

PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO  
KATEDRA INFORMATIKY

## DIPLOMOVÁ PRÁCE

Implementace expertního systému v jazyce Common Lisp



## **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

<b>1. Úvod</b>	<b>4</b>
<b>2. Praktická část</b>	<b>5</b>
2.1. Instalace . . . . .	5
2.1.1. Získání zdrojového kódu . . . . .	5
2.1.2. Prerekvizity . . . . .	5
2.1.3. Načtení knihovny . . . . .	6
2.2. Common Lisp . . . . .	8
2.3. Uživatelská příručka . . . . .	9
2.3.1. Základní pojmy . . . . .	9
2.3.2. Struktura programu . . . . .	10
2.3.3. Definice znalostní báze . . . . .	13
2.3.4. Modifikace pracovní paměti . . . . .	15
2.3.5. Inference . . . . .	15
2.3.6. Reset prostředí . . . . .	18
2.3.7. Sledování průběhu inference . . . . .	18
2.3.8. Undo/redo . . . . .	18
2.3.9. Zpětné řetězení . . . . .	19
2.3.10. Práce s více prostředími . . . . .	19
2.3.11. Grafické uživatelské rozhraní . . . . .	19
2.4. Referenční příručka . . . . .	20
2.5. Implementace . . . . .	21
<b>3. Teoretická část</b>	<b>22</b>
3.1. Expertní systémy . . . . .	22
<b>Reference</b>	<b>23</b>

## Seznam obrázků

## Seznam příkladů

1	Základní struktura exilového programu . . . . .	12
---	---	----

# 1. Úvod

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 diplomové práce je tuto 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<sup>TM</sup>,
- rozšíření odvozovacího aparátu o základní zpětné řetězení.

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.

Jazyk Common Lisp<sup>1</sup> (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 stručný, snadno pochopitelný a tudíž jednoduše rozšiřitelný.

Syntax systému CLIPS<sup>2</sup> byla zvolena proto, že jde o reálně používaný systém<sup>3</sup>, 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 množství teorie nutné pro pochopení aktuálního problému, načež se k ní v teoretické části textu vrátím, pojmy zadefinuji přesně a rozšířím o souvislosti.

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Common\\_Lisp](http://en.wikipedia.org/wiki/Common_Lisp)

<sup>2</sup><http://clipsrules.sourceforge.net>

<sup>3</sup><http://clipsrules.sourceforge.net/FAQ.html#Q6>

## 2. Praktická část

Tato sekce popisuje knihovnu ExiL<sup>1</sup>, která je výsledkem této diplomové práce. Nejprve popíšu její instalaci, pak v uživatelské příručce nastíním základní 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. Instalace

#### 2.1.1. Získání zdrojového kódu

Zdrojový kód knihovny je přiložen k této diplomové práci a lze jej také získat zklonováním<sup>2</sup> gitového<sup>3</sup> 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.

#### 2.1.2. Prerekvizity

Pro práci s knihovnou ExiL potřebujeme lispový interpreter<sup>4 5</sup>, 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<sup>6</sup>, což je plugin pro textový editor GNU Emacs<sup>7</sup> (poskytující mimo jiné pomůcky pro editaci lispového zdrojového kódu, REPL<sup>8</sup> a debugger<sup>9</sup>) s interpreterem SBCL<sup>10</sup> 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>.

---

<sup>1</sup>TODO: původ názvu

<sup>2</sup><http://git-scm.com/docs/git-clone>

<sup>3</sup><http://git-scm.com/>

<sup>4</sup>[http://en.wikipedia.org/wiki/Interpreter\\_\(computing\)](http://en.wikipedia.org/wiki/Interpreter_(computing))

<sup>5</sup>lispové interpretery jsou většinou zároveň kompilátory<sup>6</sup>, označením interpreter tedy budu nazývat obojí

<sup>6</sup><http://en.wikipedia.org/wiki/Compiler>

<sup>6</sup><http://www.common-lisp.net/project/slime/>

<sup>7</sup><http://www.gnu.org/software/emacs/>

<sup>8</sup>[http://en.wikipedia.org/wiki/Read-eval-print\\_loop](http://en.wikipedia.org/wiki/Read-eval-print_loop)

<sup>9</sup><http://en.wikipedia.org/wiki/Debugger>

<sup>10</sup><http://www.sbcl.org/>



[net/projects/slime/doc/html/Installation.html](http://net/projects/slime/doc/html/Installation.html). Prostředí poté můžeme v Emacsu spustit voláním příkazu `slime` (`M-x slime<enter>`). Při prvním spuštění se kód prostředí kompiluje, což může chvíli trvat, pak už se v editoru otevře buffer<sup>11</sup> s lispovým REPLEm.

Knihovnu jsem testoval také ve vývojovém prostředí LispWorks®<sup>12</sup> Personal Edition 6.1, pro které jsem také vytvořil minimalistické grafické uživatelské rozhraní. Součástí prostředí LispWorks je i lispový interpret. Prostředí můžeme nainstalovat podle návodu zde [lispworksinstallation](#).

Pro efektivní načtení knihovny včetně závislostí potřebujeme ještě dvě knihovny:

- ASDF<sup>13</sup> je knihovna umožňující snadnou definici struktury projektu a jeho dávkové načtení,
- quicklisp<sup>14</sup> 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.

