VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ZPŘÍSTUPNĚNÍ FUNKCÍ CLIPS Z JAZYKA RUBY

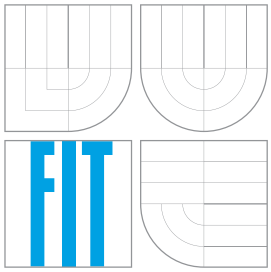
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAROSLAV ČECHO

BRNO 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ZPŘÍSTUPNĚNÍ FUNKCÍ CLIPS Z JAZYKA RUBY

MAKING CLIPS AVAILABLE FROM RUBY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAROSLAV ČECHO
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZDENĚK LETKO
SUPERVISOR

BRNO 2008

Abstrakt

Tato práce popisuje knihovnu rbClips, která zpřístupňuje funkce CLIPS v jazyce Ruby. CLIPS je software pro tvorbu expertních systémů původně vyvinutý v NASA na začátku 90. let. Nástroj je napsán v jazyce C a jeho uživatelské rozhraní je velice podobné jazyku Lisp. Ruby je moderní dynamický skriptovací jazyk, který programátorovi nabízí flexibilní syntax, otevřenost objektů, čistě objektové prostředí a další zajímavé vlastnosti. Výsledná knihovna bude použita v antivirovém programu k tvorbě expertního systému pro automatickou detekci malware.

Abstract

This thesis describes a library called rbClips that makes CLIPS functionality available from Ruby. CLIPS is a public domain tool for building expert systems that was originally developed in NASA in 90's. The tool itself is written in C but its user interface is very similar to the Lisp language. Ruby is a modern dynamic scripting language that offers programmer flexible syntax, purely object environment, openness of objects and other interesting features. The library is ment to be used to build expert system for detection of a possibly malicious code in an antivirus software.

Klíčová slova

Ruby, CLIPS, binární rozšíření, expertní systém

Keywords

Ruby, CLIPS, binary extension, expert system

Citace

Jaroslav Čecho: Zpřístupnění funkcí CLIPS z jazyka Ruby, bakalářská práce, Brno, FIT VUT v Brně, 2008

Zpřístupnění funkcí CLIPS z jazyka Ruby

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Zdeňka Letka.

.....
Jaroslav Čecho
12. května 2010

Poděkování

Rád bych poděkoval technickému vedoucímu této prace, Ryanu Hicksovi, za nápad, rady a ochotu řešit problémy, na které jsem při tvorbě narazil a svému odbornému vedoucímu Zdeňkovi Letkovi za jeho trpělivost a čas věnovaný kontrole korektnosti mých textů.

Obsah

Kapitola 1

Úvod

Počet nově se objevujících virů a jiných škodlivých počítačových kódů (malwaru) rok od roku stoupá. Velice tomu pomáhá postupné rozšiřování internetu i mezi laickou veřejnost, která se příliš netrápí dostatečným zabezpečením svých počítačů. Tento nepříliš optimistický trend způsobuje problémy snad ve všech odvětvích lidské činnosti, protože dnes už bez oddělení starajícího se o počítačovou infrastrukturu nemůže existovat žádná vetší firma. Způsobené problémy jsou hlavně finanční – rostou náklady jak na zabezpečení jednotlivých počítačových stanic, tak i celých počítačových sítí.

Logickou úvahou by mohl čtenář dojít k názoru, že na tomto trendu mimo producentů nevyžádané pošty profitují hlavně různé bezpečností firmy. Nejvíce pak antivirové společnosti, dodávající ochranná řešení proti malwaru. Bohužel opak je pravdou. Škodlivého kódu je ovšem nejenom více, ale bohužel se taktéž dost rychle mění. Jeho autoři, kterých stále přibývá, vynalézají nové a důmyslnější techniky jak předejít odhalení antivirovým programem. Což na straně antivirových firem znamená poptávku po stále větším počtu velice dobře kvalifikovaných lidí, kteří jsou schopní prozkoumat podezřelé vzorky a pro každý malware vytvořit nové virové definice.

Dostatečný počet dobře kvalifikovaných lidí bohužel na trhu práce není k dispozici, a proto se hledají jiné, automatizovanější postupy, pro zpracování podezřelých vzorků. Automatizované postupy mají navíc i jiné výhody. Počítač pracuje rychleji, mnohdy spolehlivěji a hlavně výrazně levněji než jeho lidský ekvivalent. Na druhou stranu mu chybí takzvaný "zdravý selský rozum". Jako jeden ze způsobů adresace problému lidského uvažovaní vznikly tzv. expertní systémy. Expertní systém je program, který napodobuje rozhodovací proces lidského experta podle předem nadefinovaných pravidel.

V rámci společnosti AVG Technologies s.r.o jsme se rozhodli vytvořit program s jádrem v expertním systému pro automatickou klasifikaci vzorků. Většina expertních systému je komerčních a vzhledem k tomu, že se nejedná o levné programy, zvolili jsme volně šiřitelný příklad takového systému jménem CLIPS. Nabízí plnohodnotné prostředí pro tvorbu expertního systému, které je napsáno v jazyce C. Uživatelské rozhraní, které je velice podobné jazyku LISP, je ovšem uživatelsky velice nepřívětivé. Proto jsme se rozhodli celé CLIPS vzít a zapouzdřit do nějakého vyššího programovacího jazyka. Jako cílový jazyk pro zapouzdření jsme se rozhodli využít moderní skriptovací jazyk Ruby, kvůli jeho zajímavým a neobvyklým možnostem.

Má bakalářská práce pojednává o nově vytvořené knihovně rbClips, která umožňuje používat nástroj pro tvorbu expertních systémů CLIPS v jazyce Ruby. Knihovna bude následně použita k tvorbě systému pro automatické rozpoznávání vzorků tak, jak bylo nastíněno v úvodních odstavcích. Samotný systém pro klasifikaci vzorků už ovšem není

obsahem této práce.

Zbytek této práce je rozdělen do dvou kapitol. Po této první úvodní kapitole následuje popis použitých technologií. Tedy úvod do programovacího jazyka Ruby s důvody proč jsme se rozhodli použít pravě jej. A následně také úvod do problematiky expertních systémů a nástroje CLIPS. Následující kapitola se již věnuje čistě knihovně rbClips, výsledku mé práce. Budu se zde zabývat jak návrhem a popisem rozhraní tak i různými zajímavými implementačními detaily.

Kapitola 2

Použité technologie

Jak jsem popsal v úvodu, cílem mé bakalářské práce je vytvořit knihovnu rbClips umožnující propojit CLIPS a Ruby. V následujících dvou kapitolách se oběma technologiemi budu zabývat více do hloubky a vysvětlím proč jsme se rozhodli použít právě je. Obě technologie jsou napsány v programovacím jazyce C, proto i moje knihovna bude napsána v témže jazyce. Samotný popis jazyka C jsem již ovšem do své práce nezahrnul a lze jej najít například v knize [3].

2.1 Ruby

Ruby [18] je relativně mladý dynamický skriptovací jazyk vytvořený japonským inženýrem Yukihiro Matsumoto, známým pod přezdívkou Matz. V současné době neexistuje žádná specifikace či norma popisující tento jazyk jako je tomu například u jazyků C, C++ a dalších. Z tohoto důvodu se jako reference jazyka bere samotný interpret napsaný Matzem – Matz' Ruby Interpret (MRI). Interpretů existuje více a jejich zkrácený seznam je uveden dále v textu. Absence existujícího standartu vedla v roce 2008 k vytvoření skupiny pro standardizaci jazyka v rámci japonské organizace Information-technology Promotion Agency (IPA), která vychází z MRI verze 1.8.7. V současné době je k dispozici již návrh standardu[1], který má být navržen ke schválení prvně v Japonsku u standardizační komise Japanese Industrial Standards Committee (JISC) a následně i u International Standard Organization (ISO).

2.1.1 Historie

První verze interpretu Ruby vytvořená Matzem byla zveřejněna už v roce 1995 (velice zajímavý je fakt, že jméno nového jazyka už bylo vybráno v roce 1993, tedy o cele dva roky dříve), ovšem oficiální webové stránky jazyka v angličtině byly k dispozici až o tři roky později, tedy v roce 1998. Absence kvalitních materiálů v angličtině je jedním z hlavních důvodů, proč se Ruby stal populárním prvně pouze v Japonsku a velice pomalu se dostával i do ostatních států světa. V dnešní době už naštěstí není problém najít kvalitní zdroje informací ani v angličtině ani v češtině, či dalších jazycích, a to jak ve webové, tak i v knižní podobě. Tato i další zajímavá data jsou shrnuta na Obrázku 2.1, která jsem čerpal z prezentace vytvořené k příležitosti konference RubyConf 2006 [21].

Matz vytvořil nový jazyk, protože ho nadchly možnosti skriptovacích jazyku a žádný z tehdy dostupných jazyků ho nezaujal[2]. Chtěl jazyk mocnější než Perl [11] a více objektově

[1] http://ruby-std.netlab.jp/draft_spec/agreement.html
[2] http://linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html

1993	Vybráno jméno.
1995	První zveřejněná verze interpretu (verze 0.95).
1996	Verze 1.0.
1997	Matz zaměstnán jako Ruby programátor na plný úvazek.
1998	Webové stránky v angličtině a vznik anglicky psaného mailing listu.
1999	První kniha o Ruby (psána japonsky).
2000	Začátek rozšiřování povědomí o Ruby mimo hranice Japonska.
2001	YARPC - Yet Another Ruby and Perl Conference.

Obrázek 2.1: Shrnutá historie v datech.

orientovaný než Python [12]. Na otázku, proč vytvořil další skriptovací jazyk, Matz ještě často dodává důvody syntaxe. Programovacích jazyků je mnoho a jsou svým způsobem velice podobné (vlastnostmi i schopnostmi). Matz ovšem chtěl jazyk, ve kterém ho bude bavit psát a nebude se muset příliš trápit se zápisem jednotlivých konstrukcí. Syntaxe by měla být podřízena člověku jakožto pisateli jazyka, nikoliv počítači jakožto interpretu jazyka.

V současnosti se udržují dvě hlavní větve MRI – verze 1.8 a 1.9. Větev 1.8 je považována za stabilní a již se v ní pouze opravují chyby, hlavní vývoj probíhá ve větvi 1.9.

2.1.2 Vlastnosti ruby

Ruby je dynamický skriptovací jazyk mnoha různých paradigmat – je plně objektově orientovaný, ale lze v něm bez problémů psát i imperativně či funkcionálně. Následuje popis vybraných vlastností, které bych rád vyzdvihl, či na ně upozornil. Podle potřeby budu dále v textu srovnávat Ruby s dalšími jazyky které znám – Java [5] a C++ [15].

Níže uvedený popis vlastností jsem čerpal ze své osobní zkušenosti s tímto programovacím jazykem a z velice obsáhlého popisu jazyka napsaného samotným autorem Ruby, The Ruby programming language [6], která je zrevidování a rozšíření dříve vydané knihy Ruby in Nutshell [23]. A z knihy Metaprogramming Ruby [14], která se zabývá nejen metaprogramováním v Ruby, ale také velice čitelným způsobem vysvětluje objektový model ruby a další jeho aspekty.

V dalších odstavcích předpokládám, že je čtenář seznámen se základními konstrukcemi jazyka Ruby, na nichž popíši jeho zajímavé a neobvyklé vlastnosti jako volnější syntaxi, dynamičnost, otevřenost objektů spolu s objektovým modelem a bližší popis práce s bloky. V příkladech budu mimo jiné využívat konstrukci `# =>`, která se v Ruby světě běžně používá pro uvedení návratové hodnoty příkazu. Znak `#` slouží jako úvodní znak komentáře, proto konstrukce nijak neovlivňuje vykonávání napsaného kód.

Objektovost

Ruby je plně objektově orientovaný jazyk (byl mimo jiné inspirován i Smalltalkem [4]), což znamená, že vše v Ruby je objekt. Nejsou zde žádná datová primitiva jako `int` či `double` podobně jak tomu je v Javě a C++. Číselné a jiné konstanty jsou interpretem okamžitě převáděny na instance příslušných tříd. Příklad na Obrázku 2.2 ukazuje možnost volání metody na číselné konstantě bez nutnosti explicitní konverze na objekt.

Interpret po svém startu vytvoří bezejmennou instanci třídy `Object` a v jejím kontextu následně vykoná předaný skript. Odpadá tedy nutnost explicitně vytvářet třídu s minimálně

```
123.class          # => Fixnum
```

Obrázek 2.2: Automatická konverze konstanty na odpovídající objekt.

```
"abc".class        # => String
String.class       # => Class
```

Obrázek 2.3: Ruby třída je plnohodnotný objekt.

jednou veřejnou metodou `main()`, která se spustí po startu aplikace, jako je tomu v Javě nebo funkci `main()` v případě C++. Ukázkový "Hello world!" program, který v Ruby vypadá takto: `puts "Hello world!"`, je tedy objektový kód, i když se tak na první pohled nezdá. Tvrzení lze dokázat vypsáním třídy aktuálního objektu `puts self.class`, která vrátí `Object`, i když pro pisatele v žádném objektu není (`self` je obdoba ukazatele `this` z jazyka C++, tedy reference na aktuální objekt),

Další a pro mě v době, kdy jsem se s Ruby poprvé seznamoval, velice překvapivý důsledek plné objektovosti je fakt, že třídy samy jsou objekty podobně jako v jazyce Smalltalk. Například objekt "abcd" je třídy `String`, samotná třída a objekt zároveň `String` je třídy `Class`, což jde vidět na Obrázku 2.3. Z tohoto přístupu plyne několik důsledků: (1) je potřeba rozlišovat mezi třídní metodou a instanční metodou, (2) třídy mohou mít své instanční proměnné a (3) existuje možnost dědit i na množině objektů, ze kterých se vytvářejí třídy. Z praktického pohledu se na tuto vlastnost lze koukat jako na statické proměnné a metody u jazyka C++. Pro lepší pochopení je na Obrázku 2.4 naznačeno umístění metod a proměnných jak ve třídě, tak i ve třídě třídy.

