PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

DIPLOMOVÁ PRÁCE

Implementace expertního systému v jazyce Common Lisp

2013 Jakub Kaláb

Anotace

Expertní systémy mají v praxi bohaté využití. Jejich smyslem je asistovat expertovi na danou problematiku, či jej plně nahradit. V příloze bakalářské práce implementuji prázdný expertní systém s dopředným řetězením inspirovaný systémem CLIPS jako knihovnu v programovacím jazyku Common Lisp tak, aby jej bylo možno plně integrovat do dalších programů.

Děkuji Mgr. Martinu Dostálovi, Ph.D. za vedení této diplomové práce.

Obsah

Seznam obrázků

Seznam příkladů

1. Úvod

Pojem expertního systému spadá do oblasti umělé inteligence. Jde o počítačový systém, který simuluje rozhodování experta nad zvolenou problémovou doménou. Expertní systém může experta zcela nahradit, nebo mu při rozhodování asistovat.

Ve své bakalářské práci jsem implementoval základní knihovnu pro tvorbu expertních systémů (tzv. prázdný expertní systém) s dopředným řetězením v jazyce Common Lisp. Cílem této práce je knihovnu rozšířit o následující:

- syntaktický režim pro zajištění přiměřené kompatibility se systémem CLIPS,
- možnost vrácení provedených změn včetně odvozovacích kroků,
- podpora pro ladění s jednoduchým grafickým uživatelským rozhraním pro prostředí LispWorks™,
- rozšíření odvozovacího aparátu o základní zpětné řetězení.

Jazyk Common Lisp[1] (případně jiné dialekty Lispu) je častou volbou pro implementaci umělé inteligence díky svým schopnostem v oblasti symbolických výpočtů (manipulace symbolických výrazů), na nichž řešení těchto problémů často staví. Navíc jde o velmi vysokoúrovňový, dynamicky typovaný jazyk, díky čemuž je programový kód expresivní, snadno pochopitelný a tudíž jednoduše rozšiřitelný.

Synax systému CLIPS[2] byla zvolena proto, že jde o reálně používaný systém[3], jehož syntax je Lispu velmi blízká, takže není těžké ji v Lispu napodobit.

Přestože běžnou praxí je začínat diplomovou práci teoretickou částí, definovat jednotlivé pojmy a principy a ty poté v praktické části uplatnit, rozhodl jsem se postupovat opačně, tedy začít práci praktickou částí. Domnívám se totiž (také na základě zkušeností nabytých při vypracování bakalářské práce), že je podstatně snazší, minimálně v řešené problematice, pochopit příklady bez detailní znalosti teorie, než snažit se pochopit teorii bez příkladů, na nichž si lze popisované pojmy a principy představit. V praktické části tedy uvedu jen minimální množtví teorie nutné pro pochopení práce s knihovnou, načež se k ní v teoretické části textu vrátím, pojmy zadefinuji přesně a rozšířím o souvislosti. V tuto chvíli už si bude čtenář schopen představit, jaké problémy lze pomocí expertního systému řešit a jak takový expertní systém v praxi vypadá.

> Pro popsaní některé teorie (symbolické výpočty, ...) je navíc užitečná (neli potřebná) představa, jak expertní systém funguje uvnitř.

Kdo už to dělal, čím se tohle liší

[1] http://en.wikipedia.org/wiki/Common_Lisp
[2] http://clipsrules.sourceforge.net
[3] http://clipsrules.sourceforge.net/FAQ.html#Q6

2. Praktická část

Tato sekce popisuje knihovnu ExiL[1], která je výsledkem této práce. Nejprve popíšu její instalaci a prerekvizity nutné k jejímu používání. Pak v uživatelské příručce představím její možnosti a typickou strukturu programu, který ji využívá. Poté v referenční příručce projdu všechny možnosti, které knihovna poskytuje. Načež v části věnované implementaci popíšu architekturu jejího zdrojového kódu a zmíním zajímavé části kódu implementující jednotlivá rozšíření. Nakonec uvedu několik větších příkladů použití knihovny a rozeberu několik dalších možných rozšíření a co by obnášela z pohledu implementace.

2.1. Common Lisp

- základní znalosti lispu nutné pro používání knihovny
  - loadování souborů - poté zjednodušit v kapitole o instalaci
  - package, export, import, shadow
  - seznam, atom, car, cdr, plist
  - symbol, klíč
  - prefixová syntax, S-expressions
  - funkce, makro
  - načítání souborů.
  - quotování (u funkčních alternativ, ale ty se stejně používají spíš z jiného kódu, který knihovnu volá - tudíž uživatel evidentně lisp zná)
  - lexikální prostředí - u vyhodnocení aktivací pravidla
  - destrukturující makra - tamtéž
- odkaz na practical common lisp, clhs

2.2. Instalace

[1] TODO: původ názvu

2.2.1. Získání zdrojového kódu

Zdrojový kód knihovny je přiložen k této diplomové práci a lze jej také získat zklonováním[2] gitového[3] repozitáře na adrese `git@github.com:Incanus3/ExiL.git`. Kód knihovny se nachází v podadresáři `src`, ten budu dále nazývat kořenovým adresářem knihovny či projektu.

2.2.2. Prerekvizity

Pro práci s knihovnou ExiL potřebujeme lispový interpreter[4] [5], vývojové prostředí (s interpreterem bychom si ve skutečnosti vystačili, ale přímá práce s ním není většinou příliš pohodlná) a knihovny umožňující dávkové načtení celého projektu včetně závislostí.

Knihovnu jsem vyvíjel v prostředí SLIME[6], což je plugin pro textový editor GNU Emacs[7] (poskytující mimo jiné pomůcky pro editaci lispového zdrojového kódu, REPL[8] a debugger[9]) s interpreterem SBCL[10] a tuto kombinaci mohu vřele doporučit. V operačním systému Debian GNU Linux, který jsem pro vývoj použil, lze Emacs, SLIME i SBCL nainstalovat z výchozího repozitáře a aktivovat úpravou inicializačního souboru Emacsu, viz `http://www.common-lisp.net/projects/slime/doc/html/Installation.html`. Prostředí poté můžeme v Emacsu spustit voláním příkazu `slime` (`<Alt+X>slime<ENTER>`). Při prvním spuštění se kód prostředí kompiluje, což může chvíli trvat, pak už se v editoru otevře buffer[11] s lispovým REPLem.

Knihovnu jsem testoval také ve vývojovém prostředí LispWorks®[12] Personal Edition 6.1, pro které jsem také vytvořil minimalistické grafické uživatelské rozhraní. Součástí prostředí LispWorks je i lispový interpret. Prostředí můžeme získat zde `http://www.lispworks.com/downloads/index.html` a nainstalovat podle návodu, který se zobrazí po vyplnění formuláře.

