na webu http://www.visual-prolog.com/. [26]

```
class hanoi
   predicates
       hanoi : (unsigned N).
end class hanoi

implement hanoi
   domains
       pole = string.

   clauses
       hanoi(N) :- move(N, "left", "centre", "right").

   class predicates
       move : (unsigned N, pole A, pole B, pole C).
   clauses
       move(0, _, _, _) :- !.
       move(N, A, B, C) :-
           move(N-1, A, C, B),
           stdio::writef("move a disc from % pole to the % pole\n", A, C),
           move(N-1, B, A, C).
end implement hanoi

goal
   console::init(),
   hanoi::hanoi(4).
```

Obrázek 8. Ukázka kódu z Visual Prologu [5]

GNU Prolog patří do rodiny GNU projektů a pokud nepožadujeme až tak robustní verzi (bez GUI, podpory webu a podobně) jako Visual Prolog nebo SICStus Prolog, dostaneme zdarma velice dobrý interpret. Je založen na ISO standardu a navíc obsahuje užitečné rozšíření jazyka jako globální proměnné, DC gramatiky, přes 300 vestavěných predikátů a rozhraní do jazyka C. Dále obsahuje debugger a umožňuje kompilaci souborů do binární podoby. Jeho obrovská výhoda je podpora téměř všech běžných architektur (ix86, ix64, PowerPC, SPARC atd.) a operačních systémů (Unixové systémy, Win32, Win64, Solaris,

Mac OS X), takže nepřekvapí absence GUI rozhraní. Otevřený kód je u GNU projektů samozřejmostí. Najdeme ho na webu http://gprolog.univ-paris1.fr/. [25]

Ciao Prolog je další implementace zajímavá hned v několika bodech. Stejně jako GNU Prolog je multiplatformní a pod GPL/LGPL licencí. Užívá hned několik paradigmat a to logické, funkcionální a objektové. Logické paradigma je zde založeno opět na ISO standardu, ale je rozšířeno o logiku vyššího řádu a omezujících podmínek (stejně jako v SWI-Prologu). Navíc nám interpret umožňuje do jisté míry řízení výpočtu oproti klasickému PROLOGu, a to ve formě prohledávání do šířky, iterativního prohlubování a dalších. Z dalších paradigmat nám umožňuje vytvářet funkce nebo objekty. Jeho knihovny podporují paralelní zpracování, podporu webu i externího rozhraní do jazyků C, Javy nebo databází. Oproti předchozím nabízí zajímavé rozšíření ve formě automatického generátoru dokumentace. Je schopen z kódu PROLOGu a vložených komentářů vytvoří manuál v mnoha formátech jako HTML, pdf, man. Opět nabízí možnost debuggeru, kompilace a tvorbu (podle autorů velmi rychlých a malých) spustitelných souborů. Tvorba GUI však chybí. Najdeme ho na webu http://ciao-lang.org/. [24]

V následující tabulce jsou uvedeny nejužívanější implementace, základní vlastnosti a rozdíly vzhledem k SWI-Prologu. Pro více informací doporučuji webové stránky jednotlivých autorů. [27]

Název	OS	Licence	Syntaxe	Web	Vlákna	OOT	GUI
SWI	multi	LGPL	ISO	ano	ano	ne	ano
BProlog	multi	LGPL	ISO	ne	ne	ano	ne
Ciao	multi	LGPL	ISO	ano	ano	ano	ne
GNU	multi	LGPL	ISO	ne	ne	ne	ne
LPA	Win	komerční	Edinburgh	ne	ne	ano	ano
SICStus	multi	komerční	ISO	ano	ne	ano	ano
Visual	Win	komerční	vlastní	ano	ano	ano	ano
XSB	multi	LGPL	vlastní	ano	ano	ne	ne
YAP	multi	GPL	ISO	ne	ano	ne	ne

7. Příklady reálných aplikací

Jazyk PROLOG slouží dnes především pro studijní účely, ale jak jsem zmínil v úvodu, i přesto v tomto jazyce existuje řada praktických aplikací. V této kapitole jich několik z historie i současnosti uvedu a pokusím se ukázat, že PROLOG není jen akademickým jazykem. Některé aplikace jsou na jazyku PROLOG založeny zcela, jiné využívají jen malou část. Pokud ale PROLOG už někde najdeme, jedná většinou o nějaký zajímavý projekt.

