Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Expertní systém v Prologu

2020

Denisa Nováková Andrýsková

Vedoucí práce: doc. RNDr. Miroslav Kolařík, Ph.D.

Studijní obor: Informatika, prezenční forma

Bibliografické údaje

Autor:	Denisa Nováková Andrýsková
Název práce:	Expertní systém v Prologu
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2020
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	doc. RNDr. Miroslav Kolařík, Ph.D.
Počet stran:	69
Přílohy:	1 DVD
Jazyk práce:	český

Bibliograhic info

Author:	Denisa Nováková Andrýsková
Title:	Expert System in Prolog
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2020
Study field:	Computer Science, full-time form
Supervisor:	doc. RNDr. Miroslav Kolařík, Ph.D.
Page count:	69
Supplements:	1 DVD
Thesis language:	Czech

Anotace

Tato práce prezentuje syntézu grafologie jakožto psychodiagnostické metody rozboru písma, expertních systémů a programovacího jazyka Prolog. V teoretické rovině se práce nejprve zabývá jazykem Prolog z hlediska možností jeho deklarativních rysů a z hlediska jeho logického základu. Následně pojednává obecně o problematice expertních systémů. V poslední části je představen grafologický expertní systém vytvořený v jazyku Prolog.

Synopsis

This thesis presents the synthesis of graphology as a psychodiagnostic method of handwriting analysis, expert systems and programming language Prolog. On a theoretical level the thesis deals with Prolog from the point of view of a declarative paradigm and from its logical basis. Subsequently, the thesis discuss expert systems in general. Lastly, the new graphological expert system writen in Prolog is introduced.

Klíčová slova: expertní systém; grafologie; Prolog; umělá inteligence

Keywords: artificial intelligence; expert system; graphology; Prolog

Ráda bych velice poděkovala panu doc. RNDr. Miroslavu Kolaříkovi, Ph.D. za vedení, cenné rady a ochotnou pomoc. Dále bych chtěla poděkovat své rodině, zejména mému drahému manželovi Davidovi a sestře Barboře za laskavou podporu v těžkých chvílích. Děkuji i kamarádce Markétě za kontrolu textu namísto spánku. V neposlední řadě patří mé díky i Ježíši Kristu, mému Pánu a příteli.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracovala samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce

podpis autora

Obsah

1 Úvod **8**

2 Jazyk Prolog **9**
- 2.1 Charakteristika . . . 9
- 2.2 ISO Prolog . . . 9
  - 2.2.1 Struktura programu . . . 9
  - 2.2.2 Syntaktické jednotky . . . 11
  - 2.2.3 Unifikace a dotazy . . . 12
  - 2.2.4 Backtracking . . . 14
- 2.3 Prolog v kontextu predikátové logiky 1. řádu . . . 16
  - 2.3.1 Jazyk PL . . . 17
  - 2.3.2 Termy . . . 17
  - 2.3.3 Formule . . . 18
  - 2.3.4 Struktura pro jazyk PL . . . 18
  - 2.3.5 $\mathcal{M}$-ohodnocení . . . 20
  - 2.3.6 Teorie . . . 21
  - 2.3.7 Hornovské klauzule . . . 21
  - 2.3.8 Důkaz . . . 22
  - 2.3.9 Nejobecnější unifikace . . . 23
  - 2.3.10 Robinsonův rezoluční princip . . . 24

3 Expertní systémy **26**
- 3.1 Typy expertních problémů . . . 28
- 3.2 Znalostní báze . . . 29
  - 3.2.1 Logická reprezentace . . . 30
  - 3.2.2 Produkční pravidla . . . 30
  - 3.2.3 Rámce . . . 31
  - 3.2.4 Sémantické sítě . . . 32
- 3.3 Inferenční mechanismus . . . 33
  - 3.3.1 Stavový prostor . . . 33
  - 3.3.2 Zpětné řetězení . . . 35
  - 3.3.3 Dopředné řetězení . . . 35
  - 3.3.4 Rezoluce konfliktu . . . 36
- 3.4 Neurčitost . . . 37
  - 3.4.1 Nespolehlivost informací . . . 38
  - 3.4.2 Nepřesnost jazyka . . . 40

4 Grafologický expertní systém **42**
- 4.1 Grafologie . . . 42
- 4.2 Role grafologa . . . 42
- 4.3 Charakteristika grafologického expertního systému . . . 43

5 Architektura systému 44
5.1 Typy znalostí . . . 44
5.1.1 Grafologické znalosti . . . 44
5.1.2 Pravidla . . . 45
5.1.3 Klasifikační znalosti . . . 46
5.2 Uživatelské rozhraní . . . 46
5.2.1 Globální a lokální příkazy . . . 46
5.2.2 Struktura hlavního menu . . . 47
5.2.3 Tvorba znalostní báze . . . 48
5.2.4 Provádění expertízy . . . 51
5.2.5 Možnosti nastavení . . . 53
5.3 Inference . . . 53
5.3.1 Fuzzyfikace . . . 53
5.3.2 Odvozování závěrů . . . 55
5.4 Vysvětlování závěrů . . . 56

6 Implementace 57
6.1 Reprezentace znalostí . . . 57
6.2 Zpracování uživatelských vstupů . . . 58
6.2.1 Implicitní přetypování . . . 58
6.2.2 Implementace zpětného chodu . . . 59
6.3 Inferenční mechanismus . . . 60

7 Zhodnocení 63
7.1 Vhodnost úlohy pro expertní systém . . . 63
7.2 Vhodnost použitého jazyka . . . 63
7.3 Použité uživatelské rozhraní . . . 64
7.4 Vize . . . 64

Závěr 65

Conclusions 66

A Obsah přiloženého DVD 67

Literatura 68

Seznam obrázků

1 Backtracking 16
2 Schéma expertního systému 27
3 Rámce 32
4 Sémantická síť 33
5 Funkce příslušnosti 41
6 „Funkce příslušnosti“ písmového parametru Sklon 54

1 Úvod

Grafolog se při kontaktu s písmem setkává s velkým množstvím dat. Nejenže je složité v rámci jeho expertízy brát všechna tato data v potaz tak, aby vznikl objektivní posudek, ale díky své povaze je mnoho z těchto dat vhodných k automatizovanému zpracování, neboť ač se přístupy ke grafologii liší (od striktně metodických, až po holistické), společným faktorem zůstává předpoklad, že lze postihnout určitý vztah mezi charakteristikami písma a vlastnostmi jeho pisatele.

Z hlediska expertního systému pak jde především o formalizaci těchto zákonitostí pomocí pravidel a navržení odvozovacího mechanismu, který bude tato pravidla využívat pro automatické odvozování závěrů z charakteristik písma a pomůže tak grafologa odstínit od rutinnějších aspektů jeho práce.

Struktura této práce je dělena do dvou logických celků – teoretické části, v níž se zabývám jazykem Prolog a problematikou expertních systémů; a praktické části, v níž pojednávám o vzniklém grafologickém expertním systému.

Konkrétně ve druhé kapitole je na poznatky o jazyku Prolog nahlíženo ze dvou hledisek. Nejprve je čtenář uveden do Prologu z hlediska jeho standardizované syntaxe, sémantiky a dvou klíčových principů – unifikace a backtrackingu. Poté je Prolog představen abstraktněji a pojmy s ním související jsou vysvětleny ve vztahu k predikátové logice 1. řádu.

Kapitola o expertních systémech je pojata poněkud obšírněji. Nejprve jsou charakterizovány expertní systémy jako takové a typy úloh, které řeší. Následně jsou podrobně rozebrány dvě hlavní komponenty expertního systému – znalostní báze a inferenční mechanismus. Nakonec je proveden exkurz do obecnější problematiky neurčitosti dat, s níž se musí potýkat i expertní systémy.

Čtvrtá kapitola čtenáře zasvětí do tématu grafologie, vymezí roli grafologa a expertního systému a popíše hlavní ideu grafologického expertního systému.

Pátá a šestá kapitola se zaměřuje na konkrétní podobu grafologického expertního systému. První z těchto kapitol má částečně sloužit jako popis návrhu systému, částečně jako uživatelská příručka, neboť čtenář se zde dozví, co může od systému čekat. V části o implementaci jsou prezentovány vybrané části vytvořeného systému.

V kapitole sedmé je práce uzavřena zhodnocením vzniklého systému z různých pohledů. Také jsou zde nastíněny možnosti, v rámci kterých se může systém do budoucna vyvíjet.

2 Jazyk Prolog

Tato kapitola pojednává o Prologu ze dvou hledisek – jeho standardu ISO Prolog a jeho teoretického základu, predikátové logiky 1. řádu.

2.1 Charakteristika

Jazyk Prolog tradičně zařazujeme do rodiny deklarativních jazyků, které jsou pro svou vysokoúrovňovou povahu a podporu symbolických výpočtů[1] hojně využívány či přímo navrženy pro řešení problémů v oblastech umělé inteligence. Deklarativní jazyky stojí v opozici vůči imperativnímu paradigmatu, v němž je výpočet nad nějakým problémem přesně algoritmizován posloupností kroků v syntaxi určitého programovacího jazyka. Naproti tomu v deklarativním paradigmatu „pouze" specifikujeme (deklarujeme) požadované chování a cíle výpočetního procesu. Obecně to znamená, že program napsaný deklarativním stylem je množina formalizovaných vztahů, které specifikují cíle[2], dle nichž se pak řídí budoucí výpočet, který obstarává (na tomto programu) zpravidla nezávislý algoritmus. Tyto vztahy pak mohou být v konkrétních případech definovány např. pomocí funkcí v případě funkcionálních jazyků či formulí v případě jazyků založených na principech logického programování.

2.2 ISO Prolog

Prolog je jazyk standardizovaný Mezinárodní organizací pro normalizaci (ISO)[3]. My se nyní na Prolog podíváme právě z hlediska jeho standardu (z jehož dokumentů [1, 2] budu v této kapitole vycházet primárně). Představíme si strukturu prologovského programu, jeho syntaxi i sémantiku a vhlédneme do dvou klíčových funkcionalit Prologu – unifikace a backtrackingu.

2.2.1 Struktura programu

Prologovský program je složen ze série tzv. *klauzulí* a *direktiv*, z nichž každá je ukončena symbolem '.' a odděleny jsou bílými znaky[4]. [3]

Klauzule se v prologovském kódu vyskytují buď ve formě tzv. *faktů* či *pravidel*. Určitou klauzuli vždy řadíme k nějakému *predikátu*. Přesněji však k jednomu predikátu může existovat více klauzulí. Pojďme si právě zmíněné pojmy blíže

[1] Jedná se o nenumerické výpočty, v nichž se pracuje se symboly – např. je možné provádět úpravy algebraických výrazů.

[2] Proto se deklarativní jazyky někdy nazývají *goal-driven*.

[3] International Organization for Standardization

[4] mezery, tabulátory, odřádkování, …

objasnit na příkladech. Napišme si zde několik jednoduchých faktů.

```
animal(cat).
animal(dog).
animal(bird).

produces(cow, milk).
produces(hen, eggs).
```

Vidíme, že fakty míníme jakési struktury, které jsou-li

- *unární*, představují vlastnosti objektů – v tomto případě názvy zvířat,
- *binární* (či mají-li vyšší aritu), představují vztahy mezi příslušnými objekty – zde jsou to produkty chovných zvířat.

Máme zde dvojici predikátů `animal` s aritou 1 a `produces` s aritou 2, přičemž první z nich je zastoupen třemi klauzulemi a druhý dvěma klauzulemi. Z hlediska sémantiky se díváme na jakýkoliv fakt jako na *pravdivé* tvrzení.

Syntax pravidel je v obecném tvaru vyjádřena následovně:

```
Head :- Body.
```

Levá část pravidla (`Head`) popisuje fakt, který má pravidlo jako celek definovat. Vedle toho tělo pravidla (`Body`) se může skládat z libovolného počtu tzv. *cílů* oddělených čárkami (což představuje konjunkci těchto cílů), které musí být splněny (tj. musí být vyhodnoceny jako pravdivé), aby i hlava pravidla byla pravdivá. [4] Opět si to blíže ukažme na příkladu.

```
livestock(X):-
    animal(X),
    produces(X, _).
```

Máme zde pravidlo, které má definovat chovné zvíře pomocí dvou vlastností: musí to být zvíře a musí něco produkovat. Všimněme si, že syntaktické jednotky, které jsme si před chvílí představili jako cíle, máme ve skutečnosti ve zobecněném tvaru nám již známých výše definovaných faktů. Zobecněnými je činí přítomnost velkého písmena X, které zde představuje *proměnnou* a podtržítka, které značí tzv. *anonymní proměnnou*[5].

Fakty jsou mj. pouze speciálním případem pravidel, neboť každý fakt můžeme se zachováním sémantiky přepsat jako:

```
fact(object):- true.
```

[5]Blíže si je představíme v následující kapitole 2.2.2.

Na množinu pravidel a faktů se dále budeme odkazovat jako na prologovskou databázi.

Direktivy jsou speciální části kódu určené pro kompilátor. Pomocí nich můžeme specifikovat vlastnosti určitých predikátů, vyskytujících se ve zdrojovém kódu, načítat potřebné moduly programu (jako knihovny) nebo stanovit cíle, které mají být splněny za běhu programu.

2.2.2 Syntaktické jednotky

Mimo základní pohled na program v Prologu jakožto množinu klauzulí a direktiv můžeme tento pohled přesunout na čistě syntaktickou rovinu. Z této perspektivy se můžeme dívat na program jako na množinu tzv. *termů*, které považujeme za univerzální datovou strukturu v Prologu.

Na té nejnižší úrovni se termy mohou skládat ze čtyř druhů znaků – velká písmena, malá písmena, číslice a speciální znaky. V prvé řadě ale termy (v rámci jejich definice) rozdělujeme na:

- jednoduché termy – *konstanty* a *proměnné*,
- složené termy (neboli *struktury*).

Konstanty jsou buď čísla nebo tzv. *atomy*. Čísla, která Prolog rozlišuje, včetně zápisů čísel v binární, oktalové a hexadecimální soustavě (uvozených v pořadí prefixy 0b, 0o a 0x), jsou např. tato (oddělená čárkou):

-20, 33, 0.239, -4.2e2, 9.25e-10, 0b1010, 0x6BA2.

Naproti tomu atomy představují pojmenování objektů – např. názvy predikátů či jména zvířat v předchozích příkladech. Podmínkou pro atomy je to, že musí buď začínat vždy malým písmenem, přičemž dále mohou být složeny z libovolných znaků, které nenesou speciální význam (např. *operátorů*) nebo to musí být série speciálních znaků (tedy např. tyto dva znaky ':-', které známe z pravidel, tvoří atom). Poslední možnost, jak vytvořit atom, je ta, že libovolný řetězec znaků vložíme mezi jednoduché uvozovky (tedy např. 'Obi-Wan Kenobi', '012abc' jsou také atomy).

Proměnné vždy začínají velkým písmenem a mohou být složeny z libovolných znaků, které (stejně jako v případě atomů) nemají jiný speciální význam. Z hlediska sémantiky je to pak objekt, který může za běhu programu nabýt nějaké konkrétní instance, kterou je opět term. (V angličtině se používá speciální termín *instantiated variable*; kvůli absenci ekvivalentního výrazu v češtině budu používat pojem „proměnná s instancí".) Proměnnou také může být již dříve zmíněná *anonymní proměnná*, značená podtržítkem. Taková proměnná může zcela standardně nabývat libovolných instancí s tím rozdílem, že se dále její název v kódu (např. v těle pravidla) nepoužije.

Struktury jsou v Prologu specifikovány svým *funktorem* a *komponentami* (kterými mohou být mj. i další struktury). Ve standardní syntaxi navíc používáme kulaté závorky a čárky:

```
functor(a, b, c).
```

Zde již vidíme, že dříve jsme se o faktech jakožto strukturách nezmiňovali bezdůvodně; z pohledu syntaxe Prologu jsou to vskutku struktury. Pravidla jsou však, jak ihned zjistíme, také struktury (ač trochu skryté).

Atomy, které nazýváme funktory, je v Prologu možné definovat i jako speciální operátory, jež můžeme (místo defaultní prefixovosti) stanovit jako infixové či postfixové. Strukturu představující např. operaci sčítání `+(X, Y).` je pak tedy možné používat zažitým infixovým způsobem. Prologovské pravidlo

```
f(a):- g(b, c).
```

tedy není nic jiného než struktura s funktorem :- a dalšími dvěma strukturami jakožto komponentami:

```
:-(f(a), g(b, c)).
```

Důležitými strukturami, které zde ještě stojí za zmínku, jsou *seznamy*. Běžně jsou v Prologu zapisovány jako prvky oddělené čárkami a uzavřené v hranatých závorkách – např:

```
[a, b, c, d, e].
```

Interně jsou ale implementovány jako zprava zanořené struktury s funktorem '.'[6], tedy ekvivalentní zápis výše uvedeného seznamu by byl

```
.(a, .(b, .(c, .(d, .(e, []))))),
```

kde ukončujícím prvkem je (vždy) prázdný seznam `[]`.

Díky této definici se pak dá na celý seznam nahlížet jako na dvojici *hlava* (tj. první prvek seznamu) a *tělo* (tj. seznam obsahující zbylé prvky), k čemuž se pak v Prologu používá stručnější zápis (zde obecně)

```
[Head | Body].
```

2.2.3 Unifikace a dotazy

Samotná prologovská databáze by nám nebyla příliš k užitku, pokud by nám nebyla dána k dispozici možnost se na ni dotazovat. Vrátíme-li se k našemu příkladu se zvířaty, dejme tomu, že nás zajímá odpověď na otázku: „Je kočka zvíře?“ To můžeme zjistit tak, že v prologovském interpretu položíme dotaz ve speciálním tvaru:

```
?- animal(cat).
```

[6] Definice seznamu pomocí tzv. *tečkových párů* v Lispu je v podstatě ekvivalentní.

Bude-li v databázi nalezen shodný fakt, interpret vrátí odpověď **`yes`**, v opačném případě vrátí **`no`**.[7]

Mnohem běžněji nás ale budou zajímat odpovědi na obecnější dotazy než jen ty, na které se dá odpovědět ano/ne – kupříkladu „*Jaká* zvířata produkují mléko?“. V takovém typu dotazů již uplatníme proměnné:

```
?- produces(X, milk).
```

Máme zde proměnnou X bez instance. Úkolem interpretu je v tomto případě najít instance této proměnné skrze tzv. *unifikaci* a ty vrátit jako odpověď.

Unifikace je proces, v němž jde o to, neformálně řečeno, udělat dva termy stejnými tím, že najdeme instanci ke všem proměnným bez instance. Term, který není proměnná se může unifikovat jen sám se sebou (např. atomy 'x' a 'y' by se tedy neunifikovaly). V praxi se tak děje přes aplikaci operátoru =. Mějme kupříkladu tyto dva složené termy (a položme mezi ně =):

```
f(A, g(b, C)) = f(a, g(B, c)).
```

Máme zde celkem tři proměnné bez instance – A, B, C. Po aplikaci = se nám každá z nich (popořadě) úspěšně unifikuje s atomy a, b, c. Případ neúspěšné unifikace by nastal, pokud bychom v jedné ze struktur např. nahradili funktor g za h:

```
f(A, g(b, C)) = f(a, h(B, c)).
```

Navraťme se nyní s těmito poznatky k dotazu `?- produces(X, milk).` Interpret v databázi nalezne fakt `produces(cow, milk).` a unifikuje ho s dotazem. Z proměnné X se tedy stane proměnná s instancí a Prolog vydá její hodnotu jako odpověď:

```
X = cow.
```

Můžeme se též zeptat z opačné strany – na to, *co* určité zvíře produkuje. Zadáním např. dotazu

```
?- produces(hen, Y).
```

dostaneme odpověď `Y = eggs`.

Pokud bychom ovšem zadali dotaz na zvíře, u nějž vztah `produces` není definován – např. `?- produces(cat, Y).`, obdržíme od Prologu odpověď **`no`**.

[7] Tyto odpovědi se v různých implementacích mohou lišit – např. ve SWI-Prologu se můžeme setkat s true/false.