Schopnosti dědit na množině objektů, ze kterých se vytvářejí třídy využívá i objektový model Ruby. Třída `Class` používaná pro tvorbu tříd dědí od velmi specifické třídy `Modul`, kterou rozšiřuje hlavně o metodu `new` sloužící pro tvorbu nových instancí. Modul je zjednodušená třída, kolekce metod, od které nelze vytvářet instance ovšem lze je vkládat dovnitř jiných tříd a tím nahrazovat absenci vícenásobné dědičnosti v Ruby.

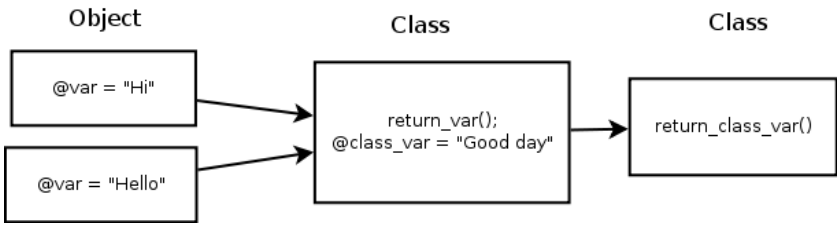
Volná syntaxe

Jak už bylo zmíněno výše, Matz chtěl vytvořit jazyk, ve kterém by ho bavilo programovat. Jazyk, ve kterém by nemusel pořád přemýšlet nad syntaxí, který by byl krásně čitelný a hlavně, pokud možno, co nejbližší běžnému lidskému jazyku. Cíl se mu povedl, protože ukázky kódu na Obrázku 2.5 opravdu při čtení připomínají anglické věty, popisující co, se má udělat. Tato vlastnost má vliv na syntaktickou analýzu kódu, která je mnohem složitější.

Jedním ze způsobů přizpůsobení gramatiky jazyka programátorovi je možnost vynechat závorky oddělující název metody od jejich parametrů v místech, kde to není syntakticky nejednoznačné. Lze tedy zavolat metodu `naDruhou` s parametrem 5 tak, jak je zvykem v Javě a C++ `naDruhou(5)`, nebo na mnoha místech více čitelněji `naDruhou 5`. Volání metody bez závorek vypadá na mnoha místech spíše jako klíčové slovo než obyčejné volání metody.

Na první pohled se to může jevit spíše jako matoucí, ale uvedený příklad z Obrázku 2.6) z webového frameworku Ruby on Rails [16] (RoR) ukazuje opak. I člověk bez znalosti Ruby nebo RoR, ovšem se znalostí jiného programovacího jazyka, je schopen s velkou úspěšností odhadnout, že prezentovaný příklad vytváří třídu `Human`, s určitou závislostí na oddělení (department), a který napsal mnoho různých článků (articles).

Tento spíše sociální efekt ("Jedná se o metodu nebo klíčové slovo?") jde v Ruby velice hluboko. Mnohdy lze už v úvodních tutoriálech najít zmínku o klíčovém slově `puts`



Obrázek 2.4: Třída je také objekt.

```
3.times do puts 'Ahoj' end      # 3x vypis "ahoj"
puts "Ahoj" if name == "jarcec" # vypis Ahoj pokud je jmeno "jarcec"
```

Obrázek 2.5: Ukázky čitelnosti kódu.

sloužícímu pro výpis na standardní výstup (obdoba `printf()` z jazyka C). Ve skutečnosti se ovšem nejedná o klíčové slovo, ale o metodu modulu `Kernel`, který každý objekt zdědí, a proto je na každém místě programu dostupná. Jedná se tedy o zcela normální metodu a lze ji stejně jako jakoukoliv jinou v případě potřeby přetížit, popřípadě z objektu odstranit...

Možnost vynechat psaní závorek na každém místě považuji za kladnou vlastnost, kterou velice často a rád využívám. Na druhou stranu, ovšem, musím upozornit na její dopady – i pouhá mezera může změnit logiku kódu. Na Obrázku 2.7 uvádím dva příklady lišící se pouhou jednou mezerou označenou symbolem ␣ mezi koncem názvu metody a otevírající závorkou. V prvním případě syntaktický analyzátor najde ihned za názvem metody závorku. Ví tedy, že uvnitř se nacházejí parametry, se kterými se metoda má volat a až následná vrácená hodnota se vynásobí dvěma. Výsledek bude 18 ($=3^2 * 2$). V druhém příkladě syntaktický analyzátor najde za názvem metody mezeru a přistupuje k volání jinak. Ví, že následuje seznam argumentů oddělený čárkami, protože závorky okolo argumentů jsou vynechány. Na řádce se ovšem žádná čárka nenachází, a tedy celý výraz $(1 + 2) * 2$ je metodě předán jako jeden argument s výsledkem 36 ($=((1 + 2) * 2)^2$).

Ruby neumožňuje pojmenovat argumenty metod podobně jako Python, ovšem díky volné syntaxi pro vytvoření asociativního pole (hashe) lze toto chování napodobit. Příkladem může být volání fiktivní metody pro hledání na Obrázku 2.8, kde opticky předávám dva pojmenované parametry – name a surname. Ve skutečnosti je tento zápis reprezentován jako tvorba asociativního pole o dvou položkách, s klíči name a surname, a až následně je toto pole předáno metodě `find` jako jeden argument. Webový framework RoR tento přístup používá velmi často.

Poslední poznámka k volné syntaxi: Ruby umožňuje stejně jako Perl otočení pořadí zápisu neúplné podmínky a jejího těla (viz. Obrázek 2.9).

Dynamičnost a otevřenost

Dynamičností Ruby rozumím provádění mnoha činností, které jsou v případě C++ a Javy prováděny už v době kompilace, až za běhu programu. Vzhledem k plné objektovosti (vše je objekt) nemá moc smysl mluvit o typovosti jazyka, protože existuje pouze jediný datový typ, kterým je objekt.

Z dynamičnosti vyplývá mnohem zajímavější a ne až tak běžná vlastnost Ruby, kterou je otevřenost. Každý objekt může být za běhu otevřen a rozšířen o nové metody, podobně jako je tomu u JavaScriptu [17]. Velice často se toho využívá u tříd (například rozšířením

```
class Human < ActiveRecord::Base
  has_many     :articles
  belongs_to   :department
end
```

Obrázek 2.6: ActiveRecord ukázka prezentující volitelné vynechání závorek.

```
puts␣naDruhou(1+2)*2
puts␣naDruhou␣(1+2)*2
```

Obrázek 2.7: Důležitý význam mezer.

vestavěné třídy o nové metody). Tuto vlastnost by do určité míry šlo nahradit dědičností (nové metody uložit do potomka), ale jen pokud se rozšiřují třídy. V Ruby je vše objekt, takže rozšířit lze nejen třídy, ale dokonce i instance jednotlivých tříd (viz. Obrázek 2.10), což byl pro mě jakožto programátora znalého jazyka C++ docela šok při seznamování se s tímto jazykem. Možnosti rozšíření instancí o nové metody používám i ve své práci, což popisuji v kapitole 3.3.5 v části o zpracování Faktů.

Díky dynamičnosti a otevřenosti se v Ruby velice pěkně metaprogramuje, což znamená psaní kódu, který generuje další kód). Jedná se o velice často využívanou schopnost jazyka a velké projekty v Ruby jsou metaprogramováním dosti ovlivněny. Příkladem může být projekt ActiveRecords[3], který zapouzdřuje činnost s relačními databázemi tak, aby se uživatel s SQL [2] vůbec nesetkal. Vše je pro něj zapouzdřeno do objektů.

V definici třídy z Obrázku 2.11 chybí deklarace metod `name` a `surname` a přesto daný kód bude fungovat. Konstruktor třídy se totiž při inicializaci podívá do databáze na strukturu tabulky kterou popisuje, v tomto případě tabulky `humans`, a vytvoří ke všem nalezeným sloupcům příslušné metody pro čtení i zápis. Tento přístup velice usnadňuje údržbu kódu. V definici třídy není seznam sloupců jednotlivých tabulek, a proto při změně databázového schématu není nutné vše upravovat a zvedat tak pravděpodobnost zavlečení chyby díky inkonzistenci seznamu sloupců v databázi a v definici třídy.

O metaprogramování v Ruby byla napsána velice pěkná kniha jménem Metaprogramming Ruby [14]. Psána je spíše laickým stylem a na praktických příkladech popisuje, co přesně se pod pojmem metapogramovaní skrývá, a jak toho v Ruby co nejefektivněji využít.

Bloky

Blok je podobně jako v C++ nebo Javě sekvence příkazů ohraničená složenými závorkami. V případě Ruby je ještě možné blok ohraničit klíčovými slovy `do` a `end`. Na rozdíl od zmíněných jazyků v Ruby jde s bloky dělat o dost větší kouzla, a proto je zmiňuji v samostatné části. První důležitý rozdíl je možnost předat bloku parametry (blok se pak vlastně chová jako bezejmenná funkce). Další rozdíl je schopnost bloky v Ruby předávat do metod.

Příkladem může být záměna velikosti písmen v poli řetězců na Obrázku 2.12. Metoda `map` třídy `Array` přijímá blok akceptující jeden parametr a pro každý prvek v poli tento blok zavolá. Návratovou hodnotu bloku (v Ruby je vše výraz, i bloky tedy vracejí hodnoty, tato vlastnost je detailněji popsána níže) uloží do pole na místo volaného prvku. Ukázka tedy zmenší velikost písmen pro každý prvek v poli.

S bloky se pojí jeden důležitý pojem – uzávěr (closure). Každý blok si s sebou nese vazby na lokální proměnné známé v době jeho vzniku. V příkladu na Obrázku 2.13 si blok

[3]Oficiální dokumentace k projektu ActiveRecords je dostupná na http://ar.rubyonrails.org/. Na adrese http://guides.rubyonrails.org/ je v sekci Models k dispozici laicky čitelný úvod do tohoto projektu.

```
find :name => 'Jarek', :surname => 'Cecho'
```

Obrázek 2.8: Předávání asociativního pole jako jednoho argumentu.

```
if osoba == 'jarcec' then print "Ahoj Jarcec"; end
print "Ahoj Jarcec" if osoba == 'jarcec'
```

Obrázek 2.9: Přehození pořadí zápisu neúplné podmínky a jejího těla.

zapamatuje svou lokální proměnnou `a` (s hodnotou 30) a nese si ji s sebou i do metody `foo`, odkud je poté zavolán (pomocí klíčového slova `yield`). Ukázka vypíše číslo 30, protože `a` je lokální proměnná z kontextu, ve kterém byl předaný blok vytvořen. Pokud by chtěla metoda `foo` vytisknout svou hodnotu proměnné `a`, musela by ji předat do bloku přes parametr.

Třída Symbol

Symbol je textový řetězec podobný řetězcům třídy `String`, který se pro odlišení píše s dvojtečkou před vlastním řetězcem. Pokud je symbol navíc složen pouze z písmen, je možné vynechat i jinak povinné uvozovky či apostrofy (ukázky možností zápisu symbolu jsou na Obrázku 2.14).

Symboly v Ruby mají několik odlišných vlastností od třídy `String`. Za prvé se jedná o objekty pouze pro čtení (nelze změnit jejich hodnotu, vlastní text symbolu). Druhý a podstatnější rozdíl je v rovnosti a identitě. V ukázce na Obrázku 2.15 se provádí dvě dvojice porovnání na objektech tříd `String` a `Symbol`. V prvním testu porovnávám objekty, zda-li se rovnají (mají-li stejný obsah). Druhý test se ptá na jednoznačný identifikátor objektu. Testuje tedy identitu objektu. Testy na rovnost samozřejmě vždy uspějí, protože jak symbol, tak řetězec obsahují stejné hodnoty. Ovšem test na identitu v případě řetězce selže, protože se jedná o dva různé objekty (vytvořené interpretem při parsování zdrojového kódu programu). Symboly jsou si ovšem identické - jedná se o jednu instanci. Tato důležitá vlastnost symboly předurčuje k použití na místech konstant nebo klíčů do asociativních polí. Symbol s daným textem existuje v paměti maximálně jednou, bez ohledu na počet výskytů v programu, což vede k šetření paměti a zvýšení výkonu aplikace.

Na úrovní zdrojových kódu MRI je tato vlastnost implementována pomocí tabulky jednoznačně převádějící textovou reprezentaci symbolu na datový typ `ID` (což je předefinovaný typ `unsigned long`). Veškerá porovnávání na rovnost symbolu jsou ve skutečnosti celočíselná porovnání, která jsou rychlejší než řetězcová, čímž se dosahuje optimalizace. Java obsahuje velice obdobný mechanismus pomocí volání `String.intern()`, C++ nic jako symbol bohužel nezná.

Drobnosti

Dále bych se chtěl už jenom krátce zmínit o několika vlastnostech Ruby, které popisem nevydají na samostatnou část, ale přesto si myslím, že by zde měly být uvedeny.

Mimo faktu, že vše je objekt, je každá jazyková konstrukce výraz (má svou návratovou hodnotu). Tedy i řídící konstrukce (`if`, `while`, ...) něco vrací - podle funkce příslušné konstrukce je to třeba hodnota posledního vykonaného příkazu nebo objekt `nil`, který je popsán níže. (Obrázek 2.16).

Ruby je plně objektový a vše je objekt, popřípadě jeho metoda. I aritmetické operátory jsou tedy ve skutečnosti metody a jejich zápis bez přístupové tečky je jen "syntax sugar" (jak se doslovně uvádí v knize [6]). Lze je tedy podobně jako v C++ přetížit nebo z daného objektu úplně odstranit (Obrázek 2.17).

```
class A                  # definovani tridy jmenem "A"
  def metodaA            # definovani metody jmenem "A"
    4                    # tato metoda bude vracet cislo 4
  end
end

a = A.new                # vytvoreni prvni instance tridy A
a.metodaA                # => 4
b = A.new                # vytvoreni druhe instance tridy A

def b.metodaB            # vytvoreni metody pouze v instanci b
  5                      # tato metoda bude vracet cislo 5
end

b.metodaB                # => 5
a.metodaB                # => Exception NoMethodError
```

Obrázek 2.10: Přidávání metod do instancí tříd.

```
# trida popisujici relacni tabulku humans
class Human < ActiveRecords::Base
end

jarcec = Human.new       # vytvor novy radek v tabulce
jarcec.name = "Jarek"    # napln sloupec jmeno
jarcec.surname = "Cecho" # napln sloupec prijmeni
jarcec.save              # uloz radek do tabulky
```

Obrázek 2.11: Ukázka použití knihovny ActiveRecords.