PROLOG je znám jako jazyk umělé inteligence, zpracování jazyka nebo vyhledávání a srovnávání vzorů v datech. Zmíněné techniky nejsou ale omezeny jen na tento jazyk. I když se v následujících příkladech řeší právě pomocí PROLOGu, existuje mnoho jiných postupů, jak dané problémy řešit i bez něj. Systém PROLOGu navíc často potřebuje vstupní data předzpracovat nějakou jinou technikou. Existuje mnoho různých aplikací užívajících technik umělé inteligence, které nemají s PROLOGem nic společného. Pro zajímavost uvedu třeba projekty Wolfram-Alpha nebo Google search engine, u kterých by se na první pohled mohlo zdát, že by zde PROLOG nalezl uplatnění.

7.1. ELIZA

ELIZA je příklad počítačového programu a primitivního zpracování (z dnešního pohledu) přirozeného jazyka. Pochází z roku 1966 od autora Joseph Weizenbaum. Tento program zpracovává uživatelský vstup, přiřazuje k němu předem vytvořený skript a vrací odpověď, která se snaží simulovat odpověď člověka na druhé straně. Využívá technik srovnávání vzorů, parsování a substituce slov. Postupně se dočkal mnoha modifikací, např. verze DOCTOR, který simuluje doktora, který odpovídá pacientovi na otázky. Podle zadávaných otázek a odpovědí na otázky od DOCTORa dokázal tento program "lidsky"komunikovat s pacientem i několik minut než pacient přišel na to, že se jedná o program a DOCTOR o léčení ve skutečnosti nic neví.
ELIZA byla nejprve napsána v jazyce SLIP, později v jazyce LISP a časem se dočkala mnoha dalších zpracování. Pro příklad uvedu verzi v PROLOGu z roku 1992 od autora V. Patel, který se právě inspiroval ELIZou od Weizenbauma. Napsal ji v Quintus Prolog a později v SWI-Prologu. Jeho verze obsahuje zjednodušující predikáty, které substituují jednotlivá slova. Např. `sr([do,not|X],[dont|Y],X,Y).` nebo `sr([machine|X],[computer|Y],X,Y)`. Následuje řada pravidel, která jako vstupní parametry obsahuje klíčové slovo, na které se má reagovat, a jeho důležitost. Výstupní argumenty jsou tvořeny číslem vzoru, samotným vzorem a možnými odpovědmi na tento vzor. Každé pravidlo obsahuje několik vzorů a tak řádově více verzí možností, jak reagovat na dané klíčové slovo. Můžeme uvést dva kratší příklady:

```
rules([[sorry,0],[
  [1,[_],0,
    [please,do,not,apologize,.],
    [apologies,are,not,necessary,.],
    [what,feelings,do,you,have,when,you,apologize,?],
    ['I',have,told,you,that,apologies,are,not,required,.]]]]).

rules([[memory,0],[
  [1,[_,your,Y],0,
    [lets,discuss,further,why,your,Y,.],
    [earlier,you,said,your,Y,.],
    [but,your,Y,?],
    [does,that,have,anything,to,do,with,the,fact,that,your,Y,?]]]])
```

Program dále obsahuje predikáty pro zpracování vstupu, např. určení typu znaku, odstranění mezer, převod čísel na znaky a naopak. Poté zpracovává samotná slova z vloženého řetězce. Pokusí se je v něm najít. Slovo podle potřeby upraví (převod písmen na malé znaky). Obsahuje pravidla pro aktualizaci odpovědí, vyhledávání podle klíčových slov a predikáty pro porovnávání vzorů v pravidlech. [15], [28]