Pro efektivní načtení knihovny včetně závislostí potřebujeme ještě dvě knihovny:

- ASDF[13] je knihovna umožňující snadnou definici struktury projektu a jeho

[2] `http://git-scm.com/docs/git-clone`
[3] `http://git-scm.com/`
[4] `http://en.wikipedia.org/wiki/Interpreter_(computing)`
[5] lispové interpretery jsou většinou zároveň kompilátory[6], označením interpreter tedy budu nazývat obojí
[6] `http://en.wikipedia.org/wiki/Compiler`
[6] `http://www.common-lisp.net/project/slime/`
[7] `http://www.gnu.org/software/emacs/`
[8] `http://en.wikipedia.org/wiki/Read-eval-print_loop`
[9] `http://en.wikipedia.org/wiki/Debugger`
[10] `http://www.sbcl.org/`
[11] emacs buffer
[12] `http://www.lispworks.com/`
[13] `http://common-lisp.net/project/asdf`

dávkové načtení,

- quicklisp[14] staví na knihovně ASDF a umožňuje pohodlně stáhnout a načíst knihovny třetích stran z internetové databáze.

Knihovna ASDF je součástí instalace interpreteru SBCL i prostředí LispWorks. Knihovnu quicklisp jsem k projektu přiložil a pokud není součástí prostředí, je automaticky načtena před načtením ExiLu.

2.2.3. Načtení knihovny

V prostředí SLIME načteme knihovnu načtením souboru `load.lisp` z kořenového adresáře knihovny, tedy zadáním

```
(load "cesta/k/projektu/src/load.lisp")
```

v REPLu). Tento soubor nejprve načte knihovnu `quicklisp`, je-li potřeba, a s její pomocí poté načte celý projekt ExiL včetně závislostí. Nakonec soubor definuje výchozí prostředí, viz sekce 2.3.11.

V prostředí LispWorks načítání pomocí knihovny `quicklisp` nefunguje správně, knihovnu je proto třeba načítat načtením souboru `load-manual.lisp` (opět z kořenového adresáře projektu). Načíst můžeme opět voláním `load` v REPLu, nebo vybráním položky `Load...` v nabídce `File` menu libovolného okna prostředí.

Všechna makra a funkce, které knihovna definuje pro přímé volání uživatelem jsou *exportována* z *package* `exil`. Před interakcí s knihovnou je tedy třeba vstoupit do package `exil-user`, který symboly z package `exil` *importuje*. Symboly z package je také možno importovat do existujícího package takto:

```
(defpackage :my-package
  (:documentation "user-defined package")
  (:use :common-lisp :exil)
  (:shadowing-import-from :exil :assert :step))
```

Package `exil` exportuje několik symbolů, které již v package `common-lisp` existují. Ty je třeba *zastínit*, jak je vidět z ukázky.

[14] http://www.quicklisp.org/beta/

2.3. Uživatelská příručka

2.3.1. Základní pojmy

Nyní stručně zadefinuji základní pojmy, nutné pro pochopení fungování knihovny ExiL a práci s ní. Význam pojmů bude jasnější, jakmile si je ukážeme na příkladech. K těmto pojmům se posléze vrátím i v teoretické části textu a jejich popis rozšířím o další souvislosti.

První dva pojmy staví na pojmu znalost, který chápeme intuitivně a nebudu se jej ani snažit definovat, nikoli na následujícím pojmu znalosti, jak ji chápeme v ExiLu (v takovém případě by byla definice cyklická).

Pojem expertního systému zatím chápejme tak, jak jsem jej představil v úvodu práce. V teoretické části rozeberu pojem v potřebné šíři.

fakt	elementární statická znalost - tvrzení
(odvozovací) pravidlo	elementární odvozovací znalost - pokud víme, že (ne)platí nějaká tvrzení, můžeme odvodit, že platí i nějaká další
znalost (v ExiLu)	množina faktů a pravidel
znalostní báze	výchozí znalost expertního systému
pracovní paměť	aktuální množina faktů
inference	odvozování - postupná aplikace odvozovacích pravidel

Pojem *pracovní paměť* není příliš intuitivní. Jde o doslovný překlad v literatuře užívaného pojmu *working memory*, kterým je označována množina faktů (tvrzení), které expertní systém v danou chvíli považuje za platné. Nejde tedy ve skutečnosti o paměť, nýbrž o obsah pomyslné paměti. Pojem pracovní množina faktů by byl jistě výstižnější, bohužel ale také značně těžkopádný.

2.3.2. Struktura programu

Příklad 1 na straně 12 ukazuje minimální strukturu programu nad knihovnou ExiL (dále exilový program). První část programu tvoří definice znalostní báze. Ta sestává z definic faktů, ze kterých expertní systém vychází, a definic odvozovacích pravidel, jež jsou následně aplikována při inferenci.

Definice faktů jsou uspořádány do skupin označených názvem (v tomto případě `world`). V ukázkovém programu si snadno vystačíme s jednou skupinou faktů, v reálných programech bude ale těchto skupin většinou více. Tato organizace umožňuje snadnou redefinici, případně odebrání, jen některých skupin faktů v případě potřeby. Definice skupiny faktů `world` v příkladu přidává do znalostní

báze informaci o počáteční pozici robota, krabice a o našem záměru přesunout krabici z pozice `A` na pozici `B`.

Následuje definice odvozovacích pravidel. Definice každého pravidla sestává z množiny podmínek, tedy předpokladů pro jeho splnění (a následnou aktivaci), a množiny důsledků, tedy libovolných lispových lispových výrazů, které jsou při aktivaci pravidla vyhodnoceny. Tyto dvě množiny jsou od sebe odděleny *symbolem* `=>`.

Podmínky odvozovacích pravidel jsou ve formě vzorů (*pattern*). Struktura vzorů je stejná jako struktura faktů (viz sekce 2.3.3.), ale na rozdíl od nich mohou obsahovat proměnné (symboly začínající otazníkem). Při vyhodnocování podmínek pravidla je zajišťěna konzistence vazeb těchto proměnných a výskyty všech proměnných v důsledcích pravidla jsou při jeho aktivaci nahrazeny jejich vazbami. Detaily viz sekce 2.3.5.

Důsledky pravidel typicky obsahují příkazy pro modifikaci pracovní paměti (viz sekce 2.3.4.), tedy přidání (`assert`), odebrání (`retract`), či úpravu (`modify`) faktů v ní. Nemusí tomu tak ale být vždycky - důsledkem aktivace pravidla může být např. vypsání výstupu, logování, zápis souboru, ale také např. ovládání externího systému.