Ruby podporuje pouze jednoduchou dědičnost, což znamená, že třída může mít maximálně jednoho předka. Ovšem absenci implementačně složité vícenásobné dědičnosti vynahrazuje možností vkládat do sebe moduly (Obrázek 2.18).

Na rozdíl od C++ či Javy jsou v Ruby povoleny na konci metod i jiné znaky než jen znaky anglické abecedy, čísla a podtržítko. Což má veliký dopad na zlepšenou čitelnost kódu, jelikož podle konvencí metody vracející boolovskou hodnotu jsou ukončeny znakem otazníku (například `Object#nil?`) a metody měnící stav samotného objektu jsou pro změnu ukončeny vykřičníkem (například `Array#map!`).

2.1.3 Implementace Ruby

Nejznámější a asi i současně nejvíce používanou implementací Ruby je interpret MRI (Mat'z ruby interpret) od autora jazyka. Ten má ovšem své velké nedostatky. Předně používá tzv. mark and sweep garbage collector. Ten v rámci své "mark" fáze zmrazí provádění programu, což se neblaze odráží na celkové rychlosti interpretace. Dále obsahuje takzvaný "Global interpreter lock" (GIL), který znemožňuje paralelní zpracování. Programy sice mohou běžet ve vláknech, ale kvůli jeho užití může v jednu chvíli běžet maximálně jedno vlákno.

Jako podklad pro následný seznam alternativních implementací jsem použil seznam interpretů a jejich srovnání na webu igvita [22].

```
["AHOJ", "NAZDAR", "HI"].map{|prvek| prvek.downcase}
      # => ["ahoj", "nazdar", "hi"]
```

Obrázek 2.12: Ukázka předání bloku metodě.

```
def foo           # definice metody foo
  a = 10          # deklarace vlastni promene a
  yield           # zavolej predany blok
end

a = 30            # druhy deklarace promene a
foo { puts a }    # zavolani funkce foo
```

Obrázek 2.13: Vazby na lokální proměné v případě předaných bloků.

Ruby Enterprise edition

Jedná se o vývojovou větev MRI implementace verze 1.8.7[4] s vylepšenou správou paměti a vyměněným garbage collectorem. Vznikla jako potřeba optimalizovat Ruby pro běh na serverech pro webové aplikace postavené na Ruby on Rails a je pro ně optimalizována. Je plně kompatibilní s Ruby verze 1.8.7 a podle oficiálních stránek v určitých případech až o 33% rychlejší. Na portále RubyInside je k dispozici záznam přednášky [19] obou autorů z konference Google Tech Talk (2009, San Francisco), o důvodech proč se do úprav pustili a také o optimalizacích, které provedli.

JRuby

JRuby[5] je implementace Ruby v Javě využívající plně možností Java Virual Machine. Například umožňuje plnou paralelizaci (žádný GIL). V současnosti se pracuje na podpoře Ruby verze 1.9. Jedná se o velmi živý projekt se 7 aktivními vývojáři (MRI naproti tomu je z největší části vyvíjeno pouze jedním člověkem).

BlueRuby

Jedná se o implementaci Ruby běžícího uvnitř SAP ERP [1] produktu, tedy další možnost jak rozšířit platformu SAP mimo jazyků ABAP [10] a Java. Zatím je projekt pouze ve stádiu experimentální implementace[6] a nehodí se pro reálné nasazení.

Rubinius

Zajímavý nápad napsat co největší část Ruby v samotném Ruby[7] - jedná se o přepsání nezbytně nutných částí interpretu do C++ a zbývajícího kódu do Ruby (například standardní třídy - `String`, `Array`, `Hash`, ...). Interpret obsahuje just in time (JIT) kompilátor – parsovaný skript přeloží prvně do binárního kódu spustitelného na dané platformě a až ten následně spustí. Tímto dosahuje zrychlení oproti MRI. Díky kompilaci do nativního kódu není zatížen GIL a umožňuje tedy plný paralelismus. V současné době je tento projekt stále ve vývoji - několik aplikací v něm již běží, ale pro reálné nasazení není příliš vhodný.

[4] http://www.rubyenterpriseedition.com/index.htm
[5] http://jruby.org
[6] https://wiki.sdn.sap.com/wiki/display/Research/BlueRuby
[7] http://rubini.us

```
:ahoj             # symbol s hodnotou "ahoj
:'ahoj, tady jezisek' # dlouhy retezec jako symbol
:'4'              # symbol s hodnotou "4"
```

Obrázek 2.14: Různé způsoby zadání symbolu.

```
'ahoj' == 'ahoj'                      # => true
'ahoj'.object_id == 'ahoj'.object_id # => false

:ahoj == :ahoj                        # => true
:ahoj.object_id == :ahoj.object_id    # => true
```

Obrázek 2.15: Porovnání a test identity symbolu a textového řetězce.

2.1.4 Binární rozšíření

Binární rozšíření skriptovacího jazyka je dynamická knihovna napsaná většinou ve stejném jazyce jako cílový interpret. Tyto knihovny při načítání zpravidla zaregistrují v interpretu nové třídy a jejich metody namapují na své vlastní funkce. Tímto postupem se rozšíří množina dostupných tříd a funkcionalita dostupná programátorovi skriptovacího jazyka. V případě MRI verze Ruby jsou rozšíření psaná v jazyce C. Lze je psát i v C++, ale kvůli odlišným překladovým a linkovacím konvencím se jedná o výjimečné případy.

Důvodů, proč napsat binární rozšíření místo kódu ve skriptovacím jazyce se dá najít několik. Prvním, a asi i nejpodstatnějším, je rychlost. Programy napsané v jazyce C budou prováděny rychleji než jejich skriptovací ekvivalenty, a proto zapouzdření nejnáročnějšího kódu z Ruby do jazyka C přináší výkonnostní vylepšení. V rámci standardních Ruby tříd je k dispozici modul pro práci s XML jménem `REXML` napsaný v čistém Ruby. Existuje k němu několik alternativ, z nichž například libxml-ruby[8](binární rozšíření zapouzdřující práci s XML knihovnou libxml[9]) je podle webových stránek projektu až o dva řády rychlejší.

Druhým důvodem může být nutnost. Standardní knihovny nemusí zpřístupňovat všechny požadované nízkoúrovňové operace operačního systému. Programátor je tak nucen napsat si mini rozšíření, které právě jím požadované vlastnosti zpřístupní. Poslední důvod je pohodlnost. Proč přepisovat celou již napsanou a navíc i odladěnou knihovnu do jiného jazyka, když stačí zpřístupnit pouze její aplikační rozhraní (API). Velice dobrým příkladem takovýchto knihoven jsou binární rozšíření pro frameworky grafických uživatelských rozhraní GTK[10] a QT[11].

Libovolnou knihovnu napsanou v jazyce C jako rozšíření pro skriptovací jazyk nelze použít přímo, protože by interpret nevěděl, jak s ní má zacházet. Je potřeba napsat rozhraní mezi touto knihovnou a interpretem, které je zodpovědné za dvě důležité činnosti: (1) musí v interpretu registrovat nabízené funkce a třídy knihovny a (2) musí poskytovat překladovou úroveň pro odlišné volací konvence knihovny a interpretu. MRI například vyžaduje u všech funkcí volatelných z Ruby, aby přijímaly a vracely pouze datový typ `VALUE` (popsán dále). Běžná knihovna nic o specifickém datovém typu interpretu neví a už vůbec netuší jak třídu `String` převést na reprezentaci řetězce v jazyce C. Za tyto typové převody je taktéž odpovědné binární rozšíření.

Tvorbu binárních rozšíření lze v základu rozdělit do dvou odlišných postupů - do postupu automatického a manuálního. V dalších odstavcích se každému přístupu budu věnovat více

[8] http://libxml.rubyforge.org/
[9] http://xmlsoft.org/
[10] http://ruby-gnome2.sourceforge.jp/
[11] http://rubyforge.org/projects/korundum/

```
if 1 == 2          # porovnej cislice 1 a 2
  puts "1 == 2"    # vypis text
end                # => nil
```

Obrázek 2.16: Každá jazyková konstrukce je výraz.

```
class Fixnum       # otevreni tridy Fixnum
  def +(oth)       # predefinovani metody +
    self * oth     # ktera bude ted nasobit
  end
end                # => nil

4 + 2              # => 8
```

Obrázek 2.17: Aritmetické operátory jsou normální metody.

do detailu.

Automatický postup

Pomocí speciálních nástrojů lze vrstvu mezi knihovnou a interpretem nechat vygenerovat automaticky. Příkladem může být projekt SWIG[12] (Simplified Wrapper and Interface Generator) sloužící pro automatickou tvorbu rozšíření pro knihovny napsané v jazyce C nebo C++. Seznam podporovaných skriptovacích jazyků je úctyhodný a mimo jiné zahrnuje známé jazyky jako Python, Perl, PHP nebo právě Ruby.

Použití podobných nástrojů je velice snadné. Většinou je bez složité a zdlouhavé konfigurace stačí spustit nad hlavičkovými soubory zpřístupňované knihovny. Jako výstup vygenerují rozhraní, zdrojový text v jazyce C nebo C++, který stačí už jen přeložit a poté nahrát v rámci programu vykonávaného v interpretu.

Vygenerované rozhraní zpravidla obsahuje pro každou nalezenou funkci knihovny jednu nově vygenerovanou funkci, která slouží jako překladový obal. Vezme parametry předané interpretem a přeloží datové typy do typů knihovny. Pro MRI je třeba převést předané objekty do jejich reprezentací v jazyce C. Tedy převést objekt typu `Fixnum` na `int`, `String` na `char *`, atd... Poté zavolat obalovanou funkci knihovny a nakonec převést její návratovou hodnotu zpět na Ruby objekt a vrátit ji interpretu. Rozhraní navíc obsahují jednu inicializační funkci, která všechny vygenerované obalovací funkce zaregistruje, aby byly v interpretu k dispozici.

Takto vygenerované rozhraní je velice přímočaré a programátor má minimální kontrolu nad jeho výslednou podobou. Zpřístupňuje všechny funkce knihovny pro jejich volání ze skriptovacího jazyka se zachováním sémantiky jejich argumentů bez jakékoliv vyšší abstrakce. Navíc zde může být problém se složitými vnitřními datovými typy, které se používají i vně knihovny.

Manuální postup

Manuálním napsáním spojovací vrstvy získá programátor plnou kontrolu nad výsledkem. Jako jednu z hlavních výhod vidím možnost vytvořit z vnitřních struktur používaných i vně knihovny třídy a z funkcí nad nimi operujícími metody této třídy. Výsledkem je krásné objektové chování, jak jsou programátoři v Ruby zvyklí na rozdíl od pouhého procedurálního

[12] http://www.swig.org/

```
module Ahoj       # deklarace modulu Ahoj
  def ahoj        # s metodou ahoj
    "Ahoj"        # vracejici text "Ahoj
  end
end

class Pozdravy   # deklarace tridy Pozdravy
  include Ahoj   # vlozeni modulu Ahoj
end

p = Pozdravy.new # tvorba instance tridy Pozdravy
p.ahoj # => "Ahoj" # zavolani metody z modulu
```

Obrázek 2.18: Moduly lze vkládat do tříd.

```
ruby extconf.rb   # vytvori makefile
make              # prelozi knihovnu
make install      # skopiruje knihovnu do systemu
```

Obrázek 2.19: Překlad binárních rozšíření v Ruby.

zpřístupnění všech funkcí. Navíc není problém zapouzdřovanou knihovnu rozšířit o další funkcionalitu, která se už ovšem bude odehrávat pouze na úrovni Ruby.

Na druhou stranu má tento postup oproti automatickému přístupu jednu značnou nevýhodu – čas programátora. Ten musí nastudovat rozhraní knihovny, navrhnout jeho vhodné zapouzdření a nakonec i celé rozhraní napsat. Díky časové náročnosti se plně manuální přístup v praxi používá pouze zřídka. Pokud je potřeba knihovnu ještě zapouzdřit na vyšší úrovni, například adaptací na objektový model, tak se spíše používá kombinace obou přístupů. Automaticky se vytvoří rozhraní knihovny přístupné ve skriptovacím jazyce a až toto rozhraní se následně zapouzdří do objektů a dalších specifických jazykových konstrukcí, které v jazyce C nebo C++ nejsou známy.

Překlad rozšíření

Vzhledem k faktu, že binární rozšíření jsou psána ve stejném jazyce jako cílový interpret skriptovacího jazyka, tak se velice podobně i překládají. Ve většině případů jsou interprety napsány v jazycích C nebo C++, překládají se tedy do binární podoby pomocí programů z balíčku `gcc`[13]. Překlad bývává řízen příkazem `make`, podle pokynů uvedených v souboru `Makefile`. Tento soubor bývá vygenerován ze šablon dodaných programátorem za pomocí nástrojů z rodiny `autotools`[14] (například script `configure`).

Binární rozšíření se pro MRI překládají také pomocí příkazu `make`. Ovšem na rozdíl od ostatních interpretů (i jiných skriptovacích jazyků) není použita sada programů `autotools`. `Makefile` popisující, jak se má rozšíření přeložit a nainstalovat totiž vygeneruje samo Ruby. Pro tyto účely existuje modul `mkmf`[15] (MaKe MakeFile), pomocí jehož metod programátor popíše na vysoké úrovni abstrakce, jak přeložit jeho rozšíření a tento skript potom distribuuje spolu se zdrojovými kódy. Celý sled příkazů pro překlad a instalaci je zobrazen na Obrázku 2.19.