7.2. Interpret jazyka Erlang

Interpret tohoto jazyka byl napsán v PROLOGu, od J. L. Armstrong, roku 1986. Syntaxe tohoto jazyka se inspirovala PROLOGem (např. obsahuje seznamy jako n–tice prvků s hlavou a tělem, čárka mezi příkazy, tečka na konci bloku kódu). Naproti tomu funkce mají návratovou hodnotu pomocí klíčového slova return, takže se syntaktický i sémantický význam jazyků liší.
Erlang není založen na faktech, pravidlech a dotazech, ani na jiných mechanismech PROLOGu a logických jazyků. Je považován za deklarativní jazyk s funkcionálními prvky. Původní verze byla napsána v NU Prologu a později v Quintus a SICStus Prologu. Dnešní implementace Erlangu není již na PROLOGu založena vůbec, přesto ho zde uvádím z historického hlediska. Erlang se velmi často užívá jako produkční systém pro simulování nějaké formy umělé inteligence (např. expertní systémy). Jazyk se skládá z množiny pravidel s určitým chováním, mezi nimiž se vybírá. Jako produkční systém ho užívá mnoho velkých společností jako např. Amazon (pro implementaci a poskytování databázových služeb), Yahoo!, Facebook (podpora funkcí chatu a pošty) nebo T-Mobile (SMS autentizační systém).

Článek od J. L. Armstronga nejprve navrhuje zjednodušený meta interpret v PROLOGu (postupné plnění jednotlivých cílů ve formě faktů, vestavěného predikátu a zavolání meta predikátu call, nebo pravidla a rekurzivního řešení cílů v jeho těle), který ale neumožňuje odkládat výpočty, výpočet přerušit a vrátit se k němu později. Představuje predikát redukce, který redukuje seznam jednotlivých položek jako cílů v PROLOGu. Následně je tento predikát upraven pro

odložení výpočtu po nějakém fixním počtu kroků. Je navrženo další vylepšení pro spuštění a redukci více úloh najednou. Poté tento predikát autor dále rozšiřuje pro funkcionální potřebu a vykonávání jazyka Erlang. Na závěr autor popisuje vývoj syntaxe jazyka. [8]

7.3. Clarissa

Clarissa je zvukové uživatelské rozhraní vytvořené NASA pro Mezinárodní vesmírnou stanici, vyvíjené od roku 2002. Tento software byl vytvořen jako virtuální pomoc pro astronauty, kteří by jinak museli pracovat ve dvojicích při různé obsluze této stanice. Clarissa se stala prvním systémem ve vesmíru, který umožňuje hlasově zadávat instrukce a pokyny, což značně zjednodušuje a urychluje některé úkoly. Systém je schopný uživatelům odpovídat rovněž v anglickém jazyce. V roce 2004 byl systém schopen rozpoznat přibližně 500 slov, nějak reagovat na 260 z nich a provést 75 přesných hlasových instrukcí. Byl schopen rozpoznat špatně zadaný příkaz nebo provést krok zpět. Musel se vypořádat s různými dialekty a akcenty angličtiny, protože polovina posádky IIS tvoří ruští astronauti. Při testování se nedokázal vypořádat nebo špatně provedl jen 10% zadaných příkazů po předložení několika tisíc trénovacích rozhovorů. Po dalším roce vývoje se chybovost snížila jen na 5%. Do budoucna se plánuje rozšířit jeho kapacita na téměř všechny procedury IIS, jako např. systém podpory života, lékařskou asistanci nebo kontrolu vybavení, kterých je dohromady přes 12.000.
Clarissa umí rozpoznat jednotlivá slova, věty a různý kontext slov, a podle toho plnit rozdílné příkazy ve zdánlivě podobné větě. Systém ohodnocuje postupně jednotlivá slova podle toho, zda a do jaké míry je správně rozpoznal. Poté se je pokouší skládat do daného kontextu a ohodnocuje jednotlivé části. Systém umí vykonat například tyto funkce:

- navigace na následující krok – např. příkazy next, previous, go to step three,
- obecná navigace – např. read step four before step ten,
- práce se zvukovým záznamem – např. record voice note, delete voice note on substep two,
- nastavení alarmu – např. set alarm for five minutes from now,
- zapnutí/vypnutí zvukového rozpoznávání – systém poté reaguje jen na přesná klíčová slova,
- dotazovací fráze – např. where are we, list voice notes,
- mód verifikace – pro kritické úlohy systém požaduje potvrzení každého kroku před jeho provedením.

