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
  - výjimky
  - dynamické proměnné
- 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 2.3.10.

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:

```
(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, nikoli na následujícím pojmu znalost, jak jej 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 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 (jak vidíme v příkladu 2), 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 2.3.6.). 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.

Vysvětlit interferenci

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, zvláště 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. Příklad 2 na straně 15 ukazuje definici znalostní báze ekvivalentní příkladu 1 s použitím strukturovaných faktů.

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`.

```
(deftemplate goal (action :default move) object from to)
(deftemplate in object location)

(deffacts world
  (in :object robot :location A)
  (in :object box :location B)
  (goal :object box :from B :to A))

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

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

(defrule stop
  ?goal <- (goal :object ?obj :to ?to)
  (in :object ?obj :location ?to)
  =>
  (retract ?goal))
```

Příklad 2: Definice znalostní báze s použitím strukturovaných faktů

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 *substitucí* proměnných navázaných při vyhodnocování jeho podmínek. Substituci 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 důsledcích.

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 pravidlech 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.

2.3.8. Zpětná inference

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). Cíle je pak možné vypsat voláním `(goals)`. Cíl můžeme také odebrat makrem `undefgoal`. Ke spuštění zpětné inference 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 hledání 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 nemění 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 důsledcích, podle vzoru cíle. Tyto vazby poté aplikuje 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. Původní cíl je poté z množiny odstraněn.

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.

Aktuální stav výpočtu pomocí zpětné inference je ztracen, zavoláme-li během krokování `defgoal` nebo `undefgoal`. Kdyby tomu tak nebylo, mohli bychom systém uvést do nekonzistentního stavu.

Implementace zpětné inference v ExiLu je velmi omezená. V důsledcích pravidel zohledňuje pouze specifikace faktů ve voláních `assert`. Nelze ji tedy aplikovat u pravidel, která fakty z pracovní paměti odstraňují, či je modifikují. Inference také neumí pracovat s negovanými cíli, tudíž ani s pravidly, která mají negované podmínky (neboť tyto by při použití pravidla mezi cíli objevily).

Použití zpětné inference nemusí nutně snížit míru nedeterminismu výpočtu. Existuje-li několik pravidel, která vedou ke splnění aktuálního cíle, bude výpočet opět nedeterministický. Na rozdíl od dopředné inference lze ale alternativní cesty výpočtu postupně prohledat. To je možné jednak díky backtrackingu - nevede-li daná cesta ke splnění všech cílů, výpočet se vrátí a zkusí se ubírat jinudy. Dále díky možnosti dotázat se na alternativní odpovědi.

2.3.9. Reset prostředí

Entitu, která uchovává aktuální stav systému, nazývám v ExiLu prostředím. Hodnoty tvořící aktuální stav prostředí rozděluji do dvou skupin. První skupinu tvoří hodnoty trvalé. Sem spadají definice šablon, znalostní báze (skupiny faktů, pravidla), definice strategií pro volbu shod a aktuálně zvolená strategie.

Druhou skupinou jsou hodnoty dočasné. Sem patří aktuální stav pracovní paměti, aktuální cíle, stav sítě RETE spolu s agendou udržující aktuální shody (splněná pravidla spolu s vazbami proměnných), hodnoty zásobníků sloužících k vrácení provedených akcí a jejich opětovné provedení (undo/redo) a hodnota zásobníku udržujícího aktuální stav zpětné inference (aby bylo možné ji krokovat nebo se dotázat na alternativní odpovědi).

Dočasné hodnoty lze uvést do výchozího stavu voláním `(clear)`. Tím systém uvedeme do stavu, kdy zná pouze definované šablony, skupiny faktů a odvozovací pravidla. Volání `(reset)` provede `(clear)` následovaný zavedením všech skupin faktů do pracovní paměti. Všechny hodnoty prostředí, tedy včetně trvalých, lze pak uvést do výchozího stavu voláním `(complete-reset)`.

Chování `reset` a `clear` vzhledem k zásobníkům pro vracení akcí může být poněkud překvapivé. Makro `reset` totiž vymaže obsah tohoto zásobníku, poté na něj však uloží akci k vrácení jeho volání. Z výpisu `(undo-stack)` tak po volání `reset` zmizí předchozí akce, uvidíme zde jen akci `(reset)`. Po volání `(undo)` se na zásobníku předchozí akce opět objeví. Totéž platí pro volání `(clear)`.

2.3.10. Práce s více prostředími

Při práci s ExiLem se nemusíme omezovat pouze na jedno prostředí (byť si s ním často vystačíme). Nové prostředí lze definovat voláním `defenv` s názvem prostředí, např. `(defenv test)`. Prostředí pak lze přepnout voláním `(setenv test)`, případně smazat voláním `(undefenv test)`.

Každé prostředí má oddělený stav, tedy trvalé i dočasné hodnoty (viz předchozí sekce). Na název aktuálního prostředí se lze dotázat voláním `(current-environment)`. Název výchozího prostředí je `default`. Seznam všech prostředí získáme voláním `(environments)`.

Máme-li už definované prostředí, např. `test`, opětovné volání `(defenv test)` skončí výjimkou. Tím je zajištěno, že si omylem nevymažeme celé prostředí. Chceme-li jej opravdu vymazat, musíme volat `(defenv test :redefine t)`.

2.3.11. Volání ExiLu z jiného kódu a naopak

Doposud jsem v ukázkách práce s ExiLem pracoval většinou s makry. Byla to následující:

definice šablon	`deftemplate undeftemplate find-template`
definice skupin faktů	`deffacts undeffacts find-fact-group`
definice pravidel	`defrule undefrule find-rule`
modifikace pracovní paměti	`assert retract modify`
definice cílů	`defgoal undefgoal`
sledování průběhu inference	`watch unwatch watchedp`
strategie výběru shody	`defstrategy setstrategy`
definice prostředí	`defenv undefenv setenv`

Tato makra berou jako parametry symboly a/nebo seznamy a tyto automaticky *quotují*. To je pohodlné, pracujeme-li s knihovnou přímo. Představme si ale, že chceme knihovnu volat z jiného kódu a například specifikace faktů, které předáváme makru `assert`, generovat nějakou funkcí.

Protože makro `assert` seznamy se specifikacemi faktů quotuje, místo aby je vyhodnotilo, nelze ho v tomto případě použít. Výsledkem volání