[13] http://gcc.gnu.org/
[14] http://www.gnu.org/software/hello/manual/automake/Autotools-Introduction.html
[15] http://ruby-doc.org/stdlib/libdoc/mkmf/rdoc/index.html

Modul `mkmf` slouží hlavně k nastavení důležitých cest pro překlad automaticky a nezávisle na dané verzi či distribuci operačního systému (cesta k hlavičkovým souborům Ruby, jeho knihovnám). Mimo této činnosti modul nabízí metody, které umožňují podobné činnosti jako skript `configure` – lokalizovat systémové knihovny, ověřováním zda-li obsahují požadované symboly (funkce, proměnné, ...) a další podobnou funkcionalitu. Takto vytvořený skript je samozřejmě plnohodnotný program v Ruby. Lze tedy použít veškeré jeho možnosti včetně použití dalších knihoven. Například lze distribuovat pouze skript bez zdrojových kódů, který si prvně stáhne nejnovější verzi projektu a až poté bude pokračovat ve své obvyklé činnosti.

Datový typ VALUE

`VALUE` je datový typ specifický pro interpret MRI. Ostatní implementace Ruby stejně jako interprety úplně jiných skriptovacích jazyků ho neznají. V binárním rozšíření tento typ reprezujte Ruby objekt. Vše je objekt, a proto se tento datový typ používá všude – každá funkce volatelná z Ruby vrací `VALUE` jako svou návratovou hodnotu. Ve skutečnosti se jedná o přejmenování datového typu `unsigned long` obsahujícího adresu struktury reprezentující daný objekt. Většinou se tedy jedná o prostý ukazatel. Ovšem ne ve všech případech. Interpret využívá faktu, že ukazatele jsou na platformě x86 v paměti zarovnány - nejnižší dva bity jsou pro ukazatele vždy nulové (platí pro 32 bitovou architekturu, pro 64 bitů se zarovnává na 8 bytů - tedy poslední čtyři bity ukazatele jsou nulové).

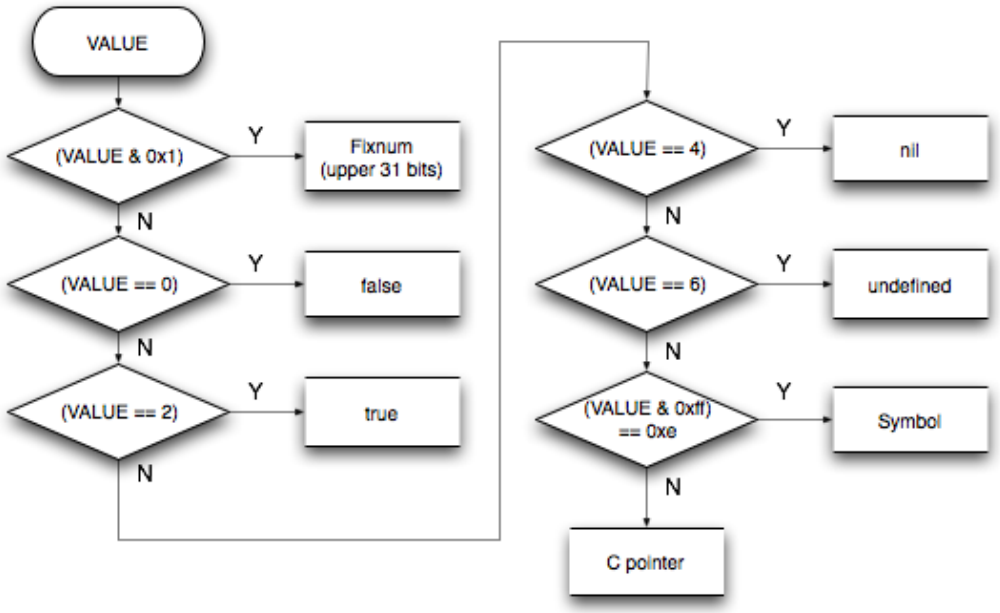
Například objekty třídy `Fixnum` (reprezentující celá čísla) nemají svou strukturu jako ostatní objekty, ale jejich hodnota je zakódována přímo do proměnné typu `VALUE` tak, že nultý bit (LSB, Least significant bit, nejméně významný bit) je nastaven na jedničku a ostatní bity jsou použity pro uložení vlastní hodnoty čísla. Tímto přístupem se ušetří dereferencování ukazatele při práci s celými čísly. Samozřejmě zobrazitelný rozsah čísel je tímto přístupem omezený na 31 bitů (číslo je uloženo v paměti jako znaménkové v doplňkovém kódu). Proto Ruby pro příliš velká celá čísla nabízí ještě třídu `Bignum`, jejíž hodnota není reprezentována přímo v místě ukazatele a nabízí podstatně větší rozsah.

Tato optimalizace není použita jen u celých čísel, ale také u singleton proměnných významných a často používaných tříd. Singleton je označení pro třídu, která má maximálně jednu instanci v celém programu. V Ruby se jedná o instance tříd `TrueClass` (instance se jmenuje `true`), `FalseClass` (instancí je objekt `false`) nebo `NilClass` (instance je `nil`, obdoba `NULL` v jazyce C, "prázdný" nebo také "žádný" objekt). Všechny tyto objekty nemají reprezentující strukturu, ale jejich obsah je přímo zakódován do typu `VALUE` pro rychlejší interpretaci skriptu. Postup interpretu při interpretaci obsahu `VALUE` je zobrazený na Obrázku 2.20, který jsem převzal z příspěvku [20] o tomto datovém typu.

2.2 Expertní systémy

Jako jeden ze způsobů adresace problému lidského přemýšlení vznikly expertní systémy. Jedná se o systémy umělé inteligence, které mají za cíl simulovat rozhodovací proces lidského experta. Tedy na základě uložených znalostí a uživatelských vstupů dojít k nějakému závěru. Například expertní systém MYCIN [7], vyvinutý na začátku 90. let minulého století, sloužil jako výpomoc lékařům k určení diagnózy pacienta. Pomocí pokládání otázek typu ano/ne nakonec uživatele informoval o jeho možných diagnózách.

Expertní systémy neadresují problém univerzálního řešení problému podobně jako General problem solver [8] (GPS). Místo toho po vzoru člověka, jeden konkrétní expertní



Obrázek 2.20: Diagram pro reprezentaci hodnoty v proměné VALUE.

systém umožňuje řešit problémy jen v určité předem omezené oblasti vědění. Používají se na místech, kde je obvyklé algoritmické řešení nevhodné, složité, popřípadě těžce udržovatelné.

Zpravidla expertní systémy obsahují několik částí:

Rozhodovací algoritmus. Jádrem expertního systému vždy musí být rozhodovací algoritmus, který řídí celý proces rozhodování vedoucí od počátečních informací až k dosaženému výsledku. Tento algoritmus je vždy spuštěn až na žádost uživatele, poté, co skončí se zadáváním úvodních informací.

Báze znalostí. Báze znalostí reprezentuje znalosti uložené v expertním systému. Zpravidla má formu pravidel když-tak (if-then), které se skládají ze dvou částí - levé a pravé strany. Levá strana obsahuje seznam podmínek, které musí platit před vykonáním samotného pravidla. Zatímco pravá strana zase obsahuje akce, které se mají vykonat po aktivaci pravidla. Tato pravidla umožňují měnit, mazat a vyvozovat různé závěry z pracovní paměti. Báze znalostí spolu s rozhodovacím algoritmem tvoří nezbytnou součást každého expertního systému.

Pracovní paměť. Pracovní paměť slouží k uchování aktuálně známých dat. Může také obsahovat vyvozené mezi závěry a nakonec zde také bude uložen konečný závěr, ke kterému expertní systém dojde. Jak zde data budou reprezentována záleží na daném expertním systému a nelze je příliš dobře generalizovat podobně jako v případě báze znalostí.

Vysvětlující podsystém. Další důležitou ovšem již nikoliv nezbytnou součástí expertního systému je vysvětlující podsystém. V mnoha případech stačí lidskému uživateli znát výsledek, ke kterému jeho expertní systém dospěl. V jiných případech je ovšem dobré znát i jak a proč k němu došel. Například v době ladění báze znalostí, je postup expertního systémů vedoucí k závěru naprosto nepostradatelný.

Uživatelské rozhraní. Poslední hlavní součástí expertních systému je uživatelské rozhraní. Slouží k ovládání celého systému, tedy k zadávání známých dat do pracovní paměti a ovládání celého prostředí – spuštění rozhodovacího algoritmu a získaní výsledku. U některých expertních systému lze přes uživatelské rozhraní také upravovat bázi znalostí,

ale to již záleží na konkrétním systému.

2.2.1 CLIPS

CLIPS je systém pro tvorbu expertních systémů napsaný v jazyce C vyvinutý v NASA na počátku 90. let. V současné době je dostupný jako public domain software a je tedy k dispozici zadarmo i pro komerční využití. Původní autor Gary Riley se systému stále věnuje a to jak opravováním nalezených chyb, tak i dalším vývojem.

CLIPS není expertním systémem, nýbrž se jedná o nástroj pro tvorbu expertních systémů. Z tohoto těžko postřehnutelného rozdílu slov vyplývá velice důležitý fakt – CLIPS samo o sobě je díky absenci jakékoliv báze znalostí naprosto k ničemu. Uživatel CLIPS musí nejprve dodat bázi znalostí a až poté se dá hovořit o expertním systému, který lze použít k rozhodovaní. Uživatelské rozhraní je uzpůsobeno i pro snadnou manipulaci s bázi znalosti.

2.2.2 Uživatelské rozhraní CLIPS

Uživatelské rozhraní CLIPS umožňuje k ovládání celého prostředí od manipulace s bázi znalostí přes práci s pracovní pamětí, až po spuštění rozhodovacího algoritmu. Rozhraním je specifický programovací jazyk, který se strukturou velice podobá jazyku LISP. Mít vlastní programovací jazyk pro ovládání celého prostředí má své výhody i nevýhody. Výhodou je nezávislost na výsledné podobě celkového uživatelského rozhraní – může se jednat o formu příkazového řádku stejně jako grafické okno očekávající jednotlivé příkazy jazyka. Popřípadě CLIPS ještě umožňují nahrát a vykonat libovolný textový soubor, ve kterém může být uložena báze znalostí s výchozí podobou pracovní paměti. Na druhou stranu ovšem tento přístup má i velkou nevýhodu – uživatel se musí naučit zcela nový jazyk. Což může přinášet problémy, zvlášť má-li výsledný expertní systém využívat běžně neprogramující obsluha.

V ukázkách uživatelského rozhraní CLIPS budu používat znak `#` pro oddělení kódu CLIPS od mého komentáře. Tyto komentáře ovšem do ukázek samotných patřit nebudou.

Datové typy uživatelského rozhraní

Než se dostanu k detailnějšímu popisu možností samotného rozhraní, popíši datové typy, se kterými vestavěný programovací jazyk pracuje. Nejzákladnějším datovým typem je `symbol`. Jedná se řetězec velkých nebo malých písmen a podtržítek či pomlček (podtržítko ani pomlčka ovšem nesmí být na první pozici). `Symbol` se používá na mnoha místech pro interní jména různých CLIPS struktur. Podobným datovým typem je `řetězec`, který se skládá s libovolné posloupnosti znaků uzavřené v uvozovkách. Obsahuje-li samotný řetězec znak pro uvozovky je potřeba ho ošetřit tzv. escape sekvencí podobně jako v jazyce C. Dále CLIPS rozlišuje dva číselné typy – jeden pro celá čísla a druhý pro desetinná čísla. Syntaxe jejich zápisu je shodná s jazykem C, proto je zde pro jejich složitost nebudu hlouběji popisovat. Poslední datový typ, který bych zde rád zmínil pro jeho použití v dalším textu je `fact_address`. Jedná se o referenci na vytvořený fakt v pracovní paměti CLIPS. Tyto reference uživatel sám explicitně nevytváří, ale jsou vytvářeny prostředím a poskytnuta uživateli k dalšímu zpracování.

Pracovní paměť

Známé skutečnosti jsou v CLIPS reprezentovány pomocí tzv. faktů. Na fakta se lze dívat jako na tvrzení, na jejíchž základě poté expertní systém odvozuje další nové skutečnosti.

```
(clovek "Jaroslav Cecho" 22 muz svobodny)
```

Obrázek 2.21: Uchování dat v rámci jednoho dlouhého seřazeného faktu.

```
(clovek "Jaroslav Cecho")              # hlavni fakt
(vek "Jaroslav Cecho" 22)              # zavisle fakta spojuci
(pohlavi "Jaroslav Cecho" muz)         # hlavni fakt s urcitymi
(rodiny-stav "Jaroslav Cecho" svobodny)# uchovavanymi atributy
```

Obrázek 2.22: Rozřezání sledovaných hodnot do separátních seřazených faktů.

Tento popis je možný ve dvou základních typech faktů – seřazených a neseřazených.

Seřazený fakt je vlastně seznam různých proměnných uložených v pracovní paměti CLIPS. Na první položku seznamu jsou kladena zvláštní omezení – musí obsahovat symbol a musí být vždy přítomna. Tato první položka je reprezentována jako jméno skupiny příbuzných seřazených faktů. Počet následujících položek je zcela libovolný a ty mohou nabývat libovolných datových typů mimo referencí.

Existují dva základní přístupy k ukládání dat pomocí seřazeného faktu. Prvním je vytvoření jednoho dlouhého faktu s pozičními položkami pro všechny popisované skutečnosti. Tedy význam hodnoty je dán její pozicí v seřazeném faktu. Příklad tohoto přístupu k uložení informací o člověku je na Obrázku 2.21. Tento přístup má několik problémů. Jedním z nich je adresace problému prázdných hodnot. Fakt z příkladu sestává ze 4 uchovávaných proměnných - jména, věku, pohlaví a rodinného stavu. Problém nastane v případě, že nebudeme o dané osobě znát například věk. Pak je otazné, co zde má být uloženo, protože vzhledem k pozičnímu charakteru všech hodnot musí být pozice obsazena. Proto je lepší každou sledovanou vlastnost dát do samostatného seřazeného faktu a spojovat je na základě jedné položky. V případě neznalosti nějaké hodnoty pak prostě fakt s chybějící vlastností chybí celý. Přepsaný příklad je zobrazen na Obrázku 2.22. Tento druhý přístup zvedá paměťovou náročnost výsledné aplikace, ovšem na druhou stranu je optimalizovanější pro vnitřní implementaci rozhodovacího algoritmu, a proto bude dosahovat lepších výkonů. Dalším problémem, který už ovšem seřazený fakt řešit neumí, jsou omezení. Například je očekávané, že věk bude číselná hodnota, ovšem do seřazeného faktu lze uložit zcela libovolné hodnoty.

