Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Portace knihovny Prawn z Ruby do Pythonu 3

Bc. Michal Molhanec

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Michal Molhanec

Informatika

Název práce

Portace knihovny Prawn z Ruby do Pythonu 3

Název anglicky

Porting of Prawn Library from Ruby to Python 3

Cíle práce

Hlavním cílem práce je portování knihovny Prawn, pro generování PDF dokumentů z Ruby do Pythonu 3. K dosažení tohoto účelu budou porovnány konstrukty a datové typy těchto jazyků a bude implementován nástroj pro částečný automatický překlad mezi nimi.

Metodika

V práci budou použity programovací jazyky Python 3 a Ruby. Portovanou knihovnou bude Prawn, sloužící pro tvorbu PDF dokumentů. Vlastní portace bude založena na poloautomatickém překladu. Jednodušší konstrukce budou překládány automaticky z Ruby AST do Pythonu, složitější konstrukce budou přepsány ručně.

Doporučený rozsah práce

50-60 stran

Klíčová slova

skriptovací jazyky, python, ruby, překlad, datové typy

Doporučené zdroje informací

BEAZLY, David a JONES, Brian K. Python Cookbook. O'Reilly, 2013. 3rd ed. 706 s. ISBN 978-1449340377.

FLANAGAN, David a Yukihiro MATSUMOTO. The Ruby programming language. 1st ed. Beijing: O'Reilly, c2008, xi, 429 s. ISBN 9780596516178.

FOWLER, Martin a PARSONS, Rebecca. Domain-specific languages. Upper Saddle River: Addison-Wesley, 2011. 597 s. ISBN 9780321712943.

FULTON, Hal Edwin. Ruby: kompendium znalostí pro začátečníky i profesionály. Vyd. 1. Brno: Zoner Press, 2009, 765 s. Encyklopedie Zoner Press. ISBN 978-80-7413-018-2.

LUTZ, Mark. Programming Python. O'Reilly, 2011. 4th ed. 1632 s. ISBN 978-0596158101.

MÜLLER, Karel. Programovací jazyky. Vyd. 1. Praha: Vydavatelství ČVUT, 2002, 219 s. ISBN 80-01-02458-x.

NEŠVERA, Šimon. Programovací jazyky: cvičení. Vyd. 1. Praha: Vydavatelství ČVUT, 2002, 114 s. ISBN 80-01-02522-5.

PARR, T. *Language implementation patterns : create your own domain-specific and general programming languages.* Raleigh: Pragmatic Bookshelf, 2010. ISBN 978-1-934356-45-6.

THOMAS, Dave. Programming Ruby 1.9 & 2.0. 4th ed. Pragmatic Bookshelf, 2013, 888 s. ISBN 978-1937785499.

Předběžný termín obhajoby

2016/17 ZS – PEF

Vedoucí práce

Ing. Marek Pícka, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 20. 2. 2016

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 20. 2. 2016

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 04. 10. 2016

Čestné prohlášení

Prohlašuji, že svou diplomovou práci „Portace knihovny Prawn z Ruby do Pythonu 3“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne

Poděkování

Rád bych touto cestou poděkoval Ing. Marku Píckovi, Ph.D, za vedení práce.

Portace knihovny Prawn z Ruby do Pythonu 3

Souhrn

Práce se zabývá automatickým překladem kódu napsaného v jazyku Ruby do Pythonu 3. Nejprve jsou prozkoumány principy překladu, jak je popisuje odborná literatura. V další části se porovnávají jednotlivé jazykové konstrukce a jsou navrženy způsoby jejich překladu.

Na základě této teorie byl vyvinut překladač, který svou funkčnost ukazuje na překladu knihovny Prawn pro generování PDF souborů.

Klíčová slova

skriptovací jazyky, python, ruby, překlad, datové typy, pdf, abstraktní syntaktický strom

Porting of Prawn Library from Ruby to Python 3

Summary

This work focuses on automatic translation of code written in Ruby programming language into Python 3. First part investigates principles of computer programming languages translation as is described in the literature. Second part compares elements of these two programming languages and suggests ways how these elements can be translated.

Based on this theory I developed Ruby-to-Python translator. Its working is demonstrated on the translation of the Prawn library for generating PDF files.

Keywords

scripting languages, python, ruby, translation, data types, pdf, abstract syntax tree

Obsah

1 Úvod **10**

2 Cíl práce a metodika **10**
2.1 Programovací jazyk . . . 10
2.2 Použité verze . . . 11

3 Teoretická východiska **13**
3.1 Doménově zaměřené jazyky . . . 13
3.2 Varianty překladu jazyků . . . 15
3.2.1 Překlad řízený vstupní syntaxí . . . 15
3.2.2 Překlad řízený pravidly . . . 17
3.2.3 Překlad řízený modelem . . . 17
3.2.4 Přístup k vlastní implementaci . . . 19

4 Porovnání vybraných konstrukcí jazyků Ruby a Python 3 a návrh mechanismu jejich překladu **22**
4.1 Neznámá hodnota . . . 23
4.2 Logické hodnoty . . . 23
4.2.1 Literály . . . 23
4.2.2 Konverze na logické hodnoty . . . 24
4.2.3 Logické operátory `and` a `or` . . . 24
4.3 Čísla . . . 26
4.3.1 Celá čísla . . . 27
4.3.2 Reálná čísla . . . 28
4.3.3 Další typy čísel . . . 28
4.3.4 Běžné operace s čísly . . . 29
4.4 Symboly . . . 30
4.5 Řetězce . . . 31
4.5.1 Binární data . . . 32
4.5.2 Jednotlivý znak . . . 32
4.5.3 Zápis literálů . . . 33
4.5.4 Konverze na řetězec . . . 33
4.6 Regulární výrazy . . . 34
4.7 Rozsahy . . . 36
4.8 Pole . . . 37
4.8.1 Vybrané operace s poli . . . 37
4.8.2 Vykrojení (slice) . . . 39
4.8.3 Procházení pole . . . 41
4.9 Slovníky . . . 41
4.9.1 Procházení . . . 41
4.10 Definice modulů . . . 43
4.11 Standardní moduly . . . 47
4.12 Standardní mixiny . . . 49
4.12.1 Mixin *Comparable* . . . 49
4.12.2 Mixin *Enumerable* . . . 50

4.13 Třídy . . . 52
4.13.1 Volání metody předka . . . 53
4.13.2 Proměnné a metody objektu třídy . . . 53
4.14 Metody . . . 53
4.14.1 Funkce a metody . . . 54
4.14.2 Názvy . . . 55
4.14.3 Operátory . . . 56
4.14.4 Parametry metod . . . 60
4.14.5 Výchozí hodnoty parametrů . . . 61
4.14.6 Návratová hodnota . . . 63
4.15 Bloky . . . 66
4.15.1 Implicitní rozklad více argumentů . . . 68
4.15.2 Přiřazení do proměnné metody . . . 69
4.15.3 Podporované speciální bloky . . . 71
4.16 Porovnávání . . . 71
4.17 Řídicí výrazy . . . 72
4.17.1 Podmínky (if, unless) . . . 73
4.17.2 Podmínky ve formě modifikátorů . . . 75
4.17.3 Ternární operátor . . . 76
4.17.4 Několikanásobný výběr (case, when) . . . 76
4.17.5 Cykly . . . 80
4.17.6 Skoky (break, next, redo) . . . 82
4.18 Běžné idiomy . . . 84
4.18.1 Hluboká a mělká kopie . . . 84
4.18.2 Pozdní inicializace . . . 87
4.18.3 Test na *nil* . . . 88

5 Vlastní řešení **90**
5.1 Celkový přehled . . . 90
5.2 Formální popis transformací . . . 92
5.2.1 Neformální shrnutí . . . 94
5.2.2 Jednotlivé transformace t_i . . . 95
5.3 Omezení implementace . . . 95
5.4 Změny nutné v knihovně Prawn . . . 95

6 Závěr **97**

Seznam použitých zdrojů **99**

A Vstupní AST **100**
A.1 Jednoduché literály . . . 100
A.2 Regulární výrazy . . . 101
A.3 Závorky . . . 102
A.4 `defined?` . . . 103
A.5 Operátory . . . 103
A.6 Přiřazení konstantě . . . 104
A.7 Rozlišení názvů . . . 104

A.8 Proměnná . . . 105
A.9 Jednoduché přiřazení . . . 105
A.10 Vícenásobné přiřazení . . . 107
A.11 Řetězce a symboly s vloženými výrazy . . . 108
A.12 Rozsahy . . . 109
A.13 Pole . . . 109
A.14 Slovníky . . . 110
A.15 Moduly . . . 110
A.16 Třídy . . . 111
A.17 Deklarace metody . . . 112
A.18 Sběr parametrů . . . 114
A.19 Parametr typu blok . . . 114
A.20 Bloky . . . 115
A.21 Předání bloku . . . 115
A.22 Volání bloku . . . 116
A.23 Return . . . 116
A.24 Volání metody . . . 116
A.25 super . . . 117
A.26 Podmínky . . . 117
A.27 Cykly . . . 119
A.28 break a next . . . 121
A.29 Vícenásobný výběr . . . 121
A.30 Výjimky . . . 123
A.31 Rejstřík uzlů vstupního AST . . . 128

B Obsah doprovodného CD **129**
B.1 Struktura . . . 129
B.2 Jak spustit příklad . . . 129
B.3 Jak spustit příklad s lokálně nainstalovanými interpretery . . . 130

Seznam obrázků

1 Typy překladu . . . 16
2 Typy kopií . . . 86
3 Schéma implementace . . . 91

1 Úvod

Pro svoji práci jsem si vybral téma skriptovacích jazyků. Vždy mne zajímalo, jak jednotlivé skriptovací jazyky přistupují k obdobným konstrukcím, v čem se liší, v čem si jsou naopak podobné. Přičemž lze jasně vidět konvergenci, kdy se skriptovací jazyky vzájemně inspirují. Například v klasické příručce *The Ruby Programming Language* (Flanagan a Matsumoto, 2008) najdeme hned několik odkazů na Python. Příkladem budiž možnost ukončení cyklu výjimkou `StopIteration`. Ruby v tomto případě převzalo nejen vlastní mechanismus, ale dokonce přímo i název této výjimky.

Do jazyků se také doplňují vlastnosti, které se stávají populární. Za příklad mohou sloužit prvky funkcionálního programování, jako jsou generátory a klauzury (uzavření, closure). Zajímavé pak je, jak se jazyky liší v jejich realizaci. Třeba právě u klauzur jsou značené rozdíly mezi Pythonem a Ruby.

Práce částečně navazuje i na moji bakalářskou práci, která se také zabývala programovacími jazyky, byť z odlišného pohledu (Molhanec, 2005). Zatímco tehdy mne zajímala syntaxe, v této práci se zabývám spíše tím, jak jednotlivé konstrukce jazyka fungují.

2 Cíl práce a metodika

Základním cílem této práce je prozkoumání možnosti překladu kódu napsaného v jazyce Ruby do Pythonu 3. Ke konkrétní demonstraci je použita knihovna Prawn napsaná v Ruby, která slouží pro generování PDF souborů.

Vlastní portace bude založena na poloautomatickém překladu. Bude vyvinut automatický překladač, který zvládá podmnožinu jazyka Ruby přeložit do Pythonu. Jednodušší konstrukce budou překládány automaticky z Ruby AST do Pythonu, složitější konstrukce budou přepsány ručně.

2.1 Programovací jazyk

Termín programovací jazyk se často chápe jen v tom nejužším smyslu, jako část, která je definována jeho syntaxí a sémantikou. Chceme-li ale v něm vytvořil jakýkoli reálný program, bude nás zajímat programovací jazyk v nejširším možném smyslu. Z komplexního pohledu lze u jednotlivých reálných programovacích jazyků rozlišit celou řadu jednotlivých prvků.

Za *interní* prvky jazyka bychom mohly označit:

- Paradigma a koncepty, na kterých je jazyk založen.

- Vlastní syntaxe a sémantika.
- Typový systém.
- Standardní knihovna.
- Používané idiomy, praktiky, konvence a doporučované best practices.

Stejně důležité je i okolí jazyka, *externí* prvky:

- Vlastní implementace jazyka. Dnes je běžné, že mají jazyky více rovnocenných implementací a je potřeba pečlivě rozlišovat, které vlastnosti jsou garantovány jazykem pro všechny korektní implementace, a co je již vlastnost konkrétní implementace.

  Například výchozí implementace Pythonu v jazyce C, tzv. CPython, implementuje správu paměti pomocí metody počítání odkazů. Díky tomu bude například soubor uzavřen v okamžiku, kdy se odkaz na jeho instanci stane nedostupným. Spolehneme-li se na to, může se stát, že kód nebude spolehlivě fungovat například v Jythonu, implementaci jazyka Python v Javě, který přenechává správu paměti platformě Java.

- Rozšiřující knihovny a způsob jejich distribuce.
- A celá řada dalších podpůrných nástrojů, podpora v editorech, nástrojích pro správu verzí a podobně.

Chceme-li tedy překládat kód v jednom jazyce do druhého, je potřeba se do různé míry vypořádat s většinou těchto prvků.

2.2 Použité verze

Tato práce používá v době psaní aktuální verze. Konkrétně u knihovny Prawn jde o verzi 2.0, u Pythonu o verzi 3.5 a u Ruby o verze 2.2 a 2.3.

Jak již bylo zmíněno v předchozí sekci, oba jazyky existují ve více implementacích. Základní, prakticky definiční, jsou implementace v jazyku C. U Pythonu se nazývá „CPython“[1], v případě Ruby „Ruby MRI“, *Matz's Ruby Interpreter*[2]. *Matz* je přezdívka autora Ruby, Yukihiro Matsumota. Tyto implementace udávají směr vývoje jazyka a jsou typicky o jednu až nekolik verzí napřed oproti ostatním.

Tato práce se soustředí na tyto výchozí implementace, už proto, že ostatní se s postupem času k nim přibližují. Přesto stojí za to představit si i další. Komentáře, které

[1] https://www.python.org/
[2] https://www.ruby-lang.org/

se týkají verzí a kompatibility s CPythonem, resp. Ruby MRI, se vztahují k době psaní této práce, tj. přibližně polovině roku 2016.

Aktivní jsou verze napsané pro platformu Java. V případě Pythonu se nazývá Jython[3], v případě Ruby JRuby[4]. Jython bohužel zatím nemá stabilní verzi podporující Python 3, JRuby je mnohem aktuálnější s podporou Ruby 2.2.

Velmi zajímavé jsou také projekty, které se snaží implementovat maximum kódu v jazyce, který realizují. V Případě Pythonu jde o projekt PyPy[5], oficiálně podporující již verzi 3.2, v případě Ruby jde o projekt Rubinius[6], podporující Ruby 2.2. Pro virtuální stroj Rubinia existuje dokonce i náznak implementace Pythonu[7].

Zajímavé jsou i implementace pro platformu .Net, IronPython[8], resp. IronRuby[9], bohužel po té, co je Microsoft přestal dotovat, tak se jejich vývoj prakticky zastavil.

Protože se Python i Ruby hojně využívají pro tvorbu webových aplikací na straně serveru, vzniklo i mnoho více či méně kompletních překladačů těchto jazyků do Javascriptu, aby se daly používat i na straně klienta. Z těch kompletnějších zmiňme PyJS[10] pro Python, který se bohužel zastavil u verze Pythonu 2.6, a překladač Opal[11] pro Ruby, který implementuje přibližně Ruby 1.9/2.0.

[3] http://www.jython.org/
[4] http://jruby.org/
[5] http://pypy.org/
[6] http://rubinius.com/
[7] https://github.com/vic/typhon
[8] http://ironpython.net/
[9] http://ironruby.net/
[10] http://pyjs.org/
[11] http://opalrb.org/

3 Teoretická východiska

K problematice programovacích jazyků existuje mnoho literatury. Už proto, že psát složitější programy přímo ve strojovém kódu nebo jazyku symbolických adres je obtížné, byly překladače a interpretry programovacích jazyků jednou z prvních aplikací, které se s vývojem počítačů vytvářely. Historicky se nicméně většina literatury věnuje spíše oblastem jako je lexikální analýza a parsování, generování strojového kódu či bajtkódu, interpretaci, implementaci správců paměti a podobně.

Oblast překladu mezi dvěma jazyky dlouho zůstávala stranou. To se změnilo s rozpoznáním a popularizací doménově zaměřených jazyků, *Domain Specific Languages*.

3.1 Doménově zaměřené jazyky

Běžný programátor se většinu svého času pohybuje v prostředí univerzálních (*general purpose*) programovacích jazyků jako je Java, Python nebo Ruby. Tyto jazyky mají velmi široké uplatnění, od použití na serveru, přes desktopové aplikace, po jednoduché skripty. Je jistě praktické moci používat jeden jazyk pro rozličné účely a nemuset se učit pro každý typ aplikace zvláštní jazyk.

Odvrácenou stránkou univerzálnosti je nicméně nutně menší přizpůsobení cílové oblasti. Ve některých oblastech to nemusí vadit, jinde je to extrémně nepraktické. A zde nastupují doménově zaměřené jazyky. Představme si například, že bychom místo jednoduchého regulárního výrazu pro české PSČ:

```
/\d\d\d ?\d\d/
```

museli psát něco ve stylu:

```
sm = StringMatcher.new
sm.add_digit
sm.add_digit
sm.add_digit
sm.add ' ', required: false
sm.add_digit
sm.add_digit
```

Doménově zaměřené jazyky nejsou samy o sobě nic nového. Zrovna výše uvedené regulární výrazy se používají již velmi dlouho, stejně tak lze jmenovat i další mikrojazyky, které se dají nalézt v knihovnách mnoha univerzálních programovacích jazyků, jako jsou formátovací řetězce ve stylu `printf` nebo `strftime`. V širším smyslu patří mezi DSL vlastně i zástupné znaky pro vyhledávání souborů (`* ?`), nástroje typu *make*, TEX, nebo nástroje jako *awk* a *sed* (Herold, 2004).

DSL mohou být samozřejmě mnohem větší, jako jsou jazyky zaměřené na vědu a statistiku: Igor Pro, Origin, Matlab, Mathematica, SAS, R, Julia apod.

Jazyky zmiňované doposud patří do skupiny tzv. *externích* DSL. To jsou samostatné, klasické programovací jazyky s vlastní syntaxí a sémantikou, jen úzce zaměřené na konkrétní oblast. Kromě nich existují i *interní* DSL, tj. jazyky které využívají prostředí hostitelského jazyka, používají jeho syntaxi netradičním způsobem tak, aby syntaxe byla pohodlnější pro daný účel (Fowler, 2011, s. 15). Mohly bychom například výše uvedený regulární výraz zapsat pomocí *fluentního* rozhraní (Fowler, 2011, s. 67) takto:

```
StringMatcher.digit.digit.digit.optional(' ').digit.digit
```

Výhodou interních DSL je možnost plně využít sílu univerzálního jazyka a odpadá nutnost mít vlastní parser. Nevýhodou je skutečnost, že výrazové prostředky jsou vždy limitovány syntaktickými i sémantickými pravidly hostitelského jazyka a někdy také poměrně složitý kód vlastní knihovny, který často musí využívat možnosti hostitelského jazyka na maximum, což jednak ztěžuje vývoj a rozšiřování interního DSL, jednak může znamenat obtíže i při ladění kódu v interním DSL napsaného, kdy se například chyba projeví výjimkou hloboko v knihovně realizující interní DSL.

Přestože jsou DSL vlastně velmi staré a vždy byly hojně používané, stály do značné míry na okraji zájmu vývojářů a teoretiků. Změnu přineslo dle mého soudu několik hlavních skutečnosti. Jednou z nich je popularita nových dynamických jazyků jako je Ruby či Boo[12], jejichž flexibilní syntaxe je ideální pro interní DSL. Například Boo podporuje makra na úrovni AST, nikoli jen prostých řetězců jako třeba klasické céčko, čímž je možné jazyk rozšiřovat a modifikovat skoro libovolně (Rahien, 2010).