```
(assert (generate-fact))
```

totiž bude přidání faktu `(generate-fact)` do pracovní paměti. K tomuto účelu poskytuje ExiL ke všem uvedeným makrům funkční alternativy, které parametry vyhodnocují. Tyto jsou označeny suffixem `f`, například `assertf`. Tímto suffixem sice v Lispu typicky označujeme destruktivní makra, která mění svůj argument, v případě exilových maker ale záměna nehrozí.

Dotazovací funkce a makra jako `facts`, `goals`, `find-fact-group`, apod. navíc nevypisují hodnoty na výstup, nýbrž vrací externí reprezentaci objektů, se kterou je možné dále manipulovat a pak ji třeba systému předat zpátky. To umožnuje například následující (nepříliš užitečné) volání:

```
(dolist (fact (facts))
  (retractf fact)
  (assertf (cons 'my-fact fact))).
```

Funkční alternativy jako `assertf` také umožnují použití složitějších konstruktů v důsledcích pravidla. Bude-li například podmínka pravidla

```
(defrule surround-by-as
  (palindrome ?p)
  =>
  (assertf `(palindrome ,(concatenate 'string "a" ?p "a"))))
```

splněna faktem `(palindrome "b")`, přibyde po jeho aktivaci do pracovní paměti fakt `(palindrome "aba")`. Funkci `assertf` bohužel nelze použít se zpětnou inferencí, neboť ta nemá šanci předvídat, jaký fakt bude jejím voláním do pracovní paměti přidán.

V důsledcích pravidla bychom také mohli chtít například upozornit jinou část programu na událost, ke které došlo. Protože ExiL nahrazuje výskyty proměnných v celé důsledkové části pravidla, lze toho snadno dosáhnout. Je ale třeba dát pozor na quotování hodnot proměnných. Uvažme pravidlo

```
(defrule move-robot
  (goal move robot ?from ?to)
  (in robot ?from)
  =>
  (retract (in robot ?from))
  (assert (in robot ?to))
  (notify 'moving-robot '?from '?to))
```

a fakt `(goal move robot A B)` v pracovní paměti. Kdybychom ve volání `notify` proměnné `?from` a `?to` nequotovali, volání by se při aktivaci pravidla vyhodnotilo jako `(notify 'moving-robot A B)`. To by pravděpodobně skončilo chybovou hláškou sdělující, že proměnné `A` a `B` nejsou definovány.

Makra `deftemplate`, `deffacts`, `defrule` a `modify` berou specifikace slotů šablony, faktů, těla pravidla a změn k provedení (v tomto pořadí) jako další parametry. Díky tomu nemusíme tyto parametry obalovat do dalšího seznamu. Jejich funkční alternativy naproti tomu očekávají tyto parametry v jednom seznamu, například

```
(deffactsf 'world (list '(in box A) '(in robot B))).
```

To umožňuje snazší generování těchto specifikací funkcemi.

2.3.12. Kompatibilita se systémem CLIPS

Dalším z požadavků zadání práce bylo přiblížit syntax exilových volání systému CLIPS, aby bylo možné programy v něm napsané snáze převést na programy exilové. Toho se mi podařilo dosáhnout jen částečně.

Systém CLIPS používá jiný formát specifikací slotů šablony, strukturovaných faktů a požadovaných změn při volání `modify`. Tuto syntax nyní ExiL podporuje také. Příklad 3 na straně 31 ukazuje definici znalostní báze ekvivalentní příkladu 2 s použitím CLIPSové syntaxe. Syntax je dostatečně odlišná na to, aby ji ExiL

rozpoznal, není tedy třeba syntaktický mód nijak přepínat. Díky tomu dokonce můžeme oba typy syntaxe kombinovat.

ExiL také po vzoru CLIPSu umožňuje omezit seznam vrácený voláním `(facts)` volitelnými číselnými parametry. První volitelný parametr udává index prvního faktu v seznamu (číslováno od 1). Druhý parametr udává index posledního faktu. Třetí parametr pak maximální počet vrácených faktů.

Makra `assert` a `retract` také nyní umožňují přidání či odebrání více faktů najednou. Makru `retract` lze navíc místo specifikací faktů k odstranění předat jejich číselné indexy v seznamu faktů. Obě možnosti lze dokonce kombinovat.

V podmínkách pravidel lze nyní také použít speciální proměnnou `?` a navázání celého faktu na proměnnou pomocí operátoru `<-`, jak jsem již popsal v sekci 2.3.5.

CLIPS dále umožňuje specifikovat u slotu šablony datový typ. To považuji v dynamickém jazyce, jakým je Common Lisp, za zbytečné. CLIPS také nabízí několik vlastních funkcí a možnost definice nových, které pak lze používat v důsledcích pravidel. To nemá v ExiLu smysl, neboť můžeme použít vestavěné nebo uživatelsky definované funkce Lispu.

CLIPS navíc poskytuje možnost definovat některé sloty šablony jako *multislot*. Ve vzorech podmínek pravidel pak můžeme používat proměnnou `$?`, která se naváže na jeden nebo více atomů multislotu, případně jednoduchého faktu. Dále je v podmínkách CLIPSu možné používat libovolně vnořené agregační funkce `and`, `or`, `not` a podobně. Ve vzorech podmínek CLIPSových pravidel lze také používat speciální proměnné, které se shodují jen s některými atomy, např. proměnná `?color&~red&~yellow` se shoduje se všemi atomy, kromě `red` a `yellow`.

Tyto možnosti ExiL neposkytuje, neboť jejich implementace by vyžadovala přepsat velkou část algoritmu RETE, který podmínky pravidel vyhodnocuje, jak popíšu v sekci zabývající se implementací.