Neseřazený fakt lze přirovnat ke strukturním proměnným v jazyce C. Ty mají svůj typ identifikovaný jménem a poté seznam jednotlivých položek. V CLIPS jsou neseřazené fakty vytvářené na základě šablon a proto se jejich popisu budu věnovat ještě před dalším popisem neseřazených faktů. Jak vyplývá z předcházejícího textu, šablona je předpis obsahující předně seznam všech možných položek pro fakt. V terminologii CLIPS se položky nazývají sloty. Jméno šablony musí být unikátní, protože se fakty vytvářejí za použití právě tohoto jména.

Každý slot musí mít své jméno, podobně jako položky ve strukturách jazyka C. Na rozdíl od nich ovšem je definice typu volitelná. Ve výchozím stavu může slot přijímat jakékoliv datové typy CLIPS. Uživatel může slotu volitelně nastavit datový typ, případně celou množinu datových typů, které bude akceptovat. Čímž se dá vyřešit dříve uvedený problém s ukládáním věku osoby. Navíc mimo specifikace typu lze slot vytvořit i s dalšími omezením. Pro číselné sloty lze nastavit minimální i maximální akceptovanou hodnotu. Slot lze nastavit jako tzv. multislot, který smí obsahovat více než jednu hodnotu, popřípadě tomuto multislotu lze nastavit minimální i maximální počet obsažených položek. Příklad vytvoření šablony pro popis člověka je na Obrázku 2.23.

Na základě vytvořené šablony lze vytvářet už jednotlivá neseřazená fakta. Zde je ana-

```
(deftemplate clovek                    # sablona pro cloveka
  (slot name)                          # uchovavajici jmeno
  (slot vek (type INTEGER))            # vek
  (slot pohlavi)                       # pohlavi
  (slot rodiny-stav)                   # a rodiny stav
)
```

Obrázek 2.23: Šablona pro reprezentaci dat o člověku.

```
(clovek                                # serazeny fakt
  (name "Jaroslav Cecho")              # s vyplnenymi sloty
  (vek 22)
  (pohlavi muz)
)
```

Obrázek 2.24: Neseřazený fakt popisující člověka.

logie se strukturní proměnou jazyka C stejná, prvně se vytvoří šablona (struktura) a poté se mohou vytvářet její fakta (proměnné s typem vytvořené struktury). První hodnota vytvářeného faktu musí být jméno již vytvořené struktury následované seznamem slotů s jejich hodnotami.

Neseřazená fakta mají několik výhod oproti seřazeným faktům, předně zapouzdřují data týkající se jedné popisované entity dohromady (v mých příkladech se jedná o popis člověka). Dále, jak jsem uvedl v popisu šablony, jednotlivé sloty lze omezit na akceptované hodnoty, což umožňuje mít čistý a kontrolovaný návrh. Neseřazené fakta také řeší problém neznámých hodnot - nezadané sloty jsou neznáme a nemusí se zde ukládat žádná hodnota značící "nic". Příklad neseřazeného faktu je na Obrázku 2.24.

V CLIPS se fakta v pracovní paměti vytvářejí pomocí funkce `assert`. Příklady uložení faktu do pracovní paměti CLIPS jsou na Obrázku 2.25. Jak jde v ukázce vidět, funkce se volá úplně stejně jak pro neseřazený, tak i pro seřazený fakt. Pro variantu seřazeného faktu ovšem šablona jména `clovek` již musí existovat jinak ukázka skončí chybou. Mazání fakt z pracovní paměti probíhá voláním funkce `retract`, která jako parametr přijímá referenci na fakt. Postup k získání reference na fakt bude vysvětlen v kapitole 2.2.2. Poslední možná operace s fakty je jejich změna, pro kterou se používá funkce `modify`, která opět očekává referenci na fakt jako svůj parametr.

Ještě bych se rád věnoval vnitřní implementaci rozdílu seřazených a neseřazených faktů v CLIPS. Na té nejnižší úrovni jsou totiž i pro seřazená fakta vytvářené šablony s právě jedním multislotem jménem `implied`. První hodnota seřazeného faktu slouží jako jméno automaticky vytvořené šablony. Celé chovaní je samozřejmě vyššími vrstvami CLIPS zapouzdřeno a uživatel si tak nemusí být vědom tohoto chování. Tento přístup s sebou ovšem přináší jednu nevýhodu. Jméno šablony musí být v rámci systému unikátní a první hodnota seřazených faktů slouží právě jako jméno šablony. Nelze tedy vytvořit šablonu a skupinu seřazených faktů stejného jméno, což vzhledem k odlišnému výskytu může uživateli způsobit zmatení. Není na první pohled hned viditelné, že se obě tato jména vybírají ze stejné množiny. Popisovaný problém lze obejít používání právě jednoho typu faktu v jedné aplikaci.

```
(assert (neserazeny-clovek "Jaroslav Cecho" 22 muz))
(assert (clovek (name "Jaroslav Cecho") (vek 22) (pohlavi muz)))
```

Obrázek 2.25: Vytváření faktů v pracovní paměti.

```
(defrule pouze-leva-strana            # pravidlo s levou stranou
  (neserazeny-fakt ?promena)          # hledajici dve fakta se
  (serzeny-fakt (prvni-slot ?promena))# stejnou promenou
-> )
```

Obrázek 2.26: Vytvoření pravidla obsahující pouze levou stranou.

Báze znalostí

CLIPS podobně jako mnoho dalších expertních systémů používá pro reprezentaci báze znalostí pravidla typu když-tak (if-then). Pravidla fungují na principu vyhledávání vzorů na množině existujících faktů. Levá strana pravidla proto obsahuje všechny vzory, které musí být nalezeny pro aktivaci pravidla. Pravá strana poté obsahuje seznam činností, které se mají stát po aktivaci pravidla. V rámci aktivace pravidel lze provádět mnoho různých činností jako například vypisování nalezených informací, ovšem nejdůležitější operace se týkají manipulace s fakty. Ta lze modifikovat, vytvářet či mazat.

Nejdříve se budu blíže věnovat možnostem levé strany pravidla. Jak již bylo řečeno, levá strana obsahuje vyhledávané vzory. Z tohoto důvodu je zde seznam faktů, které musí existovat. CLIPS zde nemá žádné restrikce a lze libovolně míchat seřazená a neseřazená fakta. Zápis se neliší od běžného zápisu faktu až na možnost místo libovolné hodnoty mimo šablony nebo skupiny podobných seřazených faktů zadat proměnnou. Ta začíná znakem otazníku následovaného jménem proměnné. Opravdová síla proměnných se projeví až když je použita stejná proměnná na více místech v rámci jednoho pravidla. Protože všechny proměnné stejného jména jsou nahrazeny stejnou hodnotou, čehož se využívá při vyhledávání faktů se stejnými ovšem předem neznámými hodnotami velice podobně jako v jazyce PROLOG [13]. Na Obrázku 2.26 je ukázka pravidla obsahující pouze levou stranou, na kterém mimo jiné prezentuji zápis proměnných. Pokud bude v systému vyhovovat danému vyhledávanému vzoru více možných kombinací faktů, tak to stejné pravidlo bude zavoláno vícekrát – jednou pro každou možnou kombinaci.

Je-li na levé straně pravidla uvedeno více hledaných faktů, je mezi ně položena logická spojka "a současně" (and). Pravidlo bude aktivováno právě tehdy, když budou v systému přítomny všechny fakta. Pro jemnější nastavení podmínek aktivace CLIPS umožňuje používat i ostatní logické spojky - "nebo" a negaci. Jejich použití je zobrazeno na Obrázku 2.27. Pravidlo z příkladu bude aktivovano pro každou dvojici faktů `prvni` a `treti`, `druhy` a `treti`, které mají stejnou druhou hodnotu.

Reference na nalezená fakta, díky kterých došlo k aktivaci pravidla, se dají v CLIPS uložit do proměnné. Na rozdíl od proměnných prezentovaných výše, zde už neplatí, že více výskytů se nahradí právě jednou hodnotou. Pokud se uživatel do proměnné rozhodne uložit referenci na aktivační fakt, proměnou tohoto jména v rámci levé strany již nelze použít. Příklad zápisu uložení reference na aktivní fakt je na Obrázku 2.28.

Jak již bylo uvedeno výše, pravá strana pravidla slouží k popsání věcí, které se mají stát po aktivaci pravidla. Lze zde volat libovolné funkce CLIPS, ovšem já se omezím na popis manipulace s fakty. Na pravé straně lze vytvářet fakta úplně stejně jako jinde v systému, tedy pomocí použití funkce `assert`. Navíc lze místo přesných hodnot použít proměnné získané z levé strany pravidla (s výjimkou proměnných referencujících fakta). Dalšími možnostmi

```
(defrule logicke-spojky                # pravidlo se spojkami
  (or
    (prvni ?promena)                   # jeden z techto faktu
    (druhy ?promena)                   # musi byt pritomen
  )
  (treti ?promena)                     # tento ovsem vzdy
-> )
```

Obrázek 2.27: Vytvoření pravidla s logickými spojkami.

```
(defrule reference-na-fakt
  ?ref <- (fakt ktery se ma najit)     # uchovej referenci
-> )
```

Obrázek 2.28: Uložení reference na aktivační fakt.

je aktivační fakta mazat pomocí funkce `retract` nebo je měnit pomocí funkce `modify`. Příklady použití všech tří funkcí jsou na Obrázku 2.29.

2.2.3 Prostředí v CLIPS

V rámci jedné instance CLIPS lze provozovat více nezávislých expertních systémů. Tato funkcionalita je umožněna tzv. prostředími. Prostředí je název pro zcela oddělenou část systému, která má vlastní bázi znalostí i pracovní paměť. Rozhodovací algoritmus musí být pro každé prostředí spuštěn samostatně. Využívají se v případě, že uživatel nechce z určitého důvodu běžet více procesů v operačním systému.

2.2.4 Rozšiřitelnost

Podobně jako lze binárními rozšířeními přidat do interpretu skriptovacího jazyka další funkcionalitu, i CLIPS lze rozšířit o nové funkce. Aplikační rozhraní CLIPS umožňuje uživateli zaregistrovat libovolnou novou funkci a poté ji z uživatelského rozhraní volat. Oproti rozšířením interpretu mají rozšíření CLIPS velké omezení. Nelze je nahrávat za běhu, tak jako v Ruby pomocí příkazu `require`. Musí tedy být do CLIPS přidány již v době kompilace. V ostatních ohledech se ovšem chovají stejně a řeší stejné problémy jako ony. Jsou odpovědné za převody typů a volajících konvencí mezi CLIPS a spojovanými knihovnami.

2.2.5 Rozhodovací algoritmus

Jak jsem již zmínil v předchozích odstavcích, rozhodovací algoritmus CLIPS funguje na základě vyhledávání vzorů na množině všech faktů přítomných v systému. Jedním z možných způsobu nalezení všech možných pravidel k aktivaci i spolu se všemi aktivačními parametry je sekvenční průchod existujících faktů. Tato naivní implementace by už pro systémy se stovkami faktů byla příliš pomalá a proto CLIPS využívá algoritmu optimalizovaného pro vyhledávání vzorů – algoritmus Rete [9] (Rete algorithm).

Detailní popis algoritmu je nad rámec této práce, proto zde uvedu pouze nástin jeho principu. V paměti vytváří stromovou strukturu, kde každý uzel mimo kořene odpovídá jednomu hledanému vzoru nějakého pravidla. Bude-li mít pravidlo celkem tři vyhledávané vzory, poté bude celé pravidlo reprezentované celkem třemi uzly. Cesta od kořene k listu odpovídá celé levé straně pravidla. V každém uzlu bude uložen ukazatel na fakta, které

```
(defrule prava-strana
  ?ref1 <- (prvni ?prom ?x)
  ?ref2 <- (clovek (name ?name) (nickname ?nick))
  ->
  (assert (treti ?x))                          # vytvor novy fakt
  (retract ?ref1)                              # vymaz prvni fakt
  (modify ?ref2 (nickname ?name) (name ?nick) ) # zmen druhy fakt
)
```

Obrázek 2.29: Ukázka pravé strany pravidla.

daný vzor splňují. V listech stromu poté fakta vyhovující pro aktivaci pravidla, jehož levá strana je ve stromu reprezentována. Jak jsou jednotlivé fakta vytvářena či modifikována, tak se propagují do tohoto stromu a při uložení až do nějakého listu mohou způsobit aktivaci pravidla.

Kapitola 3

Knihovna rbClips

V této kapitole se již důkladně budu věnovat své vlastní prací – knihovně rbClips. Postupně v následujících podkapitolách popíši její návrh, některé zajímavé implementační detaily a samozřejmě nevynechám shrnutý popis aplikačního rozhraní. Úplné detaily rozhraní už ovšem ponechám do programové dokumentace. V závěru kapitoly je uveden praktický příklad použití knihovny.

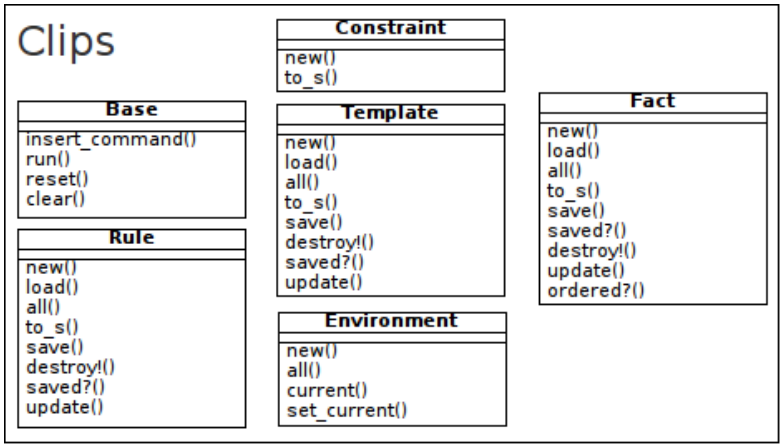
3.1 Požadavky

Cílem této prace a tedy i hlavním požadavkem je vytvořit knihovnu zpřístupňující funkce CLIPS v jazyce Ruby. A to nikoliv jako automaticky vytvořené rozšíření, které pouze zpřístupní funkce CLIPS. Musí nabídnout uživateli rozhraní, které bude objektové a bude v souladu s nepsanými konvencemi jazyka Ruby. Je vyžadováno zapouzdřit celou knihovnu do objektů a odstínit tak uživatele od nutnosti znát syntaxi uživatelského rozhraní CLIPS. Dále je nutné umožnit uživateli v rámci aktivace pravidla volat libovolné metody Ruby a tím v rámci vytvářeného expertního systému zpřístupnit celou škálu dalších již existujících knihoven Ruby.