Systém se musel vypořádat i se zajímavým problémem, kdy jeden astronaut zadával slovně příkazy do počítače a přitom běžně komunikoval s druhým. Počítač musel rozpoznat, které příkazy patří jemu, a které patří do běžného rozhovoru. Při vývoji se uplatnilo mnoho technologií jako fuzzy logika, strojové učení, lexikální a syntaktická analýza (právě zde má své místo PROLOG). Část systému Clarissa byla napsána v SICStus Prologu. Celý systém je složen z několika částí v různých jazycích. Jazyk PROLOG se užívá v částech sémantického analyzátoru a manažeru dialogů. Sémantický analyzátor dostává vstup od modulu zpracování řeči. Tento vstup je ve formě řetězců slov. Analyzátor tento vstup převádí pomocí zabudovaných vzorů na tzv. dialogy. Manažer dialogů obdrží tento vstup a vyprodukuje list abstraktních akcí k těmto dialogům. Tyto abstraktní akce jsou předány výstupnímu manažeru, který již volá konkrétní procedury a následně přes GUI komunikuje s uživatelem. [12]

7.4. IBM Watson

IBM Watson je superpočítač od firmy IBM, jehož systém je založen na problematice umělé inteligence. Tento systém je schopný porozumět otázkám zadaným v přirozeném jazyce. Analyzovat je a v krátkém čase najít správnou odpověď s danou jistotou nebo její další alternativy. Původně byl po dobu pěti let vyvíjen pro soutěž *Jeopardy!* (obdoba českého Riskuj).

V současné době je tento projekt dále vyvíjen pro běžné komerční a průmyslové projekty jako rádce ve finančním a bankovním sektoru, systém pro zdravotní péči nebo systém starající se o databázi zákazníků. Jeho hlavní cíl a odlišnost od běžných informačních systémů je ten, že se snaží přiblížit lidským kognitivním schopnostem. Základ je v porozumění a zpracování přirozeného jazyka, strukturovaných i nestrukturovaných a jiných komplexních dat. Z těchto vstupních dat dále vyvodit podle potřeby nějakou hypotézu a tu ohodnotit. Ohodnocení probíhá ve formě výběru jen relevantních odpovědí, odpovědí s nějakou vahou spolehlivosti a nalezení možných alternativních odpovědí. Při ohodnocení systém používá znalostní databázi, ale snaží se odvodit i nové možné závěry. Poslední krok je učení se nových správně odvozených faktů při každé iteraci.

Watson sice disponuje pro běžné počítače závratnými parametry jako 16 TB RAM, je složen klustrem devadesáti IBM Power 750 serverů s osmi jádrovým 3.5 GHz procesorem u každého. Jeho síla zde ale není v maximálním výpočetním výkonu, ale v nově užité technologii. Takže při zpracování přibližně 500 GB různých dat za sekundu mohl konkurovat lidskému hráči v dané soutěži nebo jiné činnosti, na kterou by se specializoval. Tvůrci se museli vypořádat s problémy jako potřeby extrémně rychlé odpovědi na otázku. Zpracováním obrovského množství dat, vytvořit v nich logickou spojitost a vrátit komplexní odpověď. Vypořádat se s velkou robustností při zadávání otázek, různým významem slov pro různé obory apod.