```
(deftemplate goal
  (slot action (default move))
  (slot object)
  (slot from)
  (slot to))

(deftemplate in
  (slot object)
  (slot location))

(deffacts world
  (in (object robot) (location A))
  (in (object box) (location B))
  (goal (object box) (from B) (to A))).

(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 ?z))
  =>
  (modify ?robot (location ?from)))

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

(defrule stop
  ?goal <- (goal (action move) (object ?obj) (to ?to))
  (in (object ?obj) (location ?to))
  =>
  (halt))
```

Příklad 3: Definice znalostní báze s použitím CLIPSové syntaxe

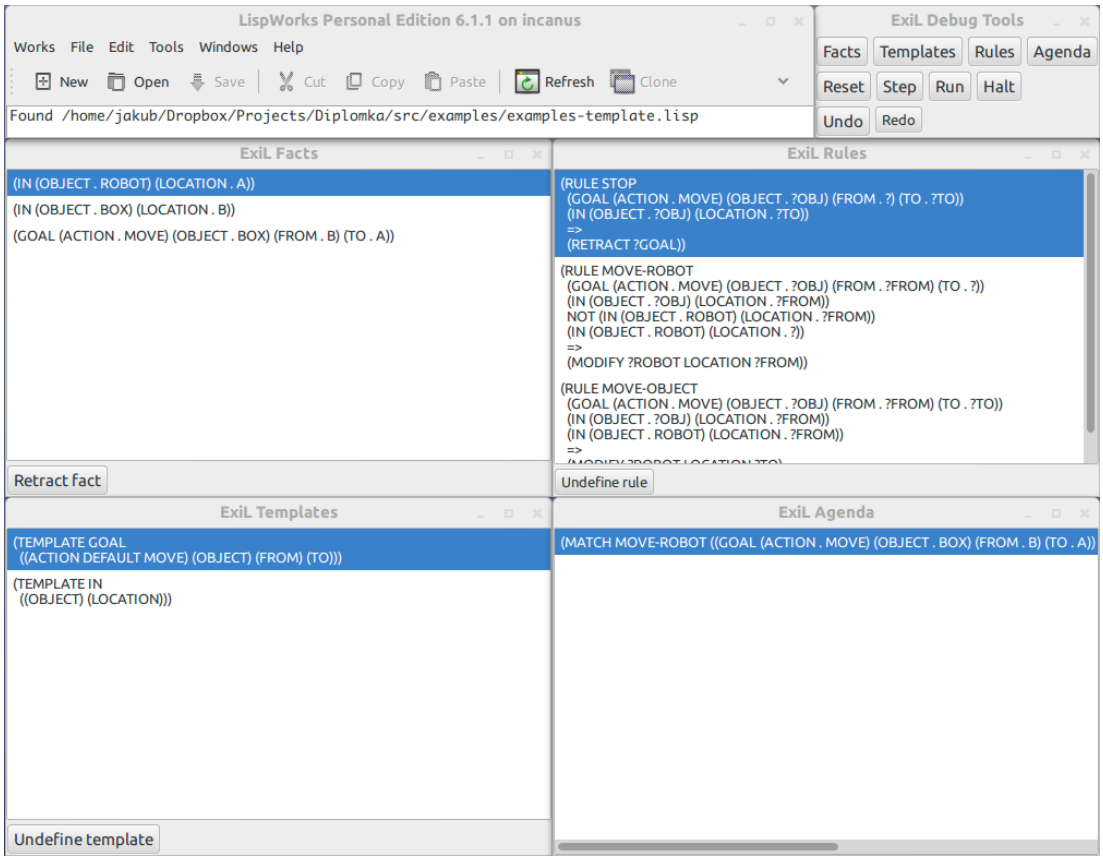
2.3.13. Grafické uživatelské rozhraní

Pro prostředí LispWorks jsem k ExiLu implementoval minimalistické grafické uživatelské rozhraní, zobrazené na obrázku 1. To sestává z hlavního okna s deseti tlačítky organizovanými do tří řad (vpravo nahoře).

První řada tlačítek - „Facts“, „Templates“, „Rules“ a „Agenda“ slouží k zobrazení podoken s jednotlivými položkami. Okno „Facts“ zobrazuje seznam faktů v pracovní paměti a umožňuje jejich odebrání tlačítkem „Retract fact“. Okno „Templates“ zobrazuje seznam definovaných šablon a umožňuje jejich odebrání tlačítkem „Undefine template“. Okno „Rules“ zobrazuje seznam definovaných odvozovacích pravidel a umožňuje jejich odebrání tlačítkem „Undefine rule“. Poslední okno „Agenda“ zobrazuje seznam aktuálních shod v agendě. Seznamy v oknech se automaticky obnovují při každé změně zobrazených hodnot.

Druhá řada tlačítek - „Reset“, „Step“, „Run“ a „Halt“ umožňuje řízení inference. Tlačítka volají stejnojmenné funkce ExiLu.

Poslední řada tlačítek - „Undo“ a „Redo“ slouží k vracení a opětovnému provedení akcí voláním stejnojmenných funkcí ExiLu.



Obrázek 1. Grafické uživatelské rozhraní

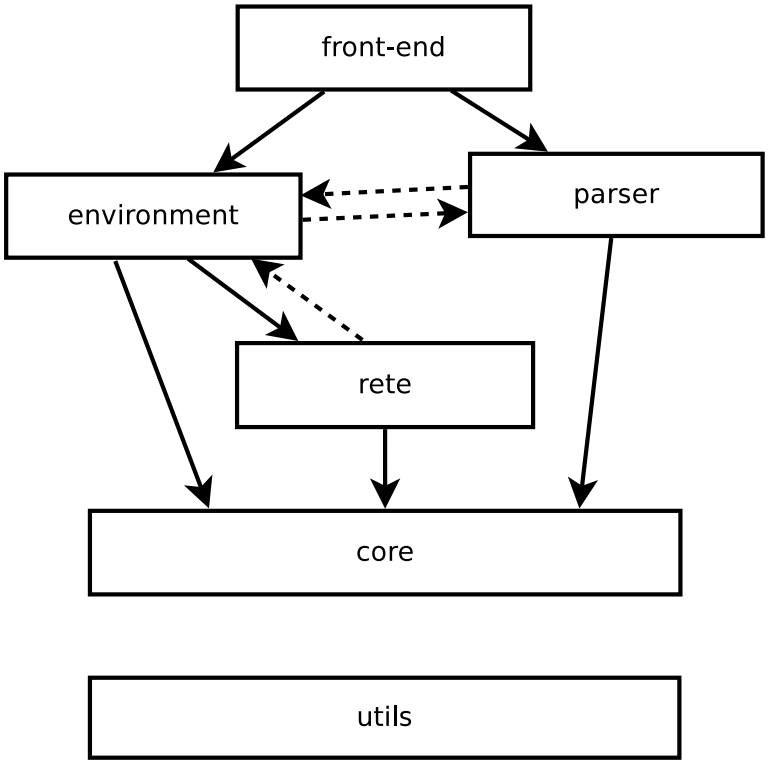
Rozhraní lze zobrazit voláním `(exil-gui:show-gui)`. Každé prostředí má vlastní rozhraní. Volání `show-gui` bez parametru zobrazí rozhraní k aktuálnímu prostředí. Jako volitelný parametr můžeme funkci předat název prostředí, jehož rozhraní chceme zobrazit. Máme-li tedy definováno více prostředí, můžeme si ke každému z nich zobrazit uživatelské rozhraní.

2.4. Implementace

Zmínit použité knihovny - iterate, xlunit

2.4.1. Architektura programu

V následující sekci budu používat obecný pojem *modul* pro ohraničenou část programu, implementující nějakou část funkcionality. Veřejné rozhraní modulu definuje vstupní body, pomocí nichž s ním mohou ostatní moduly komunikovat. Ty jsou zde reprezentovány funkcemi, makry a metodami, které modul poskytuje a specifikacemi jejich parametrů. V Common Lispu sice tuto entitu nazýváme *package*, tento pojem se však špatně skloňuje a celkově narušuje plynulost textu.



Obrázek 2. Architektura ExiLu

Obrázek 2. zobrazuje architekturu knihovny ExiL, tedy moduly, do kterých je kód knihovny rozdělen a jejich vzájemnou komunikaci. Orientace šipek určuje směr volání funkcí (metod, maker) poskytovaných moduly, např. modul `front-end` volá funkce modulu `parser` a ne naopak. Těmito voláními jsou zároveň definovány závislosti mezi moduly, ty mají tedy stejnou orientaci.