3.2 Návrh

Aplikační rozhraní popsané v kapitole 3.3 jsem navrhl po vzoru open source knihovny ActiveRecords (AR). Stejně jako AR odstiňuje programátora od nutnosti znát SQL, tak i rbClips odstinňuje programátora od nutnosti znát programovací jazyk uživatelského rozhraní CLIPS. Veškerá funkcionalita je zapouzdřena do objektů a ovládá se pomocí metod těchto objektů. Hlavní důvod proč jsem rozhraní navrhl po vzoru AR je jeho velká rozšířenost. Jakýkoliv programátor, který alespoň částečné zná rozhraní AR, se v rbClips velice rychle zorientuje a nebude mít příliš velké problémy mou knihovnu začít používat.

Celá knihovna je zapouzdřena v jednom Ruby modulu jménem `Clips`. Ten obsahuje pět hlavních tříd: `Fact`, `Template`, `Rule`, `Environment`, `Constraint`, které popisují příslušné entity v CLIPS a zapouzdřují celou práci s nimi. Dále obsahuje několik pomocných tříd, které uživatel není oprávněn vytvářet přímo. Dostává je jako již vytvořené objekty do konfiguračních bloků hlavních tříd. Tyto pomocné třídy budou upřesněny dále v textu v popisu jednotlivých hlavních tříd, kde jsou používány. Nakonec obsahuje ještě modul `Base` sloužící pro volání důležitých funkcí CLIPS, které nejsou součástí jiných velkých entit. Seznam hlavních tříd a modulů je vyobrazen na Obrázku



Obrázek 3.1: Modul Clips a jeho hlavní třídy.

Inspirace AR se projevila i v určitých návrhových vzorech, které jsou shodné napříč celým API. Mimo výjimek zmíněných u popisu konkretních tříd níže, obsahují všechny třídy společnou podmnožinu metod zobrazenou na Obrázku 3.2. Entita zastupuje libovolnou hlavní třídu knihovny rbClips, dvojznak `::` přísluší třídní metodě, zatímco znak `#` patří instanční metodě. Inspirace AR není vidět jen u těchto společných metod, ale také v chování celé knihovny. Například vytvoření instance třídy `Fact`, `Rule` nebo `Template` automaticky nevytvoří odpovídající novou entitu i v prostředí CLIPS. K vytvoření (uložení) entity v prostředí CLIPS dojde až v okamžiku zavoláním metody `save`. Velice podobně to funguje i s aktualizací entit, veškeré změny je potřeba potvrdit opětovným voláním metody `save`, jinak se změny v prostředí CLIPS nikdy neobjeví. Tato funkcionalita lze přirovnat k operaci `commit` z transakčního zpracování jazyka SQL.

3.3 Aplikační rozhraní knihovny

V následujících odstavcích se budu detailněji věnovat jednotlivým hlavním třídám v knihovně rbClips. Níže uvedený popis nepokrývá veškerou nabízenou funkcionalitu, ale vyzdvihuje základní použití a upozorňuje uživatele na případné záludnosti. Zbývající části jsou popsány v programové dokumentaci.

3.3.1 Modul Base

CLIPS je celé prostředí pro tvorbu expertních systémů, nikoliv jen knihovna pro tvorbu faktů, pravidel a dalších větších entit. Proto obsahuje také velké množství různých doplňkových funkcí, které prostředí umožňují ovládat. Právě pro tyto funkce existuje modul `Base`, kde jsou pohromadě metody, které se nedotýkají žádného většího celku CLIPS pokrytého hlavními třídami, ale nelze si bez nich výsledný expertní systém představit. Dále uvedu dvě nejdůležitější metody, se kterými se uživatel mé knihovny může setkat:

run(FixNum) Metoda `run` slouží k zahájení aplikace pravidel. Běžný postup spuštění programu v CLIPS sestává z vytvoření množiny pravidel a prvotních faktů a následného spuštění vlastního algoritmu pro jejich zpracování. Knihovna rbClips na tomto přístupu nic nezměnila, proto se do zavolání metody `run` žádná pravidla neaplikují. Pro zamezení případného nekonečného cyklu slouží volitelný argument určující maximální počet pravidel,

- `Entita#save` Vytvářené entity nejsou po vzoru ActiveRecords do CLIPS uloženy automaticky při vzniku objektu, ale je třeba je explicitně uložit pomocí metody `save`. Velice podobně se knihovna chová i případě editace entity, například změna hodnoty slotu nějakého faktu, která taktéž vyžaduje volání metody `save` k uložení i do CLIPS.

- `Entita#saved?` Metoda `saved?` slouží ke zjištění zda-li je daná entita uložena v CLIPS. Po uložení entity do CLIPS slouží ke zjištění zda-li je reprezentace entity v Ruby objektu shodná s jeho reprezentací v CLIPS. Má dvě možné návratové hodnoty – `true` a `false`.

- `Entita#destroy!` Pro vymazání entity z CLIPS slouží metoda `destroy!`. V případě úspěšného vymazání vrací `true`, pokud již daná entita v CLIPS neexistuje a nelze tedy vymazat, tak vrací `false`. V obou případech je pouze vymazána reprezentace v CLIPS, samotný Ruby objekt je ponechán nedotčen a lze ho tedy bez problému znovu uložit. V případě problému s vymazáním metoda vyhazuje výjimku příslušného typu, například nelze smazat šablonu, kterou současně používají některá neseřazená fakta.

- `Entita#update` Pokud uživatel provede na aktuálním objektu změny, kterých se chce zbavit a obnovit původní uloženou podobu z CLIPS, tak může použít metodu `update`. Tato metoda neslouží pro uložení případných změn do CLIPS, ale vždy pouze k aktualizování hodnot v Ruby objektu z CLIPS.

- `Entita::all` Vrátí pole všech entit daného typu, které se aktuálně nacházejí v aktivním prostředí CLIPS.

- `Entita::load` Vrátí jednu nebo více entit dle předaných kritérii. Jako parametr bere v případě tříd `Template` a `Rule` jméno hledané entity, tedy parametr třídy `String` a vrací právě jednu nalezenou instance případně `nil`. Pro třídu `Fact` akceptuje o dost širší možností argumentů, které zde nebudu zbytečně opisovat z programové dokumentace.

Obrázek 3.2: Společná podmnožina metod většiny tříd knihovny rbClips.

která se smí aplikovat. Návratovou hodnotou je celkový počet aplikovaných pravidel (jedno pravidlo může být aktivováno a tedy i započítáno vícekrát).

insert_command(String) Knihovna rbClips zapouzdřuje nejpoužívanější části CLIPS, bohužel ovšem ne veškerou možnou funkcionalitu. Kvůli absenci 100% pokrytí všech možností CLIPS jsem vytvořil metodu `insert_command`. Na vstupu metoda dostává jako parametr jeden validní příkaz CLIPS, který následně provede a vrací objekt `true` nebo `false` podle úspěchu vykonání. Kvůlí chování funkcí, které jsou v prostředí CLIPS nakonec zavolány, je pro správnou funkčnost nutné, aby předaný řetězec obsahoval maximálně jeden příkaz. V případě, že jich bude obsahovat více, tak se provede pouze první z nich. Navíc se do Ruby vrátí informace, že vše proběhlo v pořádku, což může být velice matoucí.

3.3.2 Třída Environment

Třída `Environment` slouží pro vytváření a manipulaci s CLIPS prostředími. Pomocí níže zmíněných API knihovny se vždy nastaví jedno prostředí jako aktivní a všechna pravidla a fakta jsou do něj uložena. Po startu programu je jedno prostředí vytvořeno automaticky,

```
include Clips
env = Environment.new          # vytvori nove prostredi
cur = Environment.current      # vrati aktualni prostredi
env.set_current                # nastavi aktualni prostredi
```

Obrázek 3.3: Příklad použití třídy Environment.

aby ho uživatel nemusel vždy vytvářet explicitně.

RbClips využívá možností, které CLIPS pro prostředí nabízí, a proto se chová bohužel mírně neobjektově. V jeden okamžik může být aktivní pouze jedno prostředí a to i v případě více vláknové aplikace. Navíc je potřeba u vícevláknových aplikací dávat pozor na potencionálně nebezpečné volání `Environment::new`. CLIPS nově vytvořené prostředí rovnou nastaví jako aktuální, což porušuje konvence rbClips. Prostředí se stává aktivní až po zavolání metody `set_current` (obdoba metody `save` z ostatních hlavních tříd). Volání konstruktoru třídy tedy vytvoří nové prostředí, a poté přepne zpět do starého prostředí. Toto přehazování je nebezpečné právě ve více vláknových aplikacích, kde mohou nastat těžko odhalitelné časově závislé chyby nad daty. Například se ukládaná pravidla či fakta mohou uložit do zdánlivě náhodně zvolených jiných prostředí. Z tohoto důvodu doporučuji vytvořit všechna potřebná prostředí ještě před samotným vytvořením vláken.

Ukázka práce s prostředími je uvedena na Obrázku 3.3.

3.3.3 Třída Constraint

Druhá hlavní třída `Constrait` zapouzdřuje kompletní práci s omezeními pro sloty neseřazených faktů. Vytvoření omezení je snadné, konstruktor akceptuje jako parametr jedno asociativní pole se čtyřmi možnými klíči. První možnost klíč `:type` jako hodnotu akceptuje jeden nebo pole možných typů. Typy jsou reprezentovaný symboly a smí nabývat hodnot `:symbol`, `:string`, `:lexeme`, `:integer`, `:float`, `:number`, `:instance_name`, `:instance_address`, `:instance`, `:external_address`, `:fact_address` případně hodnoty `:any` reprezentující jakýkoliv datový typ (implicitní hodnota). Druhý možný klíč je `:values` akceptující jednu či pole všech různých hodnot, kterých slot s tímto omezením smí nabývat. Poslední dva klíče `:range` a `:cardinatily` akceptují Ruby objekt třídy `Interval`. Klíč `:cardinality` omezuje počet možných hodnot v multislotech, zatímco `:range` omezuje možné velikosti uložitelných čísel.

Pro názornější pochopení je na Obrázku 3.4 uveden příklad, který vytváří dvě různá omezení. Slot s omezením objektu `a` bude moci obsahovat pouze hodnoty typu `řetězec` nebo `symbol`. Druhý příklad, objekt `b`, nastavuje slotu povolení pouze pro celá čísla v rozsahu 3 až 13.

Uvnitř CLIPS bohužel neexistuje možnost, jak množinu omezení uložit jako samostatnou entitu. Omezení je vždy uloženo pouze v rámci jednotlivých slotů vytvářených šablon. Z tohoto důvodu v knihovně rbClips objekty třídy `Constraint` nemají metodu `save` a nejdou tedy ani uložit. Existují pouze na úrovni interpretu Ruby. Pokud programátor nastaví nějakému slotu omezení, vezme se jeho aktuální stav a ten se použije. Pokud je tento `Constraint` objekt posléze změněn, tak se změny již zpětně nepromítnou do všech míst, kde byl použit. Proto je vhodné vytvořit všechny objekty popisující omezení na začátku programu ještě před tvorbou šablon a dále je pokud možno již neměnit.

```
a = Constraint.new :type => [:string, :symbol]
b = Constraint.new :type => :integer, :range => 3..13
```

Obrázek 3.4: Příklad použití třídy Constraint.

```
# tvorba sablony pomoci asociativniho pole
Template.new :name => 'animal', :slots => %w(name age race)

# tvorba samlony pomoci bloku
Template.new 'animal' do |t|
  t.slot :name            # vytvor slot "name"
  t.slot :age             # vytvor slot "age"
  t.slot :race            # vytvor slot "race"
end
```

Obrázek 3.5: Dvě možnosti tvorby stejné šablony.

3.3.4 Třída Template

Třída `Template` slouží pro práci s CLIPS šablonami. Vytvořit šablonu lze dvěma základními postupy. Konstruktor akceptuje buď konfigurační asociativní pole nebo jméno šablony s konfiguračním blokem. V případě konfigurace užitím asociativního pole, musí toto pole obsahovat povinně dva klíče - `:name` a `:slots`. Klíč `:name` slouží pro zadání jména šablony, které musí být v daném prostředí unikátní. Druhý požadovaný klíč `:slots` slouží jako seznam slotů vytvářené šablony. Akceptuje jako hodnotu buď pole objektů třídy `String` nebo `Symbol`, které bude reprezentováno jako jména jednotlivých slotů. V případě, kdy uživatel potřebuje změnit slotům výchozí chování, je pro něj nachystána druhá varianta klíče `:slots`, která přijímá další (již tedy vnořené) asociativní pole. Klíči tohoto vnořeného pole jsou jména slotů, přičemž hodnotami je v pořadí již třetí asociativní pole, které obsahuje seznam atributů daného slotu. Atributy slotů existují dva: `:multislot` povolující uložení více hodnot v rámci jednoho slotu a `:default` obsahující výchozí hodnotu (implicitně `nil`).

Zápis užívající postupně tři vnořená asociativní pole není zrovna pro člověka přehledný, proto existuje druhá, o dost čitelnější varianta volání konstruktoru užitím bloku. Konfigurační blok bude zavolán s jednou proměnou pomocné třídy `TemplateCreator`, která má metodu `slot` sloužící pro vytvoření nového slotu. Jméno slotu je předáno jako první argument. Volitelně lze jako druhý argument předat asociativní pole popisující atributy slotu (tedy to samé asociativní pole, které se předává jako druhé vnořené při první variantě volání konstruktoru). Oba postupy jsou zobrazeny na Obrázku 3.5.