Ukázkový příklad definuje tři odvozovací pravidla. Pravidlo `move-robot` je aktivováno, pokud chceme přesunout nějaký objekt z pozice `?from` na pozici `?to`, objekt se nachází v pozici `?from` a robot nikoli (třetí podmínka je negovaná, viz sekce 2.3.5.). Poslední podmínka slouží pouze k navázání původní pozice robota. Při aktivaci pravidla je v pracovní paměti nahrazena informace o původní pozici robota pozicí `?from`. Robot se tedy nyní nachází na stejné pozici, jako kýžený objekt.

Podmínky pravidla `move-object` vyžadují, aby byl jak robot, tak objekt určený k přesunu, na pozici `?from`. Při jeho aktivaci je robot i s objektem přesunut na pozici `?to` nahrazením faktů o původních pozicích novými, podobně jako v prvním pravidle. Definice pravidla obsahuje speciální notaci (s použitím operátoru `<-`), jejímž účelem je navázání celého faktu na proměnnou. Ten pak můžeme v důsledcích snadno ostranit z pracovní paměti. Detaily opět viz sekce 2.3.5.

Poslední pravidlo slouží k zastavení inference, pokud se již objekt nachází na cílové pozici. Inference je zde zastavena explicitním voláním `(halt)`. Druhou možností by bylo odstranit z pracovní paměti fakt definující cíl, neboť v takovou chvíli nemůže být žádné další pravidlo splněno.

Jakmile je znalostní báze nadefinována, můžeme z ní inicializovat pracovní paměť. To provedeme voláním `(reset)`, které (po případném vyčištění původních faktů) přidá do pracovní paměti fakty ve všech definovaných skupinách.

Poslední nutnou fází exilového programu je spuštění inference. To můžeme udělat nejjednodušeji voláním `(run)`. Inferenční mechanismus poté postupně vyhodnocuje, která odvozovací pravidla mají splněné všechny podmínky, v každém kroku z nich jedno vybere a aktivuje jej. Detaily viz sekce 2.3.5.

Výstup programu je následující:

```
==> (IN ROBOT B)
==> (IN BOX A)
==> (GOAL MOVE BOX A B)
Firing MOVE-ROBOT
<== (IN ROBOT B)
==> (IN ROBOT A)
Firing MOVE-OBJECT
<== (IN ROBOT A)
<== (IN BOX A)
==> (IN ROBOT B)
==> (IN BOX B)
Firing STOP
Halting
```

Řádky začínající symbolem `==>` označují fakty přibyvší do pracovní paměti, řádky začínající `<==` fakty z paměti odstraněné. Tento výstup obdržíme pouze pokud zapneme sledování faktů voláním `(watch facts)` (viz sekce **??**). První tři fakty přibydou do pracovní paměti při vyhodnocení volání `(reset)`, další pak spolu s postupnou aplikací odvozovacích pravidel. Dotážeme-li se po skončení inference na seznam faktů v pracovní paměti voláním `(facts)`, obdržíme výstup

```
((GOAL MOVE BOX A B) (IN ROBOT B) (IN BOX B)).
```

Robot i krabice jsou tedy na cílové pozici.

Kód exilového programu má deklarativní charakter. Nikde jsme nemuseli specifikovat, jakou posloupností akcí má systém k výsledku dospět. To nás ovšem nezbavuje nutnosti chápat fungování inferenčního mechanismu ExiLu. Nebudeme-li při konstrukci programu opatrní, může výpočet snadno dospět k neočekávaným výsledkům, dostat se do slepé větve, či se zacyklit. Tyto problémy jsou často způsobeny nezamýšlenou interferencí podmínek pravidel s důsledky jiných.

```
;;; definition of knowledge base
;; facts
(deffacts world
  (in box A)
  (in robot B)
  (goal move box A B))

;; inference rules
(defrule move-robot
  (goal move ?object ?from ?to)
  (in ?object ?from)
  (- in robot ?from)
  (in robot ?z)
  =>
  (retract (in robot ?z))
  (assert (in robot ?from)))

(defrule move-object
  (goal move ?object ?from ?to)
  ?rob-pos <- (in robot ?from)
  ?obj-pos <- (in ?object ?from)
  =>
  (retract ?rob-pos)
  (retract ?obj-pos)
  (assert (in robot ?to))
  (assert (in ?object ?to)))

(defrule stop
  (goal move ?object ?from ?to)
  (in ?object ?to)
  =>
  (halt))

;;; initialization of working memory
(reset)

;;; inference execution
(run)
```

Příklad 1: Základní struktura exilového programu

2.3.3. Definice znalostní báze

ExiL, stejně jako CLIPS, rozlišuje dva typy faktů - jednoduché (*simple, ordered*) a strukturované (*templated*). Stuktura jednoduchého faktu je udána pouze pořadím *atomů*, typickou volbou je např. `objekt-attribut-hodnota`:

```
(box color red),
```

či `relace-objekty`:

```
(in box hall).
```

Strukturované fakty mají naproti tomu explicitně pojmenované složky (sloty). Typicky popisují objekt s množinou pojmenovaných atributů:

```
(box :color red :size small),
```

či relaci s pojmenovanými aktory:

```
(in :object box :location hall),
```

kde `box` a `in` jsou šablony (*template*), které je třeba definovat předem. Na pořadí specifikace slotů u strukturovaných faktů nezáleží.

Vyjadřovací síla obou typů faktů je stejná, použitím explicitnějších strukturovaných faktů ale docílíme lepší čitelnosti a jednoznačnější sémantiky exilového programu, zláště třeba v případě relací na jedné množině objektů:

```
(father john george).
```

Šablonu definujeme voláním *makra* `deftemplate`, např:

```
(deftemplate in object (location :default here)).
```

Prvním parametrem je název šablony, za ním následuje libovolný počet specifikací slotů. Specifikací slotu je buď symbol - jméno slotu, nebo *seznam*, jehož hlavou (*car*) je jméno slotu a tělem (*cdr*) je *property list (plist)* s dalšími parametry. Aktuálně systém umožňuje pouze specifikaci výchozí hodnoty slotu *klíčem* `:default`. Ta je použita, není-li při specifikaci faktu, používajícího tuto šablonu, uvedena hodnota pro daný slot.

Je-li už šablona požadovaného názvu definována, ale neexistují v pracovní paměti fakty, které ji používají, je její stávající definice nahrazena. Pokud ale v pracovní paměti existují takové fakty, skončí volání `deftemplate` výjimkou.