2.2.4 Backtracking

Pro účely příkladu si teď nepatrně rozšíříme naši databázi – konkrétně pozměníme definici „hospodářského zvířete“. Definice daná naším původním pravidlem zněla: „musí být zvířetem a musí něco produkovat“. Předpokládejme ale, že žijeme např. v nějaké exotičtější zemi, kde mohou být zvyklosti hospodářského průmyslu extravagantnější než u nás, a přidejme k prvnímu pravidlu ještě další pravidlo definující hospodářské zvíře:

```
livestock(X):-
    animal(X).
```

Nyní se v rámci této databáze podíváme na vyhodnocovací proces Prologu, který je řízen tzv. *backtrackingem.*

Popišme si, co se děje od chvíle, kdy uživatel do interpretu zadá dotaz (který může být buď jediný cíl nebo konjunkce více cílů oddělených čárkami) a stiskne ENTER. V tom momentě začne Prolog prohledávat databázi od začátku (což mj. naznačuje, že záleží na pořadí, v němž jsou fakty a pravidla programátorem zapsány) a snaží se splnit první zadaný cíl (pokud je jich více), přičemž mohou nastat dva případy [4]:

- Je nalezen fakt (resp. hlava pravidla), který je možno unifikovat se zadaným cílem. Místo v databázi, kde byl tento fakt (resp. hlava pravidla) nalezen, si Prolog označí tzv. *place-markerem* pro případ potřeby *znovusplnění* (z angl. *re-satisfy*) označeného cíle. V této chvíli se také za všechny proměnné bez instance, které byly úspěšně unifikovány, dosadí jejich instance. Byl-li touto unifikující klauzulí fakt, cíl je tímto splněn, Prolog se přesouvá k dalšímu cíli v konjunkci (pokud existuje) a celý proces se opakuje. Pokud jsme ale narazili na hlavu pravidla, Prolog se nejprve musí pokusit splnit cíle v jeho těle. V případě každého z nich postupuje stejným způsobem jako tomu bylo u původního zadaného cíle.

- Nebyl nalezen žádný fakt (resp. hlava pravidla), který by bylo možno unifikovat s cílem. Splnění tohoto cíle tedy selhalo. Prolog pokračuje tím, že se nyní snaží znovusplnit naposledy splněný cíl. Vrací se tedy k tomu místu, kam byl naposledy zanechán place-marker a zkouší se vydat alternativní cestou[8] pro splnění počátečního cíle – dochází (v angličtině doslova) k backtrackingu.

Backtracking může pochopitelně probíhat až do momentu, dokud nedosáhneme počáteční úrovně, na níž může dojít k pokusu o znovusplnění prvotního cíle (zadaného dotazem) tak, že se Prolog snaží najít alternativní klauzule k unifikaci od místa, kde byl pro tento cíl položen place-marker. Pokud ani tak neuspěje, výpočet nad dotazem definitivně končí s negativní odpovědí.

[8] Tím je myšlena cesta ve stromu, protože celý vyhodnocovací proces můžeme reprezentovat stromem.

Ukažme si nyní příklad takového výpočtu na naší databázi – konkrétně pro dotaz

```
?- livestock(dog).
```

Jednotlivé kroky můžeme sledovat i na obrázku 1, který znázorňuje strom výpočtu.

1. Zadaný cíl `livestock(dog).` se unifikuje s hlavou prvního nalezeného pravidla, což jest

   ```
   livestock(X):-
       animal(X),
       produces(X, _).
   ```

2. Dále se za všechny výskyty proměnné X dosadí právě nalezená instance – `dog`.

3. Jako nový cíl pro dokázání se položí konjunkce

   ```
   ?- animal(dog), produces(dog, _).
   ```

4. Pro první z cílů je v databázi nalezen fakt `animal(dog).` a dojde k jejich unifikaci.

5. Pro druhý cíl neexistuje žádný fakt, kde by jako argument figuroval atom `dog`, dojde tedy k selhání a spuštění backtrackingu.

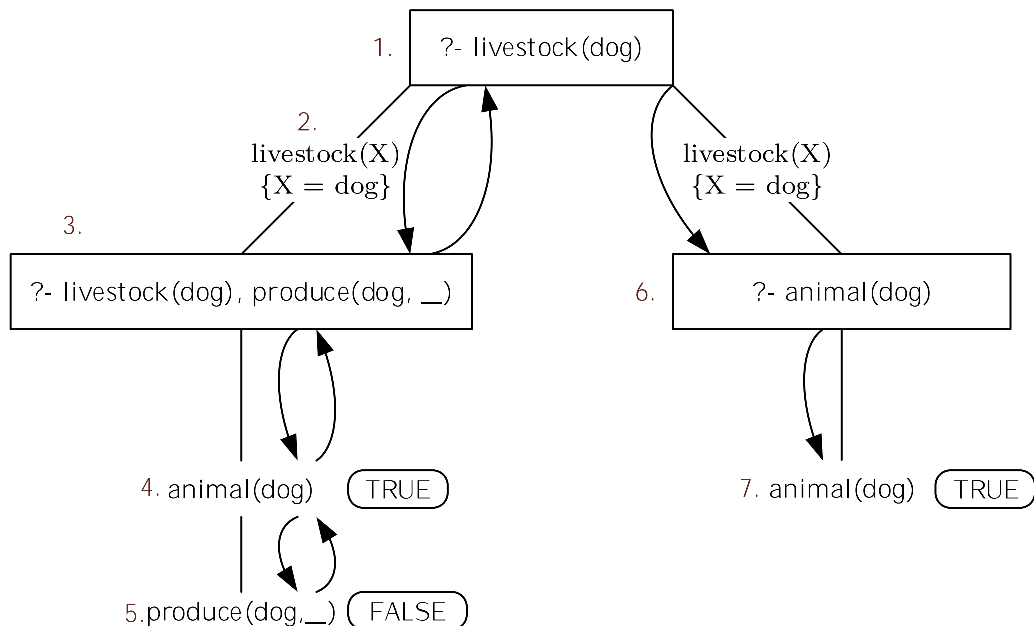
6. V dalším kroku je proveden pokus o znovusplnění cíle `livestock(dog).`, který se tentokrát unifikuje s hlavou nově vloženého pravidla

   ```
   livestock(X):-
       animal(X).
   ```

7. V posledním kroku dojde ke splnění cíle v těle pravidla

   ```
   ?- animal(dog).
   ```

   a celý výpočet končí s odpovědí **`yes`**.



Obrázek 1: Backtracking

Na závěr si všimněme, že např. otázka, zda je *kráva* hospodářské zvíře, by v případě naší databáze skončila se zápornou odpovědí, jelikož by interpret v obou pravidlech predikátu `livestock` selhal při splňování cíle `animal(cow)`. Záměrné vynechání faktů

```
animal(cow).
animal(hen).
```

z databáze tak mělo sloužit ilustraci, že správnost odpovědí prologovského interpretu na různé uživatelské dotazy je limitována pouze tou znalostí, kterou Prologu poskytne uživatel.

2.3 Prolog v kontextu predikátové logiky 1. řádu

Prolog z hlediska jeho teoretického základu spadá mezi jazyky logické, jelikož je založen na predikátové logice 1. řádu (neboli *predikátovém kalkulu 1. řádu*). Samotný název jazyka vznikl jako zkratka pro „programming in logic“.[9]

Predikátová logika 1. řádu (dále jen PL) je formální systém budovaný axiomaticky, tzn., že máme a priori zavedenou množinu axiomů, z nichž jsme pak schopni pomocí odvozovacích pravidel dedukovat další tvrzení (formule). Pro

[9] Přesněji je tento název akronymem francouzského PROgrammation en LOGique.

konkrétnost si postupně zavedeme některé důležité pojmy z PL, které budeme dále používat, či které souvisí s terminologií Prologu.

V případě formálních definic, které se v této kapitole vyskytují, jsem primárně čerpala z [5].

2.3.1 Jazyk PL

Jazykem rozumíme množinu korektně tvořených řetězců nad nějakou abecedou. Abecedu v případě PL tvoří:

- množina logických spojek $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ (většinou si ovšem vystačíme s některým z úplných systémů spojek – např. $\{\neg, \rightarrow\}$),
- kvantifikátory $\forall, \exists$ (resp. stačí např. pouze $\forall$, jelikož $\exists$ vyjádříme jako $\neg\forall\neg$),
- různé typy závorek a čárka,
- množina proměnných $\{x, y, z, \dots\}$.

Dále pak:

- množina funkčních symbolů $\mathcal{F} = \{f, g, h, \dots\}$,
- množina predikátových (relačních) symbolů $\mathcal{R} = \{p, q, r, \dots\}$,

které nám společně s jejich aritou tvoří tzv. typ jazyka $\mathcal{J} = (\mathcal{F}, \mathcal{R}, ar)$, kde $ar : \mathcal{F} \cup \mathcal{R} \rightarrow \mathbb{N}_0$.

Pomocí této abecedy pak můžeme strukturálně-induktivně definovat složitější jednotky jazyka PL – termy a formule.

Mnohé z těchto pojmů již zazněly v předchozí kapitole o Prologu. Můžeme k tomu ještě poznamenat, že konkrétní typ jazyka $\mathcal{J}$ je pro každý prologovský program unikátní a z velké části se na jeho definici podílí programátor (tím, že do databáze zavádí nově vytvořené predikáty k těm již předdefinovaným – tzv. *vestavěným predikátům*).

2.3.2 Termy

1. každá proměnná x je term,
2. je-li $f \in \mathcal{F}$ (s aritou n) a jsou-li $t_1, \dots, t_n$ termy, pak i $f(t_1, \dots, t_n)$ je term.

Tato definice termu z hlediska PL je zatím čistě syntaktická a v tomto ohledu se nám i částečně kryje s definicí termu, jak jsme si ji zaváděli ve smyslu univerzální datové struktury v Prologu – tedy jako proměnnou, konstantu a strukturu. „Proměnná“ zde uvedené definici odpovídá; stejně tak „konstanta“, která v PL představuje funkční symboly s aritou 0. Na rozdíly narazíme až u interpretace pojmu „struktura“. Aplikace funkčního symbolu f na termy $t_1, \dots, t_n$ tak, že vznikne složený term, ve skutečnosti odpovídá prologovské struktuře, která je tvořena svým funktorem (tj. funkční symbol f) a komponentami (tj. termy $t_1, \dots, t_n$). Za chvíli uvidíme, že *ne* všechny takto („prologovsky“) tvořené struktury můžeme nazývat termy ve smyslu definice PL.

2.3.3 Formule

atomická formule Je-li $r \in \mathcal{R}$ (s aritou n) a jsou-li $t_1, \ldots, t_n$ termy, pak $r(t_1, \ldots, t_n)$ je atomická formule.

Formule je definována následovně:

1. každá atomická formule je formule,
2. jsou-li φ, ψ formule, pak i $\neg\varphi, (\varphi \to \psi)$ jsou formule,
3. je-li x proměnná a φ formule, pak i $(\forall x)\varphi$ je formule.

Zde si jen uvědomme, že struktura, kterou jsme nazvali atomickou formulí, je tvořena stejným způsobem jako term. Až sémantika těchto dvou pojmů (termu a formule) nám je s konečnou platností odliší.

2.3.4 Struktura pro jazyk PL

Dosud jsme se zabývali syntaktickými prvky, pro které je ovšem potřeba zavést i příslušnou sémantiku. Tu nám pro daný jazyk $\mathcal{J} = (\mathcal{F}, \mathcal{R}, ar)$ poskytne struktura

$$\mathcal{M} = (U^{\mathcal{M}}, \mathcal{F}^{\mathcal{M}}, \mathcal{R}^{\mathcal{M}}),$$

kde

(i) $U^{\mathcal{M}}$ je univerzum obsahující prvky, kterých nabývají proměnné,

(ii) $\mathcal{F}^{\mathcal{M}} = \{f^{\mathcal{M}} : U^{\mathcal{M}^n} \to U^{\mathcal{M}} \mid f \in \mathcal{F}, ar(f) = n\}$ je množina operací, které vždy n-tici prvků z univerza $U^{\mathcal{M}}$ přiřadí opět prvek z univerza,

(iii) $\mathcal{R}^{\mathcal{M}} = \{r^{\mathcal{M}} \subseteq U^{\mathcal{M}^n} \mid r \in \mathcal{R}, ar(r) = n\}$ je množina n-árních relací mezi prvky z univerza $U^{\mathcal{M}}$.

Implicitně o množině $\mathcal{F}^{\mathcal{M}}$ (resp. $\mathcal{R}^{\mathcal{M}}$) uvažujeme jako o jistém zobrazení, které každému funkčnímu symbolu $f \in \mathcal{F}$ (resp. relačnímu symbolu $r \in \mathcal{R}$) přiřadí jeho interpretaci – tedy operaci $f^{\mathcal{M}}$ (resp. relaci $r^{\mathcal{M}}$).

Ve vztahu k Prologu nám univerzum $U^{\mathcal{M}}$ ve skutečnosti představuje množinu všech objektů, se kterými se v rámci prologovského programu pracuje – tj. převážně ve smyslu zavádění vztahů mezi těmito objekty (v příkladu z minulé kapitoly byly těmito objekty zvířata).

Konečně také můžeme přistoupit k objasnění rozlišení prologovských složených termů (struktur) ve vztahu k PL – tedy zodpovídáme otázku, které z nich můžeme považovat za logické formule a které za logické termy. Jelikož se v Prologu valná většina struktur zavádí (z hlediska sémantiky) ve smyslu predikátů, které pojmenovávají vztahy mezi danými objekty – tj. dá se uvažovat nad otázkou

platnosti/pravdivosti těchto vztahů – máme tu co do činění se strukturami jakožto *formulemi*. Tedy např. celá tato prologovská databáze je tvořena logickými formulemi:

```
animal(rabbit).
book(1984, author('George', 'Orwell')).
grandfather(X, Y):-
    father(X, Z),
    father(Z, Y).
```

Máme zde konstanty `rabbit`, `1984`, `'George'`, `'Orwell'`, proměnné `X`, `Y`, `Z`, unární relační symbol `animal`, binární relační symboly `book`, `grandfather`, `father` a nakonec binární funkční symbol `author`.[10]

U toho posledního se na chvíli zastavme. Zřejmě bychom mohli „symbol" `author` použít i jako relační symbol označující relaci „být autorem" a uložit samostatně do databáze fakt

```
author('George', 'Orwell').
```

Ovšem v případě, kdy se struktura `author` nachází jako komponenta ve struktuře `book`, s ní Prolog nakládá jako s běžným termem (ve smyslu definice PL).

Vidíme tedy, že při tom, kdy nějakou strukturu interpretujeme jako term či formuli, může záležet i na kontextu, v němž se daná struktura nachází.

Ve skutečnosti se např. struktury `grandfather` a `father` také vyskytují jako komponenty uvnitř struktury

```
:-(grandfather(X, Y), (father(X, Z), father(Z, Y))).
```

Přesto však s těmito strukturami Prolog zachází jako s (atomickými) formulemi, protože :- představuje predikát *implikace* a celá struktura tak tvoří složenou formuli přesně dle definice PL.

Ty struktury, které pak skutečně bez ohledu na kontext odpovídají definici logických (složených) termů, jsou v Prologu definovány pomocí tzv. *vyhodnotitelných funktorů* (z angl. *evaluable functors*), což jsou funktory (z pohledu PL funkční symboly $f \in \mathcal{F}$), které mohou být použity při tvorbě (aritmetických) výrazů, které mohou být posléze vyhodnoceny. Jsou to např. klasické aritmetické operátory jako $+$, $-$, $*$, $/$, $//$ (celočíselné dělení), mod či bitové posuvy $>>$, $<<$ aj.

[10] Ačkoliv to není zcela přesné používat zde terminologii PL, dopouštím se toho záměrně pro ilustraci korespondence.

2.3.5 $\mathcal{M}$-ohodnocení

Viděli jsme, že relace a operace, které tvoří interpretaci libovolného jazyka pracují s prvky univerza $U^{\mathcal{M}}$. Poslední důležitou věcí, kterou nám zbývá dodefinovat, je *zobrazení*, které tyto prvky přiřadí proměnným (potažmo termům). S tím jde samozřejmě ruku v ruce i definice zobrazení, které formulím (které jsou budovány z termů, příp. jiných formulí) přiřadí pravdivostní hodnotu.

Následující definice budou implicitně pro strukturu $\mathcal{M} = (U^{\mathcal{M}}, \mathcal{F}^{\mathcal{M}}, \mathcal{R}^{\mathcal{M}})$ jazyka $\mathcal{J} = (\mathcal{F}, \mathcal{R}, ar)$.

$\mathcal{M}$-ohodnocení proměnných je zobrazení v, přiřazující každé proměnné x prvek $v(x) \in U^{\mathcal{M}}$.

Hodnota termu t se značí jako $\|t\|_{\mathcal{M},v}$ a je definována následovně:

1. je-li t proměnná: $\|t\|_{\mathcal{M},v} = v(t)$,
2. je-li $t = f(t_1, \ldots, t_n)$: $\|t\|_{\mathcal{M},v} = f^{\mathcal{M}}\Big(\|t_1\|_{\mathcal{M},v}, \ldots, \|t_n\|_{\mathcal{M},v} \Big)$,
   kde $f \in \mathcal{F}$ a $t_1, \ldots, t_n$ jsou termy.

Pravdivostní hodnota formule φ se značí jako $\|\varphi\|_{\mathcal{M},v}$ a její definice je následující:

1. pro atomické formule $\varphi = r(t_1, \ldots, t_n)$:

$$\|\varphi\|_{\mathcal{M},v} = \begin{cases} 1 & \Big\langle \|t_1\|_{\mathcal{M},v}, \ldots, \|t_n\|_{\mathcal{M},v} \Big\rangle \in r^{\mathcal{M}} \\ 0 & \Big\langle \|t_1\|_{\mathcal{M},v}, \ldots, \|t_n\|_{\mathcal{M},v} \Big\rangle \notin r^{\mathcal{M}}, \end{cases}$$

2. pro formule φ, ψ a proměnnou x:

$$\|\neg\varphi\|_{\mathcal{M},v} = \begin{cases} 1 & \|\varphi\|_{\mathcal{M},v} = 0 \\ 0 & \|\varphi\|_{\mathcal{M},v} = 1, \end{cases}$$

$$\|\varphi \to \psi\|_{\mathcal{M},v} = \begin{cases} 1 & \|\varphi\|_{\mathcal{M},v} = 0 \text{ nebo } \|\psi\|_{\mathcal{M},v} = 1 \\ 0 & \text{jinak}, \end{cases}$$

$$\|(\forall x)\varphi\|_{\mathcal{M},v} = \begin{cases} 1 & \|\varphi\|_{\mathcal{M},u} = 1 \text{ pro každé } u, \\ & \text{kde } \forall y \neq x : u(y) = v(y)^{11} \\ 0 & \text{jinak}. \end{cases}$$

[11] Ohodnocení u a v se liší maximálně v tom, jakou hodnotu přiřazují proměnné x.

2.3.6 Teorie

Jako teorii T v jazyce $\mathcal{J}$ chápeme libovolnou množinu formulí tohoto jazyka.

Čistý prologovský program[12] se také dá považovat, jak bylo již dříve zmíněno, za konečnou množinu formulí, zachycující vztahy mezi objekty z nějakého univerza. [6] Z tohoto pohledu se tedy můžeme na program dívat jako na teorii. Z důvodu automatizace odvozování dalších formulí z této teorie uvažujeme pouze formule ve speciálním tvaru – tzv. Hornovské klauzule.

2.3.7 Hornovské klauzule

Klauzulí obecně nazýváme formuli $L_1 \vee \cdots \vee L_n$, která je ve tvaru disjunkce literálů, kde literál L_i je nějaká atomická formule či její negace.

Na Hornovu klauzuli je kladena dodatečná podmínka, že musí obsahovat maximálně jeden pozitivní literál. Ve výsledku tedy mohou nastat 4 případy, jak může taková klauzule vypadat. Nechť φ, ψ jsou atomické formule:

1. ψ (nulový počet negativních literálů a právě jeden pozitivní) – klauzule v tomto triviálním tvaru odpovídají prologovským *faktům*.

2. $\psi \vee \neg\varphi_1 \vee \cdots \vee \neg\varphi_n$ (nenulový počet negativních literálů a právě jeden pozitivní) – takový typ klauzulí je v Prologu zastoupen *pravidly*. Vzpomeneme-li si na jejich syntax, dojde nám, že mnohem blíže odpovídá ekvivalentnímu vyjádření výše uvedené klauzule, a to:

   $$(\varphi_1 \wedge \cdots \wedge \varphi_n) \rightarrow \psi.$$

   Formuli ψ jsme nazývali *hlavou*, konjunkci formulí φ_i jsme značili čárkami a nazývali jsme ji *tělem* pravidla.