3.3.5 Třída Fact

Jak bylo uvedeno v kapitole 2.2.2 CLIPS rozlišují dva typy faktů - seřazené a neseřazené. Díky velice podobné vnitřní implementaci obou druhů faktů v CLIPS, jsem je oba zapouzdřil do jediné Ruby třídy jménem `Fact`. Většína rozhraní této třídy je shodná pro oba typy faktů, ale samozřejmě existují části, ve kterých se API liší.

Podobně jako celé rozhraní rbClips i změna obsahu faktů je navrženo po vzoru ActiveRecords. Programátora AR tedy nepřekvapí, že změny faktů se neprojeví v CLIPS okamžitě, ale jsou jen lokálně uloženy v instanci třídy. Až po zavolání metody `save` se vše uloží. Bohužel CLIPS se ke změně faktu chová velice neprakticky – místo změnění jednotlivých atributů se aktualizovaný fakt prvně vymaže, a poté se vloží nový s již aktualizovanými hodnotami. Samozřejmě toto probíhá jako jedna atomická operace. Tedy nenastane stav,

```
Fact.new 'Jarcec', [20, "muz"]
Fact.new 'pohlavi', %w(muz zena)
```

Obrázek 3.6: Ukázka vytvoření seřazeného faktu.

kdy by zbytek systému mohl postřehnout, že jeden fakt byl vymazán a nový ještě nevložen. Pro rbClips to ovšem znamená obrovský problém. Instance třídy si udržuje ukazatel na strukturu faktu v CLIPS a aktualizací se tento ukazatel stává neplatným. Objekt je tedy po aktualizaci již vlastně neuložený (jeho vnitřní ukazatel neukazuje na neplatný fakt). Tento neduh se mi nepodařilo opravit tak, abych nerozbil jinou, již napsanou funkcionalitu, případně budoucí plány s rozšířením podpory pro vícevláknové aplikace. Proto je na uživateli rbClips, aby zajistil, že aktualizaci provede až v momentě, kdy už daný fakt nebude potřeba nadále měnit. Ze zkušenosti z používání AR, které je předlohou rozhraní, si myslím, že to nebude až takový problém dodržet.

Seřazený fakt

Na následujících pár řádcích bych rád popsal metody specifické pro seřazený fakt. Rozhraní třídy Fact je ovšem optimalizované hlavně pro neseřazené fakty, protože ty obsahují více možností pro čistší návrh rozhraní. Z tohoto důvodu bych doporučil používat raději neseřazenou variantu faktu.

V seřazené variantě faktu knihovna rbClips uchovává pole hodnot, které je vrací metoda `slots`. První atribut neseřazeného faktu není ve vráceném poli obsažen, protože se k němu CLIPS chovají jako ke jménu skupiny faktů. Jak již bylo zmíněno výše, ve skutečnosti se jedná o jméno automaticky vygenerované šablony. Pro získání prvního atributu (jména) slouží metoda `name`.

Vytvoření seřazeného faktu je velice jednoduché. Konstruktor v tomto případě očekává dva parametry. Prvním je jméno skupiny (objekt třídy `String` nebo `Symbol`) a druhým pole všech atributů neseřazeného faktu. Příklady možných zápisů tvorby seřazeného faktu jsou uvedeny na Obrázku 3.6.

Neseřazený fakt

Práce s neseřazenými fakty je v prostředí rbClips o dost pohodlnější než v případě seřazených faktů. Existují dvě základní metody podobné těm, které jsou k dispozici v případě seřazeného faktu. První z nich je metoda `slot(String or Symbol)` vracející hodnotu uloženou v daném slotu a metoda `template` vracející šablonu, podle které byl fakt vytvořen (tedy objekt třídy `Template`).

Navíc ovšem rbClips plně využívá dynamičnosti a otevřenosti Ruby a pro jednotlivé sloty vytváří přístupové metody shodného jména jako sloty samotné. Čímž velice napodobuje chování ActiveRecords, které vytváří přístupové metody instance podle názvů sloupců příslušných relačních tabulek. Jednotlivé instance neseřazených faktů mají různé metody, vygenerované podle šablony, ze které byly vytvořeny. Z tohoto důvodu je dobré dávat pozor a nevytvářet šablony se jmény slotů, které jsou stejné jako již existující metody. Příkladem nevhodného jména slotu je například `save`. Pokud bude šablona obsahovat takto pojmenovaný slot, tak původní metoda `save`, bude překryta metodou pro přístup k atributům faktu a nebude vůbec možné instanci uložit do prostředí CLIPS.

V případě tvorby neseřazeného faktu očekává konstruktor dvojici parametrů. Prvním z nich musí být uložená instance šablony (třída `Template`), druhým je asociativní pole, kde

```
# pro neserazeny fakt je zapotrebi sablony
animal = Template.new :name => 'animal',
  :slots => %w(name age race)
animal.save

# tvorba dvou neserazenych faktu
Fact.new animal, :name => "Azor"
Fact.new animal, :name => "Zorka", :age => 2
```

Obrázek 3.7: Ukázka vytvoření neseřazeného faktu.

klíče jsou jména slotů šablony a jejich hodnoty jsou uloženy do těchto slotů při vytváření faktu. Ukázka je k dispozici na Obrázku 3.7.

3.3.6 Třída Rule

Poslední z hlavních tříd knihovny rbClips zapouzdřuje práci s pravidly. Ty v CLIPS mají nezastupitelnou roli, protože je v nich uložena informace, co se má a za jakých podmínek provést. Rozhodovací jádro CLIPS jen vybírá podle různých, mnohdy uživatelsky nastavitelných kriterií, které pravidla se mají a v jakém pořadí provést. V rámci CLIPS jednou vytvořená pravidla již nelze dále upravovat, proto i rbClips vytváří tyto objekty needitovatelné. Výjimkou z needitovatelnosti je samozřejmě tvorba pravidla v rámci konstruktoru třídy.

Tvorba nového pravidla je mírně složitější, a proto se jí budu detailně v následujících odstavcích věnovat. Nové pravidlo se vytváří tvorbou nového objektu třídy `Rule` tak, jak je v Ruby světe zvykem voláním metody `new`. Tato metoda vyžaduje předání bloku přijímací právě jeden argument, pomocí kterého bude následně celé pravidlo sestaveno. Předaný argument je instance třídy `RuleCreator` umožnující pomocí svých metod nadefinovat jak levou, tak i pravou stranu pravidla.

Jak bylo uvedeno v kapitole 2.2.2 pravidla mohou obsahovat proměnné. Ty jsou při aktivaci pravidla zaměněny za skutečné hodnoty a vždy platí, že všechny proměnné stejného jména jsou nahrazeny právě jednou hodnotou. V rbClips roli proměnných přebírají symboly. Existují ovšem dva speciální symboly, které jsou výjimkou z uvedeného pravidla, a které se používají v podstatě pouze u seřazených faktů. Prvním je symbol `:one`, sloužící pro nahrazení libovolné hodnoty atributu, ovšem více výskytů může být nahrazeno více možnými hodnotami. Slouží k přeskakování nezajímavých hodnot neseřazených faktů, které programátora v rámci vytvářeného pravidla nezajímají. Druhým speciálním symbolem je `:any`. Chová se velice podobně jako symbol `:one`, akorát za něj může být v seřazeném faktu substituována více než jedna hodnota.

Nejdůležitější metodou pro editaci levé strany pravidla je metoda `pattern`. Slouží pro vyhledávání určitých vzorů (takzvaný pattern matching). Má velice bohaté možnosti argumentů, se kterými může být volána a vrací vždy objekt třídy `FactAddress`. Vrácený objekt může být následně použit v metodách upravujících pravou stranu pravidla. První možnost volání je určena pro vyhledávání seřazených faktů a spočívá v předaní jednoho objektu třídy `String` nebo `Symbol` a následně jednoho pole. První argument je použit jako jméno skupiny příbuzných faktů a pole je interpretováno jako vyhledávací vzor na této množině. Symboly v tomto poli jsou použity jako zástupné proměnné, které mají v rámci aplikování pravidla vždy stejnou hodnotu tak, jak bylo popsáno v předchozích odstavcích. Druhým možným voláním je předat jako první argument objekt třídy `Template` a následně jednoho

asociativního pole. Velice podobně jako v předchozím případě potom první argument slouží pro určení, na jaké podmnožině faktů se bude vyhledávat (ve všech faktech, vytvořených podle předané šablony). Asociativní pole obsahuje jako klíče sloty šablony a jako hodnoty vyhledávané vzory. Opět lze použít symboly jako zástupné hodnoty stejné přes volání všech metod vytvářeného pravidla. Poslední možností je předat metodě `pattern` jeden objekt třídy `String`, který bude beze změny uložen do CLIPS. Slouží pro vlastní specifikování vyhledávaného vzoru pro případy, kdy rbClips nedostatečně zapouzdřuje možnosti CLIPS.

Velice podobnou metodou je metoda `retract`. Jako jedna z mála upravuje jak levou, tak zároveň i pravou stranu pravidla. Na levé straně pravidla určuje vyhledávací vzor a proto akceptuje stejné možnosti parametrů jako metoda `pattern`. Navíc s ní sdílí i stejnou návratovou hodnotu, objekt třídy `FactAddress`. Na rozdíl od ní ovšem, po aktivaci pravidla nalezený fakt smaže.

Metody `pattern` a `retract` afektují levou stranu pravidla, tedy určují za jakých podmínek se pravidlo aplikuje. Pokud je jich v definici pravidla uvedeno více, tak je mezi ně dána logická spojka "a současně" (and). Tedy všechny vyhledávané vzory musí být nalezeny, aby dané pravidlo mohlo být aplikováno. Pro možnost jemnějšího definování levé strany existují tří metody s obdobným použitím - `and`, `or` a `not`. Tyto metody vyžadují předání bloku s jedním parametrem, do kterého bude předán opět objekt třídy `RuleCreator`. Vyhledávané vzory definované uvnitř předaného bloku budou hledaný s logickou spojkou podle příslušného názvu metody. Pro ještě jemnější nastavování podmínek rbClips umožňuje vzájemné zanořovaní těchto bloků do sebe. Důležitou výjimkou z prezentovaného chování je absence metody `retract`, která v bloku `not` a ve všech jemu vnořených již není k dispozici. Chybí zde z logického důvodu. Metoda `retract` slouží po aktivaci pravidla ke smazání faktu z paměti. Ovšem blok `not` zaručuje, že daný fakt neexistuje a logicky tedy není ani co vymazat. Velice obdobně je uvnitř `not` bloku změněno chování metody `pattern`, která místo objektu `FactAddress` vrací `nil`.

Velice užitečným rozšířením možností CLIPS v knihovně rbClips je možnost v rámci pravé strany volat metodu libovolného objektu Ruby. Registrovaní objektu a jeho metody se provádí metodou `rcall`, která má minimálně dva povinné argumenty. Prvním je objekt, na kterém má byt zavolána metoda specifikovaná v druhém parametru. Další parametry jsou nepovinné a slouží k předání parametrů volané metody. Samozřejmostí je možnost zadat na tomto místě symbol, který v konečném voláním metody bude nahrazen příslušnou aktivační hodnotou, za kterou byl symbol nahrazen. Tento způsob reakce na aktivaci pravidla je jeden z nejdůležitějších přínosů knihovny rbClips do výsledných expertních systému na ni založených. Možnost volat v rámci pravidla metodu na libovolném objektu Ruby umožňuje do výsledného expertního systému zakomponovat libovolnou již existující knihovnu Ruby. Navíc přináší do procesu vykonávání pravidla dynamiku typickou pro jazyk Ruby. V rámci volané metody lze totiž například získávat data z relačních databází a na jejich základě se rozhodovat o dalším postupu.

Poslední zatím nezmíněnou metodou objektu `RuleCreator` je metoda `rhs`. Akceptuje právě jeden argument třídy `String`, který bez jakýchkoliv kontrol vloží do pravé strany vytvářeného pravidla. Slouží jako záchrana v případě, že požadovaná funkcionalita není mou knihovnou zapouzdřena. Komplexnější příklad tvorby pravidla je na Obrázku 3.8.

Po vytvoření pravidla je potřeba ho ještě uložit do CLIPS pomocí metody `save`. Pravidla se začnou aktivovat až po spuštění rozhodovacího algoritmu, který se v rbClips spouští pomocí metody `run` modulu `Base`.

```
savci = Rule.new "savci" do |r|   # pravidlo "savci"
  r.pattern 'animal', :name       # musi existovat "animal"
  r.pattern 'warm-blooded', :name # a zaroven teplokrevne
  r.not do |n|                    # nesmi ovsem klast
    n.pattern 'lays-eggs', :name  # vejce
  end
  r.assert 'mammal', :name        # vytvor noveho savce
end
```

Obrázek 3.8: Ukázka vytvoření pravidla.

3.4 Implementační zajímavosti

V následujících odstavcích bych se rád věnoval implementačním zajímavostem, na které jsem v rámci této práce narazil. Nejprve bych se ovšem rád věnoval výběru způsobu tvorby samotného binárního rozšíření. Nejjednodušší přístup automatického vytvoření celého rozhraní by zachoval dostupné funkce CLIPS i s jejich parametry. Uživatel knihovny by byl nucen učit se navíc ještě syntaxi uživatelského rozhraní CLIPS, což bylo požadavky vyloučeno. V úvodních verzích návrhu jsem se proto přiklonil spíše ke kombinovanému přístupu. Nad automaticky vygenerovaným rozhraním jsem napsal vyšší zapouzdřovací logiku, která celé procedurální chování zapouzdřila do objektů. S tímto přístupem dokonce vzniklo několik funkčních prototypů knihovny na prezentaci, že to takto může fungovat (tzv. "proof of concept"). Prototypy sice fungovaly, ovšem kvůli neustálému převádění všech datových typů z Ruby objektů do jazyka C a zpět při každé operaci trpěly slabším výkonem. Například i jinak velice rychlá práce s ukazateli jazyka C byla zpomalena neustálými konverzemi do objektů, které byly předány do vyšších vrstev ke zpracování, a následnému převedení zpět na ukazatele, aby se na jejich základě provedla v CLIPS určitá operace (například nalezení určitého faktu).