Seznam názvů všech definovaných šablon můžeme získat voláním `(templates)`. Specifikaci šablony pak získáme voláním makra `find-template`, např. `(find-template in)`. Definici šablony zrušíme voláním makra `undeftemplate`, např. `(undeftemplate goal)`. To opět skončí výjimkou, existují-li v pracovní paměti fakty, které šablonu využívají.

Fakty, ze kterých expertní systém vychází, zavádíme pomocí skupin faktů. Ty definujeme makrem `deffacts`, např.:

```
(deffacts initial
  (goal move box A B)
  (in :object box :location A))
```

Prvním parametrem je název skupiny, pak následuje libovolný počet specifikací faktů. Opakovaným voláním makra `deffacts` je skupina faktů redefinována.

Specifikace faktu je vždy tvořena seznamem. Pokud jde o jednoduchý fakt, specifikací je prostě seznam atomů. Jde-li o fakt strukturovaný, je prvním prvkem specifikace název šablony, za ním následuje plist určující hodnoty slotů faktu. Pokud není hodnota některého slotu uvedena, je buď použita výchozí hodnota, pokud byla v šabloně specifikována, nebo hodnota `nil` v opačném případě.

Seznam názvů všech definovaných skupin faktů získáme voláním `(fact-groups)`. Specifikaci skupiny pak voláním makra `find-fact-group`, např. `(find-fact-group initial)`. Ke zrušení definice skupiny slouží makro `undeffacts` (voláme s názvem skupiny).

Pravidla, pomocí nichž expertní systém během inference odvozuje nové fakty, definujeme makrem `defrule`, např.:

```
(defrule move-robot
  (goal :action move :object ?obj :from ?from)
  (in :object ?obj :location ?from)
  (- in :object robot :location ?from)
  ?robot <- (in :object robot :location ?)
  =>
  (modify ?robot :location ?from)).
```

Podmínková část pravidla (před symbolem `=>`) je tvořena vzory. Ty mohou být, stejně jako fakty, jednoduché, nebo strukturované. Kromě toho umožňuje definice pravidla několik speciálních konstruktů (negace podmínky, navázání proměnné na celou podmínku). Ty popíšu podrobně v sekci 2.3.5. spolu s tím, jak jsou podmínky pravidla při inferenci vyhodnocovány.

Důsledkovou část pravidla (za symbolem `=>`) tvoří libovolný počet lispových výrazů. Jak se tyto vyhodnocují popíšu opět v sekci 2.3.5..

Opakovaným voláním makra `defrule` odvozovací pravidlo redefinujeme. K získání seznamu názvů definovaných pravidel a jejich specifikací slouží *funkce* `rules` a makro `find-rule`, podobně jako u šablon a skupin faktů. Ke zrušení definice pravidla slouží makro `undefrule`.

2.3.4. Modifikace pracovní paměti

Pracovní paměť je množina faktů, které systém v danou chvíli považuje za platné. Její obsah můžeme vypsat voláním funkce `facts`. Funkcí `reset` inicializujeme pracovní paměť ze znalostní báze. Jejím voláním jsou do pracovní paměti zavedeny fakty všech definovaných skupin faktů (viz sekce 2.3.3.).

Obsah pracovní paměťi může být dále modifikován třemi makry:

- `assert` přidává fakt(y) do pracovní paměti,
- `retract` fakt(y) z pracovní paměti odebírá a
- `modify` přímo modifikuje existující fakt.

Ta lze volat buď před započetím inference (ale po volání `reset`, neboť to dodatečné úpravy vymaže), nebo v jejím průběhu, pokud inferenci krokujeme (viz sekce 2.3.5.). Makra také typicky voláme v důsledcích pravidel.

Makra `assert` a `retract` berou jako parametry libovolný počet specifikací faktů ve stejném formátu, jako u makra `deffacts` (ale bez názvu skupiny). Makro `modify` lze použít jen u strukturovaných faktů. Toto makro bere jako první parametr specifikaci faktu, zbytek parametrů tvoří plist určující hodnoty slotů ke změně. Např.

```
(modify (in :object box :location A) :location B)
```

nahradí v pracovní paměti fakt `(in :object box :location A)` faktem `(in :object box :location B)`. Toto makro je obzvláště užitečné, navážeme-li v podmínkách pravidla celý fakt na proměnnou (viz sekce 2.3.5.).

Pracovní paměť se skutečně chová jako množina, každý fakt tedy může být v pracovní paměti jen jednou. Opětovné volání `assert` nemá žádný efekt. Volání modify, jehož výsledkem by bylo nahrazení nějakého faktu faktem, který již v pracovní paměti existuje, sice odebere původní fakt, nový však znovu nepřidá.

Všechny fakty můžeme z pracovní paměti odebrat voláním `(retract-all)`. To je ale zřídka užitečné, typicky použijeme spíše funkci `reset` pro navrácení pracovní paměti do výchozího stavu.

2.3.5. Inference

Inference (odvozování nových faktů z aktuálních) probíhá v krocích. V každém kroku jsou vyhodnoceny podmínky všech pravidel, načež je ze splněných pravidel vybráno jedno, které je posléze aktivováno. Inferenční kroky můžeme buď spouštět jednotlivě voláním `(step)`, nebo voláním `(run)` spustit cyklus, který provádí inferenční kroky, dokud je to možné. Cyklus je buď přerušen ve chvíli, kdy není splněno žádné další pravidlo, nebo voláním `(halt)` v důsledcích právě aktivovaného pravidla.

Podmínky pravidel jsou ve tvaru vzorů a jsou spojeny logickou konjunkcí, pravidlo je tedy splněno, jsou-li splněny všechny jeho podmínky. Kromě toho mohou být některé podmínky negovány. Taková podmínka je splněna tehdy, neexistuje-li v pracovní paměti žádný fakt, který by se shodoval s jejím vzorem (při zachování konzistence vazeb proměnných). Negovanou podmínku značí znak `-` (minus) na prvním místě specifikace vzoru.

Vyhodnocování podmínek pravidel probíhá ve dvou fázích. V první fázi srovnáváme vzory jednotlivých podmínek se všemi fakty v pracovní paměti a to pouze

strukturálně, tedy bez ohledu na vazby proměnných. Prvním požadavkem shody je u jednoduchých faktů stejná délka (počet atomů), u strukturovaných faktů stejná šablona. Jednoduchý fakt se nikdy nemůže shodovat se strukturovaným vzorem a naopak.