Jazyk PROLOGu se zde opět uplatňuje při parsování a sémantické analýze

zadaných otázek a dále při hledání a odvozování odpovědí. Parsování převádí zadanou otázku do strukturované formy. Následně se pomocí vzorů provádí detekce, na co je nejspíš otázka zaměřena, jaký typ odpovědi se očekává, nebo jaký je vztah objektů v otázce. Poté se aplikují určitá pravidla, sesbírají získané odpovědi a proběhne jejich ohodnocení vzhledem k zadané otázce. [10], [20]

7.5. Další aplikace

7.5.1. PigE/AUSPIG

PROLOG najdeme i v mnoha netypických aplikacích. PigE/AUSPIG je expertní systém pro chov prasat napsaný v ARITY PROLOG. Systém simuluje reprodukci a růst prasat podle zadaných podmínek a snaží se navrhnout optimální strategii. Tento starší systém pochází z roku 1991 a může se chlubit cenou pro nejlepší praktickou prezentaci aplikace v PROLOGu. Jako jeden z prvních expertních systémů dokázal lépe odhadnout danou situaci, než lidský expert. Například objevil, že pokud se nepatrně zrychlí proudění vzduchu nad teplým prasečím přístřeškem o $0.2 - 0.6 m/s$ (což je hodnota, kterou člověk nemůže přirozeně bez měřících přístrojů postřehnout), růst prasat se urychlí téměř dvojnásobně. Nyní se stále používá v zemích jako USA, Francie nebo Austrálie. [13]

7.5.2. Goldfinder

Goldfinder je znalostní systém napsán v MacPrologu, určený pro geology. Jeho účel je jednoduchý – slouží jako rádce pro nalezení zlata. Vstupní data tvoří geologické mapy. Systém je zpracuje a vrátí potencionální výskyt zlata v dané oblasti a oblasti vhodné pro vrty. [14]

7.5.3. Sportovní systémy

Existuje mnoho sportovních expertních systémů založených na PROLOGu. Perfect Pitch je sportovní rádce napsán v PDC Prologu. Slouží pro ekonomické rady pro daný sport. Tento systém se dá užít i při výběru tendrů. Dříve se používal v Austrálii.
Toernooi Assistant je systém pro tenisové turnaje, opět napsán v PDC Prologu. Poskytuje plánování zápasů, změnu rozvrhu např. v případě deště, administraci hráčů nebo zpracování výsledků zápasů.
Rotisserie/Fantasy Basketball leagues je systém pro ohodnocování hráčů a týmů v NBA. Systém je zpracován v Quintus Prologu. Každý týden stahuje nové statistiky do databáze PROLOGu, převádí je do klauzulí a počítá nové ohodnocení hráčů a týmů. Z toho pak odvozuje možné budoucí ohodnocení a výsledky zápasů. [22]

7.5.4. Ostatní systémy

CASEy je expertní systém od firmy Boeing napsaný v Quintaus Prologu. Sloužil pro zaměstnance pro obsluhu elektronických systémů na letišti. V minulosti se užívalo několik systémů pro předpověď počasí. Část serverové části webu SWI-Prologu je napsaná v samotném SWI-Prologu. Quintaus Prolog užívá mnoho švédských strojírenských firem. Uplatňuje se při automatizaci výroby, železniční správě, telekomunikaci a veřejné dopravě. [22]

Závěr

Cílem této práce bylo vytvořit sbírku řešených příkladů pro jazyk PROLOG a představit tento programovací jazyk. V teoretické části práce jsem uvedl základní teorii z predikátové logiky 1. řádu. Dále jsem uvedl teorii k základům logického programování a jazyku PROLOG. V práci jsem popsal i historické spojitosti vývoje logického programování. Byly popsány vlastnosti tohoto jazyka a jeho struktura. Popsal jsem implementaci SWI-Prologu a uvedl srovnání s ostatními implementacemi. Uvedl jsem několik příkladů implementací praktických úloh v různých systémech PROLOGu z různých období.
V praktické části jsem za užití stávájících technik vytvořil nové příklady, které nebyly doposud zpracovány, a nebo se jedná o modifikace a rozšíření známých problémů. Příklady uvádí základní i pokročilejší techniky logického programování a demonstrují teoretická omezení a vlastnosti jazyka. Vytvořil jsem několik složitějších příkladů, které by měly ukazovat sílu a typické užití PROLOGu a také jeho praktické využití.
Práce určitě neobsáhla všechna možná témata a bylo by možné vymyslet široký okruh dalších příkladů. Bylo by jí v budoucnu možno rozšířit o problematiku rychlosti výpočtů a složitosti algoritmů v jazyce PROLOG. Zajímavá by byla rovněž problematika sémantických webů, jazyka DATALOG odvozeného z jazyka PROLOG, nebo detailně se věnovat zpracování přirozeného jazyka, DC gramatikám nebo expertním systémům.