Modul `utils` definuje různorodé pomocné funkce a makra, která jsou volána všemi ostatními moduly. V obrázku 2. bych tedy mohl zahrnout šipky ze všech

ostatních modulů do modulu `utils`, to by jej však činilo zbytečně nepřehledným. Většina funkcí a maker, která modul definuje, usnadňuje nízkoúrovňovou práci se symboly a datovými strukturami - (asociativními) seznamy, množinami, stromy, *hashovacími tabulkami* apod.

Modul `core` definuje základní objekty, s nimiž zbytek knihovny pracuje. Ty reprezentují fakty, vzory, šablony a pravidla. Fakta a vzory jsou definovány ve dvou variantách - jednoduché a strukturované (*templated*), jak jsem uvedl v uživatelské příručce.

Modul `rete` implementuje algoritmus RETE, který slouží k efektivnímu vyhodnocování podmínek odvozovacích pravidel. Algoritmus je implementován *dataflow* sítí, jejíž fungování popíšu detailně v sekci 2.4.2. Algoritmus RETE je sice nejkomplexnější částí ExiLu, veřejné rozhraní modulu `rete` je však velmi jednoduché.

Modul poskytuje dvojici metod kterými síť RETE upozorníme na přidání nebo odebrání pravidel. Ty dle potřeby doplní síť o uzly potřebné k vyhodnocení podmínek těchto pravidel, případně uzly při odebrání pravidla odstraní.

Další dvojice metod upozorní síť na přidání faktu do (resp. odebrání z) pracovní paměti. Síť pak přepočítá splněné podmínky pravidel a případně upozorní prostředí na nově přibyvší nebo rozbité shody (*broken match* - pravidla, která byla splněná a už nejsou).

Modul `environment` definuje stejnojmenný objekt, který udržuje průběžný stav prostředí a řídí průběh inference. Hodnoty, které stav prostředí tvoří, jsem popsal v sekci 2.3.9. Prostředí udržuje krom jiného *referenci* na síť `rete`, kterou upozorňuje na nastalé změny.

Modul `parser` zajišťuje vytváření `core` objektů z externí reprezentace (té, kterou předáváme makrům při práci s knihovnou). Modul rozpoznává jak základní, tak CLIPSovou syntax, takže o to, kterou z nich uživatel používá se zbytek kódu nemusí dále starat.

Při parsování strukturovaných faktů či vzorů se `parser` dotazuje aktuálního prostředí na definici šablony. Při zpětné inferenci prostředí vyhodnocuje volání `assert` v důsledcích pravidel a žádá `parser` o zparsování reprezentací faktů v nich. Proto jsem mezi moduly `environment` a `parser` přidal šrafované šipky. Kromě těchto nutných interakcí spolu však moduly nekomunikují.

Modul `front-end` udržuje seznam definovaných prostředí a ví, které je právě aktivní. Kromě manipulace tohoto seznamu modul sám žádnou funkcionalitu neimplementuje. Pouze quotuje parametry předané makrům, předává je `parser`u a výsledné objekty (se zbytkem parametrů) pak aktivnímu prostředí.

Do diagramu jsem nezahrnul modul `gui`. Ten implementuje grafické uživatelské rozhraní a poskytuje jedinou metodu - `show-gui`. Rozhraní se dotazuje prostředí, se kterým je svázáno, na hodnoty aktuálního stavu a je jím upozorněno, pokud se tyto změní.