Dále jsou pak porovnávány jednotlivé atomy (u jednoduchých) či sloty (u složených) faktu vůči odpovídajícímu atomu (slotu) vzoru. Není-li atom (slot) vzoru proměnná, je jednoduše porovnán s atomem faktu. Je-li atomem proměnná, považujeme jej v této fázi automaticky za shodu. Například vzor

```
(in :object robot :location ?loc)
```

se shoduje s faktem

```
(in :object robot :location A)
```

nikoli však s fakty

```
(in :object box :location A)
(is-in :object robot :location A)
(in robot A).
```

Tímto předvýběrem tedy získáme ke každé podmínce pravidla množinu faktů, které mají stejnou strukturu a stejné hodnoty neproměnných atomů.

Ve druhé fázi vyhodnocování hledáme z předvybraných faktů takovou posloupnost (délka odpovídá počtu podmínek pravidla), kde po spárování s odpovídajícími vzory podmínek obdržíme konzistentní vazby proměnných. To znamená, že vyskytuje-li se v podmínkách pravila některá proměnná vícekrát, musí mít odpovídající fakty na daných pozicích stejný atom. Mějme například pravidlo s podmínkami

```
(goal move ?obj ?from ?to)
(in :object ?obj  :location ?from)
(in :object robot :location ?to)
```

Vzor první podmínky je jednoduchý, zatímco další dva jsou strukturované. To ale ničemu nevadí, je třeba pouze najít fakty odpovídající struktury. Posloupnost faktů

```
(goal move box A B)
(in :object box   :location B)
(in :object robot :location A)
```

neprojde druhou fází výběru, neboť vazby proměnných nejsou konzistentní. Proměnná `?from` je například v první podmínce navázána na symbol `A`, v druhé ale na `B`. Kdyby si ovšem krabice s robotem vyměnily pozice, budou vazby proměnných konzistentní a podmínky pravidla budou splněny. Proměnná `?from` by pak nabyla hodnoty `A`, proměnná `?to` hodnoty `B` a proměnná `?obj` hodnoty `box`.

Vyhodnocení negovaných podmínek si můžeme představit tak, že nejprve vyhodnotíme a navážeme proměnné všech ostatních podmínek. Pokud poté neexistuje fakt, který by se se vzorem negované podmínky shodoval a měl konzistentní vazby se zbytkem navázaných proměnných, je tato podmínka splněna. Mějme například pravidlo s podmínkami

```
(goal move box ?from ?to)
(in box ?from)
(- in robot ?from).
```

Máme-li v pracovní paměti pouze fakty

```
(goal move box A B)
(in box A)
(in robot B),
```

budou podmínky pravidla splněny, neboť po spárování vzorů prvních dvou podmínek s prvními dvěma fakty bude proměnná `?from` navázána na hodnotu `A` a neexistuje fakt, který by se shodoval se vzorem `(in robot A)`. Přesuneme-li ale robota na pozici `A`, podmínka již splněna nebude a pravidlo nelze aktivovat.

Ve vzorech podmínek pravidla můžeme využít speciální proměnné `?`. Konzistence vazby této proměnné není při vyhodnocování testována, takže vyskytuje-li se tato proměnná na více místech, chová se tak, jako kdyby byl každý výskyt označen unikátním názvem (podobně jako proměnná _ v Prologu). Použitím této proměnné dáváme najevo, že nás konkrétní hodnota daného atomu nazajímá. Ve strukturovaných vzorech podmínek není třeba tyto sloty uvádět, neboť `?` je výchozí hodnotou slotu vzoru.

Posledním speciálním konstruktem je navázání celého faktu na proměnnou. Například pravidlo

```
(defrule move
  ?fact <- (in :object ? :location A)
  =>
  (modify ?fact :location B))
```

přesune každý objekt z pozice `A` na pozici `B`. Na proměnnou můžeme navázat i jednoduchý fakt, pak ale nemůžeme použít makra `modify`. Můžeme ovšem volat `(retract ?fact)`, neboť proměnná `?fact` je při aktivaci pravidla nahrazena specifikací faktu, který byl se vzorem podmínky spárován.

Podmínky některých pravidel mohou být při vyhodnocování splněny několika různými posloupnostmi faktů. Výsledkem vyhodnocování pravidel tedy není pouze množina splněných pravidel, nýbrž množina shod (*match*). Každá shoda je tvořena pravidlem a *subsitucí* proměnných navázaných při vyhodnocování jeho podmínek. Subsituci chápejme jako zobrazení z množiny všech proměnných, vyskytujících se v podmínkách pravidla, na konkrétní hodnoty atomů (slotů). Aplikací této substituce na vzory podmínek pravidla získáme opět posloupnost faktů,

kterými byly podmínky v dané shodě splněny. Tato substituce je následně použita při aktivaci pravidla k náhradě proměnných v jeho důsledích.

ExiL uchovává aktuální množinu shod. Ta ve skutečnosti není přepočítána v první fázi inferenčního kroku, jak jsem dosud pro jednoduchost tvrdil, nýbrž automaticky po každé změně pracovní paměti či množiny pravidel (detaily viz RETE). Aktuální množinu shod nazývám, po vzoru CLIPSu, *agenda* a lze ji vypsat voláním stejnojmenné funkce. Každá shoda v agendě je opatřena časovým razítkem, shody je tedy možné uspořádat podle toho, kdy do agendy přibyly.

Je-li na začátku inferenčního kroku v agendě více shod, je třeba z nich jednu vybrat k aktivaci. Výběr shody záleží na zvolené strategii. ExiL poskytuje následující strategie výběru shody:

depth-strategy	vybírá shodu, která do agendy přibyla nejpozději
breadth-strategy	vybírá shodu, která do agendy přibyla jako první
simplicity-strategy	vybere shodu, jejíž pravidlo má nejméně podmínek
complexity-strategy	volí shodu, jejíž pravidlo má nejvíce podmínek.

Názvy prvních dvou strategií vychází z toho, že jde o prohledávání prostoru stavů systému do hloubky, či do šířky, jak blíže popíšu v teoretické části textu, spolu s motivacemi pro využití jednotlivých typů strategií a dalšími typy strategií, které se v expertních systémech používají.

Výchozí strategií je `depth-strategy`. Strategii, která bude v inferenci použita, můžeme zvolit voláním makra `setstrategy` s názvem strategie, např. `(setstrategy breadth-strategy)`. Seznam názvů strategií můžeme vypsat voláním `(strategies)`, název aktuálně zvolené strategie pak voláním `(current-strategy)`.