   Za zmínku také stojí otázka kvantifikace proměnných v pravidlech. Defaultně jsou všechny proměnné v klauzuli univerzálně kvantifikované. Ovšem je i možnost obecně kvantifikovat pouze proměnné vyskytující se v hlavě pravidla a zbylé proměnné kvantifikovat existenčně uvnitř těla [6]. Ukážeme tento způsob kvantifikace na pravidlu z příkladu z kapitoly 2.3.4, přepsaném do klasické logické notace (v níž predikát `father` nahradíme za predikátový symbol f a `grandfather` za g):

   $$(\forall x)(\forall y)\Big[(\exists z)\Big(f(x,z) \wedge f(z,y)\Big) \rightarrow g(x,y)\Big].$$

   Význam této formule pak lehko odpovídá intuici: Pro všechna x, y je x pradědou y, existuje-li z takové, že x je otcem z a z je otcem y. Ekvivalence této formule s formulí, v níž jsou všechny proměnné univerzálně kvantifikované, pak plyne z následujícího vztahu:

[12] „Čistý“ řečeno proto, jelikož v praxi mohou konkrétní implementace Prologu přidávat jistou nadstavbu základní funkcionality, která již nemusí odpovídat formálnímu modelu PL. Mohou to být kupříkladu různé predikáty s vedlejším efektem – pro práci se standardním I/O, se soubory, predikáty pro kontrolu backtrackingu atd.

$$(\forall x)(\varphi \rightarrow \psi) \leftrightarrow ((\exists x)\varphi \rightarrow \psi), \text{ jestliže } x \text{ není volná v } \psi.$$

3. $\neg\varphi_1 \vee \cdots \vee \neg\varphi_n$ (nenulový počet negativních literálů a žádný pozitivní literál) – klauzule tohoto typu jsou protějškem prologovských *dotazů*. Pokud vycházíme z toho, že máme (podobně jako ve 2. bodu) všechny proměnné univerzálně kvantifikované (obecně předpokládejme, že ve formulích φ_i se vyskytují pouze volné proměnné $x_1, \ldots, x_n$) – uděláme tedy obecný uzávěr

   $$(\forall x_1) \ldots (\forall x_n)(\neg\varphi_1 \vee \cdots \vee \neg\varphi_n).$$

   S úpravami pak dostaneme negaci formule, v níž jsou všechny proměnné kvantifikovány *existenčně*:

   $$\neg\Big[(\exists x_1) \ldots (\exists x_n)(\varphi_1 \wedge \cdots \wedge \varphi_n)\Big].$$

   Slovy můžeme formuli v hranatých závorkách (bez negace na začátku) vyjádřit jako otázku: „Existují $x_1, \ldots, x_n$ takové, že platí $(\varphi_1 \wedge \cdots \wedge \varphi_n)$?“ V tom už vidíme jasnou korespondenci s prologovskými dotazy, kde jsme po danou konjunkci cílů (dotaz) hledali unifikaci pro proměnné v databázi tak, abychom tuto konjunkci splnili. Zbývá odpovědět na to, proč je tato formule představující dotaz ve tvaru *negace*. To souvisí s tím, že „zodpovězení“ dotazu formálně představuje problém odvození sporu z negace dotazu (tj. výše uvedená formule) a množiny formulí T, která představuje program v Prologu (je to tedy množina faktů a pravidel). Pokud se to podaří, odpověď na takový dotaz je klasicky (jak to známe z Prologu) `yes`. K formalizaci této formy důkazu se dostaneme v příštích kapitolách.

4. $\emptyset$ (nulový počet literálů) – prázdná klauzule, častěji značená jako $\square$, je vyjádřením *sporu*. Je tedy vždy nepravdivá.

2.3.8 Důkaz

O PL jsme se zmiňovali jako o *axiomatickém systému*. Ten se zakládá na pětici axiomových schémat[13], jejichž platnost předpokládáme, a dvojici odvozovacích pravidel:

- **modus ponens** – z formulí φ a $\varphi \rightarrow \psi$ odvodí ψ,
- **generalizace** – z formule φ odvodí $(\forall x)\varphi$.

Dle [6] je výpočet v logickém programu ve skutečnosti konstruktivní *důkaz*, že nějaké tvrzení (resp. *cíl*, abychom byli věrní terminologii Prologu) vyplývá z tohoto programu, přesněji řečeno, je z tohoto programu dokazatelné. Podívejme se, jak je důkaz definován v PL.

[13] Ty tu nebudeme vypisovat – uvedené jsou v [5].

Důkaz formule φ z teorie T je libovolná posloupnost formulí $\varphi_1, \dots, \varphi_n$, kde $\varphi_n = \varphi$ a každá φ_i

- je axiom
- nebo je formulí z T
- nebo plyne z předchozích formulí důkazu pomocí *modus ponens* nebo pravidla *generalizace*.

Je-li formule φ dokazatelná z teorie T, značíme tento fakt jako $T \vdash \varphi$.

2.3.9 Nejobecnější unifikace

Nejobecnější unifikací θ, někdy zkracovanou jako mgu (z angl. *most general unifier*), rozumíme speciální typ substituce[14], po jejíž aplikaci (značíme obecně $\zeta_i\theta$) na formule $\zeta_1, \zeta_2, \dots, \zeta_n$ vzniknou syntakticky identické formule, tedy platí

$$\zeta_1\theta = \zeta_2\theta = \dots = \zeta_n\theta.$$

To, že je θ *nejobecnější* unifikací, znamená, že ke každé další unifikaci σ musí existovat třetí unifikace ω taková, že $\sigma = \theta \circ \omega$. Tj. σ (někdy nazývaná jako *specializovanější unifikace*) se dá vyjádřit jako složení naší nejobecnější unifikace θ s nějakou unifikací ω.

Když si vzpomeneme na unifikaci v Prologu, víme, že k ní dochází skrze aplikaci operátoru =. Na teoretické rovině ve skutečnosti dochází k hledání mgu. Demonstrujme si tento pojem na dvou prologovských strukturách $f(X, g(Y))$ a $f(a, Z)$, které chceme unifikovat. Označíme-li jednu unifikaci jako

$$\sigma = \{X = a, Y = a, Z = g(a)\}$$

a druhou jako

$$\theta = \{X = a, Z = g(Y)\},$$

pak první z nich skutečně unifikací je, neboť platí $f(X, g(Y))\sigma = f(a, Z)\sigma$, ale není to mgu, jelikož se dá vyjádřit např. takto pomocí θ a $\omega = \{Y = a\}$:

$$\sigma = \theta\omega.$$

Naproti tomu θ je mgu, neboť už se nedá dále vyjádřit pomocí obecnější unifikace.

[14] Pojem *substituce* si zde formálně zavádět nebudeme. Řádnou definici může čtenář nalézt opět v textu [5].

2.3.10 Robinsonův rezoluční princip

Nyní si definujeme odvozovací pravidlo, představené v r. 1965 J. A. Robinsonem, které ze dvou klauzulí odvodí jejich tzv. *rezolventu*.

Z klauzulí

$$(\varphi_1 \vee \cdots \vee \varphi_{i-1} \vee \boldsymbol{\psi} \vee \varphi_{i+1} \vee \cdots \vee \varphi_n),^{15}$$
$$(\varphi'_1 \vee \cdots \vee \varphi'_{i-1} \vee \boldsymbol{\neg\psi'} \vee \varphi'_{i+1} \vee \cdots \vee \varphi'_m)$$

odvodí rezolventu

$$\varphi_1\theta \vee \cdots \vee \varphi_{i-1}\theta \vee \varphi_{i+1}\theta \vee \cdots \vee \varphi_n\theta \vee \varphi'_1\theta \vee \cdots \vee \varphi'_{i-1}\theta \vee \varphi'_{i+1}\theta \vee \cdots \vee \varphi'_m\theta,$$

kde $\varphi_1, \ldots, \varphi_n, \varphi'_1, \ldots, \varphi'_m$ jsou literály a ψ, ψ' atomické formule. θ zde představuje nejobecnější unifikaci formulí ψ, ψ' (platí tedy $\psi\theta = \psi'\theta$).

Nejjednodušší speciální případ tohoto rezolučního pravidla obsahuje dvě jednoprvkové klauzule (opět unifikovatelné θ), tj.:

$$\text{Z } \psi, \neg\psi' \text{ odvodíme } \Box \text{ (spor).}$$

Popišme si nyní ideu důkazu – pomocí rezolučního pravidla – toho, že daná klauzule φ syntakticky[16] vyplývá z teorie T. Jak jsme již uvedli dříve, tento fakt značíme jako $T \vdash \varphi$. Poznamenejme jen, že ve vztahu k Prologu by T byla prologovským programem, jsou-li klauzule z T ve tvaru Hornovských klauzulí a φ by pak byla cílem, který byl interpretu Prologu zadán ve formě dotazu.

Vraťme se ale k důkazu – kapitole 2.3.7 jsme zmiňovali, že se jedná o důkaz sporem. Naším cílem tedy bude vyvodit prázdnou klauzuli $\Box$.

Na začátku přidáme do množiny předpokladů T znegovaný cíl φ (přičemž musí být opět ve tvaru klauzule, aby formule z následující množiny bylo možno vyjádřit v CNF):

$$T \cup \{\neg\varphi\}.$$

Poté je na tuto množinu (či přesněji na dvojice prvků z ní) opakovaně aplikováno rezoluční pravidlo, přičemž dochází k přidávání rezolvent do této množiny. To probíhá tak dlouho, dokud

- již není možné přidat novou rezolventu (což znamená $T \nvdash \varphi$)
- nebo ze dvou klauzulí je odvozena $\Box$ (což znamená $T \vdash \varphi$).

Formálně bychom rezoluční důkaz zavedli podobně, jako jsme zaváděli důkaz v kapitole 2.3.8.

[15] Čárkou je zde míněna konjunkce – dohromady tedy máme formuli v CNF.

[16] Dle věty o úplnosti je syntaktické vyplývání ekvivalentní se sémantickým, tím se tu ale zabývat nebudeme.

Rezoluční důkaz klauzule C z teorie T je posloupnost klauzulí $C_0, \ldots, C_n$, kde $C_n = C$ a každá C_i

- je klauzulí z T
- nebo *rezolventou* některých dvou předchozích klauzulí C_j a C_k $(j, k < i)$ z důkazu. [7]

SLD rezoluce je odvozovací princip, který je speciálním případem Robinsonovy rezoluční metody, využívajícím Hornovské klauzule. Název vznikl z anglického *Selective Linear Definite clause resolution*. Právě na tomto principu je založeno vyhodnocování dotazů v Prologu.

3 Expertní systémy

Jako základní definici expertního systému si dovolím použít definici Edwarda Feigenbauma ze Standfordské univerzity, který stál na počátku rozvoje znalostních systémů na přelomu 70. a 80. let (mj. se podílel na známých prvotinách mezi expertními systémy jako je systém Mycin či Dendral).

> *Expertní systém je inteligentní počítačový program, který používá znalosti a odvozovací mechanismus pro řešení problémů vyžadujících lidskou expertízu.* [8]

Postupně se blíže podíváme a rozšíříme základní komponenty této definice.

Zaznělo v ní, že expertní systém (dále jen ES) je *inteligentní počítačový program*, z čehož můžeme implikovat, že jeho úlohou je inteligentně a automatizovaně simulovat rozhodování lidského experta při řešení daného problému tak, aby byla pokud možno dosažena stejná kvalita úsudku. Tím ovšem není myšlena co nejpřesnější simulace kognitivních procesů při rozhodování člověka včetně „příměsi“ nechtěného subjektivního faktoru (ve formě chyb spojených s neexaktním uvažováním, založeným na lidských emocích, intuici, zapomnětlivosti, ...) do požadovaného objektivního výsledku. [9]

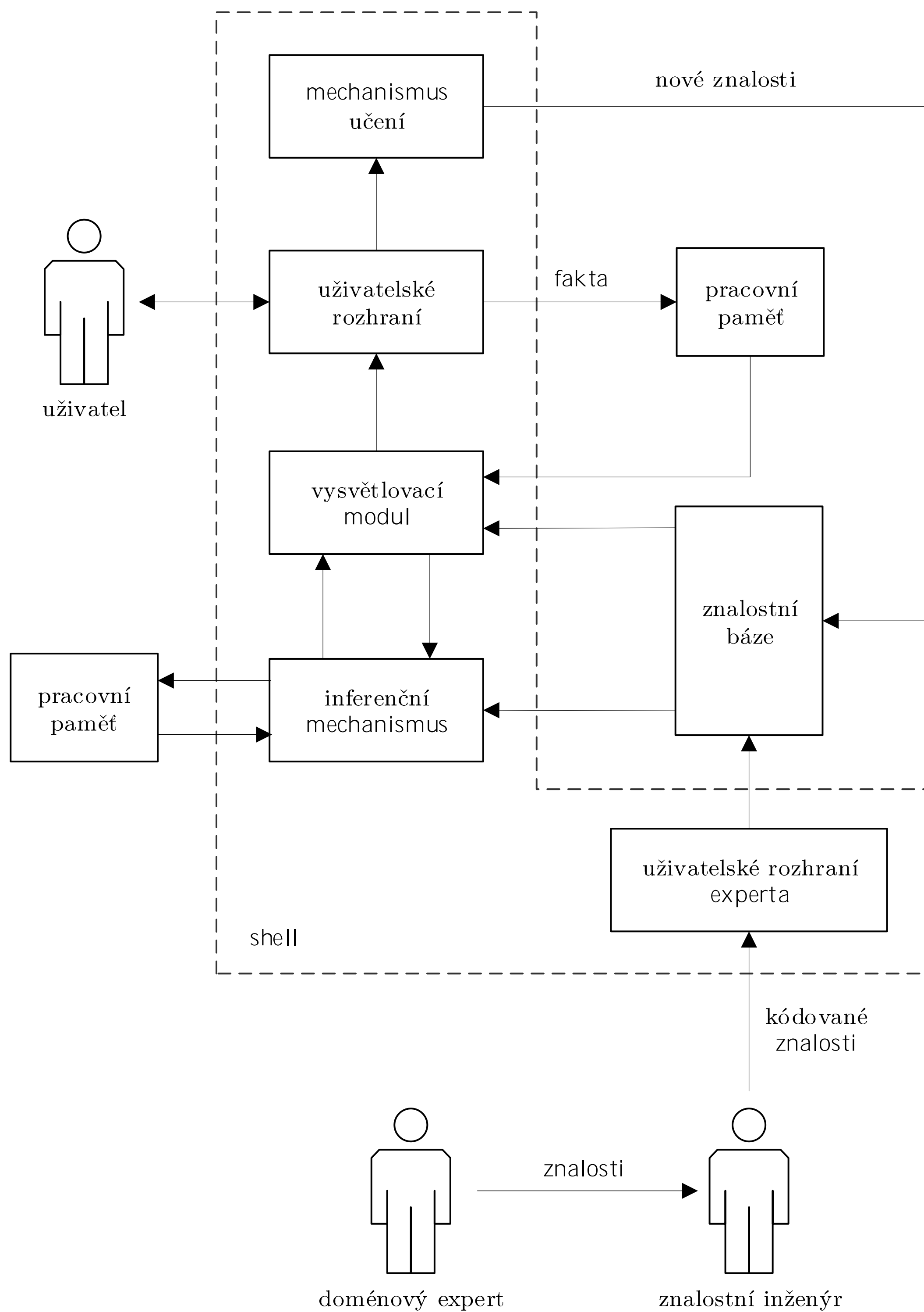
Dále jsou v ES *používány znalosti*. Na ně je kladen požadavek, aby byly vhodně zakódovány a strukturovány tak, aby mohly být následně automatizovaně zpracovávány. Podrobnosti k této problematice budou dále rozebrány v podkapitole Znalostní báze.

Inferenční (či *odvozovací*) *mechanismus* je část ES, která obstarává odvozování nových znalostí z již známých dat (tj. znalostí obsažených v tzv. bázi znalostí), což se děje podle jasně specifikovaných předpisů – tzv. odvozovacích pravidel. (Více opět v samostatné podkapitole Inferenční mechanismus.)

Tyto dvě komponenty (znalostní báze a inferenční mechanismus) dohromady tvoří hlavní části ES.

Doplňující součásti, které můžeme dále nalézt v ES, jsou např. *báze dat* (nebo také *pracovní paměť*), v níž jsou shromažďována průběžná data ke konkrétnímu řešenému problému; *vysvětlovací modul*, který zajišťuje odůvodnění svého rozhodnutí vyobrazením posloupnosti kroků, které ho k danému závěru vedly; *„učící se“ mechanismus* (learning mechanism), který umožňuje systému přizpůsobit svoje znalosti nějakému novému aspektu zkoumané domény [10] a v neposlední řadě také *uživatelské rozhraní*, skrze něž se děje samotná interakce s uživatelem. [11]

Jednu z možností, jak spolu mohou tyto komponenty ES interagovat, můžeme vidět znázorněnou na obrázku 2, a to včetně rolí, které v tomto procesu může sehrát člověk (k čemuž se dostaneme později).



Obrázek 2: Schéma expertního systému

3.1 Typy expertních problémů

V souvislosti s úvodní definicí ještě může vyvstat otázka, co vlastně můžeme považovat za *problémy vyžadující lidskou expertízu*, které ES řeší. Dle [9] (z něhož dále čerpám v této podkapitole) existují tři základní typy expertních úloh, podle nichž také rozdělujeme ES:

1. diagnostické úlohy,
2. plánovací úlohy,
3. smíšené úlohy[17].

V praxi je takových kategorií samozřejmě více, ale v každé z nich by se nakonec daly objevit principy, které jsou již pokryty tímto základním rozdělením. Uveďme si tu aspoň na ukázku další kategorie úloh: monitorování, předpovídání (např. následků některých situací), ladění (debugging), designování aj.

Za nejvýraznější rys *diagnostických úloh* považujeme to, že množina případů či *diagnóz*, které mohou nastat jako výsledek diagnostického procesu, je *konečná*. Pro ES, které takové úlohy řeší, to v podstatě znamená, že výsledný model problému složený z hypotéz, které mohou nastat, má již pevně daný; jeho úkolem je na základě vstupních dat dokázat platnost jedné z hypotéz. Jako příklad diagnostického ES můžeme uvést již zmíněný Mycin, jehož úkolem je diagnostikovat bakteriální infekce a navrhnout adekvátní léčbu či PROSPECTOR navržený pro průzkum minerálních ložisek za účelem „diagnózy“ vhodného místa pro vrt.

ES, které řeší *plánovací úlohy*, se také někdy říká *generativní*. Je tomu tak z dobrého důvodu, neboť tyto systémy pracují s počátečními daty a vědomostí o tom, jak by mělo vypadat řešení a na základě těchto informací se požadované řešení snaží vygenerovat. Nepracují tedy s konečnou množinou možných výsledků, jak je tomu u diagnostických ES, ale snaží se je vytvořit. Často může být výsledkem takového výpočtu i množina řešení ohodnocená dle různých kritérií (např. optimality, pravděpodobnosti). Příkladem plánovacího/generativního systému může být MOLGEN, což je ES sloužící k plánování experimentů v molekulární genetice či Dendral, systém pro identifikaci chemických sloučenin skrze odvozování struktur těchto látek na základě dat ze spektrometru (proto generativní).

Tzv. hybridní ES, které se používají pro řešení posledního typu úloh, jsou ve své podstatě dosti obecným pojmem, jelikož jejich funkci můžeme chápat za hranice našeho rozdělení úloh a ES (např. z hlediska způsobu reprezentace znalostí ve znalostní bázi nebo z hlediska inferenční strategie, kdy tak mohou slučovat vícero paradigmat). Z našeho pohledu hybridní systém představuje sloučení rysů diagnostických a plánovacích ES. Příkladem může být výukový expertní systém, který diagnostikuje znalosti studenta a na základě toho mu naplánuje adekvátní studijní plán.