2.4.2. Algoritmus RETE

Algoritmus RETE slouží k efektivnímu vyhodnocování splnění podmínek odvozovacích pravidel (dále jen vyhodnocování). Je implementován dataflow sítí. Ta je rozdělena do dvou částí, označovaných jako alpha a beta. Alpha část sítě vyhodnocuje splnění jednotlivých podmínek pravidla fakty nově přidanými do či odebranými z pracovní paměti. Beta část pak zajišťuje zachování konzistence vazeb proměnných mezi podmínkami.

To, že je vyhodnocování sítí RETE tak efektivní, je zajištěno maximálním sdílením částí sítě mezi pravidly. Mají-li dvě pravidla strukturálně stejný vzor nějaké podmínky (liší se jen použité proměnné), sdílí tato pravidla část alpha sítě, která tuto podmínku vyhodnocuje. Pokud mají dvě pravidla stejných několik prvních podmínek, sdílí tato pravidla kromě části alpha sítě i část beta sítě, která zajišťuje konzistenci vazeb proměnných mezi nimi. Čím více je v systému definováno odvozovacích pravidel, tím pravděpodobnější jsou shody mezi jejich podmínkami a tím více se efektivita algoritmu projeví.

Dataflow síť tvoří orientovaný acyklický graf[15]. Uzly sítě rozdělujeme na dva základní typy - paměťové a testovací. Paměťové uzly ukládají průběžné výsledky vyhodnocování. Testovací uzly provádějí různé typy testů (v závislosti na typu uzlu) nutných pro vyhodnocení podmínek pravidla.

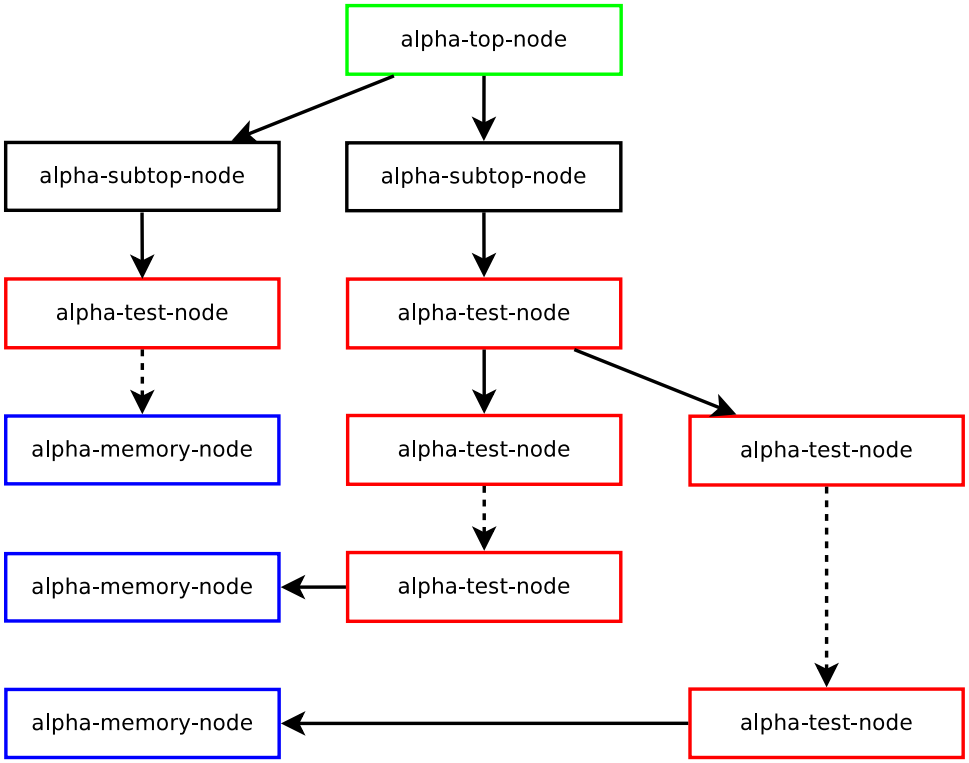
Mezi uzly sítě dochází ke dvěma typům interakcí. Hlavním typem interakce je aktivace. Ta probíhá ve směru hran grafu a uzel při ní provádí svou hlavní funkci v závislosti na typu. Testovací uzel se dále může dotazovat sousedního paměťového uzlu na obsah jeho paměti a to **proti směru orientace hran**. Abych odlišil sousední uzly ve směru hran (tedy ty, jež může uzel aktivovat) od množiny všech sousedů, budu první skupinu nazývat *potomky*.

Uzlu je při aktivaci vždy předána nějaká datová struktura. Tou je buď fakt, nebo token, který reprezentuje posloupnost faktů. Zatímco paměťové uzly po aktivaci a uložení datové struktury do své paměti vždy aktivují své potomky, testovací uzly aktivují potomky jen v případě úspěšného testu.

Obrázek 3. zobrazuje úsek alpha části sítě RETE. Uzel `alpha-top-node` je vstupním bodem této sítě. Ten je aktivován při každé změně pracovní paměti faktem, který byl přidán nebo odebrán. Potomky `alpha-top-nodu` jsou `alpha-subtop-nody`, jeden pro jednoduché fakty a jeden pro každou definovanou šablonu. `Alpha-top-node` aktivuje příslušný `alpha-subtop-node` podle typu faktu. Ten pak aktivuje `alpha-test-node` pod ním.

Šrafované šipky v obrázku 3. symbolizují, že zde může následovat posloupnost několika dalších `alpha-test-nodů`. Každá cesta z `alpha-subtop-nodu` do `alpha-memory-nodu` skrze posloupnost `alpha-test-nodů` vyhodnocuje jednu podmínku nějakého pravidla (případně skupiny pravidel). Mějme například pravidla

[15] http://en.wikipedia.org/wiki/Directed_acyclic_graph

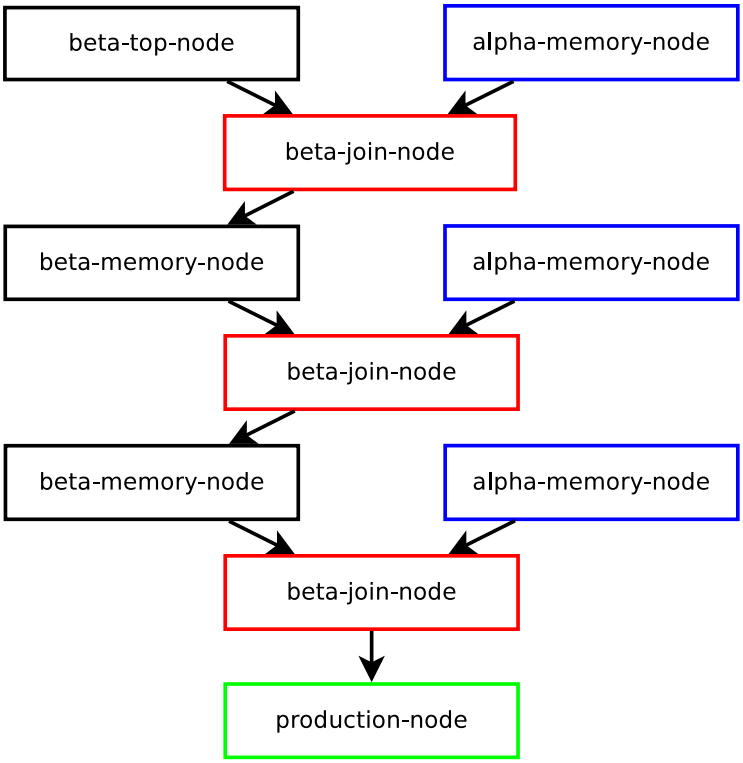


Obrázek 3. Alpha část sítě RETE