Conclusions

The aim of this thesis was to create a collection of exercises for PROLOG language and introduce this programming language. In the theoretical part I described the basic theory of the first order predicate logic. Then I mentioned the basis of the theory of logic programming and PROLOG. In my thesis I described the historical connection of the development of logic programming. The characteristics of this language and its structure were described there too. I explained the implementation of SWI-Prolog and compared it to other implementations. I gave a few examples of the implementation of practical problems in different PROLOG systems from different periods.
In the practical part I followed the usage of existing technology to build new examples which have not been processed yet, or it is a modification and extension of known issues. The examples provide basic and advanced techniques of logic programming and demonstrate the theoretical limitations and language features. I created more difficult examples which should show the power of a typical usage of Prolog and also its practical use.
This piece of work definitely do not cover all possible topics and it would be possible to come up with a wider range of other examples such as the issue of computation speed and the complexity of algorithms in Prolog. It would also be interesting the issue of semantic Webs, DATALOG language derived from PROLOG or to regard to the natural language processing and DC grammars or expert systems in details.

Reference

[1] Bramer, Max. *Logic Programming with Prolog.*
ISBN 1-85233-938-1, 2005.

[2] Bratko, Ivan. *Prolog Programming for artificial intelligence.*
ISBN 0-201-40375-7, 2001.

[3] Clocksin, William. *Programming in Prolog.*
ISBN 3-540-00678-8, 2003.

[4] Hill, Patricia; Lloyd, John. *The Gödel Programming Language.*
ISBN 978-0262082297, 1994.

[5] Lloyd, John. *Foundations of logic programming.*
ISBN 0-387-13299-6, 1997.

[6] Nilsson, Ulf; Maluszynski, Jan. *Logic, Programming and Prolog (2ed).*
ISBN 0-471-95996-0, 1995.

[7] Sterling, Leon; Shapiro, Ehud. *The art of Prolog, advanced programming techniques.*
ISBN 0-262-19338-8, 1994.

[8] Armstrong, J. L.; Virding, S. E.; Williams, M. C. *Use of Prolog for developing a new programming language.* [online]
Ellemtel Telecommunications Systems Laboratory, Sweden, 3. dubna 1992 [cit. 30. dubna 2013]
Dostupné na: http://www.erlang.se/publications/prac_appl_prolog.pdf

[9] Kniesel, Günter. *Advanced Logic Programming.* [online]
Universität Bonn, Germany, 27. června 2012 [cit. 25. dubna 2013]
Dostupné na: http://sewiki.iai.uni-bonn.de

[10] Lally, Adam; Fodor, Paul. *Natural Language Processing With Prolog in the IBM Watson System.* [online]
IBM Thomas J. Watson Research Center; Stony Brook University, 31. března 2011 [cit. 5. května 2013]
Dostupné na: http://www.cs.nmsu.edu

[11] McCarthy, John. *Programs with Common Sense.* [online]
National Physical Laboratory, Teddington, England, 1958 [cit. 15. dubna 2013].
Dostupné na: http://www-formal.stanford.edu/jmc/mcc59.pdf

[12] Rayner, M.; Hockey, B. A.; Renders, J. M; Chatzichrisafis, N; Farrel, K. *Spoken Language Processing in the Clarissa Procedure Browser.* [online] Cambridge University Press, United Kingdom, 2005 [cit. 2. května 2013] Dostupné na: http://citeseerx.ist.psu.edu

[13] Menzies, Tim; Black, John; Fleming, Joel; Murray, Dean. *An Expert System for Raising Pigs.* [online] University of New South Wales, Australia, 1991 [cit. 3. května 2013] Dostupné na: http://www.umcs.maine.edu/~cmeadow/courses/cos301/raising-pigs.pdf