Další důležitou událostí bylo vydání knihy Martina Fowlera a Rebeccy Parsons Domain-Specific Languages (Fowler, 2011). Respektovaný odborník zde formou krátkých „receptů“ popisuje mnoho aspektů a stavebních prvků interních i externích DSL. Usnadnění tvorby externích DSL pak přinesly frameworky jako je Xtext pro Javu od projektu Eclipse[13], ale patří sem třeba i MetaEdit+ Workbench [14].

Důvodem, proč věnuji DSL tolik prostoru, přestože se samotné práce přímo netýkají, je skutečnost, že externí DSL se často implementují tím způsobem, že se překládají do jiného, typicky již univerzálního jazyka. Jinak řečeno, teorie zabývající se externími DSL se mimo jiné zabývá i možnostmi překladu jednoho jazyka do jiného. A to je již oblast, která je i tématem této diplomové práce.

[12] http://boo-lang.org/
[13] https://eclipse.org/Xtext/
[14] http://www.metacase.com/mwb/

Oblast překladů mezi dvěma jazyky je samozřejmě důležitá i mimo DSL. Například C++ se někdy implementuje jako překladač do čistého C. S rostoucí popularitou webových aplikací na straně klienta se objevila řada překladačů do JavaScriptu, ať už ze speciálních, pro tento účel navržených jazyků jako je CoffeeScript[15] nebo TypeScript[16], či z klasických jazyků jako je Ruby[17] nebo Python[18], jak již bylo zmíněno v závěru kapitoly 2.2.

3.2 Varianty překladu jazyků

Jednotlivým variantám překladu programovacích jazyků se přehledně věnuje čtvrtý oddíl knihy Terence Parra *Language Implementation Patterns* (Parr, 2010).

Parr dělí překladače[19] v zásadě podle toho, jestli potřebují explicitní *sémantický model* (Fowler, 2011, s. 159) a jakým způsobem s ním pracují (Parr, 2010, s. 280):

- Překlad řízený vstupní syntaxí.
- Překlad řízený pravidly.
- Překlad řízený modelem.

3.2.1 Překlad řízený vstupní syntaxí

V tomto případě máme parser, resp. gramatiku přímo doplněnou („proloženou“) příkazy generujícími výstupní kód. Překladač nepotřebuje vytvářet interní sémantický model, výstupní kód generuje průběžně při zpracovávání vstupního souboru v jednom průchodu.

Výhodou tohoto způsobu je jednoduchost. Parr (Parr, 2010, s. 296) z tohoto důvodu doporučuje používat tuto metodu kdykoli si s ní vystačíme.

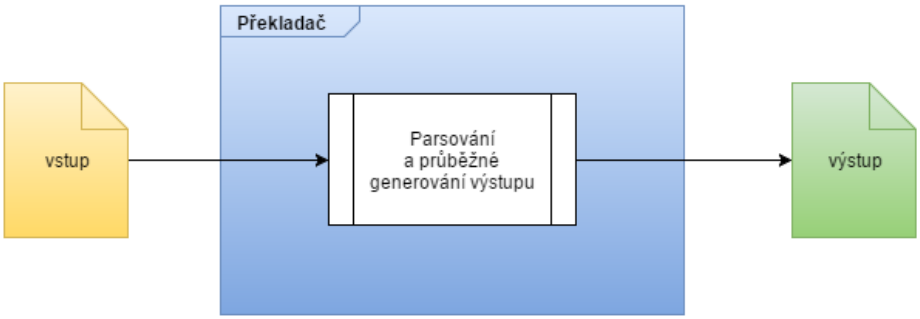
Zásadní nevýhodou této metody je nemožnost generovat výstup ve výrazně jiném pořadí než má vstup. Představme si například konverzi z jazyka, který umožňuje definovat funkce v libovolném pořadí, do jazyka, který vyžaduje, aby volání funkce předcházela její definice nebo alespoň její deklarace. V zásadě je to při jediném průchodu touto metodou neřešitelné, minimálně pokud chceme generovat pouze jediný soubor.

Otázkou je také vhodnost prokládání vstupní gramatiky či parseru kódem pro generování výstupu z hlediska čitelnosti a udržovatelnosti.

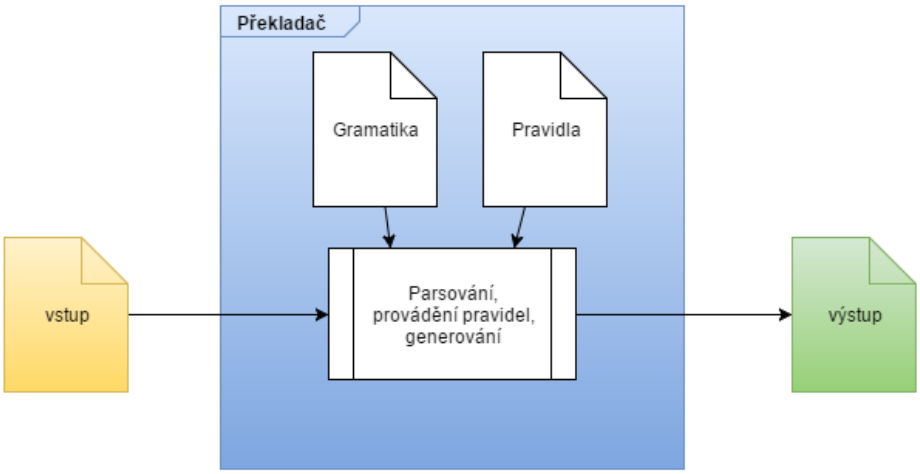
[15] http://coffeescript.org/
[16] https://www.typescriptlang.org/
[17] http://opalrb.org/
[18] http://pyjs.org/
[19] V textu používám termín *překladač* pro program překládající jeden programovací jazyk do jiného.

Obrázek 1: Typy překladu. Upraveno podle (Parr, 2010, s. 283-5)

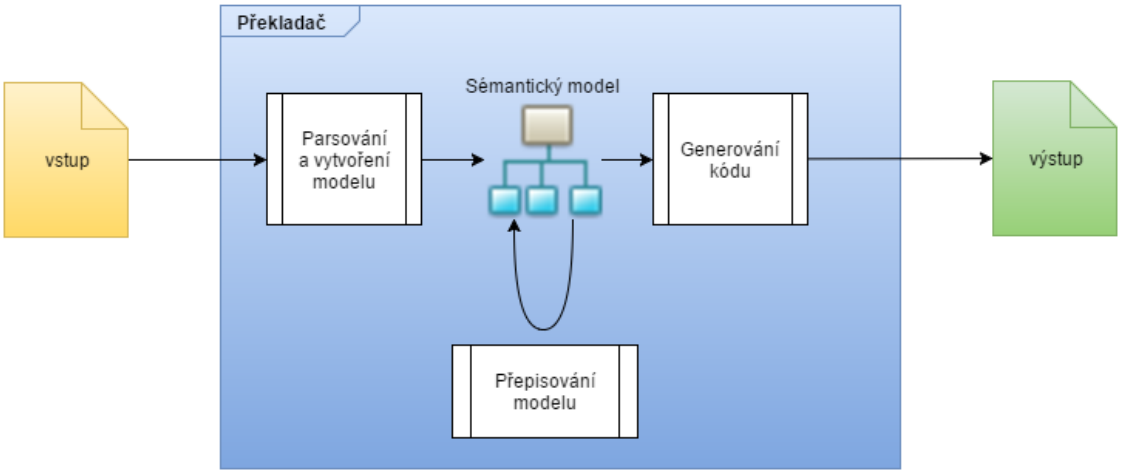
Překlad řízený vstupní syntaxí



Překlad řízený pravidly



Překlad řízený modelem



3.2.2 Překlad řízený pravidly

Při tomto přístupu zpracujeme vstup na základě gramatiky do syntaktického stromu. Přičemž gramatika, ani syntaktický strom nemusí být kompletní, pokud to pro naše účely není potřeba. V dalším kroku pak aplikujeme jednotlivá pravidla, která popisují mapování vstupního kódu na výstupní. Pravidla mohou být zapsána jak přímo v jazyku ve kterém je napsán vlastní překladač, tak lze využít i nějaký DSL vytvořený pro tento účel.

Protože máme již k dispozici syntaktický strom můžeme jím provést více průchodů s různými sadami pravidel (Parr, 2010, s. 283). Kupříkladu v dříve uvedeném případě lze při prvním průchodu vygenerovat deklarace všech funkcí a v druhém pak jejich definice.

Hodně typickým příkladem toho přístupu bývají různé XML šablonovací systémy, jake je například XSLT. Vstupní XML soubor je v tomto případě načten parserem a vytvořen strom dokumentu, ze kterého potom jednotlivá pravidla získávají hodnoty a podstromy pomocí XPath dotazů.

Tento přístup může přirozeně vést na deklaratorní, případně funkcionální paradigma. To může být někdy výhoda, protože nám dovoluje se soustředit na to *co* chceme místo toho, abychom řešili *jak*. Na druhou stranu je řadě programátorů bližší klasický imperativní přístup, což může být počáteční, vstupní překážkou.

Parr (2010, s. 296) považuje překladače řízené pravidly za elegantní, ale z výše uvedených důvodů někdy obtížné k naučení. Může být také obtížné pravidla ladit a zjišťovat, kdy a proč se jaká pravidla aplikují. Při jejich vývoji bychom neměli zapomenout dát uživatelům k dispozici možnosti dotazovat vnitřní stav překladače, mít možnost výpisu pravidel kterými se překladač řídí, v jakém pořadí je aplikuje a podobně.

Parr doporučuje používat tento typ překladačů zejména tehdy nepotřebujeme-li k překladu celou gramatiku, ale vystačíme si s její části (Parr, 2010, s. 302).

3.2.3 Překlad řízený modelem

Ostatní případy shrnuje Parr pod pojem „překlad řízený modelem“. Od předchozích přístupů se liší v tom, že vytváříme nějakou interní reprezentaci, tedy sémantický model (který může, ale nemusí odpovídat syntaktickému stromu) a tuto interní reprezentaci můžeme nejenom opakovaně procházet, ale můžeme jí i analyzovat, doplňovat do ní informace, upravovat ji a dokonce i přepisovat.

Na základě procházení sémantického modelu můžeme pak vytvářet výsledný kód různými metodami (Parr, 2010, s. 280):

- přímo vypisovat výstup,

- vytvářet řetězce,
- vyplňovat instance šablon či
- vytvářet objekty, které pak budou výstup generovat.

Přitom není nutné vytvářet přímo výsledný jazyk. Potřebujeme-li například generovat bajtkód a máme k dispozici jeho assembler, můžeme generovat jazyk, kterému tento assembler rozumí, místo přímého generování bajtkódu v binární podobě. Jistě, je to další krok navíc, mírné zneefektivnění a vytvoření závislosti na dalším produktu. Odměnou nám může být mnohem kratší a přehlednější kód našeho překladače a také nám může tento generovaný výstup usnadnit ladění našeho překladače apod.

Obdobně popisuje možnosti generování kódu i Fowler (Fowler, 2011, část VI):

- *transformační převodník* (Fowler, 2011, s. 533), který prochází sémantický model a generuje zdrojový kód a
- vyplňování šablon (Fowler, 2011, s. 539).

Pro klasická DSL reprezentující nějakou úžeji vyhraněnou oblast pak Fowler rozpoznává dva přístupy ke generování:

- *Generování znalé modelu* (Fowler, 2011, s. 555). V cílovém jazyce je implementován sémantický model ve formě knihovny, jejíž rozhraní námi generovaný kód používá. Tento přístup Fowler výrazně doporučuje.
- *Generování ignorující model* (Fowler, 2011, s. 567). Veškerá doménová logika je přímo „zadrátována“ do generovaného kódu. Tento způsob může mít smysl tam, kde je cílový jazyk primitivní a bylo by obtížné v něm vytvořit kvalitní model. Fowler také poukazuje na to, že díky neexistenci reprezentace explicitní instance modelu za běhu, resp. jeho vyjádření vlastně stavem programu, může být takto vygenerovaný kód rychlejší a méně pamětově náročný.

V našem případě píšeme příliš obecný překladač, takže se nás toto rozdělení v podstatě netýká. Navíc mapujeme na sebe dva jazyky obdobných možností. Existuje ale i obdobné dilema v takovýchto případech, které popisuji v následující části věnované vlastní implementaci.

Oba autoři také rozlišují generování *řízené vstupem* a generování *řízené výstupem* (Fowler, 2011, s. 534; Parr, 2010, s. 290) Při generování řízeném vstupem procházíme náš sémantický model a pro jeho jednotlivé prvky generujeme výstup. Naopak při generování řízeném výstupem sledujeme výslednou strukturu, kterou chceme získat a pro její jednotlivé elementy dohledáváme prvky sémantického modelu.

Parr používá analogii se šachovnicí (s. 291). Představme si, že naším úkolem je umístit na šachovnici figury pro hru šachů. Neuspořádanou skupinu figur můžeme považovat za vstupní model, plochu šachovnice za výstupní. Pokud se budeme řídit šachovnicí, tedy výstupem, budeme postupně procházet dvě krajní řady a pro každé pole prohledáme figury a vybereme tu správnou. Pokud se budeme řídit vstupem, budeme brát do ruky jednotlivé figury a umisťovat je na správné místo.

Tyto dva přístupy se mohou i vhodně doplňovat, kdy celkovou kostru výsledného souboru generujeme z hlediska výstupu, ale její jednotlivé části pak generujeme podle sémantického stromu. Můžeme také provést více průchodů různými přístupy, například v prvním, řízeném vstupem, vytvořit jiný model vhodný pro generování řízené výstupem.

Parr doporučuje generování řízené vstupem, protože je jednodušší získávat z modelu informace přímo při jeho procházení: „*we should compute information and translate phrases when it's convenient and efficient to do so, not when the the output order demands it*“ (Parr, 2010, s. 290, zvýraznění původní).

3.2.4 Přístup k vlastní implementaci

Zařaďme nyní realizovanou implementaci z hlediska této typologie překladačů.

Překlad kompletního univerzálního jazyka je poměrně složitý úkol, takže v zásadě připadá v úvahu jen možnost implementovat ho jako překlad řízený modelem. Na druhou stranu některé transformace jsou extrémně jednoduché, takže se nabízí možnost tyto jednodušší transformace psát ve formě pravidel, a protože Ruby, ve kterém je překladač implementován, je ideální jazyk na interní DSL, zdá se to být ideální varianta. Já jsem skutečně zkusil některé nejjednodušší transformace v této podobě implementovat, je ale pravda, že interní DSL má i své slabiny, zejména co se týká ladění. Z hlediska krokování a vůbec co nejjednoduššího pochopení průběhu překladu je zdaleka nejpříjemnější přímočarý imperativní kód.

Generování je řízené vstupem, vzhledem k podobnosti jazyků není potřeba měnit pořadí jednotlivých generovaných prvků. Co se týče způsobu generování je použita metoda globálních funkcí, resp. metod globálního sdíleného objektu, které generují zdrojový text. Přitom je hojně využito bloků, které Ruby nabízí. V tomto případě se dá hovořit o jednoduchém interním DSL.

Například metodu lze vygenerovat přibližně takto:

Ruby

```
$pygen.method name do
  arguments.gen
```

```
  $pygen.body do
     body.gen
  end
end
```

kde `$pygen` je globální objekt generátoru.

Zajímavé dilema vzniká při rozhodování, jestli danou transformaci implementovat již na úrovni transformace (přepisování) sémantického modelu, nebo ji alespoň částečně nahradit voláním pomocného kódu v Pythonu. Jde vlastně o obdobu Fowlerova dělení způsobu generování na metodu znalou modelu a ignorující model.

Příkladem může být test na pravdivou hodnotu. Mějme kód v Ruby ve stylu:

Ruby

```
if metoda() then vykonej() end
```

kde `metoda()` může vracet prakticky libovolnou hodnotu, včetně prázdného řetězce nebo prázdného pole, což se obojí v Ruby vyhodnotí jako pravdivé, zatímco v Pythonu ne.

První možností řešení je přepis modelu na místě a následně vygenerování upraveného řešení:

Python

```
if metoda() not in [False, None]: vykonej()
```

Tato varianta odpovídá přibližně způsobu generování ignorujícího model.

Druhou možností je napsat si vlastné modul, nazvěme ho `rb2py`, který implementuje funkci realizující tento test:

Python

```
def je_pravda(hodnota):
    return hodnota not in [False, None]
```

a následně tuto funkci v příslušných místech zavolat:

Python

```
from rb2py import je_pravda
if je_pravda(metoda()): vykonej()
```

Tento způsob má blíže ke generování znalém modelu, byť je zřejmé, že ani zdaleka nevytváříme v Pythonu kompletní model jazyka Ruby, nemluvě o tom, že ten stejně není doménovým modelem oblasti programu, který překládáme. Námi vytvořený modul `rb2py` se pak podobá vzoru, který Fowler naývá *Embedment Helper* (Fowler, 2011, s. 547). V původní podobě jde o objekt, který implementuje logiku, a který je dostupný v šablonách, aby ty nemusely obsahovat zbytečně moc komplexního kódu rozhodujícího co a jak generovat. V našem případě pracujeme s extrémně dynamickými skriptovacími jazyky, takže vlastně tuto logiku můžeme přesunout až do okamžiku běhu výsledného programu.

Představme si, že chceme transfomovat metodu `insert(index, hodnota)` pro pole například v kódu:

Ruby

```
pole.insert(index1, hodnota1).insert(index2, hodnota2)
```

Metoda `insert` v Ruby vloží hodnotu na pozici označenou indexem a vrátí změněný objekt pole, takže její volání můžeme zřetězit, jak je i v příkladu.

Pole (tj. *list*) v Pythonu má stejnojmennou metodu, která se liší tím, že nevrací nic. Je proto potřeba provést transformaci.

Půjdeme-li cestou pomocné funkce vykonávané za běhu, pak její logika může vypadat tak, že pomocná metoda otestuje parametry, jestli jde o známou transformaci, pokud ano, tak jí provede, pokud ne, tak operaci deleguje na původní objekt, kdy předpokládáme, že je to obyčejná metoda jen shodou okolností stejného názvu.

V případě metody `insert` tedy konkrétně:

Python

```
# modul rb2py
def insert(potenciální_pole, index, hodnota):
    if isinstance(potenciální_pole, list):
        # provedeme transformaci
        potenciální_pole.insert(index, hodnota)
        return potenciální_pole
    else:
        # nejde o známý typ (známou transformaci)
        # provedeme beze změny
        return potenciální_pole.insert(index, hodnota)
```

4 Porovnání vybraných konstrukcí jazyků Ruby a Python 3 a návrh mechanismu jejich překladu

Pro překlad z jednoho jazyka do druhého je potřeba porovnat jejich jednotlivé koncepty, operátory, typy a další prvky jazyka. Následující výčet se nesnaží být vyčerpávající, zejména se zaměřím na elementy, kde se jednotlivé jazyky podstatně liší, nebo, kde přes značnou podobnost konstrukcí existují rozdíly v detailech, se kterými je při převodu potřeba počítat. Až na výjimky jsou pak popsány právě konstrukce potřebné pro převod knihovny Prawn.

Na rozdíl od většiny existujících porovnání[20] se také příliš nazabývám porovnáním syntaxe, výkonu jednotlivých interpretrů jazyka, ani produktivity programátora. Zatímco většina těchto porovnání je určena pro koncové uživatele jazyka, tj. programátory, v našem případě jde o (polo)automatický převod, přičemž předpokládáme využití existujícího parseru Ruby. Volně dostupných kvalitních parserů Ruby je hned několik, takže by bylo dosti zbytečné si psát vlastní. Zájímá mne tedy vlastně až fáze, kdy od parseru získám syntaktický strom (AST). I když se zmínkám o syntaxi nebráním, v dalším textu se soustředím na porovnání chování jednotlivých konstrukcí.

Implementace

V textu jsou v takto označených místech vloženy poznámky týkající se aktuální implementace, nebo přímo odkazy na části zdrojového kódu překladače, které probíranou konstrukci řeší.

`složka/soubor.rb`
Označení konkrétního souboru, kde je transformace implementována.