[17] tj. nesou prvky diagnostických a plánovacích úloh

3.2 Znalostní báze

Znalostní báze je modul ES, který perzistentně uchovává deklarativní reprezentaci znalostí potřebných k expertíze nad konkrétní expertní doménou [12]. Slovo „deklarativní“ je použito z důvodu, že ve znalostní bázi nejsou zakódovány instrukce, které se po spuštění interpretu provedou a vznikne výpočetní proces, který dále zpracovává poznatky obsažené ve znalostní bázi. Znalosti, které popisují způsob, jak uložené poznatky zpracovávat, jsou však přitom (zdánlivě paradoxně) také obsahem znalostní báze – jsou to tzv. procedurální znalosti. Procedurální ve smyslu, že inferenčnímu mechanismu poskytují návod, jak má manipulovat s daty pro konkrétní problém, aby vzniklý výpočet (o který se stará právě inferenční mechanismus) vedl k vyřešení problému. [13]

Běžné počítačové programy organizují znalosti na dvou úrovních: data a kontrolní kód. To, čím se od nich ES odlišují, je právě zavedení třetí úrovně strukturace znalostí – báze znalostí. [13] Ta je striktně oddělena od inteligence řídícího zdrojového kódu (inferenčního mechanismu), což principiálně umožňuje i existenci tzv. prázdných expertních systémů. To jsou ES, v nichž není přítomna báze znalostí (resp. je prázdná). Jejich odvozovací mechanismus tedy není závislý na jedné konkrétní množině znalostí, což takovému systému umožňuje řešit více problémů, které však typicky musí být svým charakterem příbuzné. Výzkum této oblasti v 80. letech totiž ukázal, že realizace univerzálního prázdného ES, využitelného pro všechny typy problémů, není zcela možná kvůli přílišné odlišnosti požadavků na znalostní bázi a inferenci u různých kategorií problémů. Prázdné ES, které slouží jen pro určitou problémovou oblast, ale vůči různým znalostním bázím jsou flexibilní, se nazývají také *dedikované*. [9, 11]

Výběr reprezentace znalostí a kvalita znalostní báze je pro výslednou účinnost ES naprosto stěžejní. Dobrý návrh znalostní báze je proto mnohdy nejnáročnější disciplínou při tvorbě ES (někdy kvůli tomu nazýván jako „knowledge engineering bottleneck“). O naplnění systému znalostmi převzatými v přirozeném jazyce od doménového experta (tj. expert na určitý obor/doménu v reálném světě) se stará tzv. znalostní inženýr (knowledge engineer). Jeho úkolem je zakódovat tato „syrová“ data do strojově zpracovatelné podoby. [12, 14, 15]

To, co chápeme jako znalosti, jsou v tomto ohledu jednak jednoduché *fakty* o dané expertní doméně (tj. tvrzení, která můžeme považovat za pravdivá), dále také *pravidla*, která popisují vztahy mezi těmito fakty a postihují tak určité fenomény a zákonitosti platné v dané doméně a nadto i další doplňující informace jako heuristiky, typické situace, konkrétní doporučení a metody, které by v praxi použil lidský expert pro řešení problému – souhrnně možné charakterizovat jako výše zmiňované procedurální znalosti. [11] V praxi se ES musí vypořádat s mnoha různými druhy znalostí a musí je být schopen integrovat do koherentní znalostní báze. [16]

3.2.1 Logická reprezentace

Zazněly zde pojmy *fakty* a *pravidla*, které známe již z kapitoly o Prologu. To nám může napovědět něco o způsobech reprezentace znalostí. Z kapitol o predikátové logice ve vztahu k Prologu víme, že jak fakty, tak pravidla můžeme velmi efektivně vyjadřovat pomocí logických formulí. Tak se nám z predikátové logiky vlastně stává nástroj pro reprezentaci znalostí v ES. [17]

Logické programování je ve skutečnosti programování pomocí *deskripce*. Programátor popisuje danou aplikační doménu tak, že nejprve konceptualizuje objekty, které v dané doméně existují (nebo se jejich existence předpokládá) a relace, které by tyto objekty měly splňovat. Obojí je pak zapsáno v příslušném formálním jazyce (tím může být např. právě Prolog, který byl mj. tím jazykem, s nímž po r. 1972[18] vešlo do praxe logické programování [18]). Většina logických programovacích jazyků pracuje s formulemi v klauzulárním tvaru, což umožňuje automatizovanou rezoluci[19]. Výhodou logické reprezentace znalostí je relativní jednoduchost tvorby takové báze (logický jazyk není tomu přirozenému zas tak vzdálený); především však to, že ji podpírá silný formální pilíř ve formě predikátového kalkulu, který poskytuje nástroje, jak nad takto reprezentovanými znalostmi uvažovat. [19]

3.2.2 Produkční pravidla

Nejpopulárnějším a zároveň nejpřirozenějším formalismem pro reprezentaci znalostí jsou obecně pravidla ve tvaru

IF [situace] THEN [akce],

které nazýváme *produkčními pravidly*, protože pomocí nich můžeme na základě existující znalosti „vyprodukovat" novou znalost. [11, 17] Produkční pravidla jsou v podstatě podmnožinou predikátového kalkulu s tím rozdílem, že je do nich přidána „preskriptivní komponenta", která ukazuje, jak nakládat s informací v pravidlech během odvozování. [16] Když se podíváme na expertízu, prováděnou ES v čase, jako na posloupnost stavů systému, produkční pravidla můžeme z tohoto pohledu vnímat jako určité návody, jak měnit stav systému na základě nějakého vzoru, který je v současném stavu systému rozpoznán – tzv. *pattern-invoked programs*. Tyto „programy" nejsou volány jinými programy v běžném slova smyslu, ani mezi sebou navzájem nijak neinteragují, nýbrž jsou aktivovány nějakou kontrolní strukturou (inferenčním mechanismem), kdykoliv jsou jisté podmínky splněny v datech. [13, 20]

Obecně volba reprezentace závisí na požadavcích, které na bázi znalostí klademe. Uveďme si zde některé, kterým dobře vyhovují právě modely založené na pravidlech (tzv. rule-based či produkční systémy) či predikátové logice [9, 11]:

- modularita – znalosti by měly být vyjádřeny jako (relativně) nezávislé kusy informace,

[18] rok vyvinutí Prologu Alainem Colmerauerem a Philippem Rousselem

[19] viz kapitoly 2.3.7, 2.3.10

- inkrementálnost – databázi by mělo být možno jednoduše rozšiřovat o nové znalosti (což přímo souvisí s požadavkem modularity),
- modifikovatelnost – existující znalosti bychom měli být schopni snadno upravit (opět nezávisle na ostatních znalostech),
- transparentnost – báze znalostí by měla být pro znalostního inženýra lehce čitelná.

V kritériích ovšem není kupříkladu pokryta potřeba mít taková modulární data nějak hierarchicky strukturovaná (tedy potřeba do nezávislých dat přeci jen vnést nějaké závislosti). Jednak za čistě praktickým účelem přehlednosti z pohledu znalostního inženýra, který se často musí starat o velmi rozsáhlý komplex znalostí (což znamená i vyvarování se např. přítomnosti duplicitních znalostí v databázi), ale i z hlediska algoritmické složitosti, která závisí právě na rychlém vybavování znalostí a to zase závisí na existenci nějaké sémantické hierarchie pojmů. To v konečném důsledku vede k nutnosti kompromisu mezi těmito dvěma požadavky a motivuje to vznik dalších paradigmat reprezentace dat. Jako zástupce můžeme zmínit např. *rámce* (frames) nebo *sémantické sítě*. [9]

3.2.3 Rámce

Rámce jsou datové struktury sestávající z identifikátoru objektu (nebo i celé třídy objektů), který představují, a slotů, které v průběhu expertízy nabývají hodnot (tedy tvoří páry ve formě atribut-hodnota) a slouží tak k popisu vlastností objektu (resp. třídy objektů). [17]

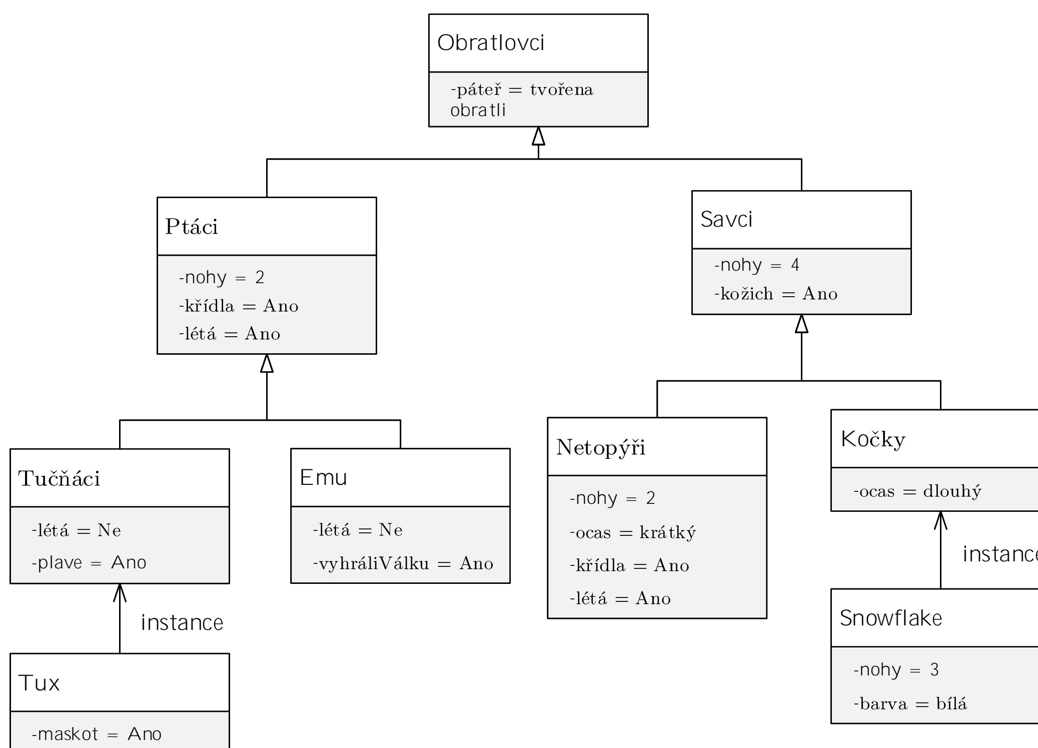
Výhoda rámců spočívá v tom, že zachycují způsob, kterým lidští experti typicky přemýšlí o svých znalostech – tj. v objektech a jejich vlastnostech (atributech), mají mezi nimi zavedené vztahy, hierarchické uspořádání a dále je používají např. k definicím pomocí specializace či generalizace. [16] Zakladatel konceptu rámců Marvin L. Minsky z MIT jejich specifikum viděl v tom, že by měly reprezentovat určité stereotypy či stereotypní situace[20], s nimiž jsou pak i spjaty očekávané vlastnosti. [21]

Propojení rámců a stanovení určité hierarchie mezi nimi docílíme jejich organizací do *taxonomií*[21], které vytvoříme použitím dvou konstruktů, které představují vztahy mezi rámci: tzv. *member-links*, které reprezentují členství v nějaké třídě a *subclass-links* představující specializaci nebo generalizaci. [16]

Obrázek 3 zachycuje jednoduchou hierarchii rámců, která popisuje malou část podkmenu obratlovců v taxonomii živočichů. Každý rámec má své záhlaví, v němž je název objektu; ve svém těle má pak sloty, které obsahují páry ve tvaru atribut-hodnota, uchovávající vlastnosti jednotlivých zvířat (či celých kategorií). Výše zmíněné member-links jsou ve schématu zaznačeny hranami s popiskem *instance*[22]. Tedy např. kočka se jménem Snowflake je instancí třídy/rámce Kočky.

[20] Dnes bychom v terminologii OOP obojí nazvali třídou.
[21] Obecně tím rozumíme klasifikační systém/schéma. [22]
[22] Jinými slovy „členství ve třídě“, do níž směřuje orientovaná hrana.

Dále představené subclass-links jsou znázorněny zbylými orientovanými hranami. Kupříkladu Kočky jsou specializací rámce Savci (či z opačného hlediska Savci jsou generalizací rámců Kočky a Netopýři).



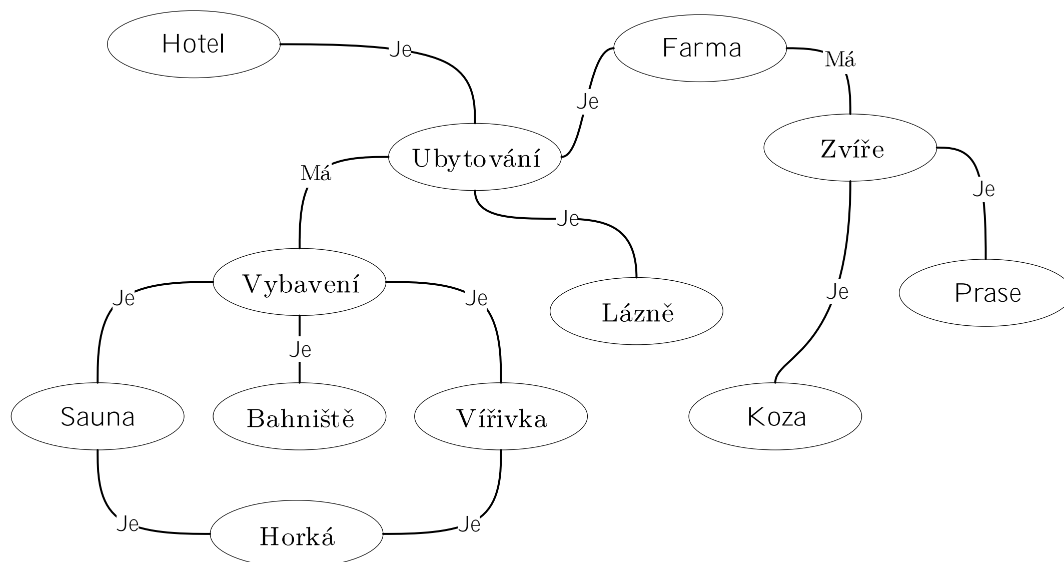
Obrázek 3: Rámce

3.2.4 Sémantické sítě

Podobnost sémantických sítí s rámci je patrná v tom, že v nich znalosti taktéž soustřeďujeme do objektů. Mezi těmito objekty také zavádíme relace, které je vzájemně propojují a vzniká tak orientovaný graf, jehož uzly představují objekty a ohodnocené (či popsané) hrany jsou relace mezi nimi. Ty nejběžnější, se kterými se setkáváme, jsou relace IS-A a HAS. [13]

Na obrázku 4 můžeme vidět jednoduchý příklad sémantické sítě se základními relacemi *být* a *mít*. Např. farma i lázně mohou *mít* bahniště[23], protože farma i lázně *jsou* ubytování, ubytování *má* vybavení a bahniště *je* vybavení.

[23] Akorát cíloví uživatelé jsou v obou případech různí.



Obrázek 4: Sémantická síť

3.3 Inferenční mechanismus

V souvislosti s produkčními pravidly jsme zmiňovali pohled na expertízu prováděnou ES jako posloupnost stavů systému, které jsou měněny aplikací těchto pravidel inferenčním mechanismem. ES je z tohoto pohledu dynamickým systémem a inferenční proces, zjednodušeně řečeno, jeho dynamiku s použitím různých přístupů „dává do pohybu“. Právě tyto „strategie“ budou předmětem této kapitoly. Pro jednoduchost se přitom zaměříme na produkční ES.

3.3.1 Stavový prostor

Jakékoli inferenční strategii nutně předchází definice *stavového prostoru*, který tvoří vnitřní model daného problému[24], který ES řeší. [23]

Stavový prostor je struktura $\mathscr{S} = (S, P, I, G)$ [17, 24], kde

$$\begin{aligned} S &= \{s_i\} \ldots \text{ konečná množina stavů,} \\ P = \{\varphi_j : S &\to S\} \ldots \text{ konečná množina přechodů mezi stavy,} \\ &\ \text{ kde } \varphi_j \text{ je parciální zobrazení,} \\ I &\ \ldots \text{ neprázdná podmnožina } S \text{ iniciálních stavů,} \\ G &\ \ldots \text{ neprázdná podmnožina } S \text{ cílových stavů.} \end{aligned}$$

[24] V obecném případě nemusí být vůbec snadné definovat stavový prostor problému, zvláště pokud jde o úlohy z přirozeného světa, které svou podstatou nejsou dobře formálně definovatelné. [17]

Stavem systému s_i rozumíme množinu všech aktuálně pravdivých tvrzení – tj. uživatelských vstupních dat společně s již odvozenými tvrzeními v pracovní paměti, a stejně tak i množinu axiomů (resp. faktů) ve znalostní bázi. Tato data o okamžitém stavu řešené úlohy také nazýváme *bází dat.* [17] Produkční pravidla zde pak hrají roli přechodů mezi stavy. Korespondence je taková, že k pravidlu p_i ve tvaru

$$[antecedent] \Longrightarrow [konsekvent/akce]$$

přísluší zobrazení φ_i definované následovně:

$$\varphi_i(s_j) = s_i,$$

kde $s_i, s_j \in S$ interpretujeme jako

$s_j \ldots$ v bázi dat je platný *antecedent*,
$s_i \ldots$ v bázi dat je platný *konsekvent*
nebo byla změněna vykonáním *akce*.

Řešení problému pak můžeme formálně vyjádřit jako posloupnost zobrazení

$$\varphi_1, \varphi_2, \ldots, \varphi_n,$$

k níž je možno přiřadit posloupnost stavů $s_0, s_1, \ldots, s_n$ takovou, že $s_0 \in I, s_n \in G$ a platí rekurzivní předpis:

$$\begin{aligned} s_1 &= \varphi_1(s_0), \\ s_i &= \varphi_i(s_{i-1}). \end{aligned}$$

Stavový prostor můžeme také reprezentovat jako orientovaný graf (obvykle strom), v němž uzly představují stavy a hrany představují přechody mezi stavy. Výpočet nad problémem je pak pouze hledáním cesty z iniciálního stavu do cílového (kterých může být v rámci problému i více). Nutno poznamenat, že použití této reprezentace samo o sobě poukazuje na nedeterministickou povahu řešených problémů, jelikož dochází k větvení na více možností volby[25]. Takový soubor hran vycházejících z jednoho uzlu (stavu) představuje tzv. *konfliktní množinu*, což je množina pravidel, jež je možno v daném stavu aplikovat. Je-li mohutnost množiny větší než 1, inferenční mechanismus musí provést tzv. *rezoluci konfliktu* (conflict resolution) a podle dané strategie zvolit jedno pravidlo, které se dále použije. Inferenční procedura prohledáváním grafu stavového prostoru v podstatě generuje podgraf tohoto grafu. [17] Jedním z ustálených témat v umělé inteligenci je to, jak toto prohledání provést co nejefektivněji. [21]

Inferenční mechanismus produkčního ES pracuje v základní kontrolní smyčce zvané *recognition-act cyklus*, složené (jak název napovídá) ze dvou fází [10, 20]:

[25] Inferenční mechanismus je tedy v podstatě nedeterministický Turingův stroj, který provádí operaci nedeterministického výběru.

1. rozpoznání (recognition) – v této fázi mechanismus na základě aktuálního stavu systému (určeného obsahem báze dat) sestaví konfliktní množinu pravidel a následně provede rezoluci konfliktu,

2. akce (act) – v tomto kroku je vybrané pravidlo „aplikováno“, tzn. na základě jeho pravé nebo levé strany (záleží na tom, v jakém směru odvozování se pohybujeme) je změněna báze dat.

Na inferenční strategie se lze dívat z několika úrovní – z hlediska směru aplikace produkčních pravidel a z hlediska způsobů rezoluce konfliktu.

3.3.2 Zpětné řetězení

Zpětné řetězení (neboli backward chaining) je inferenční strategie, která produkční pravidla aplikuje „pozpátku“. Celý proces začíná vytyčením cíle[26], jehož platnost bude třeba na základě obsahu báze dat a množiny pravidel dokázat. V případě, že cíl není triviálně splněn porovnáním s tvrzeními v bázi dat, mechanismus sestaví konfliktní množinu pravidel, jejichž konsekventy se unifikují se zadaným cílem, z níž je posléze (dle nějakých kritérií) jedno pravidlo zvoleno. Antecedent tohoto pravidla je stanoven za podcíl, jehož platnost bude ověřována v dalším recognition-act cyklu inferenčního mechanismu. Celý proces se opakuje do doby, dokud není původní cíl dokázán či vyvrácen. [20]

Tento typ odvozování se hodí v problémech, v nichž máme ohraničené množství možných závěrů. Je pak proto snaží vybrat jednu hypotézu a ověřit její pravdivost. Příkladem takových problémů mohou být identifikační či diagnostické úlohy. [12]

3.3.3 Dopředné řetězení

Dopředné řetězení (neboli forward chaining) je naproti tomu typem inference, v níž dedukce probíhá směrem od počátečních dat k závěrům. V této technice inferenční mechanismus recognition-act cyklem prochází pravidla a pro každé pravidlo testuje, zda se jeho premisy nachází mezi pravdivými tvrzeními v bázi dat. Pokud tomu tak je, konsekvent pravidla je přidán mezi pravdivá tvrzení. Vzhledem k tomu, že tento čerstvě odvozený závěr může být premisou v pravidle, které inferenční procedura již prošla, je nutné začít celý proces porovnávání znovu od prvního pravidla. Inference končí tehdy, kdy už nemůže být odvozen žádný nový závěr a zároveň jsme dosáhli konce seznamu pravidel. [25]

Problémy, které jsou vhodnými kandidáty na techniku forward chaining, se vyznačují tím, že u nich často není možné předem stanovit všechny možné závěry – to mohou být např. různé konfigurační či generativní problémy nebo úlohy, při nichž dohromady kombinujeme mnoho vstupů. [12, 14]

[26] Cílem je myšlen závěr, který může ES vrátit jako výsledek expertízy.

3.3.4 Rezoluce konfliktu

Nejzákladnějšími strategiemi rezoluce konfliktní množiny, na kterých jsou pak založeny další, vylepšující metody, jsou brute-force algoritmy prohledávání stavového prostoru – do hloubky (depth-first search) a do šířky (breadth-first search).