### 2.1.3. 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 2.3.10.

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:

---

<sup>11</sup>emacs buffer

<sup>12</sup><http://www.lispworks.com/>

<sup>13</sup><http://common-lisp.net/project/asdf>

<sup>14</sup><http://www.quicklisp.org/beta/>

```
(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.

## 2.2. Common Lisp

- základní znalosti lispu nutné pro používání knihovny
  - 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)
- odkaz na practical common lisp, clhs

## 2.3. Uživatelská příručka

### 2.3.1. Základní pojmy

- TODO: v teoretické části uvést pojmy znovu s citacemi
- TODO: production memory v exilu splývá s pravidly znalostní báze - na příhodném místě uvést, že v teorii a některých systémech se rozlišuje

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 chápeme tak, jak jsem jej představil v úvodu práce. V praktické části rozeberu pojem v potřebné šíři.

<b>fakt</b>	elementární statická znalost - tvrzení
<b>(odvozovací) pravidlo</b>	elementární odvozovací znalost - pokud víme, že (ne)platí nějaká tvrzení, můžeme odvodit, že platí i nějaká další
<b>znalost (v ExiLu)</b>	množina faktů a pravidel
<b>znalostní báze</b>	výchozí znalost expertního systému
<b>pracovní paměť</b>	aktuální množina faktů
<b>inference</b>	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 lisových výrazů, které jsou při aktivaci pravidla vyhodnoceny. Tyto dvě množiny jsou od sebe odděleny *symbol* `=>`.

Podmínky odvozovacích pravidel jsou ve formě vzorů (*pattern*). Struktura vzorů je stejná jako struktura faktů (viz sekce 2.3.3.), ale narozdíl od nich mohou obsahovat proměnné (symboly začínající otazníkem). Při vyhodnocování podmínek pravidla je zajištěna 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 úpravů (`modify`) faktů v ní. Nemusí tomu tak ale být vždycky - důsledkem aktivace pravidla může být např. vypsaní 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 odstranit 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. Details 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 přibývší do pracovní paměti, řádky začínající <== fakty z paměti odstraněné. Tento výstup obdržíme pouze pokud zapneme sledování faktů voláním (**watch facts**) (viz sekce 2.3.7.). 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 nezabývá nutností 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.

---

```
1  ;;; definition of knowledge base
2  ;; facts
3  (deffacts world
4    (in box A)
5    (in robot B)
6    (goal move box A B))
7
8  ;; inference rules
9  (defrule move-robot
10   (goal move ?object ?from ?to)
11   (in ?object ?from)
12   (- in robot ?from)
13   (in robot ?z)
14   =>
15   (retract (in robot ?z))
16   (assert (in robot ?from)))
17
18  (defrule move-object
19   (goal move ?object ?from ?to)
20   ?rob-pos <- (in robot ?from)
21   ?obj-pos <- (in ?object ?from)
22   =>
23   (retract ?rob-pos)
24   (retract ?obj-pos)
25   (assert (in robot ?to))
26   (assert (in ?object ?to)))
27
28  (defrule stop
29   (goal move ?object ?from ?to)
30   (in ?object ?to)
31   =>
32   (halt))
33
34  ;;; initialization of working memory
35  (reset)
36
37  ;;; inference execution
38  (run)
```

---

Příklad 1: Základní struktura exilového programu

### 2.3.3. Definice znalostní báze

Přesunout přílišné podrobnosti do referenční příručky

ExiL, stejně jako CLIPS, rozlišuje dva typy faktů - jednoduché (*simple*, *ordered*) a strukturované (*templated*). Struktura jednoduchého faktu je udána pouze pořadím *atomů*, typickou volbou je např. `objekt-atribut-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 *funkce* `templates` (bez parametrů). 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 funkce `fact-groups` (bez parametrů). 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 lisových 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 vypsát 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 paměti 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 fakty.

Ta lze volat buď před započítím inference (ale po volání **reset**, neboť to dočasné 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 **defacts** (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.).

Všechny fakty můžeme z pracovní paměti odebrat voláním funkce **retract-all** (bez parametrů). 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 funkce **step** (bez parametrů), nebo voláním funkce **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 funkce **halt** (bez parametrů) 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. 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 faktu (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. Např. 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 pravidla některá proměnná vícekrát, musí mít odpovídající fakty na daných pozicích stejný atom. Mějme např. 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 nevádí, 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 např. 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 s 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 s 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 podmínkách není třeba tyto sloty uvádět, neboť `?` je výchozí hodnotou slotu vzoru.

Posledním speciálním konstruktem je navázání celého faktu na proměnnou. Např. 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 s vzorem podmínky spárován.

Pravidlo může být splněno několika posloupnostmi faktů - pojem matche

Je-li při vyhodnocování podmínek nalezeno více splněných pravidel, je třeba z nich jedno vybrat k aktivaci. Výběr pravidla záleží na zvolené strategii. ExiL poskytuje následující strategie výběru pravidla:

#### **depth-strategy**

- fáze podrobně
  - výběr pravidla - strategie
  - aktivace - vyhodnocení důsledků (typicky modifikace pracovní paměti) navázání proměnných, eval
- spuštění inference, krokování (může se prolínat s ruční modifikací w.m.)
- queries - agenda, strategies

#### **2.3.6. Reset prostředí**

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

#### **2.3.7. Sledování průběhu inference**

- watchery

#### **2.3.8. Undo/redo**

- lze použít na všechny funkce/makra s vedlejším efektem

- pokud funkce nemá vedlejší efekt (fakt neexistuje, apod.), nezapiše se undo step
- queries - undo-stack, redo-stack

### **2.3.9. Zpětné řetězení**

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

### **2.3.10. Práce s více prostředími**

### **2.3.11. Grafické uživatelské rozhraní**

## **2.4. Referenční příručka**

## 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>



### **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](http://en.wikipedia.org/wiki/Expert_system)
- [8] Rete algorithm — Wikipedia, The Free Encyklopedia. 2013.  
[http://en.wikipedia.org/wiki/Rete\\_algorithm](http://en.wikipedia.org/wiki/Rete_algorithm)