Po výběru shody k aktivaci je její pravidlo aktivováno. Nejprve jsou v důsledcích pravidla nahrazeny všechny proměnné pomocí substituce shody. Výrazy v důsledcích jsou poté vyhodnoceny lispovým makrem `eval`. Tento způsob vyhodnocení vede ke značným omezením. Výrazy totiž nejsou vyhodnoceny v *lexikálním prostředí* definice pravidla, nýbrž ve výchozím (*top-level*) prostředí Lispu. Díky tomu nemůžeme v důsledcích pravidla volat lokální funkce (definované např. makrem `labels`), či přistupovat k lokálním proměnným (navázaným například v `letu` či pomoci *destrukturujících maker*).

V důsledcích každého pravidla chceme typicky alespoň jednou volat některé z maker modifikujících pracovní paměť, abychom zneplatnili podmínky pravidla, jinak se inference zacyklí. Druhou možností je přerušit inferenci voláním `(halt)`. Volání `(step)` nebo `(run)` ve chvíli, kdy již nelze dále odvozovat (žádné z pravidel nemá splněné podmínky), nemá žádný efekt.

2.3.6. Sledování průběhu inference

ExiL umožnuje sledovat několik typů událostí, ke kterým dochází během inference. K nastavení sledovaných událostí slouží makra `watch` a `unwatch`. K zjištění stavu sledování pak makro `watchedp`.

Základním výstupem programu 1 na straně 12 je

```
Firing MOVE
Firing PUSH
Firing STOP
Halting.
```

Zapneme-li sledování faktů voláním `(watch facts)`, obdržíme výstup

```
==> (IN BOX A)
==> (IN ROBOT B)
==> (GOAL MOVE BOX A B)
Firing MOVE-ROBOT
<== (IN ROBOT B)
==> (IN ROBOT A)
Firing MOVE-OBJECT
<== (IN ROBOT A)
<== (IN BOX A)
==> (IN ROBOT B)
==> (IN BOX B)
Firing STOP
Halting.
```

Sledování pravidel (`(watch rules)`), přidává informace o pravidel přidaných do (odebraných ze) znalostní báze, např.

```
==> (RULE STOP
  (GOAL MOVE ?OBJECT ?FROM ?TO)
  (IN ?OBJECT ?TO)
  =>
  (HALT)).
```

Po zapnutí sledování agendy (voláním `(watch activations)`, název je kvůli kompatibilitě se systémem CLIPS) budeme navíc informováni o shodách, které do agendy přibyly, nebo z ní byly odstraněny. Výstup programu pak bude následující:

```
==> (MATCH MOVE-ROBOT ((GOAL MOVE BOX A B) (IN BOX A) (IN ROBOT B)))
Firing MOVE-ROBOT
==> (MATCH MOVE-OBJECT ((GOAL MOVE BOX A B) (IN BOX A) (IN ROBOT A)))
==> (MATCH MOVE-ROBOT ((GOAL MOVE BOX A B) (IN BOX A) (IN ROBOT A)))
<== (MATCH MOVE-ROBOT ((GOAL MOVE BOX A B) (IN BOX A) (IN ROBOT A)))
Firing MOVE-OBJECT
==> (MATCH STOP ((GOAL MOVE BOX A B) (IN BOX B)))
Firing STOP
Halting.
```

Každá shoda je zde reprezentována názvem pravidla a posloupností faktů, které byly spárovány s jeho podmínkami. Odtud můžeme snadno odvodit substituci, jež byla při vyhodnocení použita. Negované podmínky nejsou spárovány s žádným konkrétním faktem, proto jsou zde jen tři fakty, přestože pravidlo `move-robot` má podmínky čtyři.

Je zde také vidět, že po aktivaci pravidla `move-robot` se v agendě na chvíli objeví opětovná shoda tohoto pravidla. To je způsobeno tím, že obsah agendy se přepočítává po každé změně pracovní paměti, takže se zde mohou objevit dočasné výsledky.

2.3.7. Undo/redo

Jedním z implementovaných rozšíření původního programu je schopnost vrácení provedených změn. K tomu slouží makra `undo` a `redo`. Ta lze použít k vrácení jakékoli akce s vedlejším efektem, včetně kroků inference. K vypsání zásobníků s akcemi, které je možné vrátit, jsou k dispozici makra `undo-stack` a `redo-stack`.

Pokud například vyhodnotíme prvních 32 řádků programu 1 na straně 12 a zavoláme dvakrát `undo`, bude výpis zásobníků následující (přeformátováno):

```
EXIL-USER> (undo-stack)
  1: (defrule MOVE-ROBOT
       ((GOAL MOVE ?OBJECT ?FROM ?TO)
        (IN ?OBJECT ?FROM)
        (- IN ROBOT ?FROM) (IN ROBOT ?Z)
        =>
        (RETRACT (IN ROBOT ?Z))
        (ASSERT (IN ROBOT ?FROM))))
  2: (deffacts WORLD
       ((IN BOX A) (IN ROBOT B) (GOAL MOVE BOX A B)))
```

```
EXIL-USER> (redo-stack)
  1: (defrule MOVE-OBJECT
       ((GOAL MOVE ?OBJECT ?FROM ?TO)
        ?OBJ-POS <- (IN ?OBJECT ?FROM)
        ?ROB-POS <- (IN ROBOT ?FROM)
        =>
        (RETRACT ?ROB-POS)
        (RETRACT ?OBJ-POS)
        (ASSERT (IN ROBOT ?TO))
        (ASSERT (IN ?OBJECT ?TO))))
  2: (defrule STOP
       ((GOAL MOVE ?OBJECT ?FROM ?TO)
        (IN ?OBJECT ?TO)
        =>
        (HALT)))
```

Vidíme tedy, že jsme vrátili zpět definice pravidel `move-object` a `stop` (ty bychom mohli opět provést voláním `(redo)`). Dalším voláním `(undo)` by pak byla vrácena definice pravidla `move-robot` a poté definice skupiny faktů `world`.

Nemá-li akce žádný vedlejší efekt - např. volání `assert` s faktem, který už v pracovní paměti je, či volání `run` ve chvíli, kdy už není co odvozovat - prázdná akce se na zásobník neuloží.

popsat chování undo v kombinaci s reset

2.3.8. Zpětná inference

- cíle jako patterny, vypsání
- základní inference - nejdřív fakty, pak pravidla, v jakém pořadí vybírá
- alternativní odpovědi - backtracking

Dalším z implementovaných rozšíření je možnost zpětné inference. Inference popsaná v sekci 2.3.5. je dopředná. V každém kroku jsou nalezeny všechny možnosti dalšího postupu odvozování, načež je zvolena jedna, kterou se program dále ubírá. To činí průběh inference značně nedeterministickým. Možnosti postupu, které nebyly vybrány, mohou být navíc dalším postupem ztraceny, pokud aktivace některého pravidla zneplatní podmínky jiného. Míru nedeterminismu můžeme snížit tím, že budeme navrhovat odvozovací pravidla tak, aby se výpočet neubíral nechtěnými cestami. To ale není vždy jednoduché, nebo dokonce možné.