Depth-first search pracuje následovně: pro aktuální stav vybere z konfliktní množiny první pravidlo a ověří, zda jeho aplikací přejdeme do požadovaného cílového stavu. Jestliže ano, prohledávání tímto končí. Pokud následující stav není cílový, inferenční procedura jej nastaví jakožto aktuální stav a sestaví k němu jeho konfliktní množinu. Jestliže je neprázdná, opět z ní vybere první pravidlo a celý proces se opakuje. Pokud je ovšem prázdná, backtrackingem se vynoříme v předchozí úrovni stromu a na této úrovni pokračujeme aplikací následujícího pravidla. Značná nevýhoda zde spočívá v tom, že nevíme, v jaké hloubce se nachází řešení a ani neznáme celkovou hloubku stromu v některých větvích (může se tedy stát, že algoritmus vůbec neskončí). Tento inferenční přístup je implicitně využíván Prologem (jakožto prázdným ES). [13, 26]

Breadth-first search namísto toho prochází konfliktní množinu aktuálního stavu pravidlo po pravidlu a u každého testuje, zda jeho aplikace nevede do cílového stavu. Takto postupuje ve stromu úroveň po úrovni do větší hloubky, dokud nedosáhne cílového stavu. Výhodou tohoto přístupu je, že vzhledem k tomu, že procházíme vždy všechna pravidla na dané úrovni stromu, výsledná cesta do cílového stavu bude tím pádem ta nejkratší. Nevýhodou oproti prohledávání do hloubky je prostorová složitost tohoto přístupu, protože zatímco v depth-first search si stačí pamatovat pouze cestu z iniciálního do aktuálního uzlu (cesty přerušené backtrackingem mohou být zapomenuty), breadth-first search si pamatuje všechny vygenerované uzly až do aktuální hloubky. [26]

Depth-first iterative-deepening. Neduhy obou základních algoritmů řeší asymptoticky optimální (jak z hlediska času a paměti, tak i ceny řešení) brute-force algoritmus zvaný *depth-first iterative-deepening*. Jeho princip obecně spočívá v provedení depth-first search do hloubky i (počínaje hloubkou 1), přičemž pokud v této iteraci nenajde řešení, zapomene všechny uzly vygenerované v tomto prohledání a v další iteraci znovu provede depth-first search tentokrát do hloubky $i + 1$. Tak pokračuje, dokud není nalezen cílový stav. [26]

Jelikož jsme dosud zmiňovali metody používající hrubou sílu, intuitivně asi tušíme, že v praxi bude taková inference pro rozsáhlejší stavové prostory z hlediska času či prostoru značně neefektivní. V případě, že nemůžeme zaplatit tak vysokou cenu za nalezení nejlepšího řešení, obvykle přistoupíme k použití nějaké heuristické metody, díky které jsme schopni získat řešení s alespoň přijatelnou cenou. Zmíníme alespoň některé: *hill-climbing algoritmus*, *best-first search*, *algoritmus simulovaného žíhání* aj. To, co mají tyto algoritmy společné, je, že používají evaluační funkci, přiřazující uzlům ve stromu ohodnocení, které se pak dále používá k výběru cesty. [17]

Dosud popsané metody jsou ve skutečnosti velmi obecného charakteru – použitelné pro téměř libovolnou znalostní bázi (resp. bázi dat). Nicméně v praxi je možné tyto metody kombinovat i s vysoce specifickými znalostmi, „šitými na míru" pro rozhodování konfliktu mezi konkrétními pravidly ve znalostní bázi.

Mimo jiné zde pro architekta systému vyvstává dilema, zda tuto znalost, která bude rozhodovat konfliktní množinu, zabudovat přímo do inferenčního mechanismu (jako tzv. meta-znalost) nebo ji reprezentovat samostatnou „nadvrstvou" pravidel (tzv. meta-pravidel). [20]

Meta-pravidla často představují právě znalosti úzce vázané s konkrétními pravidly. Při inferenci tak zastávají úlohu jistých „subrutin", které obstarávají určitou podmnožinu pravidel znalostní báze a jsou aplikovány pokaždé, je-li inferenčním mechanismem tato podmnožina zpracovávána jako konfliktní množina. Argumentem pro jejich použití je dřívější výzkum [27], který poukázal na vhodnost zachování inferenčního mechanismu produkčního systému co nejjednoduššího, jelikož každá přidaná funkcionalita znamená režii navíc během každého recognition-act cyklu. Jinou výhodou je explicitnost takové znalosti, která je jednak jednoduše dostupná znalostnímu inženýrovi a stejně tak použitelná vysvětlovacím modulem. Jejich velká nevýhoda spočívá v tom, že je velmi těžké takovou sofistikovanou znalost získat, natož ověřit její správnost, proto je úspěšné použití meta-pravidel možné jen v dobře prozkoumaných doménách.

Meta-znalost zakomponovaná do inferenční procedury je naproti tomu podobně jako výše popsané algoritmy spíše uniformní prostředek, který ignoruje obsah pravidel v konfliktní množině. Rozhoduje se tedy na základě obecnějších rysů – např. evaluační funkce či počet premis v pravidlech. Z toho důvodu je prakticky použitelnější. Nicméně, přináší s sebou i opozitní nevýhody: nedostatek explicitnosti (a s tím ztížená modifikovatelnost či využitelnost při vysvětlování inference) a nedostatek flexibility při rozhodování pravidel, které vyžadují spíše specifický přístup. [20]

3.4 Neurčitost

Neurčitost informací je denním chlebem přirozeného světa. V problematice ES se můžeme setkat s mnoha různými typy neurčitosti. Ta se může vyskytovat již na úrovni vstupních dat systému nebo se může zakládat na nejednoznačnosti přirozeného jazyka – tyto dvě oblasti si v této kapitole popíšeme více do hloubky.

Dále se můžeme potýkat např. s neurčitostí vyskytující se při inferenci založené na neúplných datech, kde může být premisa určitého pravidla splněna pouze částečně, s čímž jde navíc ruku v ruce problém určení hranice *nezbytnosti* a *možnosti*[27] vztahující se k informacím, na základě kterých nějaké pravidlo je nebo již není použito. V neposlední řadě můžeme narazit na problémy s neurčitostí při snaze o agregaci znalostí z více různých zdrojů (např. od různých lidských

[27]pojmy z modální logiky

expertů) – některé znalosti si mohou protiřečit a vytvářet tak nejistotu založenou na nekonzistenci dat, některé mohou úplně chybět a způsobovat tak výše zmíněný problém neúplnosti dat, v jiném případě se nám do báze znalostí mohou vloudit redundantní pravidla, která zase mohou (jak později uvidíme) zkreslovat skutečnou váhu jistoty odvozených závěrů. [28]

3.4.1 Nespolehlivost informací

Zcela očekávatelně se můžeme setkat s neurčitostí již na úrovni znalostí, se kterými ES pracuje. Nejisté mohou být fakty, které do systému zadáváme jakožto množinu předpokladů; vstupní data jak od uživatele, který o pravdivosti svých tvrzení není zcela přesvědčen, tak i výstupy (které ES dále zpracovává jako vstupy) nějakého umělého zařízení, které může produkovat různé chyby měření. Ve znalostní bázi ES můžeme dále narazit na neurčitost ve formě slabé korelace mezi premisou a závěrem pravidla, plynoucí z reálného stavu valné většiny lidských znalostí[28]. Tento typ nejistoty je běžně řešen pomocí tzv. *faktorů nejistoty*, vyjádřených číselnou hodnotou (často z intervalu $[0,1]$, ale nemusí to být pravidlem), které jsou připojeny k danému pravidlu. Lidskému expertovi je tak do rukou vložen jednoduchý nástroj, kterým může daleko realističtěji reprezentovat své vědomosti. Pro člověka ještě přirozenější způsob ovšem může být užití *fuzzy kvantifikátorů*. V takovém případě nám pro vyjádření neurčitosti slouží zájmena jako „některé“, „většina“, „téměř všechno“, která naproti umělé škále číselných hodnot daleko lépe zastupují lidskou úroveň rozlišovací schopnosti stupňů nejistoty. Takto kvantifikované pravidlo by mohlo být slovy formulované např. takto:

JESTLIŽE je *téměř celá* obloha pokryta černými mraky,
PAK na *některých* místech vypukne bouřka.

Zmíněné faktory nejistoty mohou být pochopitelně použity i pro ohodnocování faktů. V takovém případě pak v procesu inference dochází (souběžně s unifikací faktu a premisy pravidla) ke kombinaci faktorů nejistoty faktu a celého pravidla. Takto kombinovaná neurčitost se pak propaguje do odvozeného závěru. [28]

V obecnějších případech se musíme potýkat např. s následujícími problémy:

1. jak agregovat faktory nejistoty více faktů, má-li dojít k unifikaci s premisou pravidla ve tvaru konjunkce;

2. jak naložit se situací, kdy více pravidel vede k odvození stejného závěru s různými faktory nejistoty.

Je třeba říci, že neexistuje uniformní způsob, jak se vyrovnat s podobnými problémy, způsobenými neurčitostí v datech. Teoretických přístupů je mnoho. Mohou být založeny na fuzzy logice, teorii možností[29], teorii pravděpodobnosti,

[28] I Sokrates o tom věděl své, když prohlásil: „Vím, že nic nevím.“ Polemizovat by se dalo nad tím, s jakou jistotou to skutečně věděl.

[29] tj. *Possibility theory* představená L. Zadehem

teorii influenčních diagramů aj. [9] Dominujícím faktorem pro výběr některého z přístupů často bývá pouhá intuitivnost – preferovaná je ta metoda, která bude dávat takové výsledky, které by pro experta byly v realitě očekávatelné.

Podívejme se tedy alespoň pro ukázku, jak by se dalo co nejjednodušeji přistoupit k řešení dvou výše zmíněných problémů.

V 1. případě se situace dá zvládnout poměrně intuitivně (dle [12]). Máme-li pravidlo ve tvaru

$$\{A_1 \wedge A_2 \wedge \cdots \wedge A_n\} \Longrightarrow B,$$

kde premisa s participanty A_i je ve tvaru konjunkce a v bázi dat jsou přítomny fakty s jejich faktory nejistoty (zde reprezentované pomocí uspořádaných dvojic)

$$(A_1, c_1), (A_2, c_2), \ldots, (A_n, c_n),$$

výsledný faktor nejistoty c premisy položíme jako

$$c = \min\{c_1, \ldots, c_n\}.$$

Je-li pro splnění premisy nutná „účast“ všech faktů A_i, pak je pochopitelně ten „nejslabší článek“ právě tím, který nastavuje celkovou míru nejistoty.

2. situace připouští existenci několika pravidel s faktory nejistoty b_i (které vznikly kombinací faktoru nejistoty faktu A_i s faktorem příslušného pravidla, vyjadřujícím sílu vazby implikace), opět reprezentovaných dvojicemi

$$\begin{aligned} &(A_1 \Longrightarrow B, b_1),\\ &(A_2 \Longrightarrow B, b_2),\\ &\quad\vdots\\ &(A_n \Longrightarrow B, b_n). \end{aligned}$$

Jednou z možností určení kombinovaného faktoru b z uvedených faktorů b_i zvolit opačnou strategii – tedy položit

$$b = \max\{b_1, \ldots, b_n\},$$

jelikož se výše uvedená množina pravidel dá přepsat pomocí logického operátoru OR (premisa složená z A_i by tedy byla ve tvaru disjunkce). Jiný, pokročilejší přístup pak navrhuje opět [12], který byl původně implementován v systému Mycin. Jelikož ke zpracování pravidel dochází jedno po druhém, vycházíme z předpokladu, že první závěr s faktorem b_1 je již odvozen (a uložen v bázi dat jako nový fakt). K jeho kombinaci s dalším faktorem, tj. b_2, pak dojde dle následující kombinační funkce [29], kterou si označíme jako CF:

$$CF(b_1, b_2) = \begin{cases} b_1 + b_2 - b_1 b_2 & b_1, b_2 > 0,\\ b_1 + b_2 + b_1 b_2 & b_1, b_2 < 0,\\ \frac{b_1 + b_2}{1 - \min\{|b_1|, |b_2|\}} & \text{jinak.} \end{cases}$$

Definičním oborem této funkce je interval $[-1, 1]$. Výhoda tohoto přístupu je nejen to, že do výsledného faktoru nejistoty b přispívají všechny parciální faktory b_i, ale i to, že umožňuje existenci faktů se zápornými faktory, které tak mohou působit proti jiným faktům a zkombinovat tak mnohem komplexnější výsledek. Nejlepší ilustrací může být např. diagnostika nějaké nemoci. Mohou existovat jak příznaky, které přispívají k pozitivní diagnóze, tak příznaky, které se s danou nemocí mohou zcela vylučovat – tuto skutečnost můžeme pak reprezentovat právě zápornými faktory nejistoty.

3.4.2 Nepřesnost jazyka

Další typ neurčitosti je dán inherentní nepřesností jazyka, v němž vyjadřujeme informace, potažmo pravidla znalostní báze. Důsledkem toho pak při převodu znalostí do formálního jazyka pro vyjadřování pravidel dochází ke ztrátě nebo misinterpretaci znalostí. V praxi pak nedostačuje unifikace faktů s premisami pouze s pomocí lexikální analýzy, nýbrž je potřeba při každé potenciální unifikaci zkoumat alespoň přibližné významy faktů a premis.

Za běžných okolností tedy musí být fakt přesně ve tvaru A, aby došlo k lexikální unifikaci s premisou pravidla $A \implies B$. V konkrétním případě by tedy např. fakt

„baterie je *téměř* vybitá“

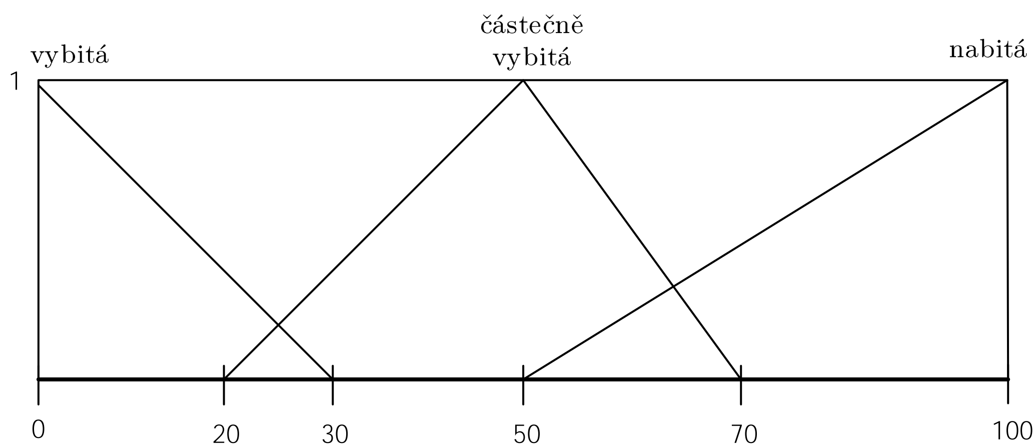
selhal v pokusu o unifikaci s premisou pravidla

„baterie je vybitá“ $\implies$ „LED se nerozsvítí“,

ačkoliv sémanticky by *téměř* vybitá baterie (z důvodu příliš slabého napětí) implikovala tentýž důsledek.

Prostředky pro to, jak se s takovým problémem alespoň částečně vyrovnat, nabízí např. tzv. *generalizované modus ponens*, které je definováno na fuzzy produkčních pravidlech. Pomocí těch je pak popsáno jen pár relevantních případů – tedy nemusíme ošetřovat každou jednotlivou nuanci úrovně nabití baterie pomocí samostatného pravidla, ale můžeme mít např. jen tři pravidla s premisami „baterie je **nabitá**“, „baterie je **částečně nabitá**“ a „baterie je **vybitá**“[30]. Tučně zvýrazněná adjektiva, popisující úroveň nabití, jsou formálně reprezentována fuzzy množinami (které se v tomto případě překrývají). Generalizované modus ponens pak představuje schéma inference, které umožňuje unifikaci přibližných faktů jako např. „baterie je z 98 % nabitá“. Tento konkrétní fakt, který vyjadřuje přibližně úplné nabití baterie, by pak „padl“ do uvedených fuzzy množin s různými hodnotami tzv. *funkce příslušnosti*.

[30] Toto rozdělení na relevantní případy je určeno především tím, co z daných případů chceme v pravidle vyvozovat.



Obrázek 5: Funkce příslušnosti

Na obrázku 5 vidíme jednu z možných podob grafu funkce příslušnosti k našemu příkladu s baterií. Horizontální osa slouží jako škála skutečných hodnot nabití baterie (dejme tomu, že v procentech). Uvedených 98 % by tedy spadalo do tohoto rozmezí. Hodnoty z této osy se pak projektují do jedné (nebo i více) ze tří fuzzy množin se *stupněm příslušnosti*, který je dán hodnotami na vertikální ose.

4 Grafologický expertní systém

4.1 Grafologie

Grafologie je disciplínou z oblasti psychodiagnostických metod, konkrétně vycházející z psychologie výrazu. Koncepce grafologie se zakládá na poměrně intuitivní premise, že existuje korelace mezi výrazovými projevy člověka (ať už jde o gestikulaci, mimiku tváře, postoj těla nebo právě psaní, jakožto zautomatizovaný pohyb řízený centrální nervovou soustavou) a určitými individuálními psychologickými charakteristikami osoby.

Z hlediska vývoje grafologie můžeme pozorovat dva základní směry přístupu. *Fragmentální* (či *empirický*) přístup, v němž jde především o analýzu dílčích znaků písma (např. velikost, sklon, tlak, ...) a pohled na tyto znaky jakožto nositele významu. V opozici vůči tomuto hledisku stojí tzv. *celostní* (či *holistická*) grafologie, která se snaží písmový obraz zachytit spíše jako celek a posoudit jej i z dojmového hlediska, přičemž se nutně neopírá o exaktní metodologii. [30]

Stěžejním bodem, kolem něhož se točí i mnoho výhrad ke grafologii jakožto seriózní empirické disciplíně, je především postižitelnost pravidel, podle nichž dochází k projekci určitých osobnostních rysů do písma. Tento problém z velké části vychází z absence pevného statistického základu pro grafologii. To nám obecně vnáší do dat nejistotu, se kterou jsem se dále musela potýkat při tvorbě expertního systému.

4.2 Role grafologa

Stojí za to také zmínit roli grafologa v celém procesu vytváření grafologického posudku, a to za účelem diferenciace úloh v grafologické expertíze, které jsou zpracovatelné algoritmicky, od úloh, v nichž je role lidského experta nezastupitelná. Cílem grafologovy práce je na základě různých metod rozboru písma (které v grafologii nejsou jednotné) analyticky zhodnotit rukopis klienta a z obsažených charakteristik písma (což je jak celkový výrazový obraz písma, tak dílčí, více či méně měřitelné znaky) extrahovat a vhodně popsat určitý psychologický profil pisatele.

Jan Jeřábek, autor knihy[31], která byla mým primárním zdrojem pro znalostní bázi systému, varuje (za účelem kritiky krajně empirického přístupu) před nebezpečím fragmentálního uvažování nad písmem, které vede k vytrhování jednotlivých znaků v písmu a potažmo jejich významů z kontextu celého písmového obrazu. [31] Tím ovšem vystihl základní překážku, které musí čelit každý počítačový program s ambicemi simulace lidského rozhodování – a sice absenci intuice a nadhledu na problémem, která by mu umožňovala „spatřit" skutečnosti nemožné zcela postihnout přesně předepsanými zákonitostmi. V případě grafologie to platí dvojnásob, jelikož vedle faktických znalostí o písmu jsou (dle [30, 31]) kladeny vysoké nároky právě na osobnost grafologa, na úroveň jeho sebepoznání

[31] Grafologie, více než diagnostika osobnosti

a schopnost empatie a dojmové intuice při kontaktu s písmem. Musí být totiž schopen analyzovat písmo jako celek a zároveň umět odstínit svůj subjektivní vztah k písmu, jelikož některé písmové znaky jsou minimálně náchylné k nejednoznačným interpretacím.

4.3 Charakteristika grafologického expertního systému

Hlavní idea tvorby grafologického expertního systému by měla především přispět ke „smíření“ tenze mezi oněmi dvěma protikladnými přístupy ke grafologii – čistě fragmentálním a čistě dojmovým. Na jedné straně by tedy takový systém měl nahradit ty oblasti grafologovy práce, v nichž takzvaně hrozí „selhání lidského faktoru“ ať už z důvodu omezenosti lidské paměti, přehlédnutí některých faktorů důležitých pro realističnost grafologického posudku (tzn. grafolog může při analýze zapomenout zakomponovat nějaký důležitý znak písma) nebo z důvodu neschopnosti za všech okolností uvažovat zcela stejně (tedy v různých situacích by mohl grafolog dospět k různým závěrům, zvláště opírá-li se i o dojmové úsudky).