`rb2py.název_funkce(parametry)`
Označení funkce pomocné knihovny rb2py v Pythonu, které implementuje dynamickou část funkčnosti. Tyto funkce jsou umístěny v souboru `rb2py/__init__.py`

[20] (Prechelt, 2000), dále pak řada zdrojů na Internetu:
http://hyperpolyglot.org/scripting
http://kronosapiens.github.io/blog/2014/05/10/from-ruby-to-python.html
http://mitsuhiko.pocoo.org/pythonruby.html
http://rosettacode.org/
http://pleac.sourceforge.net/
http://rigaux.org/language-study/syntax-across-languages.html
http://c2.com/cgi/wiki?PythonVsRuby

4.1 Neznámá hodnota

Oba jazyky mají speciální hodnotu, a jí odpovídající speciální třídu, pro vyjádření neznámé hodnoty.

V Ruby se neznámá hodnota nazývá `nil` a jde o jedinou instanci třídy `NilClass`. Idiomatickým testem na neznámou hodnotu je použití metody `nil?`. Vzhledem k tomu, že `nil` je standardní objekt, není s použitím metody žádný problém.

V Pythonu se neznámá hodnota nazývá `None` a jde o jedinou instanci třídy `NoneType`. Idiomatickým testem na neznámou hodnotu je použití operátoru identity `is`, tedy:

Python

```
objekt is None
objekt is not None
```

V obou jazycích samozřejmě funguje i běžné porovnání operátorem `==`.

Implementace

Neznámá hodnota jako literál:
`pygen/nil.rb`

Testování na neznámou hodnotu:
`fixes/send.rb` (Transformace volání `nil?` na `NilTestNode`.)
`fixes/nil_test.rb`
`pygen/nil_test.rb`

4.2 Logické hodnoty

Oba jazyky umí samozřejmě i vyjádřit pravdivostní hodnotu. Zatímco v Pythonu jde nepřekvapivě o dvě možné hodnoty třídy `bool`, Ruby je zvláštní tím, že tam má každá ze dvou logických hodnot svou vlastní třídu, `TrueClass` a `FalseClass`.

4.2.1 Literály

Ruby	Python 3
true	True
false	False

Implementace

`pygen/false.rb`

`pygen/true.rb`

4.2.2 Konverze na logické hodnoty

Následující tabulka udává hodnoty, které jsou považovány za nepravdu. V obou jazycích platí, že vše ostatní je považováno za pravdivou hodnotu.

Ruby	Python 3
false	False
nil	None
	nula libovolného číselného typu
	prázdný řetězec
	prázdné pole
	prázdná n-tice
	prázdný slovník

U vlastních tříd je v Pythonu možné, aby se jejich objekty konvertovaly na logickou hodnotu, pokud definují alespoň jednu z metod `__bool__()` nebo `__len__()`.

Převod lze nejsnáze implementovat explicitní funkcí, která vrátí logickou hodnotu podle pravidel jazyka Ruby, viz metoda `je_pravda()` na s. 20.

Implementace

`rb2py.ruby_false(value)`

`rb2py.ruby_true(value)`

4.2.3 Logické operátory `and` a `or`

V obou jazycích logické operátory `and` a `or` vracejí přímo jednu z testovaných hodnot. Tedy zápis:

levá strana `or` *pravá strana*

je ekvivalentní s:
pokud je pravda levá strana
    hodnotou je levá strana
jinak
    je hodnotou pravá strana
s tím, že *levá strana* je vyhodnocena pouze jednou.

Obdobně platí pro `and`:

levá strana `and` *pravá strana*

ekvivalence s:
pokud je pravda levá strana
    hodnotou je pravá strana
jinak
    je hodnotou levá strana
s tím, že *levá strana* je vyhodnocena pouze jednou.

Vzniká otázka, jak převést výrazy typu:

Ruby
```
a() || b()
```

Převod na:

Python
```
a() or b()
```

není korektní, pokud `a()` vrátí například prázdný řetězec nebo prázdné pole.

Bohužel ani nahrazení funkcí:

Python
```
emulace_or(a(), b())
```

není korektní, protože se pravá strana (`b()`) vyhodnotí vždy, zatímco u skutečného operátoru `or` pouze v případě, že je levá strana nepravdivá.

Převod na:

Python
```
je_pravda(a()) or je_pravda(b())
```

kde `je_pravda()` je funkce definovaná na s. 20 je pak korektní pouze tam, kde nás zajímá pouze pravdivost a nikoli sama hodnota. Překladač tuto variantu používá v podmínkách `if` a `while`, případně tam, kde nás výsledná hodnota, ani pravdivost nezajímá vůbec, tedy smyslem jsou pouze vedlejší efekty jako je vykonání metody.

Obecně je nutné převádět tyto výrazy rozdělením na vyhodnocení levé strany a samostatnou podmínku. Například:

Ruby

```
x = a() || b()
```

Python

```
dočasná_proměnná = a()
if je_pravda(dočasná_proměnná):
    x = dočasná_proměnná
else:
    x = b()
```

což lze zkrátit pomocí ternárního operátoru:

Python

```
dočasná_proměnná = a()
x = dočasná_proměnná if je_pravda(dočasná_proměnná) else b()
```

Tento postup se dá aplikovat rekurzivně:

Ruby

```
x = a() || b() || c()
```

Python

```
dočasná_proměnná = a()
dočasná_proměnná = dočasná_proměnná if je_pravda(dočasná_proměnná) else b()
x = dočasná_proměnná if je_pravda(dočasná_proměnná) else c()
```

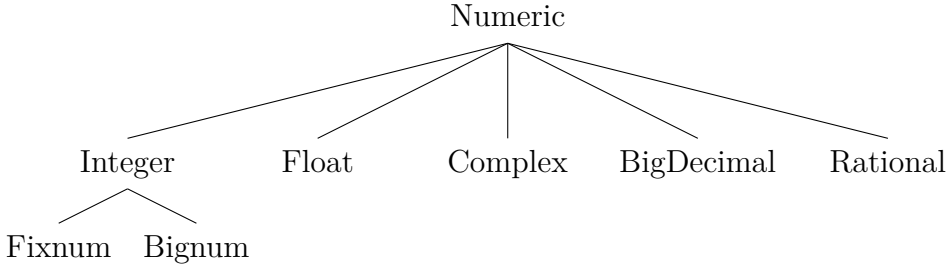
Implementace

`pyfixes/and_or.rb`

`pygen/and_or.rb`

4.3 Čísla

Hierarchie čísel v Ruby (Flanagan a Matsumoto, 2008, s. 42):



V Pythonu striktně vzato hierarchie číselných typů není, *int*, *float* a další typy jsou přímo potomky základní třídy *object*. V Pythonu existují jakési „falešné“ třídy, tzv. *abstraktní základní třídy, Abstract Base Classes*, pro testování příslušnosti objektů k třídám. Pro čísla se nachází v modulu `numbers`, kromě dokumentace popsáno též v PEP[21] 3141 *A Type Hierarchy for Numbers* [22]. Hierarchie v ní je následující:

Number
|
Complex
|
Real
|
Rational
|
Integral

4.3.1 Celá čísla

V Ruby to jsou dva potomci třídy `Integer`, `Fixnum` a `Bignum`. `Fixnum` pro čísla, která se vejdou do nativního slova platformy, kde program běží, a `Bignum` pro prakticky neomezená čísla vyjádřená ve dvojkovém doplňku, resp. omezená jen velikostí dostupné paměti. Konverze mezi Fixnum a Bignum je obousměrně automatická. V Pythonu celá čísla reprezentuje typ `int`, který obdobně zvládá čísla libovolné velikosti. Na rozdíl od Ruby je způsob interní reprezentace před uživatelem programovacího jazyka skryt.

Podporované prefixy číselných literálů (velikost písmen ani v jednom jazyce není důležitá):

Ruby	**Python 3**
desítková celá čísla	
`0d`	
šestnáctková celá čísla	
`0x`	`0x`
osmičková celá čísla	
`0`	
`0o`	`0o`
dvojková celá čísla	
`0b`	`0b`

[21] V Pythonu se návrhy na rozšíření nazývají PEP – Python Enhancement Proposals, viz: https://www.python.org/dev/peps/

[22] https://www.python.org/dev/peps/pep-3141/

Čísla zapsaná bez prefixu se považují za desítková. V Pythonu 3 není možné, aby nenulové číslo bez prefixu začínalo číslicí 0, protože to v řadě jiných jazyků, mj. i v Pythonu 2 a Ruby, znamená osmičkový zápis.

V Ruby je dále možné používat podtržení v číselném literálu pro seskupování číslic (např. `PSČ = 141_00`). Pro Python je tato možnost zvažována, jako návrh (PEP 515 *Underscores in Numeric Literals*[23]).

Jde-li nám jen o zachování funkčnosti, není potřeba převod celých čísel řešit, stačí vypsat hodnotu, kterou získáme od parseru Ruby.

4.3.2 Reálná čísla

Oba jazyky se spoléhají na reprezentaci reálných čísel poskytovaných hostitelskou platformou, dnes v drtivé většině odpovídající IEEE 754. Zápis literálů je prakticky stejný, oba jazyky podporují i semilogaritmický tvar. Převod literálů je opět relativně jednoduchý, stačí vypsat hodnotu získanou parsováním.

4.3.3 Další typy čísel

Oba jazyky podporují i komplexní čísla, racionální čísla, tj. zlomky, a reálná čísla vyjádřená na desítkovém základě. Od Ruby 1.9 jsou komplexní čísla a zlomky přímo součástí jazyka, v předcházejících verzích byly k dispozici v modulu `complex`, resp. `rational`. Reálná čísla o desítkovém základu jsou pak ve standardní knihovně `bigdecimal`. V Pythonu 3 jsou komplexní čísla přímo součást jazyka, zlomky jsou pak k dispozici ve standarním balíku `fractions` a desítková reálná čísla v balíku `decimal`.

Vzhledem k jejich relativně méně četnému používání, resp. vzhledem k tomu, že je převáděná knihovna Prawn nepotřebuje, jsem jejich převod neřešil. V principu by to nemělo být obtížné.

Implementace

`pygen/float.rb`

`pygen/int.rb`

[23] https://www.python.org/dev/peps/pep-0515/

4.3.4 Běžné operace s čísly

Ruby	Python 3
unární znaménka	
$+, -$	$+, -$
sčítání, odečítání, násobení	
$+, -, *$	$+, -, *$
reálné dělení	
/ pokud alespoň jeden operand je reálné číslo; metoda `div`	/
celočíselné dělení	
/ pokud jsou oba operandy celá čísla; metoda `fdiv`	//
zbytek po celočelném dělení	
%	%
celočíselné dělení spolu se zbytkem	
metoda `a.divmod(b)`	funkce `divmod(a, b)`
mocnina	
**	** funkce `pow`
operace s bity and, or, xor, jedničkový doplněk	
$\&, \lvert, \wedge, \sim$	$\&, \lvert, \wedge, \sim$
bitový posun vlevo a vpravo	
$<<, >>$	$<<, >>$

Ruby a Python celočíselné dělení (a tím pádem i zbytek po dělení) zaokrouhlují směrem k negativnímu nekonečnu, tj. $-7/3 = -3$. Naproti tomu jazyky jako C nebo Java zakrouhlují směrem k 0, tj. $-7/3 = -2$ (Flanagan a Matsumoto, 2008, s. 45).

Překlad operátorů je s výjimkou dělení a rozdílu přímočarý.

Protože v Ruby závisí to, jestli bude dělení celočíselné či nikoli na typu parametrů, zatímco v Pythonu ne, máme v zásadě dvě možnosti. Tím prvním, dosti komplikovaným je statická analýza kódu a určení typu parametrů při překladu z Ruby do Pythonu.

Jednodušší, a někdy prakticky jediné možné řešení, byť za cenu zpomalení za běhu, je využít dynamických vlastnostní jazyka:

Python

```
def ruby_div(a, b):
```

```
if isinstance(a, float) or isinstance(b, float):
    return a / b
else:
    return a // b
```

Tyto varianty lze zkombinovat a použít statickou analýzu tam, kde je to možné a dynamickou mít jako záložní řešení pro situace, kdy nelze určit typ proměnné.

U rozdílu není problém v případě čísel, ale je nutné pamatovat na to, že v Ruby se používá také rozdíl polí (viz část „Vybrané operace s poli“, s. 38). Z toho plyne, že nejsme-li si například díky statické analýze jisti typy operandů, je potřeba nahradit rozdíl volám funkce, která nejprve typy operandů zkontroluje a podle výsledku provede příslušnou operaci.

Implementace

`pygen/operator_binary.rb`
`rb2py.difference(left, right)`
`rb2py.division(left, right)`

4.4 Symboly

Hezkým, méně tradičním typem v Ruby jsou symboly. Symbol je vlastně *internovaný* neměnný řetězec. Za *internované* považujeme řetězce tehdy, platí-li o nich, že pokud si jsou rovny, tak jsou i reprezentovány stejným objektem. Formálně:

$$a = b \implies a \equiv b$$

kde $\equiv$ značí objektovou identitu. Protože obrácená implikace platí automaticky, mohli bychom psát i ekvivalenci.

V Ruby je použití symbolů hojné, ať už jako klíče slovníků a pojmenovaných parametrů funkcí, tak i jako prvky výčtu apod. V zásadě kdekoli, kde potřebujeme označení něčeho (např. dny v týdnu) aniž bychom dané označení potřebovali spojit s nějakou hodnotou.

V Pythonu přímo symboly nejsou, ale při běžném použití je lze jednoduše nahradit řetězci. Jako optimalizaci bychom je mohli internovat pomocí funkce `sys.intern()`. U knihovny Prawn si bohužel s tímto přístupem nevystačíme. PDF interně používá také systém typů, jako jsou řetězce, čísla nebo jména objektů. A knihovna Prawn na sebe mapuje typy Ruby a typy PDF. U většiny typů to problém není, ale symboly v Ruby jsou právě mapovány na jména objektů v PDF. Tím pádem potřebujeme odlišit řetězce a symboly. Nabízí se možná řešení:

- Označit si řetězce speciálním prefixem. Jednoduché, ale v principu nelze zaručit unikátnost prefixu.
- Mít speciální množinu všech řetězců reprezentující symboly. To by fungovalo teoreticky dobře, ale Python nezaručuje, že různě vytvořené objekty představující stejné řetězce budou opravdu různé objekty. Jinak řečeno, že budou mít různý objektový identifikátor. Ve skutečnosti třeba základní verze Pythonu, CPython, automaticky internuje krátké řetězce.
- Použití vlastní třídy k reprezentaci symbolů. Asi jediné opravdu robustní řešení. Alternativně by šlo použít výčet reprezentovaný třídou `Enum` standardního modulu `enum`.

Implementace

`pygen/symbol.rb`
`pygen/symbol_interpolated.rb`
`rb2py/symbolcls.py`
`rb2py.is_symbol(value)`
`rb2py.to_sym0(object)`

4.5 Řetězce

Z hlediska portace kódu mezi Ruby a Pythonem je jednou z nejproblematičtějších oblastí práce s řetězci, protože zatímco v Pythonu jsou řetězce neměnné (podobně jako třeba v Javě), v Ruby se měnit dají.

Řešení je v zásadě možné dvojí:

- Identifikovat místa, kdy se řetězce mění a nahradit příslušný kód. To vyžaduje modifikaci překládané knihovny.
- Implementovat vlastní třídu řetězců.

V obou jazycích došlo se zavedením verzí Ruby 1.9, resp. Python 3 k velké změně co se týče řetězců. Do té doby byly považovány prakticky za pole bajtů, zatímco v současných verzích jde již o pole znaků, resp. správněji tzv. *kódových bodů (code points)*[24]. Tj. jeden znak může být reprezentován větším počtem bajtů.

[24] Kódový bod je číslo, které reprezentuje daný znak v konkrétní znakové sadě. Kromě bežných znaků mohou mít např. v Unicode vlastní kódové číslo i některé formátovací značky nebo diakritická znaménka. Konkrétně v Unicode je např. možné znak s diakritikou vyjádřit buď jako jeden kódový bod, nebo jako dvojici kódového bodu znaku bez diakritiky a kódového bodu samostatného diakritického znaménka.

Implementace se mírně liší. V Ruby má řetězec explicitně přidružené kódování, které lze zjišťovat, a nemusí jít o kódování znakové sady Unicode (pomiňme, že každá znaková sada se dá považovat za podmnožinu Unicode). V Pythonu řetězec vždy může obsahovat libovolný znak Unicode znakové sady, interně pak používá jedno z kódování *latin1*, *UCS-2*, nebo *UCS-4*, podle toho, jaké znaky obsahuje. Detaily implementace popisuje PEP 393 *Flexible String Representation*[25]. Důležité je, že jednotlivé kódové body mají vždy hodnoty odpovídající znakové sadě Unicode.

Řetězec v Ruby nemusí být platný v přidruženém kódování. To lze otestovat metodou `valid_encoding?`. Metoda `force_encoding()` pak změní kódování přidružené k řetězci aniž by prováděla jeho konverzi. Jinak řečeno, změní pouze interpretaci bajtů, které ho tvoří. Knihovna Prawn toho používá pro zjišťování, zda je řetězec platný v kódování UTF-8:

Ruby

```
def is_utf8?(str)
  str.force_encoding(::Encoding::UTF_8)
  str.valid_encoding?
end
```

Nejprve tedy řekne, aby Ruby chápalo bajty v řetězci jako kódování UTF-8 a pak se zeptá, jestli je tento řetězec platný.

Přenos do Pythonu nelze udělat přímočaře, protože v Pythonu nemůže být řetězec neplatný.

4.5.1 Binární data

Z výše uvedeného plyne i další rozdíl. Pro čistě binární se i v nových verzích Ruby používají řetězce, pouze mají kódování nastavené buď na speciální hodnotu `ASCII_8BIT`.

V Pythonu oproti tomu existují samostatné třídy `bytes` (podobně jako řetězce neměnitelné), `bytearray` (měnitelné), případně lze užít obecnější třídy, jako je `array` ze stejnojmenného modulu, nebo prosté pole čísel[26] .

4.5.2 Jednotlivý znak

V obou jazycích také neexistuje speciální typ pro znak, v případě potřeby je reprezentován řetězcem o délce jednoho znaku.

[25] https://www.python.org/dev/peps/pep-0393/

[26] Obyčejné pole čísel je překvapivě praktický způsob realizace. Porovnání různých způsobů implementace měnitelných binárních řetězců z různých hledisek prezentoval Brandon Rhodes na montrealské konferenci PyCon 2015, viz záznam: https://youtu.be/z9Hmys8ojno

4.5.3 Zápis literálů

Oba jazyky mají bohaté možnosti zápisu literálů. Ruby tradičně podporuje interpolaci řetězců, tj. možnost vkládat do řetězců libovolné výrazy, které budou vyhodnoceny a výsledná hodnota nahradí výraz v řetězci. V Pythonu jde o novinku verze 3.6[27]. Pro naše účely je důležité, že použitý parser již sám takovýto řetězec rozdělí na sérií spojení řetězců a výrazů, což výrazně ulehčí práci.

4.5.4 Konverze na řetězec

Oba jazyky nabízejí snadnou možnost konverze objektů na řetězec. Základní konverzi nabízejí metody `objekt.to_s()` v Ruby, resp. `objekt.__str__()` v Pythonu, bývá zvykem jí volat funkčním zápisem jako `str(objekt)`.

Oba jazyky nabízejí i druhou možnost konverze objektů na řetězec. Je to konverze zvláště vhodná k účelům jako je logování a ladění.

V Ruby slouží k tomuto účelu metoda `objekt.inspect()`. Výchozí implementace vypíše třídu objektu, identifikátor objektu a rekurzivně vypíše jednotlivé složky objektu:

Ruby

```
class A
  def initialize
    @x = 1
    @y = 'ahoj'
    @z = Time.now
  end
end
a = A.new()
puts a.inspect()
```

vypíše:

```
#<A:0xa2577a4 @x=1, @y="ahoj", @z=2016-05-18 14:11:05 +0200>
```

V Pythonu existuje metoda `objekt.__repr__()`, resp. funkce `repr()`, která také vypisuje podrobnější infomace o objektu. Výchozí implementace vypíše třídu objektu a jeho identifikátor. Je ovšem doporučeno, aby tam, kde je to možné, vypisovala řetězec, který se blíží zápisu vytváření ekvivalentního objektu v syntaxi Pythonu, nebo dokonce

[27] https://docs.python.org/3.6/whatsnew/3.6.html, podrobnosti viz PEP 498 *Literal String Interpolation*: https://www.python.org/dev/peps/pep-0498/

to přímo takový zápis byl. V ideálním případě tedy `repr()` vrátí text, který je možné předat standardní funkci `eval()`, která ho vyhodnotí jako Pythonovský kód a vrátí objekt reprezentující stejnou hodnotu jako objekt původní.