Zpětná inference naproti tomu umožňuje definovat cíle, kterých chceme dosáhnout. K tomu slouží makro `defgoal`, kterému předáme vzor ve stejném formátu, jako u podmínek pravidel. Definice cíle ovšem nepodporuje negaci ani navázání faktu na proměnnou (k tomu ani není důvod).

Ke spuštění zpětné inference pak slouží funkce `back-step` a `back-run`, podobně jako u inference dopředné.

Mějme následující znalostní bázi:

```
(deffacts world
  (have-money))

(defrule buy-car
  (have-money)
  =>
  (retract (have-money))
  (assert (have-car)))

(defrule pay-rent
  (have-money)
  =>
  (retract (have-money))
  (assert (rent-payed))).
```

Spustíme-li dopřednou inferenci, systém nám vesele doporučí nákup auta. Mít nové auto je sice pěkné, hrozí-li nám ale vyhození z pronajatého bytu, není nákup auta pravděpodobně cestou, kterou bychom se chtěli ubírat. Systém by nám v tuto chvíli mohl stejně dobře doporučit správnou cestu. Že bylo vybráno zrovna první pravidlo je výsledkem toho, jak funguje síť RETE, která pravidla vyhodnocuje. Za daných okolností ale nechceme špatnou variantu ani připouštět.

V tomto případě bychom mohli upravit definici programu tak, že do znalostní báze přidáme informaci o cíli, kterou budou pravidla zohledňovat, podobně jako v příkladu 1 na straně 12. Muset ale programovat zohlednění cíle v každém pravidle je přinejmenším otravné. U větších programů to navíc může být velmi náročné, neboť cíl bude třeba programově modifikovat v průběhu výpočtu.

S použitím zpětné inference je problém podstatně jednodušší. Zavoláme-li

```
(reset)
(defgoal (rent-payed))
(back-run),
```

bude výsledkem výstup

```
All goals have been satisfied
(RENT-PAYED) satisfied by (RULE PAY-RENT
  (HAVE-MONEY)
  =>
  (RETRACT (HAVE-MONEY))
  (ASSERT (RENT-PAYED)))
(HAVE-MONEY) satisfied by (HAVE-MONEY).
```

Zde vidíme, že po spuštění zpětné inference nezačal systém bezhlavě provádět akce, ke kterým měl dostatečné prostředky. Místo toho systém uvážil zadaný cíl a jal se hledat akce, které k jeho splnění směřují.

Uvažme nyní složitější příklad (definice šablon vynechána):

```
(deffacts world
  (female jane)
  (male john)
  (parent :parent jane :child george)
  (parent :parent john :child george))

(defrule father-is-male-parent
  (male ?father)
  (parent :parent ?father :child ?child)
  =>
  (assert (father :father ?father :child ?child)))

(defrule mother-is-female-parent
  (female ?mother)
  (parent :parent ?mother :child ?child)
  =>
  (assert (mother :mother ?mother :child ?child)))
```

Zajímá-li nás, kdo je matkou George, můžeme zkusit spustit dopřednou inferenci. Po jejím skončení bude v pracovní paměti jak informace o Georgově matce, tak o jeho otci. Systém se tedy v tomto případě dobral správného výsledku, vypočítal ale i další fakty, které nás nezajímaly. Dokážeme si snadno představit, že ve větším programu může být výpočet všech odvoditelných závěrů velmi výpočetně a tudíž i časově náročný.

Spustíme-li naopak zpětnou inferenci voláním

```
(reset)
(defgoal (mother :mother ?mother-of-george :child george))
(back-run),
```

je výsledkem

```
All goals have been satisfied
(MOTHER (MOTHER . ?MOTHER-OF-GEORGE) (CHILD . GEORGE)) satisfied by
  (RULE MOTHER-IS-FEMALE-PARENT
    (FEMALE ?MOTHER)
    (PARENT (PARENT . ?MOTHER) (CHILD . ?CHILD))
    =>
    (ASSERT (MOTHER MOTHER ?MOTHER CHILD ?CHILD)))
(FEMALE ?MOTHER) satisfied by (FEMALE JANE)
(PARENT (PARENT . JANE) (CHILD . GEORGE)) satisfied by
  (PARENT (PARENT . JANE) (CHILD . GEORGE))
These variable bindings have been used:
((?MOTHER-OF-GEORGE . JANE))
```

Systém tedy vyvodil pouze závěr, který nás zajímal.

Zpětná inference umožnuje také výpočet alternativních odpovědí, tedy dalších cest výpočtu (a vazeb proměnných), které vedou ke splnění všech cílů. Na další alternativní odpověď se dotážeme jednoduše opětovným voláním `(back-run)`.

Zajímají-li nás například oba rodiče George, můžeme zadat cíl

```
(defgoal (parent :parent ?parent :child george)).
```

Pokud poté třikrát zavoláme `(back-run)`, obdržíme výstup

```
All goals have been satisfied
(PARENT (PARENT . ?PARENT) (CHILD . GEORGE)) satisfied by
  (PARENT (PARENT . JANE) (CHILD . GEORGE))
These variable bindings have been used:
((?PARENT . JANE))

All goals have been satisfied
(PARENT (PARENT . ?PARENT) (CHILD . GEORGE)) satisfied by
  (PARENT (PARENT . JOHN) (CHILD . GEORGE))
These variable bindings have been used:
((?PARENT . JOHN))

No feasible answer found.
```

Systém tedy najde obě možné odpovědi (vazby proměnných), vedoucí ke splnění cíle, načež oznámí, že další odpověď už neexistuje.

Zpětná inference nemodifikuje obsah pracovní paměti. To ani není možné vzhledem k tomu, že ve chvíli, kdy inference zvolí pravidlo, jehož důsledky vedou ke splnění aktuálního cíle, nejsou jeho podmínky často ještě splněny. Místo toho pracuje zpětná inference pouze s množinou cílů. Jednotlivé cíle jsou postupně vybírány a hledají se cesty k jejich splnění. Inference nejprve zkoumá fakty v pracovní paměti. Není-li aktuální cíl splněn žádným z platných faktů, uvažuje dále

jednotlivá pravidla. Najde-li pravidlo, které by po aktivaci vedlo ke splnění aktuálního cíle, naváže proměnné použité v jeho aktivacích na jeho podmínky (tedy opačně než u inference dopředné) a ty pak přidá do množiny cílů. Použité vazby proměnných se navíc aplikují i na zbytek cílů.