V těchto nedostatcích by měl systém grafologovi posloužit jako persistentní databáze znalostí, nástroj, v němž může ve strukturované podobě formulovat vlastní pravidla plynoucí z nabitých teoretických znalostí a vlastních zkušeností a v neposlední řadě stabilní mechanismus odvozování nových znalostí na základě vložených pravidel. Naproti tomu by měl expertní systém ponechat grafologovi volnost v těch oblastech, v nichž je pro interpretaci informací důležitý lidský vhled a intuice. Z hlediska tohoto kritéria by měly být výsledky vyhodnocené systémem ještě stále dostatečně transparentní a „syrové“ na to, aby si o nich mohl grafolog udělat vlastní úsudek a nespoléhal pouze na „skryté mechanismy programu“. Do toho se řadí i požadavek, aby byl systém schopen vyhodnocovací proces vysvětlit.

5 Architektura systému

V této kapitole bude popsán návrh grafologického expertního systému GESto[32]. Text také částečně slouží jako uživatelská příručka.

V rámci dodržení kritéria transparentnosti uchovávaných a odvozovaných znalostí z pohledu grafologa jsem systém navrhla tak, aby grafolog mohl být zároveň uživatelem i znalostním inženýrem. V systému tedy můžeme rozlišit dvě základní komponenty:

- subsystém pro tvorbu a správu znalostních bází,
- modul pro provádění expertízy (tj. načtení konkrétní znalostní báze a usuzování nad jejími znalostmi).

5.1 Typy znalostí

Uveďme si zde přehled různých znalostí, se kterými je možné se v rámci grafologického expertního systému setkat, a na které bude odkazováno v dalším textu.

5.1.1 Grafologické znalosti

V prvé řadě zde máme znalosti specifické pro úlohu řešenou expertním systémem, které lze jednoduše shrnout jako znalosti týkající se písma a znalosti týkající se pisatele. Blíže jsou to:

- písmové znaky – nejjednodušší typ znalostí týkajících se písma. Mají obvykle podobu jednoslovného přívlastku (ale může být použito slov i více), popisujícího písmo či nějaký jeho aspekt – např. *elastické*, *beztvaré*, *obroušené a oblé tvary*, *pečlivé vypracování tvarů písmen* aj. Jejich úlohou je doplňovat popis písma v tom, co nebylo postihnuto základním popisným nástrojem – tzv. *parametry písma*.
- parametry písma – jsou oproti písmovým znakům komplexnější v tom, že pojmenovávají nějaký aspekt písma, u něhož lze zároveň rozlišit vícero různých hodnot (které mohou být popisného či kvantitativního charakteru). Příkladem může být parametr *Vázání* s hodnotami *úhel*, *girlanda*, *arkáda*, *nitka*, *dvojoblouk*; nebo parametr *Sklon* s hodnotami *levý*, *stojatý*, *pravý*. Blíže budou parametry písma definovány v kontextu tvorby znalostní báze (kapitola 5.2.3) a v kontextu provádění expertízy (kapitola 5.2.4).
- osobnostní charakteristiky – výrazy jakkoliv popisující osobnost pisatele, jež lze dále rozdělit do různých psychologických oblastí (viz dále). Obvykle jsou svázány k konkrétními hodnotami parametrů, tedy stojí v roli jejich *významů*.

[32]Zkratka pro Graphological Expert System. Slovo gesto má odkazovat na fakt, že grafologie vychází z psychologie výrazu – psaný projev je z tohoto pohledu kolekce gest.

5.1.2 Pravidla

Pravidla lze v systému vytvářet kombinací tří základních relací:

1. *v písmu je parametr P s hodnotou H*,
2. *v písmu je písmový znak Z*,
3. *pisatel má osobnostní charakteristiku Ch*

Z těchto relací může znalostní inženýr poskládat tzv. *typ pravidla*, který je určitou „šablonou“ pro tvorbu konkrétních pravidel (s již konkrétními písmovými znaky či psychologickými charakteristikami). Typ pravidla se skládá z premisy, která může obsahovat libovolný počet výše uvedených relací spojených konjunkcí, a závěru, který obsahuje jen jedinou relaci. Je to z toho důvodu, že implikuje-li stejná premisa více různých závěrů (tj. tvrzení v závěru by byla v konjunkci), lze totéž vyjádřit více pravidly se stejnou premisou.

Dále pravidla řadíme do dvou základních druhů:

- interní – nejzákladnější typ pravidel, která si systém tvoří automaticky v průběhu definice nových písmových parametrů (viz 5.2.3). Interní pravidlo může být z hlediska použitých relací pouze dvojího typu.

  Jeho závěr se týká buď pisatelovy osobnosti

  JESTLIŽE *v písmu je parametr P s hodnotou H*,
PAK *pisatel má osobnostní charakteristiku Ch*,

  nebo se opět týká charakteristiky písma

  JESTLIŽE *v písmu je parametr P s hodnotou H*,
PAK *v písmu je písmový znak Z.*

- uživatelská – definuje je uživatel a na rozdíl od interních pravidel mohou být vytvořena použitím libovolného *typu pravidel*, tedy i toho, který má premisu ve tvaru konjunkce. Jejich účelem je doplňovat základní znalost obsaženou v interních pravidlech a jsou tedy vhodná pro uchovávání grafologových heuristik.

Důležitou součástí pravidel je také *faktor jistoty*, což je reálné číslo z intervalu $[0, 100]$ představující jistotu, s jakou platí závěr pravidla, či přesněji řečeno, vyjadřující sílu implikace mezi premisou a závěrem v pravidle.

5.1.3 Klasifikační znalosti

Posledním druhem znalostí jsou ty, které mají předchozí znalosti (zvláště ty grafologické) nějakým způsobem sdružovat. Jsou jimi:

- kategorie – slouží ke klasifikaci písmových parametrů,
- psychologické oblasti – představují různá hlediska, z jakých se lze dívat na osobnost člověka. Může to být např. *temperament*, *emocionalita*, *sociabilita*, *morálka* aj. Slouží k zařazení osobnostních charakteristik.

5.2 Uživatelské rozhraní

Jako uživatelské rozhraní jsem zvolila textový interaktivní shell, jehož základní ideou je klást uživateli otázky, případně nabízet možnosti k výběru a očekávat odpověď.

5.2.1 Globální a lokální příkazy

V každém jednotlivém stavu programu je vypsána otázka či menu a následně symbol **|:**, za nímž je očekávána odpověď na otázku či číslo volby z dané nabídky. Mimoto shell zčásti funguje i jako příkazový řádek, protože kromě standardní odpovědi na vypsanou otázku či menu může uživatel zadat jeden z globálních či lokální příkazů, umožňuje-li to daný kontext. Na tyto příkazy jsou navázány akce, které se ihned provedou bez ohledu na to, v rámci jaké otázky či menu byly vyvolány. Po jejich provedení program automaticky očekává odpověď na původní otázku nebo položí novou.

Globální příkazy fungují až na výjimečné situace kdekoliv v programu. Jsou-li někde zakázány, je to spíše z bezpečnostních důvodů. Popišme si je nyní blíže.

- **back/b** – slouží k návratu uživatele většinou na předchozí dialog. Není-li přímo předchozí, znamená to, že akce spojená s přímým předchozím dialogem byla již dokončena (může to být např. výběr znalostní báze či rozpracované expertízy). Případ, kdy tento příkaz nefunguje, je kupříkladu moment, kdy by jeho vykonání vedlo k narušení integrity dat ve znalostní bázi.
- **main/m** – funguje v podstatě jako vícenásobný back, přičemž uživatele z aktuálního místa přesune zpět na hlavní dialog. Zakázán je právě tehdy, když je zakázán i back.
- **help/help <argument>** – prostý help bez argumentu vyvolá nápovědu, která obsahuje dodatečné či vysvětlující informace k aktuální otázce nebo nabídce. Help s argumentem je těsně vázán spíše ke konkrétním menu, v nichž je na výběr více očíslovaných voleb. Zadáním help a daného čísla se pak zobrazí nápověda k dané volbě.

- **?** – tento příkaz opět vypíše aktuální otázku, kterou program uživateli pokládá. Jeho použití je vhodné v případě, kdy si uživatel kupříkladu vyvolal obsáhlejší nápovědu, po jejímž vypsání by mohlo být snadné ztratit přehled o tom, na co se program ptal původně.
- **info** – vypíše úvodní informace, které se zobrazují ihned po spuštění programu.

Lokální příkazy naproti tomu fungují pouze na určitých místech programu, ačkoliv princip je zcela stejný jako u globálních. Zde si uvedeme pouze jediný, ostatní jsou použitelné v rámci provádění expertízy, proto si je blíže rozebereme v příslušné podkapitole 5.2.4.

- **print** – slouží k vypsání (obecně řečeno) určitého stavu pracovní paměti, což může být např. opětovné vytisknutí odvozených závěrů či seznamu znalostních bází, které jsou k dispozici. Informace o tom, zda je příkaz v aktuálním místě programu dostupný, je uvedená v nápovědě na daném místě.

5.2.2 Struktura hlavního menu

Po spuštění programu se ocitneme v hlavním menu, jehož možnosti jsou seskupeny dle role, již grafolog momentálně zastává. První tři možnosti menu slouží grafologovi v roli uživatele, zatímco další tři může využívat jakožto znalostní inženýr.

```
Co chcete udělat?
(1) provést novou expertízu
(2) načíst rozpracovanou expertízu
(3) smazat rozpracovanou expertízu

(4) vytvořit novou znalostní bázi
(5) upravit existující znalostní bázi
(6) smazat znalostní bázi

(s) nastavení
(e) konec
|:
```

Ve volbě `(1)` dostane uživatel k výběru seznam existujících znalostních bází. Zvolená se načte a dále se bude používat v průběhu expertízy jako databáze pravidel (příp. dalších faktů) pro odvozování závěrů. Dále budou od uživatele získány informace identifikující aktuální expertízu – jméno pisatele, název expertízy a tzv. *hranice ignorance*, což je reálná hodnota z intervalu $[0, 100]$, která stanovuje mez, pod níž se budou všechny odvozené závěry s faktorem jistoty

menším než tato hodnota ignorovat. Samotný proces vykonávání expertízy bude dále popsán v podkapitole tomu dedikované.

Volba `(2)` funguje obdobně až na to, že uživatel volí ze seznamu již rozpracovaných expertíz, které byly v průběhu uloženy. Tím pádem na začátku již nedochází ke sběru úvodních informací. Změnila-li se někdy v čase znalostní báze dané uložené expertízy (čímž např. hrozí, že uložené odvozené závěry budou neaktuální), tato expertíza i přes to zůstává zachována, nicméně uživatel dostane doporučení k jejímu smazání.

Odstraňování expertíz v možnosti `(3)` (resp. znalostních bází v `(6)`) je možné provádět hromadně výběrem posloupnosti čísel přiřazeným k jednotlivým položkám. V případě odstraňování rozpracovaných expertíz je navíc možno před definitivním smazáním uložit výsledky dané expertízy do souboru zadáním příkazu **save <číslo expertízy>**. Umístění souboru je dáno cestou, kterou lze definovat v nastavení (viz 5.2.5).

Možnosti `(4)` a `(5)` uživatele přesměrují na menu tvorby znalostní báze, kterému bude podrobněji věnována následující podkapitola 5.2.3. Rozdíly těchto voleb spočívají v tom, že v prvním případě dostane uživatel za úkol zadat unikátní název nové znalostní báze a následně je přesměrován na omezenou verzi zmíněného menu, v němž je možné pouze definovat nové prvky. Oproti tomu ve druhém případě dostane na výběr z existujících znalostních bází a dle definovaných prvků ve zvolené bázi se ukáže rozšířené menu, v němž je možná jejich editace, odstraňování či jen prostý výpis.

5.2.3 Tvorba znalostní báze

Z uživatelského pohledu je v případě tvorby znalostní báze nejzajímavější její definice, proto si v této části popíšeme především tuto funkcionalitu.

Základní menu pouze pro definici znalostní báze vypadá následovně:

```
DEFINOVAT:
(1) nový parametr písma
(2) nové pravidlo
(3) nový typ pravidla
(4) nápovědu k parametru písma
|:
```

V první možnosti tvoříme nový parametr písma, který je v prvé řadě identifikován svým jménem. Zadáme-li existující jméno, dialog nás přesměruje na možnost editovat parametr s tímto jménem. Dále pro každý parametr vybíráme kategorii a typ parametru, který může být buď *popisný* nebo *kvantitativní*. Podle tohoto typu se nám tvorba konkrétního parametru může vydat poněkud odlišnými cestami.

Popisné parametry očekávají definici hodnot popisného (slovního) charakteru, kterých může být libovolný počet. Ke každé hodnotě může být dále defino-

ván libovolný počet významů, které jsou s konkrétní hodnotou svázány faktorem jistoty, který (pro připomenutí) vyjadřuje sílu implikace v tvrzení: „*Je-li v písmu parametr P s hodnotou H, pak platí význam V.*“ Mimoto rozlišujeme dva typy významu dle toho, zda se týkají pisatele či písma, neboť je počítáno s možností, že jistý parametr písma může značit přítomnost jiného písmového znaku. Význam, který je charakteristikou pisatele, navíc můžeme přiřadit do některé z psychologických oblastí (viz 5.1.3), které si opět může definovat znalostní inženýr.

Typickým příkladem popisného parametru je např. *Písmový obraz* s hodnotami *vyvážený*, *neuspořádaný*, *hustý*, *řídký*, *strnulý* a *neživotný*.

Kvantitativní parametry lze dále specifikovat buď jako *objektivní* anebo *poměrné parametry*. Od popisných parametrů se nikterak neliší procesem přiřazování významů k jejich hodnotám (příp. další specifikací a klasifikací těchto významů, jak jsme již uváděli), proto v tom se zde opakovat nebudeme. Podívejme se nejdříve na jednodušší, poměrné parametry.

Poměrné parametry se vyznačují tím, že u nich lze definovat pouze dvojice (slovy popsaných) hodnot, obvykle s protikladným významem, které lze v písmu zachytit v určitém poměru.

Příkladem poměrného parametru může být *Plnost-hubenost*, jehož hodnoty jsou přítomny přímo v názvu – tedy *plnost* a *hubenost*.

Objektivní parametry mají představovat skupinu měřitelných, a proto z objektivního hlediska nejlépe zachytitelných, znaků v písmu[33]. Jelikož měříme vždy v daných jednotkách (ať už jsou to jednotky úhlu či vzdálenosti), nejprve znalostní inženýr definuje jednotku. Dále systém očekává zadání libovolného[34] počtu diskrétních (opět slovy pojmenovaných) hodnot, k nimž jsou zároveň jednoznačně přiřazeny intervaly hodnot v zadané číselné jednotce. Tyto intervaly se mohou libovolně překrývat, ale jejich sjednocení musí tvořit jeden spojitý interval – definiční obor. Důvod zavedení konečného počtu hodnot (resp. intervalů) v případě parametru písma s nekonečným počtem skutečných (číselných) hodnot je tento: reálně není možné pro každou hodnotu z nekonečné množiny zavádět vlastní význam, proto skutečným nositelem významů jsou spíše určití reprezentanti na této nekonečné množině – pojmenované intervaly.

Zároveň není žádoucí (jelikož by to neodpovídalo realitě) skokový přechod od jednoho významu ke druhému (plynoucí z přechodu mezi dvěma definovanými intervaly) jen na základě nepatrné změny (číselné) hodnoty z definičního oboru. Při řešení tohoto problému jsem se inspirovala *teorií fuzzy množin.* Z hlediska tohoto teoretického základu můžeme objektivní parametry písma interpretovat jako tzv. *lingvistické proměnné* a hodnoty tohoto parametru (v podobě intervalů) jako *fuzzy množiny.* [32] Na těchto fuzzy množinách je dále definovaná *funkce příslušnosti*, která v klasickém významu přiřazuje hodnotám z definičního oboru (tj. hodnoty parametrů určované v jednotkách) *stupně příslušnosti* k jedné či více fuzzy množinám (viz také kapitola 3.4.2).

[33]Z hlediska teorie se zde dostáváme do krajně empirického přístupu – viz 4.1.
[34]pochopitelně konečného

Pro potřeby našeho problému jsem nicméně význam funkce příslušnosti lehce pozměnila. Znalost stupně příslušnosti (v obvyklém slova smyslu) pro nás není z hlediska funkcionality zas tak důležitá; mnohem podstatnější je přiřazení faktoru jistoty k významům korespondujícím s danými diskrétními hodnotami/intervaly (resp. fuzzy množinami). K tomu tedy bude sloužit i pozměněná (teď již myšleno nejen z hlediska významu, ale i její konečné podoby) funkce příslušnosti[35], jejíž konkrétní definice bude blíže rozebrána v kapitole 5.3.1.

Dle konkrétní podoby definovaných intervalů, může systém dále uživatele vyzvat k zadání dodatečných atributů, které později slouží k definici naší „funkce příslušnosti“. Ty jsou následující (všechno jsou to hodnoty z daného definičního oboru):

- **střední hodnota** – hodnota, jež by z hlediska statistiky vyjadřovala vážený průměr hodnot (v tomto případě) spojité náhodné veličiny, kterou je daný objektivní písmový znak. Jelikož se ale v tomto problému nemůžeme striktně opírat o statistická data (viz důvody v 4.1), tato hodnota představuje podobný koncept spíše intuitivně a její zadání je tedy na uživateli (nezadá-li ji, nastaví se automaticky jako polovina definičního oboru). Např. pro parametr Velikost písma to může být mezi pisateli nejběžněji vyskytující se velikost, která je uvedena v knize [31] jako hodnota z rozmezí 2–3 mm. Praktický smysl této hodnoty navíc spočívá v tom, že „funkce příslušnosti“ v této hodnotě nabývá svého maxima.

- **minimální hranice normality** – hodnota ostře menší než střední hodnota, která se dá při výskytu v písmu ještě považovat za normální. Objasnění smyslu si proveďme kupříkladu opět na parametru Velikost písma. Jeho definičním oborem může být např. interval $[0, 12]$ (v mm). Jeho minimální hranice normality pak může být třeba 1 mm. Pod touto hranicí je sice stále ještě možné definovat hodnoty (až do nuly), nicméně písmo s tak malou velikostí by již bylo vysoce netradiční, ba raritní. Praktické dopady to má rovněž na „funkci příslušnosti“, protože ta je navržená tak, aby zleva vzhledem k této hodnotě měla rostoucí charakter. Podíváme-li se na to opačně, z hlediska reality je žádoucí, aby faktor jistoty spjatý s významem příslušejícím k intervalu, který minimální hranici normality obsahuje (zpravidla levý krajní), postupně klesal společně s tím, jak se hodnoty definičního oboru blíží od hranice normality až k nule (či jinému levému krajnímu bodu v obecném případě). Společně se zvyšující se „nenormalitou“ písma se totiž musí snižovat i pravděpodobnost, že skutečně platí význam spjatý s daným (nejlevějším krajním) intervalem konkrétního písmového parametru.

- **maximální hranice normality** – hodnota ostře větší než střední hodnota, jejíž význam je obdobný jako u minimální hranice normality (pochopitelně bráno z opačného pohledu).

[35] Z toho důvodu ji dále budu uvádět pouze v uvozovkách a budu mít vždy na mysli pozměněnou funkci příslušnosti.

Jako typický příklad objektivního parametru zde můžeme uvést *Sklon* s hodnotami *levý*, *stojatý* a *pravý*.

Při tvorbě nového pravidla (volba `(2)`) je možné volit z existujících typů pravidel nebo lze na místě vytvořit nový. Tento typ pak „vyplňujeme“ příslušnými objekty (parametry písma, písmovými znaky nebo vlastnostmi pisatele) a tak vznikne konkrétní pravidlo. Parametry písma (díky svému výjimečnému postavení mezi ostatními znalostmi) musí být vždy voleny z existujících. Mimoto systém zabrání tvorbě pravidla, vyhodnotí-li, že neexistuje dostatečný počet různých hodnot různých parametrů tak, aby se ve vytvářeném pravidle neopakovaly. (Stejně tak je obecně zabráněno tvorbě duplicitních pravidel.) Ostatní objekty mohou i nemusí být též vybírány z existujících.

Budování typů pravidel (ve volbě `(3)`) pak probíhá výběrem relací z kapitoly 5.1.2 do premisy a následně do závěru, takže se ve výsledku uživateli zobrazí kupříkladu taková struktura:

```
IF v písmu je parametr P s hodnotou H (2× AND),
   v písmu je znak Z (3× AND),
   THEN pisatel má vlastnost V.
```

Vyskytuje-li se jedna relace v konjunkci vícekrát, je to těsně za ní pro stručnost značeno příslušným počtem AND v závorce. Systém též brání přidávání strukturálně stejných typů pravidel.

Poslední možností z menu je definice nápověd k parametrům písma. Uživatel v ní dostane na výběr, zda chce nápovědu definovat přímo k parametru písma nebo k hodnotě *popisného* parametru. U ostatních – kvantitativních parametrů – to není umožněno z toho důvodu, že jejich hodnoty, ač slovy popsané, obvykle označují nějakou kvantitu (např. levý, pravý, malé, střední, ...), tím pádem samy o sobě další vysvětlení nepotřebují.