```
(defrule rule1
  (in :object box :location ?locaction)
  =>
  ...)
(defrule rule2
  (in :object box :location ?loc)
  =>
  ...).
```

Tato pravidla mohou sdílet část alpha sítě pro vyhodnocení svých podmínek, neboť ty se liší pouze názvy proměnných. Pro vyhodnocení těchto podmínek budeme potřebovat jeden `alpha-subtop-node` pro šablonu `in`, jeden `alpha-test-node` pro testování, zda je hodnota slotu `object` rovna hodnotě `box` a jeden `alpha-memory-node` pro uložení faktů, které tuto podmínku splňují. Pro slot `location` `alpha-test-node` nepotřebujeme, neboť alpha část sítě se hodnotami proměnných nezabývá.

Obrázek 4. zobrazuje úsek beta části sítě, spolu s `alpha-memory-nody`, kterými sem vstupují fakta splňující jednotlivé podmínky pravidel. Každý `beta-memory-node` udržuje množinu tokenů reprezentujících posloupnosti faktů, z nichž každý splňuje podmínku nějakého pravidla. `Beta-join-nody` pak tes-



Obrázek 4. Beta část sítě RETE

tují konzistenci vazeb proměnných mezi těmito podmínkami. `beta-top-node` je speciální typ `beta-memory-nodu`, který udržuje v paměti pouze jeden prázdný token. `production-node` je pak speciální typ `beta-memory-nodu`, který při aktivaci upozorní prostředí, že byly splněny všechny podmínky pravidla, případně že po odstranění faktu už nejsou všechny splněny.

Mějme například pravidlo