Řada tříd ve standardní knihovně Pythonu se tím samozřejmě řídí, hezkým příkladem je standarní třída `datetime` reprezentující datum a čas:

Python
```
import datetime
nyní = datetime.datetime.now()

# běžná konverze na řetězec
print(str(nyní))
# => 2016-05-18 14:17:46.376346

# print() volá str() implicitně, takže ho není potřeba uvádět
print(nyní)
# => 2016-05-18 14:17:46.376346

# podrobná reprezentace objektu
print(repr(nyní))
# => datetime.datetime(2016, 5, 18, 14, 17, 46, 376346)

print(eval(repr(nyní)) == nyní)
# => True
```

I přes drobné rozdíly v sémantice mezi metodami `inspect()` a `__repr__()` je možné je z hlediska překladu považovat za odpovídající a tedy převádět definici `inspect()` na `__repr__()` a volání `objekt.inspect()` na volání `repr(objekt)`.

Implementace

`pygen/string.rb`
`pygen/string_interpolated.rb`
`rb2py/string.py`

4.6 Regulární výrazy

Regulární výrazy byly zmíněny již v úvodu jako příklad doménově zaměřeného jazyka. Ruby má po vzoru Perlu integrovány regulární výrazy přímo v jazyce, zatímco v Pythonu jsou běžnou součástí knihovny. Tato skutečnost v zásadě nepředstavuje výrazný

problém pro konverzi. Pro implementaci regulárních výrazů nicméně tyto jazyky používají různé knihovny. Zatímco Python používá jako výchozí svoji vlastní implementaci, Ruby nejprve od verze 1.9 začalo používat knihovnu Oniguruma[28] a od verze 2.0 její fork Onigmo[29].

Reálné regulární výrazy v programovacích jazycích mají dvě části:

- *Vzor.* To, čím se vlastně zabývá teorie regulárních jazyků. Popisuje, jaké kombinace znaků jsou povoleny. Základní syntaxe je naštěstí jednotná. Regulární výrazy mají dnes nicméně velmi bohaté prostředky a jim odpovídající syntaxe je již dosti pestrá.

  Aktuální verze překladače nemá ambice řešit konverzi vzorů regulárních výrazů a předpokládá, že vzor je napsaný tak, aby fungoval v obou jazycích. Pokud bychom požadovali maximální možnou kompatibilitu, je schůdnější cestou, než konverze vzorů, využít toho, že knihovna Onigmo přímo obsahuje i modul pro Python.

- *Příznaky.* Ovlivňují, jakým způsobem bude interpretr regulárních výrazů provádět jejich zpracování. Ruby zná tyto příznaky:

  - `i` – Nerozlišovat velikost písmen. V Pythonu tomu odpovídá `re.I`, resp. `re.IGNORECASE`.
  - `m` – Tečka bude zástupným znakem také pro konec řádku. V Pythonu tomu odpovídá `re.S`, resp. `re.DOTALL`.
  - `x` – Takzvané *rozšířené* regulární výrazy. Prázdné znaky ve vzoru jsou ignorovány a může obsahovat komentáře. V Pythonu tomu odpovídá `re.X`, resp. `re.VERBOSE`.
  - `o` – Týká se interpolace řetězců ve vzorech regulárních výrazů. Python nemá interpolaci řetězců a tím pádem ani obdobný příznak.
  - `e, n, s, u` – Týkají se kódování řetězců. V Pythonu 3 řetězce vždy mohou obsahovat všechny znaky Unicode, takže tyto příznaky není potřeba převádět.

  Protože Python má odlišně nastavené výchozí chování regulárních výrazů, jsou pro zvýšení kompatibility regulárních výrazů navíc automaticky nastaveny tyto příznaky:

[28] https://github.com/kkos/oniguruma
[29] https://github.com/k-takata/Onigmo

– `re.ASCII` – Zástupné znaky `\w`, `\W`, `\d`, `\D`, `\s` a `\S` odpovídají pouze klasickým ASCII znakům, nikoli kategoriím Unicode. To v Ruby platí vždy.

– `re.MULTILINE` – Zástupné znaky `^`, resp. `$` odpovídají začátku a konci nejen řetězce, ale též každého obsaženého řádku. To platí v Ruby také vždy. Všimněme si terminologické nejednotnosti, v Ruby znamená *multiline* to, co v Pythonu *dotall*.

Implementace

`pyfixes/regexp.rb`

`pygen/regopt.rb`

`rb2py/string.py`

`rb2py.create_regexp(pattern, flags)`

4.7 Rozsahy

Zajímavým typem je rozsah (*range*). Představuje interval. Jakkoli mají rozsahy v obou jazycích mnohem širší možnosti, popíšeme pouze ty zdaleka nejužívanější a to celočíselné.

V Ruby existují dva typy celočíselných rozsahů:

- První typ rozsahu se zapisuje se dvěma tečkami: `a..b` a obsahuje všechna celá čísla od `a` po `b` včetně obou hranic.
- Druhý způsob je se třemi tečkami `a...b` a obsahuje všechna čísla od `a` po `b-1`, tj. obsahuje pouze počáteční hranici.

Alternativně lze rozsahy v Ruby vytvářet konstruktorem třídy *Range*, jehož parametry jsou počáteční a koncová hodnota a nepovinný příznak, zda se má zahrnout i koncová hodnota.

V Pythonu se rozsahy tvoří explicitně konstruktorem třídy *range*, který má jako parametry počáteční a koncovou hodnotu a krok. Pokud není počáteční hodnota uvedena, předpokládá se nula, pokud není uveden krok, předpokládá se krok o velikosti jedničky.

Implementace

`fixes/range.rb`

`pygen/range.rb`

4.8 Pole

Oba jazyky mají jako základní typy kolekcí integrovaných přímo v jazyku pole a slovníky. V Pythonu existují také *n-tice (tuple)*, což jsou defakto neměnitelná pole. Z hlediska překladu z Ruby do Pythonu je důležité, že vracíme-li z metody více hodnot, pak v Ruby budou vráceny jako měnitelné pole, zatímco v Pythonu jako neměnná n-tice, viz dále v části věnované návratovým hodnotám metod (viz s. 63).

Implementace

`pygen/array.rb`

4.8.1 Vybrané operace s poli

Získání prvku, *nil\None* v případě jeho neexistence

Ruby: `pole[index]`

Python: `pole[index] if index in pole else None`

Získání prvku, výjimka v případě jeho neexistence

Ruby: `pole.fetch(index)`

Python: `pole[index]`

Získání prvku, náhradní hodnota v případě jeho neexistence

Ruby: `pole.fetch(index, default)`

Python: `pole[index] if index in pole else default`

Délka pole

Ruby: `pole.length`, `pole.size`, případně `pole.count` bez parametrů[30]

Python: `len(pole)`[31]

První prvek, nebo *nil\None*

Ruby: `pole.first`

Python: `array[0] if len(array) > 0 else None`

Poslední prvek, nebo nil

Ruby: `pole.last`

Python: `array[-1] if len(array) > 0 else None`

Je pole prázdné?

Ruby: `pole.empty?`

Python: stačí otestovat objekt `pole`, neboť prázdné pole se vyhodnotí jako *False*

Vyjmutí posledního prvku, nebo *nil\None*

Ruby: `pole.pop`

[30] Existují i varianty metody `count`, které počítají počet výskytů daného objektu v poli, resp. počet prvků pole splňujících zadaný predikát.

[31] `len(objekt)` je v zásadě idiomatický způsob zápisu `objekt.__len__()`

Python: `pole.pop() if len(pole) > 0 else None`

Vložení prvku na konec pole

Ruby: `pole.push(prvek)` (vrátí `pole`)

Python: `pole.append(prvek)` (nevrací nic)

Vložení prvku před daný index

Ruby: `pole.insert(index, prvek)` (vrátí `pole`)

Python: `pole.insert(index, prvek)` (nevrací nic)

Rozdíl polí se zachováním pořadí

Tato operace odstraní z prvního pole všechny prvky, které se vyskytují alespoň jednou i ve druhém. Nemění pořadí prvků pole, zachovává duplicity. Počet výskytů ve druhém poli není důležitý.

Ruby: `první_pole - druhé_pole`

Python: `[prvek for prvek in první_pole if prvek not in druhé_pole]`

Pokud by počet prvků polí byl větší, lze získat lepší asymptotickou časovou složitost pomocí:

```
s = set(druhé_pole)
[prvek for prvek in první_pole if prvek not in s]
```

To je ve skutečnosti i optimalizace, jakou interně používá Ruby MRI, podíváme-li se přímo do implementace:

Ruby MRI 2.3.1, *array.c*

```
static VALUE
rb_ary_diff(VALUE ary1, VALUE ary2)
{
    VALUE ary3;
    VALUE hash;
    long i;

    hash = ary_make_hash(to_ary(ary2));
    ary3 = rb_ary_new();

    for (i=0; i<RARRAY_LEN(ary1); i++) {
        if (st_lookup(rb_hash_tbl_raw(hash), RARRAY_AREF(ary1, i), 0)) continue;
        rb_ary_push(ary3, rb_ary_elt(ary1, i));
    }
    ary_recycle_hash(hash);
    return ary3;
```

```
}
```

Z toho také plyne důležitá (dokumentovaná) vlastnost, že je potřeba, aby objekty v poli korektně implementovaly metody, které jsou potřeba pro *hashování*. V případě Ruby jde o metody `hash()` a `eql?`, v případě Pythonu pak o `__hash__()` a `__eq__()`.

Implementace

```
pyfixes/array.rb
rb2py.array_detect(block, array)
rb2py.array_empty(array)
rb2py.array_first(array)
rb2py.array_last(array)
rb2py.difference(left, right)
```

4.8.2 Vykrojení (slice)

Oba jazyky podporují operaci vykrojení. Tato operace nám umožňuje určit část sekvence, typicky pole nebo řetězce. Za limitní případ vykrojení lze považovat prosté indexování, které určuje jeden prvek.

V Ruby se k vykrojení nejčastěji používá operátoru hranatých závorek. Výsek lze určit buď pomocí rozsahů (viz s. 36), nebo jako dvojici počáteční prvek a počet prvků, které jsou odděleny čárkou:

Ruby

```
pole = [1, 2, 3, 4, 5]

# prosté indexování => jeden prvek
puts pole[2]      # => 3

# počátek, délka
puts pole[2, 2]   # => [3, 4]

# inkluzivní rozsah
puts pole[2..4]   # => [3, 4, 5]

# exkluzivní rozsah
puts pole[2...4]  # => [3, 4]
```

Lze používat i záporné indexy, které se počítají odzadu, tj. $-i = delka - i$:

Ruby

```
# záporné indexy
pole = [1, 2, 3, 4, 5]

# prosté indexování => jeden prvek
puts pole[-2]      # => 4

# počátek, délka
puts pole[-2, 2]   # => [4, 5]

# inkluzivní rozsah
puts pole[-2..4]   # => [4, 5]

# exkluzivní rozsah
puts pole[-2...4]  # => [4]
```

Python nabízí jedinou syntaxi, která je obdobou exkluzivního rozsahu, zapisuje se nicméně s dvojtečkou „`:`“ místo dvou teček „`..`“:

Python

```
pole = [1, 2, 3, 4, 5]

# prosté indexování => jeden prvek
print(pole[2])      # => 3
print(pole[-2])     # => 4

# exkluzivní rozsah
print(pole[2:4])    # => [3, 4]
print(pole[-2:4])   # => [4]
```

Z hledika překladu je pro nás klíčové rozlišit, kdy je rozsah v Ruby použit samostatně, a kdy je použit u pole pro specifikaci rozsahu, protože v takovém případě je potřeba ho převést na operaci vykrojení, nikoli na volání konstruktoru `range()`.

Python nemá inkluzivní rozsahy a to způsobuje problém při jejich převodu, pokud by měly končit posledním prvkem pole v situaci, kdy je specifikován záporným indexem `-1`:

Inkluzivní rozsah	Exkluzivní rozsah
`0..3`	`0...4`
`-4..-2`	`-4...-1`
`-4..-1`	`-4...?`

Zatímco u ostatních indexů stačí přičíst jedničku, pro poslední prvek bychom dostali nulu a tedy naopak index úplně prvního prvku.

Python to řeší tím, že vynechání prvku znamená automaticky poslední prvek:

Python

```
pole[-4:]
```

Implementace

pyfixes/range_to_slice.rb

4.8.3 Procházení pole

Pro procházení všech prvků pole existují v zásadě dva způsoby. Ten používanější je volání metody `each()` s blokem, která je zavolán pro každý prvek:

Ruby

```
['a', 'b', 'c'].each() {|hodnota| puts(hodnota)}
```

Překladu tohoto typu průchodu je věnována stať v části týkající se bloků, s. 66.

Méně používaným způsobem v Ruby, zato hojně v Pythonu, je konstrukce `for-in`. Tato konstrukce je popsána v části věnované cyklům, s 80.

4.9 Slovníky

Dalším základním typem, který mají oba jazyky, je slovník. V Ruby je představován třídou `Hash` a v Pythonu třídou `dict`.

4.9.1 Procházení

Pokud chceme procházet všechny prvky slovníku v Ruby, tak podobně jako v případě jiných kolekcí můžeme použít metodu `each()`, resp. její varianty. Záleží totiž na tom, jestli chceme procházet klíče, hodnoty, nebo dvojice klíč-hodnota. K procházení klíčů slouží metoda `each_key()`, k procházení hodnot `each_value()`, k procházení dvojic klíč-hodnota lze použít buď explicitně pojmenovanou `each_pair()`, nebo běžnou metodu `each()`, která se chová stejně.

V Pythonu slouží k získání klíčů slovníku metoda `keys()`, k získání hodnot `values()` a k získání dvojic klíč-hodnota metoda `items()`. Tyto metody vrací *pohledy*, které lze procházet cyklem `for-in`. Pokud explicitní metodu nepoužijeme a cyklu `for-in` předáme vlastní objekt slovníku, získáme klíče stejně jako u metody `keys()`. Následující dva zápisy jsou tedy ekvivalentní:

Python

```
for klíč in slovník: print(klíč)
for klíč in slovník.keys(): print(klíč)
```

V části věnované blokům, s. 66, je popsán obecný překlad volání metody `each` v Ruby právě na cyklus `for-in`. Pokud iterujeme slovník je nicméně potřeba počítat s tím, že je potřeba přidat volání metody `items`. Což vyvolává otázku, jak zjistit, zda proměnná odkazuje zrovna na slovník, nebo na jiný objekt. Potřebujeme totiž odlišit procházení slovníku od procházení pole, které obsahuje dvojice hodnot opět ve formě polí. Zatímco v Ruby je procházíme stejně:

Ruby

```
slovník = {a: 1, b: 2}
pole = [[1, 2], [10, 20]]
slovník.each() {|k, v| puts("#{k} #{v}")}
pole.each() {|k, v| puts("#{k} #{v}")}
```

v Pythonu je potřeba procházení rozlišit:

Python

```
slovník = {'a': 1, 'b': 2}
pole = [[1, 2], [10, 20]]
for k, v in slovník.items(): print("%s %s" % (k, v))
for k, v in pole: print("%s %s" % (k, v))
```

Možnosti řešení jsou:

- Pokud máme jistotu, že kromě procházení slovníku nikdy nedochází k popsané situaci, kdy by metoda `each()` generovala do bloku více než jeden parametr, pak můžeme jakýkoli objekt procházený s metodou `each()` s blokem s více než jedním parametrem považovat za slovník.
- Tam, kde je to možné můžeme použít statickou analýzu kódu ke zjištění, že proměnná obsahuje slovník, během překladu z Ruby do Pythonu.

- Dynamické řešení. Zjišťovat za běhu, jde-li o slovník:

Python

```
def each(objekt):
    if isinstance(objekt, dict):
        return objekt.items()
    else:
        return objekt
```

Implementace

`rb2py.each(object)`

V Ruby od verze 1.9 jsou nicméně prvky slovníku procházeny v daném pořadí, které odpovídá pořadí, v jakém byly do slovníku vloženy. Python má ekvivalent ve třídě `OrderedDict` standardního modulu `collections`.

Implementace

`pyfixes/hash.rb`

`pygen/hash.rb`

4.10 Definice modulů

V Ruby je modul, tj. instance třídy *Module*, základní stavební prvek programu, mimo jiné už proto, že třída, *Class*, je přímým potomkem *Module*.

V Ruby slouží moduly ke dvěma hlavním účelům:

- Jako jmenný prostor pro logicky související funkce a třídy.
- Jako tzv. *mixin*, tj. kolekci metod, kterou je možné vložit do třídy. Jde vlastně o určitou formu vícenásobné dědičnosti.

V Pythonu se terminologicky rozlišují moduly (*module*) a balíky (*package*). Balík je vlastně modul obsahující další moduly či vnořené balíky (*subpackage*).

V Ruby není hierarchie modulů svázána s umístěním souborů na disku, ale uvádí se ve vlastním kódu pomocí klíčového slova `module`. Oproti tomu v Pythonu je hierarchie dána umístněním souborů. Balíkům odpovídají adresáře a koncovým modulům soubory. Pokud má balík nějaký kód, tzv. *řádný (regular)* balík, pak je umístěn v adresáři balíku v souboru `__init__.py`. Pokud balík slouží jen k rozlišení jmenné hierarchie, tak tento soubor nemusí obsahovat. Pak jde o tzv. *jmenný (namespace)* balík.

Převod modulů je v zásadě přímočarý, jediné obtížnější je rozlišit, k jakému ze dvou účelů modul primárně slouží. Moduly zajišťující jmenné prostory převedeme na odpovídající hierarchii balíků a modulů, moduly zajišťující mixiny převedeme na standardní třídy a jejich vložení do třídy nahradíme klasickou vícenásobnou dědičností, kterou Python podporuje.

Implementace

Překladač momentálně nemá autodetekci modulů, které se používají jako mixiny implementovanou. Je potřeba takové moduly explicitně vyjmenovat v poli `$HINTS_MODULE`. Viz soubor `hints.rb`

Jediná situace, kdy se modul automaticky stane třídou, je pokud je obsažen v jiné třídě. Což v Ruby lze, v Pythonu zjevně nemá smysl.

Zvažovanou možností je konvertovat jednoduše všechny moduly, s výjimkou toho úplně vnějšího, na třídy.

Při nahrazení modulů zajišťujících mixiny postupujeme takto:

- Pro každý modul vytvoříme třídu.
- Pro každou třídu:
  - Pro každý vložený modul připoj stejnojmenou třídu na začátek seznamu předků.

Výsledek je tedy ten, že seznam předků obsahuje nejprve třídy realizující moduly a to v pořadí opačném, než jsou ve zdrojovém textu v Ruby a případná původní třída předka je na konci. Toto pořadí je důležité, aby správně fungovalo pořadí volání metod předků.[32]

Ukažme si to na příkladě:

Ruby

```
class Předek
  def metoda
    puts('Předek')
  end
end
module Modul1
  def metoda
```

[32] Python 3 používá pro vyhledávání metod předků při vícenásobné dědičnosti tzv. *linearizaci C3* vyvinutou původně pro jazyk Dylan. Více viz: https://www.python.org/download/releases/2.3/mro/

```
    puts('Modul1')
    super()
  end
end
module Modul2
  def metoda
    puts('Modul2')
    super()
  end
end
module Modul3
  def metoda
    puts('Modul3')
    super()
  end
end
class E < Předek
  include Modul1
  def metoda
    puts('E')
    super()
  end
end
class F < E
  include Modul2
  include Modul3
  def metoda
    puts('F')
    super()
  end
end
objekt = F.new
objekt.metoda()
```

vypíše:

```
F
Modul3
Modul2
```

```
E
Modul1
Předek
```

Tomu odpovídá v Pythonu:

Python

```
class Předek:
  def metoda(self):
    print('Předek')

class Modul1:
  def metoda(self):
    print('Modul1')
    super().metoda()

class Modul2:
  def metoda(self):
    print('Modul2')
    super().metoda()

class Modul3:
  def metoda(self):
    print('Modul3')
    super().metoda()

class E(Modul1, Předek):
  def metoda(self):
    print('E')
    super().metoda()

class F(Modul3, Modul2, E):
  def metoda(self):
    print('F')
    super().metoda()

objekt = F()
objekt.metoda()
```

Všimněme si, že v Pythonu nevolá `super()` přímo metodu předka, ale vrací vlastně jeho „podobjekt“, na kterém ji musíme zavolat sami.