[14] Hawkes, D. D. *GOLDFINDER: A knowledge-based system for mineral prospecting.* [online] The University of Birmingham, United Kingdom, 30. dubna 1991 [cit. 4. května 2013] Dostupné na: http://jgs.lyellcollection.org/content/149/3/465.abstract

[15] Patel, Viren. *ELIZA in Prolog.* [online] New York University, United States, 10. dubna 1992 [cit. 4. května 2013] Dostupné na: http://cs.nyu.edu/courses/fall12/

[16] Naish, Lee. *Higher-order logic programming in Prolog.* [online] University of Melbourne, Australia, 1. února 1996 [cit. 21. ledna 2014] Dostupné na: http://pdf.aminer.org

[17] Enderton, Herbert B. *Second-order and Higher-order Logic.* [online] Stanford University, United States, 21. června 2012 [cit. 21. ledna 2014] Dostupné na: http://plato.stanford.edu/archives/fall2012

[18] Miller, Dale. *Logic, Higher-order.* [online] École polytechnique Université Paris-Saclay, France, 5. února 1991 [cit. 21. ledna 2014] Dostupné na: http://www.lix.polytechnique.fr/labo/dale.miller/papers/

[19] Datalog programming language. *Datalog User Manual.* [online] Datalog programming language, 20. prosince 2013 [cit. 20. ledna 2014] Dostupné na: http://www.ccs.neu.edu/home/ramsdell/tools/datalog/datalog.html

[20] International Business Machines Corporation. *IBM Watson.* [online] International Business Machines Corporation, United States, 2013 [cit. 4. května 2013] Dostupné na: http://www-03.ibm.com/innovation/us/watson/

[21] Mercury programming language. *Mercury.* [online]
Mercury programming language, 14. ledna 2014 [cit. 20. ledna 2014]
Dostupné na: http://www.mercurylang.org/index.html

[22] Roth, Al. *The Practical Application of Prolog.* [online]
Prolog Vendors Group, United Kingdom, 10. prosince 2002 [cit. 5. května 2013]
Dostupné na:
http://www.drdobbs.com/parallel/the-practical-application-of-prolog

[23] SWI-Prolog, Reference manual. *SWI-Prolog documentation.* [online]
Dostupné na: http://www.swi-prolog.org/pldoc/refman

[24] The Ciao System. *Ciao Prolog.* [online]
Dostupné na: http://ciao-lang.org/

[25] The GNU Prolog web site. *GNU Prolog.* [online]
Dostupné na: http://www.gprolog.org/

[26] Visual Prolog web site. *Visual Prolog.* [online]
Dostupné na: http://www.visual-prolog.com/

[27] Wikipedia, The Free Encyclopedia. *Comparison of Prolog implementations.* [online]
Wikipedia, The Free Encyclopedia, 8. března 2013 [cit. 12. dubna 2013]
Dostupné na:
http://en.wikipedia.org/wiki/comparison_of_prolog_implementations

[28] Wikipedia, The Free Encyclopedia. *ELIZA.* [online]
Wikipedia, The Free Encyclopedia, 10. března 2013 [cit. 6. května 2013]
Dostupné na: http://en.wikipedia.org/wiki/eliza

[29] Wikipedia, The Free Encyclopedia. *Q-Systems.* [online]
Wikipedia, The Free Encyclopedia, 20. března 2013 [cit. 17. dubna 2013]
Dostupné na: http://en.wikipedia.org/wiki/q-systems

[30] Wikipedia, The Free Encyclopedia. *SWI-Prolog.* [online]
Wikipedia, The Free Encyclopedia, 13. března 2013 [cit. 20. dubna 2013]
Dostupné na: http://en.wikipedia.org/wiki/swi-prolog

A. Obsah přiloženého CD

`bin/` Instalační soubor interpretu SWI-Prolog.

`doc/` Text diplomové práce ve formátu PDF a soubory pro vygenerování tohoto PDF.

`src/` sbírka příkladů tvořená .pl soubory reprezentující jednotlivé příklady.

`src/img` Složka obrázků pro jednotlivé příklady sbírky.

`readme.txt` Instrukce pro instalaci a spuštění interpretu a příkladů.