```
(defrule rule
  (goal :object ?obj :from ?from :to ?to)
  (in :object ?obj :location ?from)
  (in :object robot :location ?to)
  =>
  ...).
```

Alpha část sítě pro toto pravidlo bude obsahovat tři `alpha-memory-nody`, každý pro jednu jeho podmínku (kdyby však byla v poslední podmínce místo hodnoty `robot` proměnná, vystačili bychom si se dvěma `alpha-memory-nody`).

Beta část sítě bude vypadat tak, jako na obrázku 4. První (shora) `beta-test-node` ve skutečnosti nemá co testovat, neboť jde o první podmínku pravidla. První `beta-memory-node` bude udržovat tokeny délky 1 reprezentující „posloupnost“ faktů, které splňují první podmínku. Druhý `beta-join-node`

bude testovat konzistenci vazeb mezi druhou a první podmínkou pravidla. Druhý `beta-memory-node` bude udržovat tokeny délky 2 reprezentující dvojice faktů, splňující první dvě podmínky pravidla. Třetí `beta-join-node` bude testovat konzistenci vazeb mezi třetí podmínkou pravidla a předchozími dvěma. `Production-node` pak bude udržovat tokeny délky 3 reprezentující trojice faktů splňující všechny podmínky pravidla.

Přidávejme nyní postupně fakty do pracovní paměti. Přidáním `(in :object box :location A)` dojde (po průchodu alpha částí sítě) k aktivaci druhého `alpha-memory-nodu`. Ten uloží fakt do své paměti a aktivuje „zprava“ druhý `beta-join-node`. Ten prohledá paměť `beta-memory-nodu` nad sebou, zda neobsahuje nějaký token s konzistentními vazbami proměnných. Ta je ale prázdná, takže zde se aktivace zastaví.

Přidáním `(goal :object box :from A :to B)` dojde k aktivaci prvního `alpha-memory-nodu`. Ten po uložení faktu aktivuje první `beta-join-node`. Ten nemá co testovat, takže aktivuje `beta-memory-node` pod sebou. Ten uloží jednoprvkový token s přidaným faktem a aktivuje opět druhý `beta-join-node`, tentokrát však „zleva“. `Beta-join-node` tentokrát prohledá obsah paměti `alpha-memory-nodu` nad sebou, zda neobsahuje fakt konzistentní s tokenem. `Beta-join-node` zde testuje, zda jsou hodnoty slotů `object` a `location` faktu v alpha paměti shodné s hodnotami `object` a `from` prvního faktu tokenu v beta paměti.

Test zde skončí úspěšně, takže druhý `beta-join-node` aktivuje `beta-memory-node` pod sebou a předá mu dvouprvkový token s oběma fakty. Ten aktivuje „zleva“ třetí `beta-join-node`. Ten následně prohledá paměť `alpha-memory-nodu` nad sebou. Ta je ale prázdná, takže zde aktivace skončí.

Po přidání faktu `(in :object robot :location B)` dojde k aktivaci třetího `alpha-memory-nodu`. Ten fakt uloží a aktivuje „zprava“ třetí `beta-join-node`. Ten prohledá paměť `beta-memory-nodu` nad sebou, zda neobsahuje konzistentní tokeny. Ta obsahuje dvouprvkový token s prvními dvěma fakty. `Beta-join-node` tedy srovná hodnotu slotu `location` přidaného faktu s hodnotou slotu `to` prvního faktu tokenu. Protože tyto hodnoty se shodují, aktivuje `beta-join-node production-node` pod sebou a předá mu tříprvkový token s posloupností faktů, které splňují všechny podmínky pravidla. Ten token uloží a upozorní prostředí, že pravidlo bylo splněno posloupností faktů v tokenu.

Jak jsme viděli, `beta-join-node` může být aktivován buď zprava `alpha-memory-nodem` s novým faktem, který splňuje nějakou podmínku pravidla, nebo zleva `beta-memory-nodem` s tokenem, jehož fakty splňují předchozí podmínky. V každém případě `beta-join-node` prohledá obsah paměti druhého paměťového uzlu nad ním, aby nalezl tokeny konzistentní s novým faktem, nebo fakty konzistentní s novým tokenem.

Speciálním typem `beta-join-nodu` je `beta-negative-node`, který je zprava aktivován `alpha-memory-nodem` udržujícím fakty, které splňují negovanou podmínku pravidla. Ten naopak testuje, že druhá paměť **neobsahuje** konzistentní

fakty/tokeny a v takovém případě aktivuje potomky.

Při odebrání faktu z pracovní paměti se síť chová velmi podobně. Paměťové uzly ale v tomto případě odebírají fakty/tokeny ze svých pamětí. Je-li při odebrání faktu aktivován `production-node`, upozorní prostředí, že pravidlo už není původní posloupností faktů splněno.

Odlišně se ovšem při odebrání faktu chovají `beta-negative-nody`. Každý `beta-negative-node` je potomkem dvou paměťových uzlů - jednoho `beta-memory-nodu` a jednoho `alpha-memory-nodu`. Pokud byl odebíraný fakt/token v paměti jednoho z těchto paměťových uzlů jediným faktem/tokenem, konzistentním s fakty/tokeny v druhém paměťovém uzlu, je nyní negovaná podmínka pravidla splněna a `beta-negative-node` aktivuje `beta-memory-node` pod sebou všemi tokeny z paměti `beta-memory-nodu` nad sebou, jako by šlo o nově přidané tokeny.

Kdybychom nyní přidali pravidlo, které má první dvě podmínky stejné jako předchozí pravidlo, většina beta části sítě by zůstala stejná. Druhý `beta-memory-node` by ale získal jako potomka další `beta-join-node`, který by testoval konzistenci vazeb proměnných ve třetí, odlišné podmínce vůči prvním dvěma. Pod ním by pak následovaly další uzly testující zbytek podmínek a nový `production-node`. Kdybychom přidali pravidlo, které má všechny podmínky stejné, síť by se dokonce nezměnila vůbec. Původní `production-node` by si jen zapamatoval, že jeho aktivace nyní značí splnění obou pravidel.

Každý typ uzlu sítě RETE je v programu reprezentován jedním typem objektu stejného názvu. Každý z těchto objektů implementuje metodu `activate`, jejímž zavoláním je uzel aktivován. Většina kódu implementujícího modul `rete` je tvořena právě definicemi metod `activate`. Další část kódu pak implementuje dotváření sítě při přidání pravidla, které je netriviální. Program zde prochází podmínky pravidla, přičemž zároveň postupuje v síti odshora a zkoumá, zda je třeba vytvářet nové uzly, nebo lze použít existující, případně je nově propojit.

Teorii potřebnou k implementaci algoritmu RETE jsem nastudoval z [3].

2.4.3. Kompozitní podmínky pravidel

Z fungování beta částí sítě RETE (viz sekce 2.4.2.) vidíme, že vyhodnocování podmínek pravidla je inherentně sekvenční. `Beta-join-nody` zde implementují logickou spojku *a* mezi podmínkami pravidla, která je zde zleva asociativní.