Implementace

`fixes/module.rb`

`pygen/module.rb`

4.11 Standardní moduly

Jak bylo již uvedeno v úvodní kapitole této práce, jazyk tvoří nejenom syntaxe a sémantika, ale řada dalších prvků. Jedním z nich jsou i standardní moduly. Pokud jsou jednoduché a napsané v Ruby, lze uvažovat o jejich překladu tímto překladačem. Na druhou stranu jsou to právě standardní moduly, které často používají složité techniky metaprogramování, nebo zapouzdřují systémové služby. Jako takové bývají mnohdy psány přímo v jazyku, ve kterém je daná implementace jazyka realizována, ať už jde o céčko v případě základní Ruby MRI implementace, nebo třeba Java v případě JRuby.

Velmi typickým příkladem je knihovna *Zlib*[33], která implementuje způsob komprese známý jako *deflate*, resp. *gzip*.[34]

Je zjevné, že v každém jazyce se rozhraní liší, a to i v případě, že zapouzdřuje stejnou nativní knihovnu. V tomto případě je úkol ještě poměrně jednoduchý. Prawn používá dvě funkce (z hlediska Ruby tedy metody modulu):

- `Zlib::Deflate.deflate(data)` pro kompresi a
- `Zlib::Inflate.inflate(data)` pro rozbalení.

Tyto metody mají naštěstí v Pythonu přímé ekvivalenty:

[33] http://www.zlib.net/

[34] Přesněji řečeno *deflate* je:

- Metoda komprese a způsob, jakým je výsledek komprese uložen v paměti počítače, tj. formát generovaného proudu bitů. Toto popisuje (RFC 1951).
- Zároveň se tímto termínem vyjadřuje samotný proces komprese, kdy rozbalení se značuje jako *inflate*.

Výsledný proud bitů je pak „zabalen“ do obálky, která poskytuje kontrolní součet a případně další údaje. Zlib nabízí dvě jednoduché obálky:

- Jednodušší se jmenuje stejně jako knihovna, *zlib* (RFC 1950).
- Druhá obálka, určená hlavně pro přenos souborů, je *gzip*, což je i název stejnojmenného programu, kterým lze soubory komprimovat. Na unixových systémech bývá součástí základní programové výbavy. (RFC 1952)

Jak dokumentace, tak zmiňovaná knihovna jsou volně dostupné, takže exituje i řada dalších implementací těchto algoritmů, mj. i moje vlastní: http://alpng.sourceforge.net/inflate.html. Jak Ruby MRI, tak i CPython nicméně zapouzdřují původní céčkovou knihovnu Zlib.

- `zlib.compress(data)` pro kompresi a
- `zlib.decompress(data)` pro rozbalení.

Obecně je nejpřijatelnější strategií vytvořit v Pythonu modul pro každou použitou knihovnu a v ní implementovat použitou podmnožinu původní knihovny pomocí nástrojů Pythonu:

Python

```
# modul rb2py.zlib
import zlib

class Zlib:

    class Deflate:
        @staticmethod
        def deflate(stream):
            return zlib.compress(bytes(stream, 'UTF-8'))

    class Inflate:
        @staticmethod
        def inflate(stream):
            return zlib.decompress(stream).decode('UTF-8')
```

Všimněme si, že nicméně musíme řešit rozpor, který plyne z toho, že Python 3 má pro binární data vlastní typ, zatímco v Ruby se používají řetězce. Více viz kapitola věnovaná řetězcům, s. 31.

Díky tomu pak můžeme převést původní kód:

Ruby

```
# modul PDF::Core::Filters::FlateDecode
def self.encode(stream, params = nil)
  Zlib::Deflate.deflate(stream)
end
def self.decode(stream, params = nil)
  Zlib::Inflate.inflate(stream)
end
```

automaticky na ekvivalent v Pythonu:

Python

```
# modul PDF.Core.Filters.FlateDecode
from rb2py.zlib import *

def encode(stream, params=None):
    _result = None
    _result = Zlib.Deflate.deflate(stream)
    return _result

def decode(stream, params=None):
    _result = None
    _result = Zlib.Inflate.inflate(stream)
    return _result
```

Tento kód vznikne standardním překladem. V postatě ani není potřeba, aby překladač z Ruby do Pythonu vědel, že `Deflate` a `Inflate` jsou třídy, které emulujeme. On pozná, že jde o třídu nebo modul, protože jejich názvy začínají velkým písmenem, tj. jde o konstanty. Přesto může být někdy užitečné informaci o emulovaných třídách a jejich metodách překladači poskytnout. Poud například víme, že metoda `decode` vrací řetězec, může to překladač využít pro optimalizace založené na statické typové analýze.

Implementace

Implementace emulovaných funkcí a tříd se nachází ve složce `rb2py`.

4.12 Standardní mixiny

Termín mixin byl již popsán v části věnované modulům, s. 43. Mixiny jsou vkládány do třídy pomocí klíčového slova `include`.

Je potřeba umět převádět některé standardní, hojně využívané mixiny, jako je *Comparable* nebo *Enumerable*.

4.12.1 Mixin *Comparable*

V sekci o operátorech je popsán porovnávací operátor `<=>` (s. 59). Vložíme-li do třídy mixin *Comparable*, tak doplní naši třídu o operátory `<`, `<=`, `>`, `>=`, `==` a metodu `between?`, které budou implementovány pomocí našeho operátoru `<=>`.

Python nabízí velmi podobnou funkčnost ve standardním modulu *functools* v podobě *dekorátoru třídy* `total_ordering`. Dekorátory zatím nebyly popisovány, protože nemají v Ruby přímý ekvivalent a tudíž při překladu z Ruby do Pythonu nevznikají. Jde

o zvláštní syntaxy pro volání funkce s objektem třídy nebo metody. Název dekorátoru se zapisuje se zavináčem před definici dekorovaného objektu. Například zápis:

Python
```
@total_ordering
class Třída:
    ...
```

je zcela ekvivalentní:

Python
```
class Třída:
   ...
Třída = total_ordering(Třída)
```

Dekorátor *total_ordering* pak pro třídu, která definuje alespoň jeden z operátorů `<`, `<=`, `>`, `>=`, automaticky doplní definice těch ostatních.

V principu tak můžeme použít `total_ordering` při překladu třídy, která používá mixin *Comparable*. Drobnou nevýhodou je mírná neefektivita, kdy budou ostatní operátory volat námi vygenerovaný operátor `<`, který bude volat metodu vzniklou překladem původního operátoru `<=>` (ve vlastní implementaci ji nazývám `_cmp()`). Pokud bychom všechny operátory generovaly při překladu rovnou, což je velmi jednoduché, ušetřili bychom jedno volání.

Implementace
`pyfixes/comparable.rb`

4.12.2 Mixin *Enumerable*

Tento mixin je o poznání složitější. Pro třídu implementující metodu `each` (některé metody tohoto mixinu vyžadují i `<=>`) přídává 51 dalších metod (platí pro Ruby 2.2.3).

Řada z nich je obdobou funkcionálního programování, například:

- `all?(blok)` Vrací pravdu, pokud blok vrátí pro všechny prvky pravdivou hodnotu.
- `any?(blok)` Vrací pravdu, pokud blok vrátí alespoň pro jeden prvek pravdivou hodnotu.
- `map(blok)` Vrací pole, kde každý prvek je hodnota, kterou vrátil blok pro hodnotu prvku.

Python v těchto případech používá funkcionální zápis, tj. `map(funkce, sekvence)` místo `sekvence.map(blok)`. Převést blok na vnořenou funkci není velký problém.

Pro Pythonovské funkce, jako je zmiňovaná `map()`, musí být objekt tzv. *iterovatelný*. Co to znamená je popsáno v části věnované `for-in` cyklu na s. 81.

Konverze z `each()` na *iterovatelný* objekt je vlastně docela jednoduchá. Vezměme si příklad z knihovny Prawn, ze třídy PDF::Core::FilterList:

Ruby

```
def each(&block)
  @list.each do |filter|
    block.call(filter)
  end
end
```

Tento kód převedeme na:

Python

```
def __iter__(self):
    for filter in self._list:
        yield filter
```

Python třídu implementující metodu `__iter__()`, která předává hodnoty pomocí `yield` považuje za *iterovatelnou*. Python je pak schopen takovou třídu procházet `for-in` cyklem, používat ji s funkcemi `all()`, `map()` a podobně.

Implementace

`pyfixes/enumerable.rb`

Překlad `each` na `__iter__`:

`pyfixes/custom_methods_call.rb`
`pyfixes/custom_method_def.rb`
`pyfixes/yield.rb`
`pygen/argument.rb`
`pygen/yield.rb`

4.13 Třídy

Co se týče tříd, jsou si oba jazyky naštěstí docela podobné. Ruby má pouze jednoduchou dědičnost, takže z hlediska převodu nevzniká problém.

V Ruby je základem hierarchie tříd od verze 1.9 třída `BasicObject` (Flanagan a Matsumoto, 2008, s. 235). Ta implementuje jen naprosté minimum metod, a byla přidána primárně pro usnadnění implementace *delegace*, tj. pro situace, kdy přeposíláme zprávy jinému objektu.

Pro běžné účely, a do verze 1.9 i skutečným, základem hierarchie je třída `Object`. Ta je také použita jako předek implicitně, neuvedeme-li při definici třídy jinou třídu. Třída `Object` také vkládá mixin `Kernel`, který implementuje standardní „globální funkce“. Technicky jde o metody modulu, ale protože se téměř vždy nacházíme v rámci objektu třídy, která je potomkem třídy `Object`, tak jsou nám tyto metody automaticky přímo dostupné.

V Pythonu 3 je situace jednodušší, základem hierarchie je třída `object`.

Pro zbytek kapitoly o třídách je nutné pečlivě rozlišit:

- Obyčejný *objekt*, tj. *instanci třídy*.
- *Objekt třídy*, tj. objekt který reprezentuje třídu samotnou. V Ruby jde o instanci třídy `Class`.

Elementární funkčnost tříd je v obou jazycích stejná:

- Je možné vytvářet instance. V Ruby to znamená zavolat metodu *objektu třídy* `new`. Python je zvláštní tím, že instanci třídy vytvoříme zápisem, kdy jméno třídy použijeme jako funkci. Například `objekt = datetime()`.
- Třída může obsahovat metody, které její instance sdílejí. Při volání pak proměnná `self` (v Pythonu explicitní parametr metody) obsahuje instanci, vůči které metodu voláme.
- Objekty mohou obsahovat *instanční proměnné*. Ty definují interní stav objektu. V Ruby jejich názvy začínají znakem zavináč (`@proměnná`) a jsou vždy soukromé.

Implementace

`fixes/class.rb`

`pygen/class.rb`

4.13.1 Volání metody předka

Předefinujeme-li metodu v potomkovi a chceme zavolat zděděnou stejnojmenou metodu předka, slouží nám v obou jazycích metoda `super()`. V Pythonu vrátí podobjekt předka, na kterém metodu voláme, v Ruby přímo zavolá stejnojmenou metodu předka. Ruby má jednu zvláštnost. Neuvedeme-li explicitně žádné argumenty při volání metody `super`, Ruby automaticky předá všechny argumenty aktuální metody metodě předka. To znamená, že chceme-li zavolat metodu předka bez jakýchkoli argumentů, musíme explicitně uvést prázdné závorky.

Příklad na použití metody `super()` byl uveden v části věnované modulům, s. 44.

Implementace

`pyfixes/super.rb`

`pygen/super.rb`

4.13.2 Proměnné a metody objektu třídy

V obou jazycích mohou proměnné existovat i na úrovni *objektu třídy*, tedy proměnné sdílené všemi instancemi. V Ruby je situace poněkud komplikovanější, protože kromě běžných *proměnných třídy*, jejichž názvy začínají dvojicí zavináčů (`@@proměnná`), mohou existovat i *instanční proměnné objektu třídy*. Ty se naštěstí používají výjimečně a v Pythonu nemají přímý ekvivalent. I proto je v rámci překladače neřeším. Pro případné detaily viz (Flanagan a Matsumoto, 2008, s. 230; Shaughnessy, 2014, s. 120).

Kromě metod volaných vůči instanci třídy umožňují oba jazyky definovat i metody na úrovni třídy, které lze volat bez nutnosti vytvoření její instance. V Pythonu je potřeba rozlišovat mezi *metodou třídy (classmethod)*, které je jako první argument předán *objekt třídy*, podobně jako instanční metodě je předána instance, od *statických metod (staticmethod)*, které toto nemají.

Implementace

`fixes/defs.rb`

4.14 Metody

Výkonné části kódu je jedna z oblastí, kde se jazyky výrazně liší, což představuje problém z hlediska převodu.

4.14.1 Funkce a metody

Do značné míry formálně má Ruby pouze metody, zatímco Python má jak metody, tak i obyčejné funkce. Rozdíl je v tom, že funkce v Pythonu nemá parametr *self* odkazující na aktuální instanci nad kterou je metoda volána. V Pythonu se *self* předává explicitně, je potřeba uvést ho v seznamu parametrů metod na prvním místě. Tím pádem může mít i jiný název, byť konvenci *self* je dobré dodržovat. V Ruby se *self*, jako ve většině objektových jazyků, předává implicitně a existuje vždy.

Z hlediska převodu potřebujeme rozlišit několik situací:

- Metody přidané do modulu *Kernel*. Protože tento modul je vkládán do třídy *Object*, automaticky to znamená, že jeho metody jsou k dispozici odkudkoli bez nutnosti explicitního určení. Jinak řečeno, tento idiom se používá právě tam, kde potřebujeme „klasickou“ globální funkci. Naštěstí jsou takovéto konstrukce v běžném objektovém kódu vzácné (v knihovně Prawn se nevyskytují), protože Python přímý ekvivalent globálních funkcí nemá, všechny funkce patří do nějakého modulu a je buď potřeba je při volání kvalifikovat jménem modulu, resp. jménem pod kterým byl importován, nebo je explicitně vložit do jmenného prostoru současného modulu.

- Metody na úrovni běžného modulu, tj. modulu, který není vkládán do tříd. Tyto metod převedeme do Pythonu na obyčejné funkce na úrovni modulu. Instanční proměnné modulu je potřeba přeložit na běžné proměnné modulu. V Pythonu se právě proměnným modulu říká „globální“, protože klasické globální proměnné, tj. dostupné odkudkoli bez explicitní kvalifikace, Python nemá.

- Rozšiřování existujících tříd. V Pythonu není problém rozšiřovat běžné třídy, na rozdíl od Ruby nelze ale rozšiřovat základní třídy typu *object* nebo *str* (řetězec).

- *Eigenmetody*, tj. metody, které jsou přidruženy přímo konkrétní instanci, takže je objekty stejné třídy nesdílejí. Ruby v takovém případě vytvoří proxy třídu, která dané metody obsahuje, v Pythonu mohou objekty obsahovat metody přímo.

- Ostatní případy jsou v zásadě běžné metody volané buď jako instanční, nebo jako metody třídy (tj. instance *Class*). Obojí má v Pythonu své přímé ekvivalenty.

Implementace

`fixes/def.rb`

`fixes/defs.rb`

`pygen/def.rb`

4.14.2 Názvy

Hezkou zvláštností Ruby oproti většině programovacích jazyků je možnost, aby název metody končil znakem `?`, `!` či `=`.

Metody končící otazníkem typicky zjišťují pravdivost predikátu, tj. mají návratovou hodnotu, kterou lze testovat v podmínce (*false* nebo *nil* v případě nepravdy a cokoli jiného pro pravdivou hodnotu). Příkladem může být metoda `empty?`, která zjišťuje, zda je pole prázdné.

V Pythonu mají takové metody typicky název začínající `is_`. Stejná metoda by se tedy jmenovala `is_empty()`. To je i způsob překladu, který používá současná verze překladače.

Metody končící vykřičníkem slouží ke zdůraznění určité „nebezpečnosti“, často tak jsou označovány metody, které mění objekt, na rozdíl od stejnojmenné metody bez vykřičníku, která vrací objekt nový. Kupříkladu metoda pole `sort()` vrací nové seřazené pole, zatímco metoda `sort!` změní pole původní.

Metody, jejichž název končí znakem `=` slouží jako *settery*, jinak řečeno Ruby přeloží kód:

Ruby

```
objekt.vlastnost = hodnota
```

jako:

Ruby

```
objekt.vlastnost=(hodnota)
```

Pro *gettery* žádná zvláštní syntaxe potřeba není, protože v Ruby lze při volání metody vynechat závorky, tj.:

Ruby

```
objekt.vlastnost()
```

je to samé jako:

Ruby

```
objekt.vlastnost
```

Tato pěkná vlastnost způsobuje těžkosti při překladu do Pythonu, protože někdy nejsme schopni zjistit, jestli přistupujeme k vlastnosti objektu, nebo jde o volání běžné

metody. Možným řešením je vygenerovat pro každou vlastnost objektu stejnojmennou metodu, která vrátí jeho hodnotu. Respektive, protože metoda a instanční proměnná se nemohou v Pythonu jmenovat stejně, je potřeba v takovém případě upravit jméno proměnné, např. připojením podtržítka na začátek.

Implementace

`pyfixes/method_rename.rb`

4.14.3 Operátory

Při předefinování operátorů pak používáme přímo symboly, tj. například plus definujeme jako:

Ruby

```
class Třída
  def +(druhý_operand)
    ...
  end
end
```

Z tohoto existují dvě výjimky:

- Pro odlišení unárních a binárních operátorů mají unární operátory připojen znak zavináč.
- Operátor indexování stojící na levé straně má tvar `[]=`.

V Pythonu se používají speciálně pojmenované metody. Následující tabulka ukazuje převod jmen metod operátorů Ruby na názvy metod v Pythonu:

Ruby	Python 3
unární operátory	
`+@`	`__pos__`
`-@`	`__neg__`
`~@`	`__invert__`
`!@`	přímo není
indexy	
`[]`	`__getitem__`
`[]=`	`__setitem__`
binární operátory	
`+`	`__add__`
`-`	`__sub__`
`*`	`__mul__`
`**`	`__pow__`
`/`	`__truediv__`
`%`	`__mod__`
`&`	`__and__`
`^`	`__xor__`
`<<`	`__lshift__`
`>>`	`__rshift__`
`==`	`__eq__`
`!=`	`__ne__`
`===`	přímo není
`=~`	přímo není
`!~`	přímo není
`<=>`	přímo není
`<`	`__lt__`
`<=`	`__le__`
`>`	`__gt__`
`=>`	`__ge__`

Python nemá operátor `!`, pro negaci používá `not`. Ten přímo předefinovat nejde, lze ale definovat speciální metody `__bool__()` či `__len__()` pro konverzi na logické hodnoty a tím pádem i ovlivnit chování `not`. Co se týče metody `__len__()`, ta slouží zejména k tomu, aby objekt, který představuje kolekci, informoval o počtu svých prvků. Připomeňme, že Python vyhodnocuje jako nepravdu i prázdné kolekce. Viz část *Konverze na logické hodnoty*, s. 24.

U operátorů indexování povoluje Ruby na rozdíl od Pythonu několikanásobný klíč.