5.2.4 Provádění expertízy

Prováděním expertízy je v kontextu našeho programu myšlen proces získávání dat týkajících se písma od uživatele souběžně s kombinací těchto dat se znalostmi ze zvolené znalostní báze. Cílem tohoto procesu je odvození nových znalostí neboli závěrů.

Po založení nové expertízy (ve volbě `(1)` hlavního menu) či načtení existující (ve volbě `(2)`) je možné volit ze dvou režimů, v nichž může být expertíza vykonávána. První z nich je režim „Krok za krokem“, v němž uživatel prochází postupně všechny parametry písma jeden po druhém, zadává systému data, případně se může vracet zpět k parametrům, jejichž hodnoty již definoval a měnit je.

Ve druhém, tzv. výběrovém módu, si naproti tomu uživatel může parametry písma či další písmové znaky volit nesouvisle a nemusí tak procházet všechny.

Pro lepší přehled jsou v tomto režimu písmové parametry rozděleny dle svých kategorií. Odlišení již zpracovaných je realizováno pomocí barevného zdůraznění.

Co se týče ovládání průběhu expertízy, máme zde opět množinu (tentokráte lokálních) příkazů.

- **eval** – vede k okamžitému vyhodnocení expertízy a výpisu výsledků.
- **end** – tento příkaz ihned ukončí probíhající expertízu, přesune uživatele na hlavní dialog a ve vedlejším efektu dojde k uložení aktuální expertízy mezi rozpracované.
- **del <argument>** – při výběru popisných parametrů (jejichž hodnoty lze vybírat ze seznamu) je možné tímto příkazem přesunout již definované hodnoty zpět do seznamu nedefinovaných.
- **next/n** – realizuje přechod od jednoho parametru k následujícímu v režimu „Krok za krokem“ a v režimu výběrovém uživatele vrátí na výběr jiného parametru.

Kromě těchto příkazů jsou zde funkční i globální příkazy, které jsme si již představili, ovšem hrají zde roli některé aspekty, které stojí za to zmínit zvlášť.

- **back/b** – v režimu „Krok za krokem“ funguje jako protějšek příkazu next, tedy pro návrat k předchozímu písmovému parametru. Ve výběrovém režimu je pak jeho funkce stejná jako u next.
- **help/help <číslo>** – od běžného použití příkazu help kdekoliv v programu se liší tím, že vyvolává nápovědy (existují-li) přímo k parametrům písma nebo jejich hodnotám, které měl možnost během tvorby expertízy definovat sám grafolog.

Představme si zde ještě samotné zadávání dat expertízy – a to především z hlediska představených typů písmových parametrů.

Popisné parametry se vyznačují tím, že při svém zpracování očekávají hodnoty popsané slovně, které uživatel může volit přímo ze seznamu. Výlučně u tohoto typu parametrů je také možno zobrazit nápovědu přímo k samotným hodnotám.

Poměrné parametry jako svůj vstup berou číselnou hodnotu – konkrétně reálnou hodnotu z intervalu $[0, 100]$, která vyjadřuje, nakolik jedna z hodnot tohoto parametru převažuje v poměru vůči té druhé (zpravidla v určitém ohledu protikladné hodnotě). Zadá-li uživatel hodnotu 50, znamená to, že jsou obě hodnoty v písmu přítomny vyváženě.

Objektivní parametry jako svůj vstup očekávají číselnou hodnotu v daných jednotkách a daném rozsahu, přičemž obojí je stanovené znalostním inženýrem (viz 5.2.3). Ze zadaného čísla dále systém určí jednu nebo více hodnot tohoto

parametru. Tyto určené hodnoty jsou naproti tomu vyjádřeny slovně (rovněž dle definice ve znalostní bázi) a je k nim navíc přiřazen faktor jistoty, opět interně vypočítaný (mechanismus těchto výpočtů je blíže popsán v kapitole 5.3.1).

Zadá-li navíc uživatel u objektivního parametru, který již navštívil a zpracoval, příkaz **print**, vypíše se mu funkce, dle níž se určily faktory jistoty u vypočítaných hodnot.

5.2.5 Možnosti nastavení

V nastavení je možné zvolit novou cestu pro ukládání výsledků expertíz (do běžného textového souboru). Ve výchozím případě je název cílového adresáře Uložené-výsledky a nachází se v pracovním adresáři aplikace.

Dále je možné nastavit jiné než výchozí barvy či je zcela vypnout; případně se může uživatel po jejich deaktivování opět vrátit k výchozímu nastavení.

Třetí možností nastavení je aktivace automatického ukládání expertíz v průběhu jejich vykonávání, aniž by musel uživatel používat příkaz **end**. Ty jsou pak k opětovnému načtení klasicky dostupné ve volbě `(2)` hlavního menu.

5.3 Inference

Během expertízy se můžeme setkat se dvěma typy odvozování nových znalostí.

5.3.1 Fuzzyfikace

V průběhu zadávání naměřených hodnot objektivních písmových parametrů do expertízy je třeba z těchto vstupů odvodit hodnoty, s nimiž je svázán význam; a k těmto významům dále jejich faktory jistoty (podrobněji o tom bylo psáno v kapitole 5.2.3). K tomu používáme „funkci příslušnosti“ definovanou následovně:

$$f(x) = \begin{cases} 100\Big(\frac{1}{\gamma(a,b)-a} \cdot (x-a)\Big) & x \in [a, \gamma(a,b)], \\ 100\Big(\frac{-1}{b-\gamma(a,b)} \cdot (x-\gamma(a,b))+1\Big) & x \in [\gamma(a,b), b]. \end{cases}$$

Máme zde hodnoty a, b, které nám stanovují libovolný interval $[a, b]$ z množiny intervalů definovaných znalostním inženýrem (pro daný objektivní parametr písma). Hodnota x (tj. naměřená hodnota) je dále systémem namapována[36] na popisnou hodnotu spojenou s intervalem $[a, b]$, do něhož x náleží (padne-li do více intervalů, pak je více i namapovaných hodnot a „funkce příslušnosti“ se vyhodnocuje pro každý příslušný interval). Funkční hodnota funkce f pak stanovuje faktor jistoty přiřazený k této namapované hodnotě, který se dále kombinuje s faktory jistoty v pravidlech do výsledného faktoru jistoty, který náleží konečnému významu (tento proces bude popsán v následující kapitole).

[36]Bráno z pohledu teorie fuzzy množiny, tento proces se nazývá *fuzzyfikace*. [32]

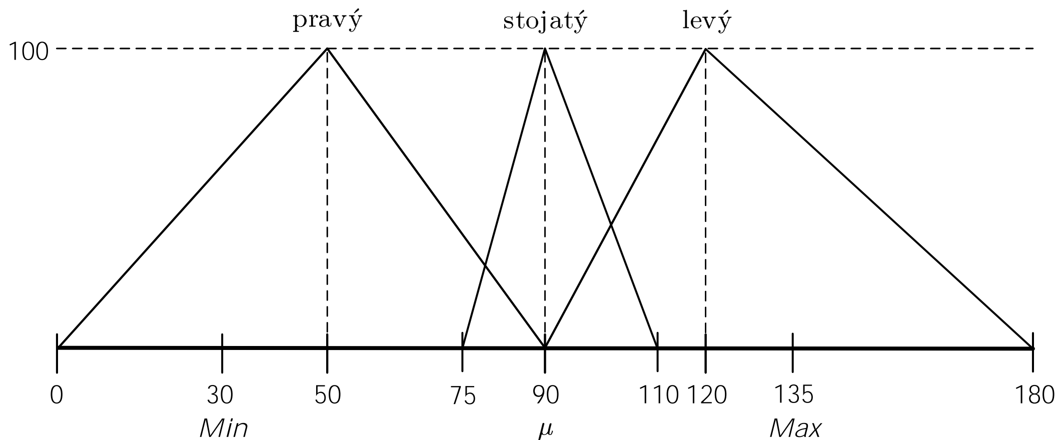
Mimoto se ve funkci f vyskytuje funkce γ, která stanovuje bod, v němž bude mít funkce f v daném intervalu $[a, b]$ své maximum. Jeho určení se liší dle různých kritérií kladených na interval $[a, b]$. Označíme-li minimální hranici normality jako Min, maximální jako Max a střední hodnotu jako μ, funkce γ je definována následovně:

$$\gamma(a,b) = \begin{cases} Min + \frac{b-Min}{3} & \text{je-li } [a,b] \text{ \textit{levý} krajní interval,} \\ Max - \frac{Max-a}{3} & \text{je-li } [a,b] \text{ \textit{pravý} krajní interval,} \\ \mu & [a,b] \text{ \textit{není} krajní a } \mu \in (a,b), \\ a + \frac{b-a}{2} & [a,b] \text{ \textit{není} krajní a } \mu \notin (a,b). \end{cases}$$

Uveďme si nyní příklad pro písmový parametr *Sklon*, jehož hodnoty jsou *pravý*, *stojatý* a *levý* s příslušejícími intervaly hodnot v jednotkách ° (popořadě): $[0, 90]$, $[75, 110]$, $[90, 180]$. Minimální hranice normality byla stanovena jako 30°, maximální jako 135° a střední hodnota jako 90°.

V případě krajních intervalů vychází hodnota funkce γ jako: $\gamma(0, 90) = 50$ a $\gamma(90, 180) = 120$; v případě prostředního intervalu je pak rovna střední hodnotě μ.

Na obrázku dále vidíme graf „funkce příslušnosti“ f.



Obrázek 6: „Funkce příslušnosti“ písmového parametru Sklon

Může nastat i případ, kdy dvojice krajních intervalů neexistuje, nýbrž celý definiční obor pokrývá jediný interval (v němž mohou být samozřejmě obsaženy další). V takovém případě hranice normality určovány nejsou.

5.3.2 Odvozování závěrů

K odvozování závěrů ze získaných dat a znalostí ve znalostní bázi dojde poté, co si uživatel explicitně vyžádá vyhodnocení expertízy zadáním příkazu **eval** nebo

- projdou-li se v případě *režimu „Krok za krokem“* všechny parametry písma,
- není-li už nic ke zpracování (tj. všechny písmové parametry a znaky již byly definovány a zobrazují se zabarveně) v případě *výběrového režimu*.

V případě grafologické expertízy je pro nás žádoucí odvodit pokud možno všechny možné závěry. Jedná se tedy o generativní problém, a proto je vhodné jako směr inference používat dopředné řetězení (viz 3.3.3). Když uživatel zadává systému data, vznikají v pracovní paměti nové fakty společně s jejich faktory jistoty (které zadá uživatel nebo vzniknou fuzzyfikací). Tyto fakty jsou pak během tohoto procesu odvozování rozpoznávány jako premisy pravidel, jejichž závěr je eventuálně osamostatněn a vložen do pracovní paměti jako nový fakt s novým faktorem jistoty.

Ukažme si nyní určení tohoto nového faktoru jistoty závěru – ten si označíme jako c – a to podle různé podoby použitých pravidel:

1. **pravidlo s jednoduchou premisou** – označíme-li faktor jistoty tohoto pravidla jako r a faktor jistoty faktu, který se nachází v premise, jako f, pak c zkombinujeme jednoduše jako
$$c = \frac{f \cdot r}{100}.$$
2. **pravidlo s premisou ve tvaru konjunkce** – faktor jistoty tohoto pravidla označme opět jako r; pro jeho premisu (s n participanty konjunkce) existují v pracovní paměti fakty s faktory jistoty $f_1, \ldots, f_n$. Položíme
$$f = \min\{f_1, \ldots, f_n\}$$
a toto f zkombinujeme společně s r do c stejně jako v předchozím případě.
3. **pravidla se stejným závěrem** – je-li odvozen stejný závěr ze dvou různých pravidel, poprvé s faktorem jistoty c_1 a podruhé s c_2, pak je výsledné c posíleno pomocí vztahu (který je inspirován kombinační funkcí Mycinu – viz 3.4.1)
$$c = c_1 + c_2 - \frac{c_1 \cdot c_2}{100}.$$

K osamostatnění závěru dojde v případě, je-li jeho vypočítaný faktor jistoty větší nebo roven *hranici ignorance*. Nicméně tím, že umožňujeme odvození jednoho závěru z více různých pravidel s možností posílení výsledného faktoru jistoty, je s tím třeba počítat a ihned daný závěr „neodvrhnout“, jestliže podmínku

nesplňuje. Namísto toho takový závěr dostává „druhou šanci“, jestliže systém zjistí, že stále existují nepoužitá pravidla se shodným závěrem.

Tím se již dostáváme k poslední věci – totiž způsobu, jakým inference probíhá v případě interních a uživatelských pravidel.

Interní pravidla jsou základním typem pravidel, tudíž je velká pravděpodobnost, že jich bude hodně (více než uživatelských pravidel). K jejich zpracování tedy dochází spíše reaktivně – prochází se fakty v pracovní paměti a ty se pokouší unifikovat s premisami interních pravidel. Složitost v takovém případě tedy závisí na počtu faktů v pracovní paměti a na interním vyhledávacím mechanismu Prologu, což je mnohem lepší přístup než procházení všech interních pravidel a testování, zda se jejich premisa nevyskytuje v pracovní paměti.

Naproti tomu v případě *uživatelských pravidel* volíme přístup opačný zejména kvůli obecnosti jejich premis, které mohou být ve tvaru konjunkce. Procházíme tedy všechna uživatelská pravidla a ověřujeme, zda je jejich premisa splnitelná v aktuální databázi faktů.

V obou případech postupujeme iterativně, dokud je možné vyvozovat nové závěry.

5.4 Vysvětlování závěrů

Po zobrazení závěrů je k dispozici volba pro vysvětlení procesu odvozování vybraného závěru. Je možné použít dvě varianty vysvětlování:

- jednoduché – pouze sesbírá a vypíše vstupy od uživatele, které přispěly k odvození tohoto závěru,

- pokročilejší – vypíše celý proces odvozování včetně použitých pravidel a důkazu platnosti jejich premis.

V obou režimech v podstatě dochází k inferenci opačným směrem – tedy ke zpětnému řetězení. Postupuje se od zvoleného závěru přes použitá pravidla, která tento závěr implikují, směrem k premisám těchto pravidel. Platnost těchto premis je opět třeba prokázat – mohou to být buď uživatelem zadané fakty nebo se nachází jako závěry v jiných pravidlech, jejichž premisy je opět potřeba prokázat jako platné. Podobným způsobem může vysvětlování probíhat do libovolného zanoření.

6 Implementace

V této části textu budou z implementačního hlediska prezentovány vybrané partie z návrhu expertního systému, který byl popsán v předchozí kapitole.

6.1 Reprezentace znalostí

Z důvodu použití Prologu jako programovacího jazyka se mi pro reprezentaci znalostní báze nabídly prologovské fakty a pravidla.

Tři základní relace představené v kapitole 5.1.2 jsem s použitím dvou základních unárních predikátů `in_writing` a `writer_is` přirozeně reprezentovala jako struktury:

1. `in_writing(parameter(P,H))` – `P` je (proměnná představující) název parametru, `H` jeho hodnota,
2. `in_writing(Z)` – `Z` je písmový znak,
3. `writer_is(Ch)` – `Ch` je charakteristika osobnosti pisatele.

Pro větší flexibilitu v následném odvozování – aby bylo možné snadno provádět dopředné i zpětné řetězení a nebyli jsme přitom omezeni prologovskými pravidly a vyhodnocovacím procesem, který nad nimi provádí Prolog – jsem reprezentaci pravidel znalostní báze redukovala pouze na prologovské fakty.

Podoba obecného uživatelského pravidla je následující fakt:

```
rule(user, Id, Premise ---> Conclusion : CF).
```

Proměnnou `Id` můžeme vždy unifikovat jako

```
Id = TypeId-Num,
```

kde `TypeId` je identifikátor *typu pravidla* a za `Num` se pokaždé vygeneruje nové číslo, takže vznikne unikátní identifikátor pravidla. Dále premisa pravidla `Premise` je v obecném případě prologovský seznam, který obsahuje libovolný počet tří výše uvedených struktur, naproti tomu `Conclusion` je samostatně stojící struktura (opět jedna z těch tří). Nakonec `CF` představuje faktor jistoty. Tyto popsané objekty jsou pak propojeny pomocí uživatelsky definovaných prologovských operátorů `--->` a `:`.

Podoba interního pravidla je obdobná – pouze chybí označení `user`, identifikátor pravidla je (až na číslo `N`) pevně daný a z hlediska stavby je interní pravidlo speciálním případem uživatelského pravidla, neboť jeho premisou je jednoprvkový seznam s prvním typem struktury:

```
rule(parameter_meaning_rule-N,
 [in_writing(parameter(P,H))]
            --->
   Conclusion : CF).
```

6.2 Zpracování uživatelských vstupů

O zpracování uživatelských vstupů se stará zejména modul answer-processor.pl. Hlavní predikát, který obstarává zobrazení otázky/menu, načtení vstupu od uživatele a zpracování tohoto vstupu, je predikát `user_dialog` zastoupený dvěma klauzulemi, na jejichž funkci se blíže podíváme.

```
user_dialog(Id, Param, QParam, Output) :-
    stack(push, Id),
    question(Id, QParam),
    read_input(Input),
    process_answer(Id, Input, Param, QParam, Output),
    stack(pop, Id), !.
```

Každý `user_dialog` má svůj identifikátor `Id`, který rozlišuje kontext jeho použití na různých místech v programu; podle stejného `Id` se navíc vyhledává i příslušná otázka, kterou program položí (predikát `question`) a též i příslušná reakce na uživatelův vstup (predikát `process_answer`). Na začátku se navíc dané `Id` vloží na zásobník (predikát `stack`) a po dokončení aktuálního dialogu se zase odebere. Tento úkon slouží k implementaci příkazu **back**, k čemuž se dostaneme v podkapitole 6.2.2, kde si také uvedeme druhou klauzuli predikátu `user_dialog`.

Predikát `read_input` pouze načte do proměnné `Input` vstup, který před tím normalizuje – odstraní z něj nadbytečné bílé znaky. Vstup v této podobě pak dostává `process_answer`. Mimoto zde máme dodatečné parametry `Param` a `QParam` – první z nich slouží čistě pro potřeby `process_answer`, ve druhém mohou být predikátu `question` předány dodatečné argumenty k tisku otázky; `QParam` se nicméně předává i predikátu `process_answer`, jelikož ten může skončit chybovým hlášením a v takovém případě může uživatel přistoupit k znovuvyvolání původní otázky příkazem **?**. Proměnná `Output` je navázána na návratovou hodnotu predikátu `process_answer`, který může, ale i nemusí vracet nějaký výsledek.

6.2.1 Implicitní přetypování

Zpracování odpovědi uživatele se řídí nejen dle `Id` predikátu `user_dialog`, ale i podle typu daného vstupu. To obecně umožňuje jednoznačnější přiřazení adekvátní reakce na uživatelovu odpověď nebo vyvolávání výchozích chyb dle typu vstupu. V neposlední řadě pak také jednodušší implementaci příkazu **back**, při němž je používáno nucené selhávání (pomocí predikátu `fail`) a nechtěný nedeterminismus by celou záležitost pouze komplikoval.

Modul parser.pl pomocí DCG (definite clause grammar) implementuje typovací systém, který rozpozná a přiřadí typ danému vstupu. Typy se mohou pohybovat na škále od nejběžnějších, jako je řetězec či číslo, až po ty méně tradiční, které spíše vyhovují potřebám programu – tj. speciální sekvence (např. příkazy, které jsme zmiňovali v 5.2.1), povolené tvary intervalů, barvy zadané ve formátu RGB aj.

Některé typy vstupů navíc nejsou pevně dané, nýbrž je u nich umožněno implicitní přetypování, jehož fungování si zběžně ukážeme v kontextu predikátu `process_answer`.

```
process_answer(Id, Input, Param, QParam, Output) :-
   (
      exist_predicate(backtrack/2)
   ->
      !, fail
   ;
      assign_type(Input, Type),
      process_match(Id, [Type, Input], Param, QParam, Output), !
   ).
```

Predikát `assign_type` zde zajišťuje přiřazení typu ke vstupu `Input`. Obojí se pak unifikuje v jedné z mnoha klauzulí predikátu `process_match`, který již zcela konkrétně reaguje na daný vstup příslušnou akcí, což může být mj. i chybová hláška. Důležité je však to, že k unifikaci vůbec došlo. Pokud ne, predikát `process_match` selže a vyhodnocovací proces se pokusí alternativně splnit predikát `assign_type`, přičemž je-li typ `Type` přetypovatelný, dojde k jeho přetypování a ke splnění `process_match` nyní může dojít s tímto novým typem.