Zkusme nyní krokovat (použitím `(back-step)`) předchozí příklad s dotazem na matku George s průběžným výpisem cílů pomocí `(goals)`.

```
GOALS: ((MOTHER :MOTHER ?MOTHER-OF-GEORGE :CHILD GEORGE))
(MOTHER (MOTHER . ?MOTHER-OF-GEORGE) (CHILD . GEORGE)) satisfied by
  (RULE MOTHER-IS-FEMALE-PARENT
    (FEMALE ?MOTHER)
    (PARENT (PARENT . ?MOTHER) (CHILD . ?CHILD))
    =>
    (ASSERT (MOTHER MOTHER ?MOTHER CHILD ?CHILD)))

GOALS: ((FEMALE ?MOTHER) (PARENT :PARENT ?MOTHER :CHILD GEORGE))
(FEMALE ?MOTHER) satisfied by (FEMALE JANE)

GOALS: ((PARENT :PARENT JANE :CHILD GEORGE))
(PARENT (PARENT . JANE) (CHILD . GEORGE)) satisfied by
  (PARENT (PARENT . JANE) (CHILD . GEORGE))
```

V prvním kroku je proměnná `?mother-of-george`, použitá v definici cíle, nahrazena proměnnou `?mother`, použitou v důslecích pravidla `mother-if-female-parent`. Proměnná `?child` v důsledcích je navázána na `george` a touto vazbou jsou nahrazeny výskyty proměnné v podmínkách pravidla. Podmínky jsou poté přidány do množiny cílů.

V druhém kroku je nový cíl `(female ?mother)` porovnán s faktem `(female jane)` v pracovní paměti, je jím splněn a v posledním cíli je proměnná `?mother` navázána na `jane`. Tento cíl je pak v posledním kroku triviálně splněn identickým faktem.

- pouze assert v důsledcích
- zpětná inference nemusí nutně snížit míru nedeterminismu, pokud existuje hodně pravidel, která splní daný subcíl

2.3.9. CLIPSová syntax

- deftemplate, fact specifiery
- volání facts s číslem
- projít příručku clipsu a vzpomenout si, co v ExiLu ještě nebylo

2.3.10. Reset prostředí

- durable/volatile slots
- clean, reset, complete reset (neměl by se jmenovat complete clean?)

2.3.11. Práce s více prostředími

2.3.12. Grafické uživatelské rozhraní

2.4. Referenční příručka

- reprezentace objektů vracené findery + názvy vracené makry jako rules či current strategy - proč klíčky - různé package
- kombinace s funkčními alternativami.
- clipsová syntax - deftemplate, specifikace faktů, volání facts s čísly, singleton proměnná

ExiL umožňuje definici vlastní strategie makrem `defstrategy`. Makro bere jako parametry název strategie a výběrovou funkci. Výběrové funkci je předána agenda (množina shod), z nichž funkce musí jednu vrátit. Definice vlastní strategie ovšem není v praxi příliš použitelná, neboť pro její implementaci je třeba znát vnitřní implementaci shody. Definice mohou vypadat například takto:

```
(defmethod newer-than-p ((match1 match) (match2 match))
  (> (timestamp match1) (timestamp match2)))

(defmethod simpler-than-p ((rule1 rule) (rule2 rule))
  (< (length (conditions rule1))
     (length (conditions rule2))))
(defmethod simpler-than-p ((match1 match) (match2 match))
  (simpler-than-p (match-rule match1) (match-rule match2)))

(defstrategy breadth-stragegy #'newer-than-p)
(defstrategy simplicity-strategy #'simpler-than-p)
```

Symboly názvů metod `timestamp`, `conditions` a `match-rule` jsou navíc interní v jiném package, než běžně uživatel pracuje, bylo by tedy třeba je plně kvalifikovat.

2.5. Implementace

Implementace or v podmínkách pravidla by vyžadovala výraznou změnu v algoritmu, který vytváří síť RETE, neboť join node je problematické znovu využít. Implementace forall by vyžadovala změnu vyhodnocování vazeb proměnných a celkově join algoritmu. Zvážit problémy implementace všech podobných rozšíření - or, and, negace celé konjunkce či disjunkce, exists, forall, viz http://www.csie.ntu.edu.tw/~sylee/courses/clips/bpg/node5.4.html

- architektura programu (objektový návrh, nedostatky Lispu, je environment jako god class důsledkem těchto problémů?)
- síť RETE, její vytváření → výhody - sdílení uzlů
- implementovaná rozšíření
  - undo/redo - kopie sítě rete, testování ekvivalence
  - zpětné řetězení - zásobníky, backtracking, obchází rete
  - kompozitní podmínky (not, and, or, forall) - neimplementováno - popsat, jak výrazné změny rete by vyžadovalo a jak by se projevilo na efektivitě (hlavní výhodou rete je sdílení uzlů, které by zde bylo dost problematické)
  - syntaktický mód - parser, implementace složených podmínek pro atom (~asdf&asf) by vyžadovalá podobné změny jako implementace kompozitních podmínek
  - gui - capi, propojení s environmentem - notify

3. Teoretická část

3.1. Expertní systémy

Reference

[1] Jackson, P.: *Introduction to Expert Systems*. Addison Wesley, 1998, ISBN 0-201-87686-8.

[2] Norvig, P.: *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers, 1991, ISBN 1-55860-191-0.

[3] Doorenbos, R. B.: *Production Matching for Large Learning Systems*. Dizertační práce, Carnegie Mellon University, 1995.
http://reports-archive.adm.cs.cmu.edu/anon/1995/CMU-CS-95-113.pdf

[4] Siebel, P.: *Practical Common Lisp*. Apress, 2005, ISBN 1-59059-239-5.
http://www.gigamonkeys.com/book/

[5] CLIPS: *Tool for Building Expert Systems*. 2013.
http://clipsrules.sourceforge.net/OnlineDocs.html

[6] LispWorks Ltd.: *Common Lisp HyperSpec*. 2005.
http://www.lispworks.com/documentation/HyperSpec/Front/

[7] Expert system — Wikipedia, The Free Encyklopedia. 2013.
http://en.wikipedia.org/wiki/Expert_system

[8] Rete algorithm — Wikipedia, The Free Encyklopedia. 2013.
http://en.wikipedia.org/wiki/Rete_algorithm