V Pythonu lze téhož docílit tím, že operátor indexování vrátí objekt také definující operátor indexování. Například:

Ruby

```
class C
  def [](a, b)
    a * 100 + b
  end
end
c = C.new()
puts c[2, 3]
```

vypíše 203. V Pythonu by ekvivalent byl:

Python

```
class C:
    class Řádek:
        def __init__(self, a): self.a = a
        def __getitem__(self, b): return self.a * 100 + b
    def __getitem__(self, a):
        return C.Řádek(a)

c = C()
print(c[2][3])
```

Operátor << se v Ruby také hojně používá jako připojovací, dají se s jeho pomocí připojovat znaky k řetězci, do výstupních proudů nebo třeba prvky do pole:

Ruby

```
řetězec = "Ahoj "
pole = [1, 2]
výstupní_proud = $stdout
řetězec << "světe\n"
pole << 3 << 4
výstupní_proud << řetězec << pole
```

vypíše:

```
Ahoj světe
[1, 2, 3, 4]
```

Všimněme si, že v případě pole vrátí operátor jako svoji hodnotu výsledné změněné pole a v případě proudu tento proud, takže operátor lze řetězit.

V případě využití operátoru `<<` k připojování může být v Pythonu vhodné ho nahradit metodami jako je `append()`, nebo voláním funkce `print()`.

Operátor `===` se používá v konstrukci *case-when*, kterou Python nemá a neobsahuje tudíž ani odpovídající operátor. Podrobnosti o tomto operátoru jsou uvedeny v části věnované operátorům porovnávání (s. 72) a v části popisující konstrukci *case-when* (s. 76).

Operátory končící vlnovkou slouží pro implementaci regulárních výrazů. V Pythonu jsou regulární výrazy implementovány jako běžná knihovna, takže Python tyto operátory přímo neobsahuje. V případě potřeby by se dalo s výhodou použít operátorů `in` a `not in`, které lze předefinovat implementováním metody `__contains__()`.

Operátor `<=>`, tzv. *spaceship* operátor, slouží k porovnávání, vrací hodnotu menší než nula pokud je levý operand menší než pravý, hodnotu větší než nula, pokud je levý operand větší a nulu pokud si jsou operandy rovny. V Pythonu 3 již není (existoval v Pythonu 2), ale lze ho snadno převést na běžné porovnávací operátory. Pro případy řazení lze využít funkci `cmp_to_key()` z modulu `functools`, která dokáže funkci, odpovídající chováním tomuto operátoru, převést na funkci klíče pro porovnávání, tj. na funkci, která pro každý porovnávaný objekt vrátí hodnotu (typicky číslo), kterou lze pak jednoduše porovnat.

Například funkce `locale.strcoll(první_řetězec, druhý_řetězec)` porovnává dva řetězce v závislosti na aktuálním lokálním prostředí právě podle pravidel popsaných pro *spaceship* operátor. Pokud budeme chtít vypsat prvky pole seřazené touto funkcí můžeme využít `cmp_to_key()` následujícín způsobem:

Python

```
for prvek in sorted(pole, key=cmp_to_key(locale.strcoll)):
    print(prvek)
```

Implementace

```
fixes/send.rb
fixes/operator_append.rb
fixes/operator_binary.rb
fixes/operator_case_cmp.rb
fixes/operator_comparison.rb
fixes/operator_format.rb
fixes/operator_index.rb
fixes/operator_match.rb
fixes/operator_not.rb
fixes/operator_unary.rb
pyfixes/operator_index.rb
pygen/operator_append.rb
pygen/operator_binary.rb
pygen/operator_comparison.rb
pygen/operator_format.rb
pygen/operator_index.rb
pygen/operator_match.rb
pygen/operator_not.rb
pygen/operator_unary.rb
```

4.14.4 Parametry metod

V obou jazycích máme čtyři základní typy parametrů, v Ruby pak ještě speciální pátý:

- poziční parametry,
- poziční sběrný parametr,
- pojmenované parametry,
- pojmenovaný sběrný parametr a
- v Ruby existuje ještě speciální parametr, kterým je blok.

Hezkou vlastností Pythonu je, že poziční a pojmenované parametry se nerozlišují při deklaraci metody, ale při volání se můžeme rozhodnout, jestli je chceme volat pozičně, nebo pomocí jména. Výjimkou jsou některé funkce implementované v C, které lze volat pouze pozičně (např. `abs()`). Převod parametrů se nám tím pádem zjednoduší, neboť poziční i pojmenované parametry převádíme na stejný zápis.

Pro úplnost dodejme, že v Pythonu existuje zápis umožňující vynucení užití parametru pomocí jména, ale pro naše účely je zbytečné ho používat.

Sběrné parametry slouží k tomu, aby „posbíraly“ parametry, explicitně nevyjmenované, do pole (Ruby), resp. n-tice (Python), v případě pozičních, resp. do slovníku v případě pojmenovaných parametrů. Oba jazyky používají stejný zápis s hvězdičkou (tzv. *splat*), kdy jedna hvězdička se používá pro parametry poziční, a dvě hvězdičky pro parametry předávané jménem.

Tato syntaxe se používá jak v seznamu parametrů, tak i pro volání. Ilustrujme si na příkladu:[35]

Ruby

```
def fun(*poziční, **pojmenované)
  p(poziční)
  p(pojmenované)
end

pole = [1, 2, 3]
slovník = {a: 7, b: 8, c:9}

fun(*pole, **slovník)
```

vypíše:

```
[1, 2, 3]
{:a=>7, :b=>8, :c=>9}
```

Implementace

`pygen/argument.rb`

`pygen/splat.rb`

4.14.5 Výchozí hodnoty parametrů

V Pythonu existuje v souvislosti s výchozí hodnotou parametrů jedna zvláštnost. Jsou vyhodnocovány už v okamžiku, kdy interpretr narazí na definice funkce a jsou po celou dobu své existence spojeny s tímto jedním konkrétním objektem. V Ruby jsou oproti

[35] Metoda `p()` vypisuje svoje parametry na konzolu, obdobně jako `puts()`, ale na rozdíl od ní převádí tyto parametry na řetězce metodou `inspect()` a nikoli `to_s()`. Rozdíl je popsán v sekci věnované řetězcům, s. 33.

tomu vyhodnocovány při volání. To má zásadní důsledky v situaci, kdy je výchozí hodnota parametru měnitelná, často třeba prázdné pole nebo slovník.

V Pythonu následující kód:

Python

```
def výchozí(a=[]):
    a.append(1)
    print(a)
výchozí()
výchozí()
výchozí()
```

vypíše:

```
[1]
[1, 1]
[1, 1, 1]
```

Ekvivalent v Ruby:

Ruby

```
def výchozí(a=[])
  a << 1
  p(a)
end
výchozí()
výchozí()
výchozí()
```

oproti tomu zobrazí:

```
[1]
[1]
[1]
```

Tzn. zatímco v Pythonu se výchozí hodnota změní trvale i pro příští volání metody, v Ruby se vždy vytvoří nové pole.

Řešení nabízí tradiční Pythonský idiom: používat jako výchozí hodnoty vždy *None* a na začátku metody provést inicializaci ručně:

Python

```
def výchozí(a=None):
    if a is None: a = []
    a.append(1)
    print(a)
```

Možnou optimalizací je pak použití tohoto idiomu pouze v případech, kdy je výchozí hodnota měnitelná. To znamená, že můžeme ponechat hodnoty jako jsou čísla, *true*, *false*, nebo *nil*.

Protože se výchozí hodnoty vyhodnocují v Ruby až při volání, znamená to také, že můžeme použít hodnotu předchozího parametru jako výchozí hodnotu dalšího:

Ruby
```
def výchozí(a, b=a)
  puts(a, b)
end
```

I v tomto případě je potřeba provést stejnou transformaci na:

Ruby
```
def výchozí(a, b=None):
    if b is None:
        b = a
    print(a, b)
```

Implementace

pyfixes/opt_arg.rb

4.14.6 Návratová hodnota

Jedním z oříšků automatického překladu mezi Ruby a Pythonem je návratová hodnota metody. Zatímco v Pythonu, pokud neuvedeme explicitní příkaz `return`, je návratovou hodnotou *None*, v Ruby je v takovém případě návratovou hodnotou hodnota posledního vykonaného příkazu v těle metody.

V principu nejjednodušším, a reálně skoro jediným možným řešením tohoto problému je ukládat si v těle metody hodnotu vykonaných výrazů a na konci metody ji vrátit. Situace je ještě o něco komplikovanější tím, že zatímco prakticky všechny jazykové konstrukce v Ruby jsou výrazy (například i přiřazení, podmínka ap.), tak v Pythonu o výrazy nejde.

Při překladu je tedy nutné rozlišovat následující situace:

- Posledním vykonaným příkazem v těle metody v Ruby nebyl výraz. Jde vlastně o příkazy, které explicitně přerušují zpracování metody, jako je explicitní `return` nebo vyvolání výjimky. Z hlediska překladu je tato situace nejjednodušší, protože ji není potřeba ošetřovat.

- Poslední vykonaný příkaz je výrazem v Ruby i v Pythonu. I toto je poměrně snadno řešitelná situace, kdy stačí přiřadit výsledek příkazu do proměnné, kterou na konci metody vrátíme. V mojí implementaci jsem ji pojmenoval `_result` .

- Přikaz je výrazem v Ruby, ale nikoli v Pythonu. Jde o nejsložitější situaci. Je nutné pro každou konstrukci explicitně zjistit, jakým způsobem počítá výslednou hodnotu a rozdělit takový příkaz do dvou: nejprve spočítat hodnotu, přiřadit ji pomocné proměnné `_result` a následně ji použít ve vlastním příkazu.

  Ukažme si to na příkladu přiřazení. Původní kód

  Ruby

  ```
  def fun
    a = 4
  end
  ```

  převedeme na

  Python

  ```
  def fun():
      _result = 4
      a = _result
      return _result
  ```

  Vlastní implementace ještě explicitně na začátku metody inicializuje `_result` hodnotou *None*, abychom měli jistotu, že tato proměnná existuje. Alternativou by bylo analyzovat tok kódu a tuto inicializaci provádět jen v případě, kdybychom seznali, že existuje takový průchod metodou, který návratovou hodnotu nenastavuje. Zobecnění takové analýzy by se dalo použít pro optimalizaci, kdybychom do proměnné `_result` přiřazovali pouze takové výrazy, jejichž hodnota může být vrácena. Například:

  Ruby

```
def fun
  a = 4
  b = 5
end
```

Aktuální verze překladače tento kód přeloží jako:

Python

```
def fun():
    _result = None
    _result = 4
    a = _result
    _result = 5
    b = _result
    return _result
```

Hodnota prvního přiřazení do proměnné `a` nicméně nikdy nebude vrácena. Popsanou optimalizací by se tedy dal tento kód zjednodušit na:

Python

```
def fun():
    a = 4
    _result = 5
    b = _result
    return _result
```

Oba jazyky také podporují vracení více hodnot. V případě Pythonu budou vráceny jako n-tice, v případě Ruby jako pole, Ruby ani ekvivalent n-tic nemá. Základní rozdíl mezi polem a n-ticí je v tom, že n-tice je neměnná.

Ve většině případů, kdy nám více návratových hodnot slouží k tomu, abychom je přiřadili do více proměnných ve výrazech typu:

```
adresář, soubor = rozděl_jméno_souboru(celá_cesta)
```

nám tento rozdíl nevadí. Pokud bychom ale narazili na příklad, kdy se s vrácenou hodnotou pracuje jako se skutečným polem, museli bychom situaci řešit buď tím, že z metody vrátíme explicitně pole, nebo tím, že n-tici zkonvertujeme na pole po volání metody.

Implementace

`pyfixes/def.rb`

`pygen/return.rb`

4.15 Bloky

V Ruby existuje speciální parametr typu blok. Jde vlastně o klauzuru, která je předána funkci. Parametr tohoto typu má dosti výjimečné postavení:

- Může být jen jeden.
- Může být uveden explicitně v seznamu parametrů, ale nemusí. Pokud je uveden, musí jeho název začínat znakem `&` a musí jít o poslední parametr. Není-li uveden, lze zjišťovat voláním metody `block_given?`, zda byl předán, a zavolat ho příkazem `yield`.
- Při volání metody se blok typicky zapisuje za zápis volání (tj. až za uzavírající kulatou závorku) mezi klíčová slova `do` a `end`, resp. mezi složené závorky.

Ukažme si blok na příkladu:

Ruby

```
def fun
  if block_given?
    yield
  else
    puts('Žádný blok nebyl předán')
  end
end

fun()                   # vypíše: Žádný blok nebyl předán
fun() { puts 'Ahoj' }  # vypíše: Ahoj
```

Blokům lze dále předávat parametry a získat návratovou hodnotu. Velmi typické je použití bloků pro procházení kolekce metodou `each()`:

Ruby

```
[1, 2, 3].each {|hodnota| puts(hodnota)}
```

Python přímý ekvivalent bloků nemá. Co se týče procházení kolekce pomocí `each()` lze výše uvedený zápis nahradit konstrukcí `for-in`:

Python

```
for hodnota in [1, 2, 3]:
    print(hodnota)
```

Mimochodem, tato konstrukce existuje i v Ruby, ale skoro se nepoužívá. Od verze Ruby 1.9 se nicméně konstrukce `for-in` od metody `each()` liší jednou vlastností: proměnné bloku (v našem případě *hodnota*) jsou lokální pro blok, tj. *hodnota* není vidět mimo blok (viz též pasáž věnovaná cyklu `for-in`, s. 80).

S převodem ostatních konstrukcí je to složitější. Python obecně anonymní klauzury nemá. Pokud by tělo klauzury tvořil jediný výraz, šlo by použít *lambda funkci*. V ostatních případech by šlo buď převést bloky opět na `for-in` cyklus, bez ohledu na to, že by nešlo o procházení kolekce, nebo definovat lokální vnořenou funkci s vygenerovaným názvem a předat jí funkci. Tato druhá varianta je obzvláště vhodná v případech funkcionálních metod typu `map()`.

Ukažme si možnosti převodu na příkladu:

Ruby

```
čtverce = [1, 2, 3].map() {|hodnota| hodnota**2}
```

Převod pomocí *lambda funkce*:[36]

Python

```
čtverce = list(map(lambda hodnota: hodnota**2, [1, 2, 3]))
```

Převod pomocí `for-in`:

Python

```
čtverce = []
for hodnota in [1, 2, 3]:
    čtverce.append(hodnota**2)
```

Převod pomocí „anonymní“ funkce:

Python

[36] Protože v Pythonu 3 vrací funkce `map()` tzv. *iterátor*, je potřeba jej explicitně převést na pole voláním konstruktoru třídy `list`.

```
def _blok(hodnota):
    return hodnota**2
čtverce = list(map(_blok, [1, 2, 3]))
```

Třetí způsob je nejobecnější. Je potřeba si u něj ale dát pozor ještě na některé problémy.

Implementace

```
fixes/block.rb
fixes/block_fix_node.rb
fixes/block_given_test.rb
fixes/custom_block.rb
pyfixes/blockpass.rb
pyfixes/block_emulation_contains_block.rb
pyfixes/block_fix_node.rb
pyfixes/block_given_test.rb
pyfixes/custom_block.rb
pygen/block_given_test.rb
pygen/block_pass.rb
```

4.15.1 Implicitní rozklad více argumentů

V Ruby může mít blok, který je parametrem `map()` rovnou více argumentů.

Uvažujme například pole dvojic reprezentovaných opět polem. V Ruby pak může blok mít dva argumenty, přičemž každá dvojice se automatiky rozdělí:

Ruby

```
pole_polí = [[1, 'a'], [2, 'b'], [3, 'c']]
pole_polí.map() do
  |a, b|
  return ">>#{a} #{b}<<"
end
```

V Pythonu je naproti tomu potřeba provést rozklad ručně:

Python

```
pole_polí = [[1, 'a'], [2, 'b'], [3, 'c']]
def _blok(args):
    a, b = args
```

```
    return ">>%s %s<<" % (a, b)
map(_blok, pole_polí)
```

Implementace

pyfixes/block_multiple_arguments.rb

4.15.2 Přiřazení do proměnné metody

Uvažujme kód:

Ruby

```
def metoda
  lokální_proměnná = 1
  čtverce = [1, 2, 3].map do |hodnota|
    lokální_proměnná = hodnota
    hodnota**2
  end
  puts(lokální_proměnná, čtverce)
end
```

Překladem na explicitní blok získáme:

Python

```
def metoda(self):
    lokální_proměnná = 1
    def _blok(hodnota):
        lokální_proměnná = hodnota
        return hodnota**2
    čtverce = list(map(_blok, [1, 2, 3]))
    print(lokální_proměnná, čtverce)
```

To bohužel nebude fungovat, protože přiřazení `lokální_proměnná = hodnota` vytvoří novou lokální proměnnou funkce `_blok`. Pro správnou funkčnost je potřeba přidat příkaz `nonlocal`:

Python

```
def metoda(self):
    lokální_proměnná = 1
```

```
def _blok(hodnota):
    nonlocal lokální_proměnná
    lokální_proměnná = hodnota
    return hodnota**2
čtverce = list(map(_blok, [1, 2, 3]))
print(lokální_proměnná, čtverce)
```

Implementace

pyfixes/block_emulation_nonlocals.rb

4.15.3 Podporované speciální bloky

Implementace

Kromě obecné implementace je implementována speciální podpora pro následující typy bloků:

```
any?
collect
delete_if
detect
each
each_with_index
each_with_object
find
Hash
gsub
inject
loop
map
map!
open
partition
reverse_each
select
sort
times
upto
```

4.16 Porovnávání

Zatímco Python obsahuje jen dva porovnávací operátory, klasické `==` a pak operátor identity `is`, Ruby má pro porovnávání objektů hned několik možností s mírně odlišnými významy (Flanagan a Matsumoto, 2008, s. 76):

- **Metoda equal?** Klasická identita objektů. `a.equal?(b)` je vlastně totéž co `a.object_id == b.object_id` (Flanagan a Matsumoto, 2008, s. 76). Odpovídá operátoru `is` v Pythonu.

- **Operátor ==** Běžné porovnání. Pokud ho třída, nebo některý její předek nedefinuje, pak výchozí implementací je metoda `equal?`, tedy identita objektů. Stejné je to i v Pythonu.

- **Metoda eql?** Není-li předefinována, pak funguje stejně jako `equal?`. Doporučuje se, aby u vlastních tříd šlo o přísnější verzi operátoru `==`, která neprovádí konverzi typů (Flanagan a Matsumoto, 2008, s. 77).

  Důležité je, že standardní slovník v Ruby (třída *Hash*) používá `eql?` pro porovnání klíčů.

- **Operátor ===** Tento operátor slouží primárně pro řídicí konstrukci *case-when* (viz s. 76). Ve výchozí implementaci předá porovnání operátoru `==`. Řada standardních tříd ale implementuje tento operátor jinak:

  – třída zjišťuje, jestli je parametr její instancí,

  – regulární výraz zjišťuje, jestli parametr vyhovuje danému vzoru,

  – rozsah zjišťuje, zda hodnota parametru v něm leží a podobně.

  Na to je potřeba při překladu pamatovat a tyto situace ošetřit.

Implementace

`fixes/operator_case_cmp.rb`
`fixes/operator_comparison.rb`
`pyfixes/case.rb`

4.17 Řídicí výrazy

Zvláštnostní Ruby je, že skoro vše je výraz, včetně konstrukcí jako je *if* nebo *while*. V Pythonu výrazem nejsou a tento rozpor je potřeba vyřešit. Nejenom, že vrací hodnotu, kterou lze například přiřadit proměnné, ale také to znamená, že je-li například `if` posledním příkazem v těle metody, pak se jeho hodnota stane implicitně návratovou hodnotou této metody.

Další, na co je potřeba si při překladu dát v této souvislosti pozor je skutečnost, že tím pádem mohou tyto výrazy stát v podmínce. Například typické je přiřazení v podmínce konstrukce `while`:

Ruby

```
while řádek = vstupní_proud.čti_řádek()
  ...
end
```

Takovýto kód čte ze vstupního proudu řádky a zároveň každý přečtený řádek uloží do proměnné `řádek`. V Pythonu není přiřazení výraz a je potřeba podobný kód nahradit:

Python
```
řádek = vstupní_proud.čti_řádek()
while řádek:
    ...
    řádek = vstupní_proud.čti_řádek()
```

Implementace

`pyfixes/while.rb`

4.17.1 Podmínky (if, unless)

Ruby má běžnou podmínku *if-elsif-else*, která má přímou obdobu v Pythonu, který se liší jen tím, že místo klíčového slova `elsif` používá `elif`.