Konstrukt `( ... -> ... ; ... )` z výše uvedené ukázky kódu představuje pouze klasické *if-then-else*. Funkce predikátu `backtrack` bude objasněna v následující kapitole.

6.2.2 Implementace zpětného chodu

Zadá-li uživatel příkaz **back**, tento vstup se (typově) vyhodnotí na speciální sekvenci kontrolující backtracking. Dojde ke splnění odpovídajícího `process_match`, který nejprve ověří, zda je pro aktuální `Id` povolen návrat (pokud ne, predikát selže běžným způsobem – tj. je vyvolána chyba). Dále se zjistí, jaké `Id` bylo na zásobník vloženo jako předposlední a to se společně s jeho úrovní zanoření na zásobníku (`Level`)[37] uloží do pracovní paměti ve formě predikátu

```
backtrack(Id, Level),
```

který určuje, do jakého místa má probíhat backtracking. Ihned poté predikát `process_match` selže skrze sekvenci `!, fail`,[38] čímž se spustí backtracking, který povede až ke splnění druhé klauzule `user_dialog`.

```
user_dialog(Id, Param, QParam, Output) :-
   exist_predicate(backtrack/2),
   backtrack(GoalId, GoalLevel),
   (
```

[37] Úroveň zanoření může být potřebná ve chvíli, je-li na zásobníku uloženo více stejných `Id`.

[38] ! – tzv. *cut* zde slouží k zahození všech alternativních cest, které mohou být spojeny s aktuálním predikátem; tj. brání tomu, aby se splnila alternativní klauzule predikátu `process_match` a ve spojení s `fail` zároveň zajistí jeho selhání.

```
    stack(top, GoalId, GoalLevel)
->
    stack(pop,_),
    wm_change(Id, undo),
    retract(backtrack(GoalId, GoalLevel)),
    user_dialog(Id, Param, QParam, Output)
;
    stack(pop,_),
    wm_change(Id, undo),
    fail
).
```

V této klauzuli se testuje podmínka, zda se nacházíme již v cílové úrovni zanoření zásobníku, která je uchovávána v predikátu `backtrack`. Pokud tomu tak není, `user_dialog` selže, opět se spustí backtracking, který znovu povede ke splnění alternativní klauzule `user_dialog`, ale o úroveň výše. Jako vedlejší efekt se při každém průchodu tímto predikátem odstraní vrchol zásobníku a vrátí se změny provedené v paměti v rámci `process_answer` s aktuálním `Id`. Je-li podmínka v `user_dialog` tentokrát splněna, z pracovní paměti se odstraní `backtrack`, který vedl k selhávání a znovu se provede `user_dialog`, k němuž se chtěl uživatel příkazem **back** vrátit.

6.3 Inferenční mechanismus

Odvozování z *uživatelských pravidel* je konstruováno jako klasický „cyklus řízený selháním“ (z angl. *failure driven loop*), který zajišťuje průchod všech uživatelských pravidel. Zde je cyklus implementován predikátem `deduce_from_rules`.

```
deduce_from_rules :-
    exist_predicate(rule/3),
    used_rules(UsedRulesList),
    rule(user,Id,RuleBody),
    (
        not(member(Id, UsedRulesList))
    ),
    fail, !.

deduce_from_rules.
```

V tomto typu cyklu se nám obecně vyskytuje

- *generátor* – predikát splnitelný vícero způsoby – což je v tomto případě predikát `rule`,
- predikáty s vedlejším efektem – zde je to `apply_rule`,
- predikát `fail`, který spustí backtracking vedoucí ke splnění alternativní klauzule generátoru.

Nakonec je přítomna alternativní klauzule predikátu `deduce_from_rules`, která je splněna, dojde-li k vyčerpání splnitelných klauzulí generátoru. Navíc vždy před pokusem aplikovat dané pravidlo testujeme, zda už nebylo použito.

Co se týče predikátu `apply_rule`, který obstarává zpracování pravidla, jeho podoba je následující.

```
apply_rule(Id, PremiseList ---> Conclusion : RuleCF) :-
   satisfiable_premise_list(PremiseList, 100, MinCF), !,
   add_rule_to_used(Id),
   seclude_conclusion(Conclusion, MinCF, RuleCF).
```

Ve stručnosti se aplikace pravidla skládá ze dvou úkonů – ověření platnosti premisy a případné odloučení závěru jakožto nového faktu. Platnost premisy testuje predikát `satisfiable_premise_list` dle návrhu uvedeném v 5.3.2. Jelikož nám jde o to získat nejmenší faktor jistoty faktu z konjunkce faktů v premise, jako výchozí hodnota pro porovnávání je vložena hodnota 100. Výsledný faktor jistoty je pak vrácen v proměnné `MinCF`.

Ve chvíli opuštění predikátu `satisfiable_premise_list` je již zřejmé, že pravidlo bude aplikováno, proto se zabrání pokusům o alternativní splnění premisy pomocí *cut* (k němuž by docházelo v rámci cyklu při vyvolání `fail`) a `Id` pravidla se uloží mezi použitá.

V predikátu `seclude_conclusion` je (opět dle návrhu z výše odkazované kapitoly) implementována kombinace faktorů jistoty `MinCF` a `RuleCF` (tj. faktor jistoty pravidla) do nového faktoru jistoty (označme jej `NewCF`) a uložení závěru `Conclusion` jakožto nového faktu ve tvaru `conclusion(Conclusion, NewCF)` do pracovní paměti. K tomuto odloučení závěru přitom dojde jen za podmínky, že `NewCF` není menší než hranice ignorance nebo je k dispozici více nepoužitých pravidel se závěrem `Conclusion`.

Odvozování z *interních pravidel* není o tolik rozdílné, ačkoliv je (oproti inferenci nad uživatelskými pravidly) řízeno databází faktů.

```
deduce_from_basic_rules :-
   exist_predicate(fact/2),
   forall(fact(Fact, PremiseCF),
      find_and_apply_rule(Fact, PremiseCF)),
   exist_predicate(conclusion/2),
   forall(conclusion(in_writing(PremiseArg), PremiseCF),
      find_and_apply_rule(in_writing(PremiseArg), PremiseCF)), !.

deduce_from_basic_rules.
```

V praxi to znamená, že pomocí vestavěného predikátu `forall` procházíme databázi faktů (tj. predikátů `fact` a `conclusion`) a k nim hledáme aplikovatelná pravidla (tj. pravidla s premisou, která by se unifikovala s daným faktem) pomocí predikátu `find_and_apply_rule`.

```
find_and_apply_rule(Premise, PremiseCF) :-
   rule(Id, [Premise] --> Conclusion : RuleCF),
```

```
    (
       used_rules(UsedRules),
       not(member(Id, UsedRules))
    ->
       add_rule_to_used(Id),
       seclude_conclusion(Conclusion, PremiseCF, RuleCF)
    ),
    fail, !.

find_and_apply_rule(_,_).
```

Tento predikát je opět realizován jako „cyklus řízený selháním“, který ale tentokrát neprochází všechna interní pravidla, ale jen ta se zadanou premisou `Premise`. V těle tohoto predikátu pak obdobně jako v `apply_rule` dochází k osamostatnění závěru `Conclusion`.

7 Zhodnocení

Kapitola obsahuje zhodnocení vytvořeného expertního systému GESto z různých aspektů. Na konci prezentuje možnou vizi jeho budoucího vývoje.

7.1 Vhodnost úlohy pro expertní systém

Nedlouho po zahájení formalizace některých částí grafologie pro potřeby vznikajícího systému jsem si začala klást otázku, zda je to vůbec vhodná úloha pro expertní systém. Dospěla jsem ke kladné odpovědi, ovšem s výhradami. Vzhledem k tomu, že je primárním cílem odvodit všechny možné výsledky, omezuje nás to ve smyslu použití sofistikovanějších metod (souhrnně popsaných v kapitole Expertní systémy), např. pro rozhodování konfliktní množiny bez toho, aniž by se systém musel ponořit hlouběji do sémantiky zpracovávaných dat. Tj. například by některou cestu stromem předem zavrhl, protože by na základě významu předchozích dat zjistil, že by se tím odvodil výsledek, který by se rozporoval s již odvozenými daty.

Jelikož práci s daty nezakládám čistě na dvouhodnotové logice, poskytuje mi to zatím z hlediska sémantiky větší flexibilitu – uživatel tak může odvodit dvě sémanticky protikladná tvrzení, ale s různými faktory jistoty, což může zároveň plně odpovídat realitě. Například v nějakém „divočejším“ písmu by se mohla v některých chvílích objevovat jak *uvolněnost*, tak *strnulost*. Koneckonců i *poměrné parametry písma* jsou postaveny na hodnotách s protikladným významem.

Přesto právě sémantická analýza je další široké pole, v němž by mohl být systém zlepšován.

Druhou výraznější překážkou byla neexaktnost grafologie. Absence statistického základu podstatně ztěžuje tvorbu znalostní báze a výběr faktorů jistoty pro pravidla. Ukázková báze znalostí, který byla v rámci systému vytvořena, tedy nijak nezaručuje, že výsledky budou odpovídat realitě. Tento neduh jsem se snažila vyvážit versatilitou systému ve vztahu ke znalostní bázi. Vytvořila jsem tak nástroj, který může pomoci grafologovi vytvořit mnoho bází s různými hodnotami se zachováním stejného odvozovacího mechanismu. V tomto ohledu vznikl dedikovaný expertní systém (viz 3.2).

7.2 Vhodnost použitého jazyka

Při tvorbě systému jsem shledala, že unifikace a backtracking jsou nesmírně mocné nástroje a z tohoto pohledu mi Prolog implementaci v mnohém ulehčil. Jediná počáteční obtíž byla sžít se s tímto novým paradigmatem. V počátcích tak vznikaly velmi nešikovné verze, v nichž kupříkladu nebylo možné implementovat zpětný chod v dialozích tak, aby to nevedlo k nekonečnému zanořování na zásobníku. Až poté, co jsem více začala využívat potenciálu backtrackingu, se mi tento úkol a mnoho jiných výrazně usnadnily.

7.3 Použité uživatelské rozhraní

Textové rozhraní jsem volila zejména pro jeho jednoduchost. Také mi připadala idea komunikace s uživatelem pomocí dialogů bližší duchu prvotních expertních systémů. Nevýhodou ovšem může být nižší uživatelská přívětivost a menší schopnosti vizualizace, které by se v některých případech hodily (např. zobrazení grafu „funkce příslušnosti“ nebo vizuální nápověda, která by k danému parametru písma a jeho hodnotám uchovávala vzorky písma). I zde je proto prostor pro vylepšování systému.

7.4 Vize

Do budoucna by se implementace nových funkcionalit v systému mohla ubírat vícero směry. Jednak v tom, co jsem již zmínila – sémantické analýze a grafickém uživatelském prostředí, jednak bych si také představovala větší možnosti upravitelnosti, např. „funkce příslušnosti“ přímo uživatelem. Kupříkladu by mohl měnit konstantu $\frac{1}{3}$, které si čtenář mohl všimnout v navržené funkci γ (v 5.3.1), která určuje, „jak daleko“ se bude vyskytovat maximum ve „funkci příslušnosti“ od hranic normality. Dále by mohla být „funkce příslušnosti“ modelovaná jinými než jen lineárními funkcemi a grafolog by si mohl vybírat podle toho, co mu dává lepší výsledky.

O trochu větší vizí by pak mohly být nějaké náznaky „učícího se mechanismu“. Obecně řečeno, systém by se mohl poučit z expertíz, které v něm již probíhaly a na základě těchto starších dat, příp. zpětné vazby grafologa, zlepšovat vyhodnocování výsledků.

Závěr

Tato práce měla dva teoretické cíle – popsat problematiku jazyka Prolog a expertních systémů. Vznikla dvě poměrně obsáhlá pojednání edukativního charakteru, jejichž záměrem je uvést čtenáře do těchto témat z různých hledisek. V části o Prologu jsem tento jazyk popsala jak z jeho techničtější stránky, tak i z hlediska jeho teoretického základu. Text o expertních systémech jsem vystavěla především na definici expertních systémů, kterou formuloval Edward Feigenbaum – tak jsem se postupně dotkla všech jeho základních komponent a nakonec jsem odbočila i do obecnějšího tématu neurčitosti dat.

V rámci praktického cíle jsem se naučila jazyk Prolog a naprogramovala v něm dedikovaný grafologický expertní systém s interaktivním textovým uživatelským rozhraním. Ten zahrnuje editor pro přidávání nových znalostních bází a jejich úpravu. Pomocí něho může znalostní inženýr definovat písmové parametry a pravidla, a tak ovlivňovat výsledky expertízy (tj. analýzy konkrétního písma). V rámci vytvořeného systému dále vznikla ukázková znalostní báze inspirovaná knihou *Grafologie, více než diagnostika osobnosti* od Jana Jeřábka, prezentující schopnosti systému.

Expertíza, kterou grafolog s využitím znalostní báze provádí, spočívá v zadání dat o konkrétním písmu systému a následném odvození závěrů, což jsou charakteristiky osobnosti pisatele. Expertízy je mj. možné přerušit a uložit (i automaticky) k pozdějšímu pokračování. V opětovně načtené expertíze je také možné aktualizovat již zadaná data a znovu je vyhodnotit. V neposlední řadě systém nabízí jednoduché rozhraní pro vysvětlení procesu odvození vybrané charakteristiky pisatele.

Mimoto uživatele celým programem provází kontextová nápověda, kterou je možné vyvolat na libovolném místě. Nápověda k písmovým parametrům je součástí znalostní báze a definuje ji znalostní inženýr.

Grafologické expertní systémy jsou poměrně neprobádanou oblastí bez volně dostupných řešení. Existující práce na toto téma se s problémem vyrovnávají převážně z hlediska počítačového zpracování obrazu. I v tomto ohledu může být moje práce přínosem.

Conclusions

Two primary theoretical objectives of this thesis were focused on describing problematics of the Prolog programming language and expert systems. Created educationally focused text is rather comprehensive; its main purpose is to introduce reader into those subjects from different perspectives. In the section about Prolog, I have described this language in both ways – in the sense of more technical aspects and a theoretical basis. The text about expert systems was built on the Edward Feigenbaum's definition of expert system in order to mention all basic components of an expert system. Lastly, I digressed slightly into more general discussion about the uncertainty of data.

In terms of practical goals, I have learnt Prolog and used it to create dedicated graphological expert system with interactive text user interface. The system includes knowledge base editor for adding new knowledge bases and their subsequent modification. It enables knowledge engineer to define writing parameters and rules that influence results of expertise (analysis of specific writing). Within this system, an exemplary knowledge base inspired by *Grafologie, více než diagnostika osobnosti* authored by Jan Jeřábek was created to showcase the abilities of the system.

Expertise done by graphologist using a knowledge base consists of collecting data describing the writing followed by inference of conclusions which represent personality traits of a writer. Expertises can be interrupted, saved (automatically as well) and later loaded again. In any expertise, saved data can be modified at any time and conclusions re-evaluated. System also offers simple interface for explanation of inference process of selected personality trait.

Moreover, contextual help system is guiding the user throughout the application. Description of writing parameters is part of knowledge base and is defined by knowledge engineer.

Graphological expert systems are relatively unexplored area without any freely available solutions. Existing work on this topic is largely focusing on perspective of computer image processing. My work can be beneficial in this regard as well.

A Obsah přiloženého DVD

`doc/`

Obsahem adresáře je text práce ve formátu PDF, včetně všech ostatních zdrojových souborů.

`src/`

V tomto adresáři se nachází kompletní zdrojové kódy programu GESTO. Též je zde výchozí adresář pro export výsledků expertízy do textového souboru s názvem Uložené-výsledky.

`readme.txt`

Soubor obsahuje podrobné instrukce pro instalaci a spuštění programu GESTO.

Navíc DVD obsahuje:

`install/`

Zde se nachází instalátor SWI-Prologu pro MS Windows.

Literatura

[1] ISO/IEC-JTC-1/SC-22. *ISO/IEC 13211-1:1995 — Information technology — Programming languages — Prolog — Part 1: General core*. 1995.

[2] ISO/IEC-JTC-1/SC-22. *ISO/IEC 13211-2:2000 — Information technology — Programming languages — Prolog — Part 2: Modules*. 2000.

[3] COVINGTON, M.A. *ISO Prolog: A Summary of the Draft Proposed Standard*. Dostupný z: ⟨http://fsl.cs.illinois.edu/images/9/9c/PrologStandard.pdf⟩.

[4] CLOCKSIN, W.F.; MELLISH, C.S. *Programming in Prolog*. Fifth. 2003. ISBN 3540006788.

[5] BĚLOHLÁVEK, R. *Matematická logika – poznámky k přednáškám*. Dostupný z: ⟨http://belohlavek.inf.upol.cz/vyuka/ML.pdf⟩.

[6] STERLING, L.; SHAPIRO, E. *The Art of Prolog*. Second. 1999. ISBN 0-262-19338-8.

[7] FOJTÍK, V. *Rezoluční metoda ve výrokové logice s aplikací na logické programování*. Dostupný z: ⟨http://msekce.karlin.mff.cuni.cz/~tuma/Aplikace17/Prace/Vit%20Fojtik.pdf⟩.

[8] FEIGENBAUM, E.A. Artificial Intelligence. *Scientific American*. 1982, roč. 247, č. 4.

[9] MAŘÍK, V.; ŠTĚPÁNKOVÁ, O.; LAŽANSKÝ, J. *Umělá inteligence 2*. 1997. ISBN 80-200-0504-8.

[10] MCDERMOTT, J.; FORGY, C. Production System Conflict Resolution Strategies. *ACM SIGART Bulletin*. 1977, č. 63.

[11] BRATKO, I. *Prolog Programming for Artificial Intelligence*. 1986. ISBN 0-201-14224-4.

[12] MERRITT, D. *Building Expert Systems in Prolog*.

[13] NAU, D.S. Expert Computer Systems. *IEEE Computer*. 1983, roč. 16, č. 2.

[14] BERKA, P. *Expertní systémy*. First. 1998. ISBN 8070798734.

[15] WOODS, W.A. Important Issues in Knowledge Representation. *Proceedings of the IEEE*. 1986, roč. 74, č. 10.

[16] FIKES, R.; KEHLER, T. The Role of Framed-Based Representation in Reasoning. *Communications of the ACM*. 1985, roč. 28, č. 9.

[17] MAŘÍK, V.; ŠTĚPÁNKOVÁ, O.; LAŽANSKÝ, J. *Umělá inteligence 1*. 1993. ISBN 80-200-0496-3.

[18] DAHL, V.; FRASER, S. Logic Programming as a Representation of Knowledge. *IEEE Computer*. 1983, roč. 16, č. 10.

[19] GENESERETH, M.R.; GINSBERG, M.L. Logic Programming. *Communications of the ACM*. 1985, roč. 28, č. 9.

[20] MORRIS, M.E. Meta-Level Reasoning for Conflict Resolution in Backward Chaining. *IEEE WESTEX*. 1986.

[21] WALTZ, D.L. Artificial Intelligence. *Scientific American*. 1982, roč. 247, č. 4.

[22] USMAN, M.; BRITTO, R.; BÖRSTLER, J.; MENDES, E. Taxonomies in software engineering: A Systematic mapping study and a revised taxonomy development method. *Information and Software Technology*. Roč. 85.

[23] MURRELL, H. *State Space Representation and Search* [online]. [cit. 2020-2-4]. Dostupný z: ⟨http://www.cs.ukzn.ac.za/~hughm/ai/notes/searches.pdf⟩.

[24] CHAN, C.-C. *State Space Search* [online]. [cit. 2020-2-4]. Dostupný z: ⟨http://www.cs.uakron.edu/~chanc/cs460/StateSpaceSearch.htm⟩.

[25] NEAPOLITAN, R.E. Forward-chaining versus a graph approach as the inference engine in expert systems. *Proceedings of Applications of Artificial Intelligence III, SPIE*. 1986, roč. 635.

[26] KORF, R.E. Depth-First Iterative-Deepening. *Artificial Intelligence*. 1985, roč. 27, č. 1.

[27] BUCHANAN, B.G.; SHORTLIFFE, E.H. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. 1984.

[28] BONISSONE, P.P.; TONG, R.M. Reasoning with Uncertainty in Expert Systems. *International Journal of Man-Machine Studies*. 1985, roč. 22.

[29] JACKSON, P. Introduction to Expert Systems. 1999.

[30] BAKOVÁ, H. *Psychologie písma – humanistický přístup v poznávání osobnosti z rukopisu*. First. 2015. ISBN 9.

[31] JEŘÁBEK, J. *Grafologie, více než diagnostika osobnosti*. Sixth. 2014. ISBN 978-80-257-1141-5.

[32] MODRÁK, O. *Teorie automatického řízení II. – Fuzzy řízení a regulace*. Dostupný z: ⟨https://www.uiam.sk/~bakosova/wwwRTP/tar2fuz.pdf⟩.