Proto jsem se nakonec rozhodl celé rozhraní napsat v jazyce C a dodávat jako čistě binární rozšíření. Po důkladném přečtení dokumentace k Ruby jsem navíc zjistil, že umožňuje do objektů uložit i ukazatel na libovolnou strukturu, o kterou se navíc stará sám interpret. Garbage collector před uvolněním objektu obsahujícího ukazatel zavolá nejprve registrovanou uvolňovací funkci na tento uložený ukazatel a až následně objekt odstraní. Nejčastěji je registrovaná funkce `free()`, ale v případě potřeby lze pro složitější struktury zaregistrovat zcela libovolnou funkci (přijímací právě jeden parametr typu `void *`). Ta může strukturu postupně uvolnit tak, aby nedošlo k žádným únikům paměti.

Tímto způsobem do objektů ukládám vlastně vytvořené struktury obsahující minimálně jednu proměnou. Tou je ukazatel `void *` ukazující na vnitřní struktury CLIPS. Například Ruby objekt reprezentující fakt má ve vnořeném ukazateli uložen odkaz na strukturu, která mimo jiné obsahuje odkaz na jeho strukturu v CLIPS. Tím pádem mám k dispozici rychlé spojení mezi Ruby objektem a jeho CLIPS reprezentací. Nemusím pokaždé popisovanou entitu vyhledávat a všechny operace jsou podstatně rychlejší.

3.5 Překlad

Zdrojové kódy knihovny rbClips jsou k dispozici na přiloženém CD. Na témže CD je k dispozici i celý Git repositář (Git je nástroj pro udržování verzí souborů podobně jako CVS nebo Subversion). Čtenář má tedy přístup k celé historii projektu, ne jen k jeho odevzdané verzi. Vývoj knihovny dnem odevzdání bakalářské práce navíc nekončí a nejaktuálnější verzi

lze stáhnout z veřejného git repositáře serveru Github [1].

Knihovna má dvě hlavní závislosti - CLIPS a Ruby. CLIPS je možné stáhnout z oficiálních stránek projektu a to zvlášť balíček se zdrojovými kódy a s různými `makefile` soubory. `Makefile` soubor pro překlad knihovny bohužel obsahuje chybu. Chybí v něm flag `-fPIC`, bez kterého se nedá dynamická knihovna vytvořit. Proto jsem se rozhodl umístit zdrojové kódy CLIPS, oproti kterým je má bakalářská práce vytvořena a otestována, přímo do svého repositáře. Odpadá tedy nutnost ručního stahování CLIPS.

Druhá hlavní závislost – interpret Ruby – již v dodaném repositáři z důvodu přítomnosti ve většině distribucí a velice aktivnímu vývoji přítomna není. Pro překlad je potřeba mít v systému nainstalován nejen samotný interpret, ale i jeho případné vývojové balíčky. Požadovaná verze je 1.9.x, dnes již pouze udržovaná větev 1.8 není rbClips podporována a knihovna nepůjde s touto verzí přeložit. Pro případ absence v nějaké neobvyklé distribuci je archív s podporovanou verzí Ruby přítomen také na přiloženém CD.

Z obvyklých závislostí je potřeba mít prostředí pro překlad, tedy hlavně překladač `gcc`. Z rodiny nástrojů `autotools` je potřeba pouze balíček s programem `make`. Ostatní nejsou potřeba, protože `Makefile` pro překlad binárního rozšíření vytvoří Ruby skript, který je pro tyto účely dodán. V kořenové složce projektu je již vytvořen hlavní soubor `Makefile`, který obsahuje všechny části překladu pohromadě. Pro samotný překlad tedy stačí spustit program `make` v kořenové složce knihovny. Jako první se přeloží zahrnutá verze CLIPS a až následní vlastní knihovna rbClips. V poslední fázi překladu se spustí automatické testy pro ověření správnosti a funkčnosti překladu.

3.6 Praktický příklad

Jako poslední podkapitolu o knihovně rbClips bych rád uvedl trošku složitější příklad spolu s jeho slovním popisem. Ukázka je zobrazena na Obrázku 3.9 a ukazuje vyhledávaní savců na množině různých zvířat a jejich vlastností.

Nejprve vytvořím všechny fakta o zvířatech a jejich vlastnostech za použití seřazeného faktu (poznámka `#1`). Za povšimnutí stojí forma zápisu, kdy nepoužívám jeden dlouhý fakt se seznamem všech vlastností daného zvířete. Místo toho mám vždy vlastnost a jako její atribut jméno zvířete ke kterému náleží. Tento způsob mi jednodušší následný zápis pravidel.

První pravidlo (poznámka `#2`) vyhledává savce podle dvou vlastností. Za prvé musí byt teplokrevní a za druhé nesmějí klást vajíčka. Pravidlo krásně prezentuje použití symbolu jako zástupné proměnné pro jméno zvířete a použití bloku `not`. Druhé pravidlo (poznámka `#3`) vyjadřuje lidsky zapsanou větu: "Je-li rodič savcem, poté i jeho potomek je savcem". Opět lze krásně vidět funkci symbolů jakožto zástupných znaků a to jak pro jméno rodiče, tak i pro jméno jeho potomka. Každé pravidlo bude spuštěno pro všechny platné kombinace, které budou v množině vstupních faktů nalezeny. Navíc spuštění rozhodovacího algoritmu (poznámka `#4`) není omezeno žádným maximálním počtem aplikovaných pravidel, takže rozhodovací algoritmus svou činnost ukončí až již nebude mít žádná pravidla k aktivaci.

Po dokončení ukázkového programu přibudou do množiny faktů celkem 4 nová fakta pro savce. Na základě prvního pravidla se bude jednat o identifikování savce kočky a psa a na základě druhého pravidla ještě fakta pro jejich potomky, tedy kotě a štěně.

[1] Zdrojové kódy projektu lze ze serveru Github stáhnout pomocí příkazu
git clone git://github.com/jarcec/rbclips.git

```
#1
Fact.new('animal', %w(dog)).save
Fact.new('animal', %w(cat)).save
Fact.new('animal', %w(duck)).save
Fact.new('animal', %w(turtle)).save
Fact.new('warm-blooded', %w(dog)).save
Fact.new('warm-blooded', %w(cat)).save
Fact.new('warm-blooded', %w(duck)).save
Fact.new('lays-eggs', %w(duck)).save
Fact.new('lays-eggs', %w(turtle)).save
Fact.new('child-of', %w(dog puppy)).save
Fact.new('child-of', %w(cat kitten)).save
Fact.new('child-of', %w(tutrle hatchling)).save

#2
mammal1 = Rule.new "mammal" do |r|
  r.pattern 'animal', :name
  r.pattern 'warm-blooded', :name
  r.not do |n|
    n.pattern 'lays-eggs', :name
  end
  r.assert 'mammal', :name
end
mammal1.save

#3
mammal2 = Rule.new "childs" do |r|
  r.pattern 'mammal', :name
  r.pattern 'child-of', :name, :young
  r.assert 'mammal', :young
end
mammal2.save

#4
Base.run
```

Obrázek 3.9: Příklad použití knihovny rbClips

Kapitola 4

Závěr

Knihovnu rbClips jsem úspěšně naprogramoval podle všech požadavků a odzkoušel na základě testů, které jsou součástí knihovny. V současné době probíhá její další testovaní pro odhalení různých chyb, kterých jsem si při vlastním testování nevšiml. Dalším krokem bude použít knihovnu k tvorbě expertního systému v Ruby pro automatickou klasifikaci vzorků malware, tak jak jsem zmínil v úvodu.

Mezi výhody knihovny rbClips a přednosti jejího použití oproti přímému použití CLIPS mohu uvést:

Zapouzdření procedurálního chování. Manuální přístup k tvorbě zpřístupňovacího binárního rozšíření mi umožnil zapouzdřit jednotlivé skupiny funkcí do tříd. Například všechny funkce pracující s fakty jsou, ať už přímo či nepřímo, volány pomocí metod třídy Fact. Samotná třída navíc celé rozhraní přizpůsobuje objektovému návrhu a nepsaným konvencím jazyka Ruby. Programátor používající rbClips si vůbec nemusí být vědom, že na nižší úrovni se používá pouze procedurální knihovna napsaná v jazyce C.

Zapouzdření syntaxe CLIPS. Zapouzdření celého rozhraní knihovny rbClips do objektů umožnilo schovat uživatelské rozhraní CLIPS. Programátor se tak nemusí učit další jazyk, navíc od Ruby velice odlišný, kterým by ovládal prostředí CLIPS. Některé méně využívané konstrukce CLIPS nejsou bohužel v rbClips přímo podporovány, a proto jsem přidal možnost vkládat a vykonávat validní úryvky CLIPS kódu přímo.

Zpětné volání Ruby metod. Knihovna obsahuje možnost jako akci pro pravidla (konsekvent) nastavit metodu libovolného objektu Ruby. Ta se po aktivaci pravidla zavolá. Do takto volané metody lze předat libovolné parametry a to nejen ty známé v době tvorby pravidla, ale třeba i ty které vedly k aktivaci pravidla.

Využitý všech dostupných Ruby knihoven. Ve výsledném expertním systému jehož základem bude knihovna rbClips, lze využívat naprosto libovolnou, již existující knihovnu Ruby. Díky možnosti volání libovolné metody v rámci aktivace pravidel lze například v době vyhodnocení pravidla pouštět dotazy v relačních databázích nebo spouštet externí utility.

Přímý přístup ke CLIPS. Pro případ, že uživatelem požadovaná funkcionalita CLIPS není knihovnou rbClips zpřístupněna, je možné ovládat celý expertní systém i za použití uživatelského rozhraní CLIPS. Tato výhoda se může obzvláště hodit v případě přechodu již existujícího systému na knihovnu rbClips, protože existující zdrojové kódy lze použít přímo. Přechod lze uskutečnit po částech a není potřeba hned všechny zdrojové kódy přepsat do Ruby.

Knihovna by mohla sloužit jako základ doplňku ke stávajícímu antivirovému jádru. To klasifikuje viry na základě vyhledávání různých vzorů v prohledávaných souborech. K tomu

by šlo vytvořit druhé jádro kvalifikující malware na základě jeho chování (tedy behaviour detekce), jejímž základem by byla právě knihovna rbClips. Operační systém by hlásil akce jednotlivých programů. Například alokace velkého bloku souvislé paměti, rozšifrování dat do této paměti a její následné spuštění. rbClips knihovna by poté rozhodovala zda-li je chovaní programů správné a přípustné. Případně by označila daný program za malware a oznámila vše uživateli. Samozřejmě by výsledné nové jádro antiviru mohlo využívat veškeré možnosti, které nabízí jak CLIPS, tak i Ruby. Například nechat zkontrolovat stávajícím jádrem už rozšifrovaný kód malwaru, který předtím nemusel být detekován.

Literatura

[1] André Maassen, Markus Schoenen, Detlev Frick, Andreas Gadatsch: *SAP R/3*. Computer Press, 2007, ISBN 978-80-251-1750-7.

[2] Anthony Molinaro: *SQL Cookbook*. O'Reilly Media, 2005, ISBN 0-596-00976-3.

[3] Brian W. Kernighan, Dennis M. Ritchie: *Programovací jazyk C*. Computer Press, březen 2006, ISBN 80-251-0897-X.

[4] Chamond Liu: *Smalltalk, Objects, and Design*. iUniverse, 2000, ISBN 1-583-48490-6.

[5] David Flanagan: *Java In A Nutshell*. O'Reilly Media, 2005, ISBN 0-596-00773-6.

[6] David Flanagan, Yukihiro Matsumoto: *The Ruby Programming Language*. O'Reilly Media, Inc., 2008, ISBN 0-596-51617-7.

[7] Edward H. Shortliffe: *Computer-based Medical Consultations: MYCIN*. Elsevier Science Ltd, 1976, ISBN 0-444-00179-4.

[8] George W. Ernst: *Gps: A Case Study in Generality and Problem Solving*. Academic Press, 1969, ISBN 0-122-41050-5.

[9] Josph C. Giarratano, Gary D. Riley: *Expert Systems: Principles and programming*. Thomson Learning, Inc, 2005, ISBN 0-534-38447-1.

[10] Julia Kirchne: *ABAP Basics*. SAP PRESS, 2007, ISBN 1-592-29153-8.

[11] Larry Wall, Tom Christiansen, Jon Orwant: *Programming Perl*. O'Reilly Media, 2000, ISBN 0-596-00027-8.

[12] Mark Lutz: *Programming Python*. O'Reilly Media, 2006, ISBN 0-596-00925-9.

[13] Max Bramer: *Logic Programming with Prolog*. Springer, 2005, ISBN 1-852-33938-1.

[14] Paolo Perrotta: *Metaprogramming Ruby*. The Pragmatic Programmers LLC, 2009, ISBN 1-934-35647-6.

[15] Ray Lischner: *C++ in a Nutshell*. O'Reilly Media, 2003, ISBN 0-596-00298-X.

[16] Sam Ruby, Dave Thomas, David Heinemeier Hansson: *Agile Web Development with Rails*. Pragmatic Bookshelf, 2009, ISBN 1-934-35616-6.

[17] Shelley Powers: *Learning JavaScript*. O'Reilly Media, 2008, ISBN 0-596-52187-1.

[18] WWW stránky: Oficiální webová prezentace jazyka ruby a MRI. `http://www.ruby-lang.org`.

[19] WWW stránky: Prezentace autorů Ruby EE interpretu o optimalizacích MRI interpretu. http://www.rubyinside.com/how-phusion-built-a-more-efficient-ruby-1-8-interpreter-2906.htm.

[20] WWW stránky: Příspěvek o datovém typu VALUE MRI interpretu. http://www.oreillynet.com/ruby/blog/2006/01/the_ruby_value_1.htm.

[21] WWW stránky: Shrnutí historie ruby prezentované na konferenci RubyConf v roce 2006. http://blog.nicksieger.com/articles/2006/10/20/rubyconf-history-of-ruby.

[22] WWW stránky: Srovnání různých implementací jazyka Ruby. http://www.igvita.com/2009/11/20/state-of-ruby-vms-ruby-renaissance.

[23] Yukihiro Matsumoto: *Ruby in a Nutshell*. O'Reilly Media, Inc., 2001, ISBN 0-596-00214-9.