Je ale potřeba říci, že použitý parser Ruby AST konstrukci:

Ruby
```
if podmínka1
  ...
elsif podmínka2
  ...
end
```

reprezentuje jako:

Ruby
```
if podmínka1
  ...
else
  if podmínka2
    ...
  end
end
```

(viz též s. 117).

Kromě toho má Ruby navíc oproti Pythonu také klíčové slovo `unless`, které má obrácený pravdivostní test, takže je ekvivalentní výrazu `if not`. Výraz `unless` může být spojen s `else`, ale není možné ho kombinovat s `elsif`.

Použitý parser výraz `unless` vrací ve skutečnosti jako `if` s prohozenými těly, tj. pro:

Ruby

```
unless podmínka
  a()
else
  b()
end
```

vrátí parser AST odpovídající:

Ruby

```
if podmínka
  b()
else
  a()
end
```

Tím vlastně vyřeší překlad `unless` na `if` za nás.

V případě chybějící varianty `else` pak nechá tělo za if prázdné:

Ruby

```
unless podmínka
  a()
end
```

převede na:

Ruby

```
if podmínka
else
  a()
end
```

Tuto variantu je potřeba ohlídat, protože v Pythonu není prázdné tělo povoleno, je potřeba ho explicitně vyznačit klíčovým slovem `pass`:

Python

```
if podmínka:
    pass
else:
    a()
```

V tomto případě je ale jednodušší a čitelnější variantou provést konverzi na `if not`:

Python

```
if not(podmínka):
    a()
```

Hodnotou výrazu `if` nebo `unless` je hodnota posledního provedeného výrazu v těle vybrané varianty. Pokud žádný proveden není, tedy v případě, že podmínka nebyla splněna a chybí `else`, nebo tělo vybrané varianty neobsahuje žádné výrazy, pak je hodnotou podmínky `nil`.

Implementace

`pyfixes/if.rb`
`pygen/if.rb`

4.17.2 Podmínky ve formě modifikátorů

Na rozdíl od Pythonu má Ruby ještě jednu zvláštní formu podmínek, kdy `if` nebo `unless` může stát napravo od příkazu, který se vykoná v případě (ne)splnění podmínky:

Ruby

```
příkaz if podmínka
příkaz unless podmínka
```

Protože parser tuto podobu převede na běžný tvar, není potřeba jí zvláště ošetřovat při překladu.

4.17.3 Ternární operátor

Oba jazyky mají i ternární operátor. Zatímco v Ruby má podobu běžnou i v jiných jazycích vycházející ze syntaxe jazyka C:

Ruby

```
podmínka ? výraz_pro_splněnou_podmínku : výraz_pro_nesplněnou_podmínku
```

v Pythonu má dosti netradiční formu s jiným pořadím jednotlivých částí:

Python

```
výraz_pro_splněnou_podmínku if podmínka else výraz_pro_nesplněnou_podmínku
```

Bohužel parser i tuto formu převede na standardní tvar `if`. To je v tomto případě na škodu, protože běžný tvar `if` není v Pythonu výrazem a není ho možné použít na místech, kde se typicky ternární operátor používá. V zásadě jsou při překladu z Ruby do Pythonu možné dva způsoby řešení:

- Protože uzly AST obsahují odkaz na část zdrojového souboru, ze které vznikly, lze tuto informaci využít ke zjištění, v jakém tvaru byla podmínka zapsána v původním zdrojovém souboru.
- Využít heuristiky. To je i varianta, kterou jsem použil já. Podmínka se chápe jako ternární operátor například pokud je použita v definici slovníku, ať už jako klíč nebo hodnota, nebo pokud je uzávorkována.

Implementace

`pyfixes/ternary.rb`

4.17.4 Několikanásobný výběr (case, when)

Ruby má na rozdíl od Pythonu několikanásobný výběr, tj. konstrukci známou z programovacích jazyků pod různými názvy, jako *select*, *switch* a podobně. V Ruby se používá dvojice klíčových slov `case` a `when`.

Tato konstrukce může mít dvě podoby. Ta jednodušší je vlastně jen jiný zápis pro sekvenci několika *if … elsif … elsif … else*. Například:

Ruby

```
if den == :pondeli
  puts 'pondělí'
elsif den == :utery
  puts 'úterý nebo středa'
elsif den == :streda
  puts 'úterý nebo středa'
else
  puts 'neplatný den'
end
```

lze napsat i jako:

Ruby

```
case
when den == :pondeli
  puts 'pondělí'
when den == :utery
  puts 'úterý nebo středa'
when den == :streda
  puts 'úterý nebo středa'
else
  puts 'neplatný den'
end
```

V tomto případě není s převodem do Pythonu na sérii *if-elif* zjevně žádný problém. Jedinou zvláštností je, že můžeme alternativy logické disjunkce kromě operátorů `or` či `||` (svislítka) oddělit i čárkou:

Ruby

```
when den == :utery, den == :streda
  puts 'úterý nebo středa'
```

Klasičtější forma *case-when*, více odpovídající jiným jazykům, testuje hodnotu jedné proměnné vůči několika variantám. Předchozí příklad bychom tedy mohli přepsat jako:

Ruby

```
case den
when :pondeli
```

```
  puts 'pondělí'
when :utery, :streda
  puts 'úterý nebo středa'
else
  puts 'neplatný den'
end
```

Vidíme, že i v tomto případě můžeme čárkou oddělovat disjunktní alternativy.

Ač v tomto případě bude poslední varianta fungovat stejně jako ty předchozí, je zde velký principielní rozdíl. Zatímco v předchozích variantách jsme explicitně uváděli porovnávací operátor, v tomto případě se vždy použije operátor `===`. Repektive v kódu:

Ruby
```
case hodnota
when varianta_1
  ...
when varianta_2
  ...
end
```

se nejprve provede `varianta_1.===(hodnota)` a pak `varianta_2.===(hodnota)`. Všimněme si, že objekt, jehož metoda `===` se volá, je objekt jmenovaný za `when` a jeho parametrem je objekt jmenovaný za `case`, tedy pořadí je vlastně opačné oproti pořadí uvedenému v zápisu zdrojového kódu.

Na toto si musíme dát při převodu do Pythonu pozor, protože operátor `===` může být obecně definován jinak než `==`. Zejména to platí pro třídy, kdy operátor testuje příslušnost objektu k třídě, tedy vlastně `Třída === objekt` je obdoba `objekt.is_a?(Třída)`.

Další věcí, kterou je potřeba si pohlídat jsou vedlejší efekty, které může mít vyhodnocení porovnávané proměnné. Kdybychom kód:

Ruby
```
case metoda()
when varianta_1
  ...
when varianta_2
  ...
end
```

čistě mechanicky převedli na:

Python

```
if case_cmp(metoda(), varianta_1):
    ...
elif case_cmp(metoda(), varianta_2):
    ...
```

kde `case_cmp` je metoda realizující v Pythonu neexistující operátor `===`, pak voláme `metoda()` dvakrát a to nechceme. Je potřeba ji vyhodnotit pouze jednou a vypočítanou hodnotu si uložit:

Python

```
hodnota = metoda()
if case_cmp(hodnota, varianta_1):
    ...
elif case_cmp(hodnota, varianta_2):
    ...
```

Také je potřeba ošetřit, že konstrukce `case-when` je v Ruby výraz. Vrací hodnotu posledního provedeného výrazu, nebo *nil* v případě, že žádná varianta nebyla zvolena a chybí průchod `else`.

Pokud je naše proměnná ukládající návratovou hodnotu `_result` (viz s. 63), pak ji můžeme zároveň využít pro výše zmíněnou úschovu vypočítané hodnoty. Pokud *case-when* nemá `else`, pak je potřeba ho doplnit a bude obsahovat příkaz nastavující `_result` na `None`:

Python

```
_result = metoda()
if case_cmp(_result, varianta_1):
    ...
elif case_cmp(_result, varianta_2):
    ...
else:
    _result = None
```

Implementace

`fixes/operator_case_cmp.rb`
`pyfixes/case.rb`
`pyfixes/when.rb`

4.17.5 Cykly

Ruby nabízí několik typů cyklů. Pro průchod prvky kolekce lze použít konstrukci `for-in`:

Ruby

```
for prvek in [1, 2, 3]
  puts(prvek)
end
```

Tuto konstrukci lze použít s libovolným objektem, který definuje metodu `each()`. Cyklus:

Ruby

```
for a, b, c in objekt
  ...
end
```

je ekvivalentní:

Ruby

```
objekt.each() do
  |a, b, c|
  ...
end
```

s tím rozdílem, že v Ruby jsou od verze 1.9 parametry bloku (v našem případě `a`, `b`, `c`) pro blok lokální, zatímco při použití `for-in` cyklu jde o klasické lokální proměnné metody. Z toho také plyne, že pokud již bude existovat proměnná stejného jména, jaké má proměnná `for-in` cyklu, a bude mít před jeho vykonáním nějakou hodnotu, bude tato hodnota přepsána, zatímco v případě použití `each()` nikoli:

Ruby

```
i = 5
puts(i)   # => vypíše 5

# přepíše hodnotu i
for i in [1, 2, 3]; end
puts(i)   # => vypíše 3
```

```
# hodnota i se nezmění
[10, 20, 30].each() {|i|}
puts(i)   # => vypíše 3
```

Hodnotou výrazu `for-in` je iterovaný objekt, nebo hodnota předaná příkazu `break`, pokud byl použit k ukončení cyklu.

V Pythonu existuje stejnojmenná konstrukce, pravidla pro lokální proměnnou jsou shodná. Pravidla pro objekty, které lze takto procházet jsou v Pythonu poněkud složitější. Takové objekty se v Pythonu označují jako *iterovatelné* a musí implementovat patřičným způsobem buď metodu `__iter__()` vracející objekt nazývaný *iterátor*, nebo metodu `__getitem__()` vracející procházené hodnoty.

Překlad cyklu `for-in` pro standardní typy jako pole nebo slovník je přímočarý, překlad pro vlastní typy implementující `each()` je již složitější. V našem případě to pro převod knihovny Prawn naštěstí nepotřebujeme.

Cyklus `for-in` se již v Ruby příliš nepoužívá, nicméně má smysl převádět procházení kolekcí pomocí `each` v Ruby do procházení pomocí `for-in` v Pythonu, viz s. 66.

V případě cyklu `loop` jde o jeden z příkladů toho, kdy se metoda implementovaná ve třídě `Kernel`, a tudíž dostupná v každém objektu, tváří jako klíčové slovo jazyka. Jde o jednoduchý nekonečný cyklus, jehož parametrem je blok, který lze kromě obligátního `break` také ukončit vyvoláním výjimky `StopIteration`. Metoda `loop` tuto výjimku zachytí, čímž zastaví její propagaci, a jednoduše se vrátí.

Návratová hodnota `loop` je buď hodnota předaná příkazu `break`, nebo *nil* v případě vyvolání výjimky `StopIteration`.

Podobně jako existuje v Ruby dvojice `if` a `unless`, tak i standardní obecný cyklus `while` má protějšek `until`, který testuje podmínku na nepravdivou hodntu. Hodnotou výrazu `while` nebo `until` je *nil*, pokud nebyl ukončen příkazem `break`, kterému byla předána hodnota.

Tyto cykly mají, podobně jako podmínky, i tvar ve formě modifikátorů, které se píší za příkaz, který se má opakovat:

Ruby

```
násobek *= 2 while násobek < 1000
```

Použitý parser nicméně reprezentuje shodně jak klasickou formu, tak formu modifikátorů, takže z hlediska překladu není potřeba ošetřovat zvlášť tuto formu.

Pro úplnost se zmiňme ještě o jedné podobě cyklů. Pokud je `while` nebo `until` použito jako modifikátor obecného sledu příkazů ohraničeného klíčovými slovy `begin` a `end`, pak se tento sled příkazů vykoná vždy alespoň jednou:

Ruby

```
begin
  puts('v těle cyklu')
end while false
```

vypíše „v těle cyklu“. Vzhledem k příliš velké podobnosti s tvarem:

Ruby

```
puts('v těle cyklu') while false
```

který neproběhne ani jednou, se nicméně formu `begin-end-while/until` nedoporučuje používat a (Flanagan a Matsumoto, 2008, s. 128) upozorňuje na to, že je možné, že budoucí verze mohou dokonce tuto formu úplně zakázat. [37] Aktuální verze Ruby (2.3.0) ale při použití této konstrukce nevypisuje ani varování, byť ji lze chápat jako zastaralou *(deprecated)*. Vzhledem k jejímu malému použití se jí není potřeba při překladu zabývat.

Ostatní formy `while` a `until` přeložíme na standardní Pythonský cyklus `while`, resp. `while not`.

Implementace

`fixes/loop.rb`
`pyfixes/while.rb`
`pygen/loop.rb`
`pygen/until.rb`
`pygen/while.rb`

4.17.6 Skoky (break, next, redo)

Blok nebo cyklus je možné předčasně ukončit příkazem `break`. Stejnojmenným příkazem je možné přerušit i cykly v Pythonu. Toto jednoduché, a naštěstí zdaleka nejčastejší, použití tak není problém přeložit do Pythonu.

Pokud bychom při překladu nahradili blok funkcí (viz s. 66), pak bychom ale museli `break` nahradit vyvoláním výjimky. Což ve skutečnosti odpovídá i tomu, jak je `break` uvnitř bloku implementováno i v Ruby MRI na úrovni bajtkódu (Shaughnessy, 2014, s. 90).

Další rozdíl existuje v tom, že v Ruby lze příkazu `break` předat hodnotu, která je pak dostupná buď jako hodnota výrazu cyklu, nebo jako výsledek volání cyklu metodou `yield`.

[37] Hlavní tvůrce Ruby, Yukihiro Matsumoto, dokonce vyjádřil lítost, že tuto vlastnost do jazyka přidal a touhu jí z něho odstranit. Viz jeho zpráva v e-malové konferenci *Ruby-Core* z 23. 11. 2005: http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/6745

V klasických cyklech se naštěstí tato možnost příliš nevyužívá, v případě bloků přeložených na funkce bychom návratovou hodnotu museli předat prostřednictvím objektu výjimky, nebo nějakou sdílenou pomocnou proměnnou.

Příkaz Ruby `next` slouží k k ukončení současné iterace a pokračováním další. V bloku pak může mít parametr, který bude použit jako návratová hodnota současné iterace. Python má pro cykly obdobný příkaz `continue`.

Pokud bychom při překladu nahradili blok funkcí (viz s. 66), pak je ale potřeba nahradit `next` za `return`. Případná návratová hodnota by se pak stala návratovou hodnotou `return`.

Ruby má navíc ještě příkaz `redo`, který zopakuje aktuální iteraci. Python ekvivalent nemá. V případě potřeby by se tento příkaz dal nahradit cyklem s příznakovou proměnnou:

Ruby

```
puts 'Hádejte čísla mezi 1 a 10'
číslo = rand 1..10
pokus = 1
hádané = nil
until hádané == číslo
  print "#{pokus}. pokus: "
  # to_i vrátí 0, pokud nejde o platné celé číslo
  hádané = gets.strip.to_i
  if hádané < 1 or hádané > 10
    puts 'Prosím zadejte číslo mezi 1 a 10'
    redo
  end
  puts "Tipuješ #{hádané}"
  pokus += 1
end
puts "Uhádl jste na #{pokus - 1}. pokus"
```

Python

```
from random import randint
print('Hádejte čísla mezi 1 a 10')
číslo = randint(1, 10)
pokus = 1
hádané = None
```

```
while hádané != číslo:
    redo = True
    while redo:
        redo = False
        try:
            # int() vyvolá ValueError pokud vstup není platné celé číslo
            hádané = int(input('%d. pokus: ' % pokus).strip())
            if hádané < 1 or hádané > 10:
                raise ValueError
        except ValueError:
            print('Prosím zadejte číslo mezi 1 a 10')
            redo = True
    print("Tipuješ %d" % hádané)
    pokus += 1
print("Uhádl jste na %d. pokus" % (pokus - 1))
```

Implementace

`pyfixes/block_emulation_control_expressions.rb`
`pygen/break.rb`
`pygen/next.rb`

4.18 Běžné idiomy

Rozumět běžným obratům v obou jazycích je důležité pro jejich vzájemný překlad. Ukažme si to hned na prvním příkladu.

4.18.1 Hluboká a mělká kopie

První idiom se týká vytváření hlubokých kopií objektů. Než se dostaneme k samotnému idiomu, podívejme se na možnosti kopírování objektů v obou jazycích.

Oba jazyky používají klasický referenční objektový model, kdy proměnné obsahují pouze odkaz (referenci) na instanci třídy jinde v paměti. Běžné přiřazení mezi proměnnými ve výsledku tedy vede jen na to, že obě proměnné odkazují na stejný objekt.

Někdy je ale potřeba vytvořit skutečnou kopii objektu. Python podporuje vytváření kopie prostřednictvím standardního modulu *copy*. Jeho funkce `copy()` umožňuje vytvářet mělké kopie, funkce `deepcopy()` kopie hluboké.

Rozdíl mezi mělkou a hlubokou kopií se týká objektů, které obsahují odkazy na další objekty:

- *Mělká kopie* vytvoří kopii pouze prvního, vnějšího objektu. Jeho vnitřní odkazy povedou na původní objekty. Nový a původní objekt je tak budou sdílet.
- *Hluboká kopie* rekurzivně zkopíruje i odkazované objekty.

Rozdíl ilustruje obrázek.

Objekty v Pythonu mohou kontrolovat kopírování objektů implementováním metod `__copy__()`, resp. `__deepcopy__()`.

Ruby má přímo podporu jen pro mělké kopírování. Na rozdíl od Pythonu nabízí pro mělké kopie dokonce hned dvě metody: `clone()` a `dup()`. Mezi nimi jsou jemné rozdíly, které jdou už mimo rámec této práce. Z hlediska implementace vlastního překladače je budeme považovat za ekvivalentní. Zájemci najdou popis rozdílů například v (Flanagan a Matsumoto, 2008, s. 83).

Podobně jako v Pythonu mohou i v Ruby objekty svoje kopírování ovlivňovat. Kromě možnosti vlastní implementace metod `clone()` a `dup()` je zde i „kopírovací konstruktor“ `initialize_copy()`. Pokud je definován, pak se použije místo běžného konstruktoru `initialize()` a jako parametr dostane původní objekt.

Protože podpora hluboké kopie přímo v jazyku není, vytvořil se pro ni idiom:

Ruby

```
nový_objekt = Marshal.load(Marshal.dump(původní_objekt))
```

Ten využívá standardních metod `Marshal.dump()`, která převede objekt na binární řetězec a `Marshal.load()`, která z tohoto řetězce vytvoří nový objekt. Mimochodem, Python interně používá obdobný postup, nicméně tím, že se vyhne serializaci na řetězec a jeho parsování zpátky, může být o dost rychlejší. Já jsem ve vlastní implementaci překladače zprvu tento idiom také používal, ale pro složitější stromy objektů je v Ruby opravdu pomalý a i velmi paměťově náročný.