2.4.4. Undo/redo

Možnost vrácení provedených akcí a jejich opětovného provedení (undo/redo) je implementována na úrovni prostředí, které udržuje aktuální stav systému. Hodnoty, které stav tvoří, jsem popsal v sekci 2.3.9. a možnosti, které undo/redo poskytuje, v sekci 2.3.7.

Prostředí je v programu reprezentováno objektem `environment`. Stav systému je uchováván ve slotech tohoto objektu.

Prostředí dále udržuje dva zásobníky - `undo` a `redo`. Každý z těchto zásobníků ukládá seznamy tvořené popiskem akce k vrácení, uzávěrem, který akci vrátí a uzávěrem, který ji opět provede. První uzávěr ve svém lexikálním prostředí uchovává hodnoty potřebné k vrácení akce. Zásobníky jsou manipulovány pouze několika makry a metodami k tomu určenými, zbytek kódu k zásobníkům přímo nepřistupuje.

Tělo každé akce, která mění hodnoty prostředí, je obaleno buď voláním makra `with-undo`, nebo `with-saved-slots`. Volání `with-undo` je kromě těla akce předán uzávěr, který zajistí její vrácení. Makro `with-undo` po vyhodnocení těla akce zajistí, že se na zásobník `undo` uloží předaný uzávěr spolu s uzávěrem, který akci opět provede při volání `redo`. Tento druhý uzávěr pouze vyhodnotí původní tělo akce.

Makro `with-saved-slot` zjednodušuje volání `with-undo`. Toto makro bere kromě těla akce seznam slotů prostředí, které je třeba před akcí uložit. Makro pak automaticky vytvoří uzávěr, který předá makru `with-undo`. Tento uzávěr si ve svém lexikálním prostředí pamatuje kopie původních hodnot požadovaných slotů prostředí a při vyhodnocení je nastaví zpět na tyto hodnoty.

Prostředí ke každému svému slotu definuje metodu `copy-<slot>`, např. `copy-facts`. Makro `with-saved-slots` tedy vytvoří potřebný uzávěr tak, že v `let`u naváže výsledky volání těchto metod pro každý požadovaný slot. V tomto `let`u pak vytvoří anonymní (lambda) funkci, která hodnoty prostředí nastaví zpět na ty, které `let` navázal.

Funkce `undo` pak jednoduše odstraní seznam z vrcholu zásobníku `undo`, zavolá uzávěr, který vrátí poslední akci a druhý uzávěr, který akci opět provede, uloží na vrchol zásobníku `redo`. Funkce `redo` funguje symetricky.

Všechny objekty definované modulem `core` - šablony, fakty, vzory a pravidla jsou implementovány jako neměnné (*immutable*). Každá jejich případná změna tedy znamená vytvoření nového objektu. Díky tomu není na úrovni prostředí nutné tyto objekty kopírovat. Kopírovat je třeba pouze datové struktury, které udržují kolekce těchto objektů.

Nakonec je třeba zajistit, aby každé volání `with-undo` uložilo na zásobník právě jeden uzávěr k vrácení akce. Například pokud samostatně voláme makro `assert` pro přidání nějakého faktu do pracovní paměti, chceme, aby toto volání bylo možné vrátit. Pokud ale voláme `step`, jehož výsledkem je aktivace nějakého pravidla, které ve svých důsledcích volá makro `assert`, chceme, aby se volání `step` uložilo na zásobník jako jedna akce. Volání `assert`, ke kterému v průběhu této akce dojde, už samostatně ukládat nechceme.

K tomuto účelu definuje prostředí *dynamickou proměnnou* `*undo-enabled*`. Makro `with-undo` ukládá uzávěr na zásobník jen v případě, že je hodnota této proměnná `true`. Tělo akce pak makro vyhodnocuje s hodnotou této proměnné navázanou na `false`. Tím je zajištěno, že se uzávěr na zásobník `undo` uloží vždy

jen v „nejvnějšnějším“ volání makra `with-undo`.

Díky makrům `with-undo` a `with-saved-slots` je možné velmi snadno přidat do prostředí další akce s možností jejich vrácení. Stačí jen tělo akce obalit jedním z těchto maker bez nutnosti vědět, jak je vrácení akce implementováno. Pokud potřebujeme do prostředí přidat další slot, jehož hodnotu je třeba při volání `undo` obnovovat, stačí k tomuto slotu implementovat metodu `copy`.

Nejsložitějším problémem při implementaci undo/redo bylo implementovat metodu `copy-rete`. Ta kopíruje dataflow síť RETE uloženou ve slotu `rete` prostředí. Kopírovnání sítě RETE je složité, neboť síť obsahuje cykly. Síť se sice z pohledu aktivace uzlů chová jako acyklický graf, některé uzly však uchovávájí reference na své sousedy proti směru hran tohoto grafu.

Pro účely kopírování sítě RETE jsem implementoval obecnou funkci vyššího řádu[16] pro průchod cyklickým grafem. Tato funkce využívá techniky memoizace[17]. Funkce bere kromě výchozího uzlu grafu tři funkce, které aplikuje na navštěvované uzly před memoizací a jimiž agreguje hodnoty vrácené sousedy uzlu.

Fungování funkce je pro textový popis příliš složité, považuji ji však za jednu z nejzajímavějších částí nového kódu, hlavně díky její obecnosti. Zrojový kód funkce je v souboru `rete/graph-traversal.lisp`, kód kopírující síť rete, který funkci využívá pak v souboru `rete/rete-copy.lisp`. Celkově je kód zajišťující kopírování sítě RETE asi třikrát delší, než zbytek kódu implementující funkcionalitu undo/redo. Ten se nachází v souboru `environment/env-undo.lisp`.

2.4.5. Zpětná inference

2.4.6. Kompatibilita se systémem CLIPS

Syntaktickou kompatibilitu se systémem CLIPS zajišťuje nově přidaný modul `parser`. Ten převádí externí reprezentace objektů - šablon, faktů, vzorů a pravidel - na interní objekty (ve smyslu instance třídy) definované v modulu `core`. K tomuto účelu definuje `parser` metodu `parse-<object>` pro každý typ `core` objektu. Externími reprezentacemi jsou (případně vnořené) seznamy symbolů, které předáváme funkcím a makrům modulu `front-end`, které jsem popsal v uživatelské příručce.

Metody `parseru` jednoduše analyzují tyto seznamy, rozhodují, zda jde o základní, nebo CLIPSovou syntax a podle toho je převádí na jednotnou reprezentaci, kterou pak předávají konstruktorům `core` objektů. Tyto vnitřní objekty jsou pak metodami `parseru` vraceny. Metody jsou volány modulem `front-end`, který pak získané `core` objekty předává prostředí.

2.4.7. Grafické uživatelské rozhraní

[16] http://en.wikipedia.org/wiki/Higher-order_function
[17] http://en.wikipedia.org/wiki/Memoization

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