Z hlediska překladu je důležité rozumět tomuto idiomu, už proto, že se vyskytuje i v knihovně Prawn, a výše uvedenou konstrukci překládat jako:

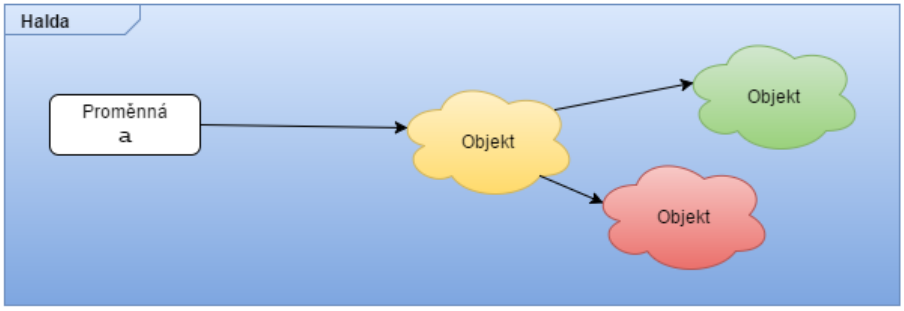
Python

```
import copy
nový_objekt = copy.deepcopy(původní_objekt)
```

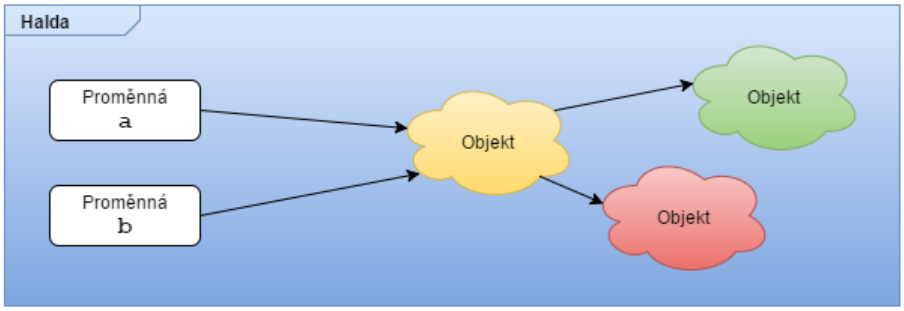
Vraťme se ještě na chvíli k mělké kopii. Abychom ji mohly provádět, je potřeba převést `initialize_copy()` na `__copy__()`. Zde je potřeba vyřešit rozpor, že jazyky volí vzájemně protichůdný přístup. Zatímco v Ruby je `initialize_copy()` metoda nového objektu, která dostane ten původní jako parametr, v Pythonu se `__copy__()` volá na původním objektu a nový objekt vrací jako svou návratovou hodnotu.

Obrázek 2: Typy kopií

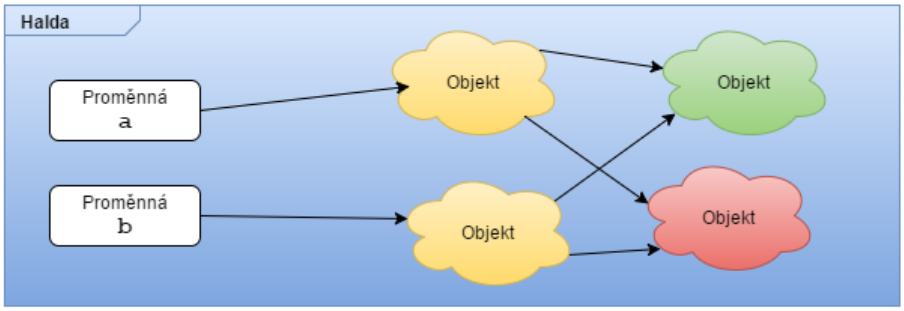
Výchozí situace



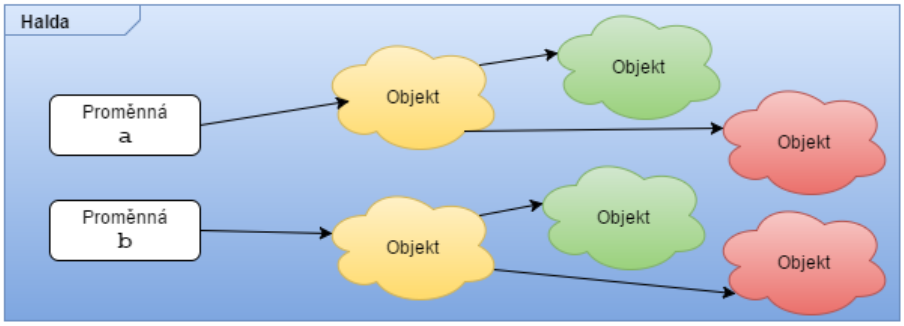
Přiřazení proměnné



Mělká kopie



Hluboká kopie



Pro překlad to znamená, že implementuje-li třída `initialize_copy()`, pak je záhodno vygenerovat takovouto metodu `__copy__()`:

Python

```
class Třída:
    def initialize_copy(self, původní):
        ...
    def __copy__(self):
        kopie = Třída()
        kopie.initialize_copy(self)
        return kopie
```

Implementace

`pyfixes/copy.rb`

4.18.2 Pozdní inicializace

V Ruby je běžná pozdní inicializace, také nazývaná *Nil Guards* (Perrotta, 2011, s. 226), typu:

Ruby

```
def metoda
  @proměnná ||= výchozí_hodnota
end
```

Pokud proměnná ještě neexistuje, je jí přiřazena výchozí hodnota a ta je i zároveň vrácena z metody. Pokud proměnná existuje, je rovnou vrácena její hodnota. To funguje díky tomu, že neexistující proměnná se chová jakoby existovala a obsahovala hodnotu `nil`.

Napsat přímý ekvivalent v Pythonu je problematické ze dvou důvodů:

- Neexistuje přímý ekvivalent operátoru `||=`.
- Přístup k neexistující proměnné vyhodí výjimku *AttributeError*.

Výše uvedený kód je v principu nutné přeložit jako:

Python

```
def metoda(self):
    if not hasattr(self, 'proměnná'):
        self.proměnná = výchozí_hodnota
    return self.proměnná
```

resp. idiomatičtěji[38] pro Python:

Python

```
def metoda(self):
    try:
        return self.proměnná
    except AttributeError:
        self.proměnná = výchozí_hodnota
        return self.proměnná
```

Implementace

`fixes/or_asgn.rb`

`pyfixes/or_asgn.rb`

4.18.3 Test na *nil*

Další idiom byl již zmíněn v sekci o Neznámé hodnotě (s. 23). Test na *nil*:

Ruby

```
proměnná.nil?
```

musíme převést na:

Python

```
proměnná is None
```

resp. u instančních proměnných:

Ruby

```
@proměnná.nil?
```

[38] Ze slovníčku, který je součástí dokumentace Pythonu: „EAFP – Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false.“

vzniká obdobná situace jako v případě výše uvedené pozdní inicializace a přesnější by bylo:

Python

```
(not hasattr(self, 'proměnná')) or (self.proměnná is None)
```

Implementace

`fixes/nil_test.rb`

`pygen/nil_test.rb`

5 Vlastní řešení

5.1 Celkový přehled

Vlastní překlad se dá rozdělit do několika fází, které ilustruje obrázek. Modrá plocha znázoňuje mnou implementovaný překladač.

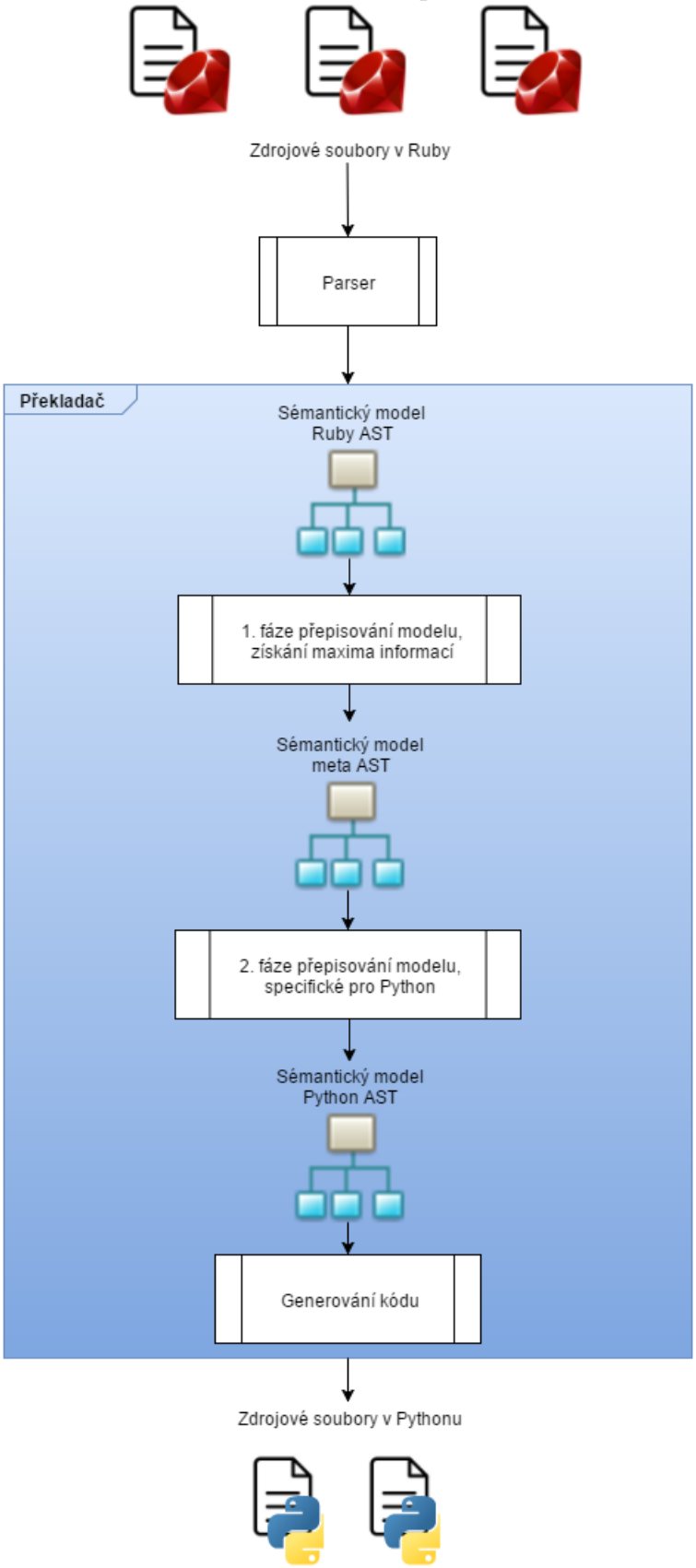
K terminologii bych chtěl poznamenat, že ne zcela přesně používám jako ekvivalenty termíny sémantický model a syntaktický strom/AST, byť samozřejmě první z nich je mnohem širší. V našem případě je ale vstupním i výstupním modelem právě AST konkrétního jazyka.

- Vstupem je kolekce zdrojových souborů napsaných v jazyku Ruby.
- Ty jsou zpracovány parserem. Jak již bylo zmíněno v úvodních částech práce, vlastní parser jsem si nepsal, neboť parsery pro Ruby jsou k dispozici. Ruby nám dává přímo možnost využít parser, který je součástí interpreteru, nicméně tím se omezíme na parsování jen takového kódu, který odpovídá verzi interpreteru, na kterém Ruby spouštíme.

  Já jsem zvolil parser s prostým jménem *Parser*[39]. Ten je aktivně vyvíjený, napsaný v čistém Ruby, dobře kompatibilní a má dobrou dokumentaci.
- Tím ze zdrojových souborů získáme jednotlivé syntaktické stromy. Nyní začne první fáze překladu. V té se snažíme sémantický strom odpovídající jazyku Ruby transformovat na univerzální model, „meta AST“. Procházíme vstupní model a snažíme se z něho získat maximum informací.
- Tím získáme meta AST. Meta AST by nemělo být závislé na Ruby, ani Pythonu. Aktuálně to není zcela splněno, je nicméně snaha oddělit alespoň transformace specifické pro Ruby (1. fáze) a specifické pro Python (2. fáze).
- Druhá fáze přepisování modelu realizuje právě transformace specifické pro Python. Zatímco v první fázi informace extrahujeme, v této fázi již může dojít i k eliminaci informací, pokud nejsou pro generování Pythonu potřeba.
- Výsledkem je strom, který odpovídá výslednému zdrojovému kódu v Pythonu, tj. jeho následným parsováním bychom velmi teoreticky dostali znovu tento strom.
- Závěrečnou fází je generování. V ideální případě by mělo stačit projít výsledný model a generovat jednotlivé elementy. S tím, že šablony jednotlivých výsledných konstrukcí by již neměli obsahovat žádnou logiku, snad s výjimkou té úplně

[39] https://github.com/whitequark/parser

Obrázek 3: Schéma implementace



nejprostší. Aktuální implementace to ne zcela splňuje a bylo by dobré některé složitější šablony nahradit transformacemi, které by proběhly v druhé fázi.

- Výstupem jsou pak soubory v Pythonu tvořící balík (*package*) nebo modul (*modul*). Umístíme-li je na cestu, kde je uvidí interpretr Pythonu, pak by ho mělo jít z Pythonu běžně používat.

5.2 Formální popis transformací

Transformaci jednoho jazyka na druhý můžeme formálně popsat. Takovýto popis je užitečný pro pochopení reálné implementace, protože se v něm můžeme oprostit od implementačních detailů, které si pak realizace nutně vyžádá.

Vlastní transformace pracuje tak, že provádí konverzi abstraktního syntaktického stromu, kdy nahrazuje, přidává a odebírá jednotlivé uzly. Jak již bylo zmíněno, *celkový překlad (transformace)* pracuje ve dvou fázích:

1. Parser nám vrátí strom reprezentovaný jednotnými uzly (s výjimkou literálů, ale to můžeme zatím zanedbat).

2. V první fázi se provede transformace na obecný „metastrom“. V této fázi je cílem získat co nejvíce informací ze vstupního syntaktického stromu. Výsledkem je informačně bohatý strom nezávislý na výsledném (generovaném) jazyku.

3. Ve druhé fázi jsou pak prováděny modifikace, které vyžaduje cílový jazyk. Pokud bychom generovali pro více cílových jazyků, můžeme první fázi provést pouze jednou.

4. Výsledkem je strom, který odpovídá výslednému zdrojovému kódu, tj. jeho následným parsováním bychom dostali znovu tento strom.

Výsledkem aplikace *celkového překladu* je kompletní překlad zdrojového kódu z Ruby do Pythonu.

Definujme si použité pojmy.

Uzel n je obecně trojice prvků:

$$n = (Class, Children, Specific)$$

kde $Class$ je třída, jejíž je uzel instancí a $Children$ je n-tice *dětí* uzlu n, může být i prázdná. *Specific* je pak n-tice atributů, které jsou specifické pro danou třídu. Platí, že všechny uzly stejné třídy mají stejný počet a typ těchto atributů (liší se pochopitelně v jejich hodnotách). I tato množina může být prázdná.

Označmě N množinu všech možných uzlů, IN množimu všech možných vstupních uzlů (tj. těch, které získáme od parseru) a OUT množinu všech možných výstupních uzlů (tj. těch, které jsou předány generátoru pythonovského kódu).

Vstupní uzel n_{in} je pak trojice:

$$n_{in} = (\mathsf{Unprocessed}, Children, (Symbol))$$

tj., jde o instanci třídy Unprocessed, která navíc obsahuje *Symbol*.

Platí:

$$\forall n_i \in IN : Class(n_i) = \mathsf{Unprocessed}$$

$$\forall n_i \in OUT : Class(n_i) \neq \mathsf{Unprocessed}$$

jinak řečeno, všechny vstupní uzly před započetím transformací jsou třídy Unprocessed a všechny uzly po ukončení transformací jsou jiné třídy.

Sled příkazů je uzel n_{sp} pro který platí:

$$Class(n_{sp}) = \mathsf{StatementList}$$

Nazvěme *celkovou transformací* T množinu jednotlivých transformací t_i:

$$T = \{t_1, t_2, \ldots, t_n\}$$

kde t_i je uspořádaná trojice:

$$t_i = (D_i, f_i, g_i)$$

kde D_i je množina individuálních transformací, které musí proběhnout dříve než transformace t_i. D_i může být prázdná.

f_i je *transformační funkce*:

$$f_i : N \to N*$$

kde oborem hodnot je uspořádaná n-tice uzlů. Může být i prázdná.

g_i je *extrakční funkce*:

$$g_i : N \to N*$$

kde oborem hodnot je uspořádaná n-tice uzlů. Může být i prázdná.

Celková transformace T se pak provede vykonáním *jednotlivých transformací* t_i za dodržení těchto pravidel

- *Provedení jednotlivé transformace.* Provedením t_i se rozumí aplikováním f_i a g_i na všechny uzly stromu n_j a nahrazení uzlu n_j za uzly $f_i(n_j)$. Všimněme si, že

funkční hodnotou f_i je obecně množina uzlů, tj. může to být jeden uzel, více uzlů, či žádný. V tom případě je původní uzel odstraněn bez náhrady.

- *Extrakce příkazů.* Dále je vyhledán nejbližší rodič uzlu n_j, který je *Sled příkazů*, a mezi jeho děti jsou vloženy uzly $g_i(n_j)$. Tyto uzly jsou vloženy přímo před uzel, který je součástí podstromu obsahujícího n_j před provedením $t_i(n_j)$.

- *Implicitní transformace.* Protože se transformace t_i týká vždy jen určitých uzlů, je hodnotou příslušné transformační funkce pro ostatní uzly jednoprvková množina obsahující původní nezměněný uzel, tedy jde o identitu:

$$f_i(n_j) = \{n_j\}$$

$$g_i(n_j) = \{\}$$

- Vždy se provede stejná t_i pro všechny uzly stromu a teprve poté se provede další. Jinak řečeno, v každý okamžik transformace mohou být ve stromu pouze uzly, které jsou výsledkem transformace t_i nebo t_{i-1} (kde t_0 chápeme jako uzly před započetím transformování).

- Před provedením t_i jsou provedeny všechny transformace $t_j \in D_i$.

- Před provedením $f_i(n_j)$ je provedeno $f_i(n_k)$ pro všechny $n_k \in Children(n_j)$. Tj. před provedením transformace uzlu již musí být provedena stejná transformace všech jeho dětí. Transformace postupují tedy od listů ke kořeni (jde tedy o analogii prohledávání do hloubky, nikoli do šířky).

5.2.1 Neformální shrnutí

Celková transformace je realizována jako série mnoha malých *jednotlivých transformací*. Ty jsou po sobě aplikovány vždy na všechny uzly AST, od listů ke kořeni AST. Uzel je okamžitě nahrazen návratovou hodnotou *transformační funkce*, což je n-tice uzlů. Pokud je prázdná, dojde k odstranění uzlu. Protože za určitých okolností nestačí provést náhradu uzlů lokálně, například při přiřazení v podmínce příkazu `if`, je někdy potřeba přidat při překladu nové příkazy do lexikálně nejbližšího sledu příkazů (*statement list*).

Implementace

Aktuální implementace nemá explicitně vyjádřeno D_i, tj. závislosti, místo toho jsou závislosti dané pořadím, ve kterém se *jednotlivé transformace* provádějí.

5.2.2 Jednotlivé transformace t_i

Jednotlivé transformace popisuje předchozí kapitola „Porovnání vybraných konstrukcí jazyků Ruby a Python 3 a návrh mechanismu jejich překladu“. Její text obsahuje kromě diskuze i přímé odkazy na zdrojové soubory, které dané transformace implementují.

5.3 Omezení implementace

Zjevným, plánovaným omezením je realizace pouze podmnožiny jazyka Ruby.

Překladač také předpokládá, že překládaný kód tvoří knihovnu, tj. modul a případné vnořené moduly. Překladač zvládne situaci, kdy je na nejvyšší úrovni kód v různých modulech. Naopak není určen pro překlad samostatných skriptů, resp. kódu, který se nenachází v žádném modulu.

V případě nepodporovaných konstrukcí je cílem:

1. aby překladač buď oznámil již při překladu co nejspecifičtější chybu,
2. nebo alespoň běh výsledného programu skončil chybou (výjimkou) za běhu,
3. případně skončilo chybou již načtení přeloženého zdrojového souboru interpretrem Pyhonu.

První možnost je nejlepší, zbylé dvě jsou akceptovatelné. Za chybu překladače lze naopak považovat situaci, kdy Python kód provede bez vyvolání výjimky, ale ten se chová jinak, než se chová původní kód v Ruby (například vypočte jinou hodnotu).

5.4 Změny nutné v knihovně Prawn

I když překladač přeloží Prawn skoro beze změn, přece jen některé změny jsou vyžadovány.

- Za prvé, překladač si neporadí s konstrukcí:

Ruby

```
def metoda_a
end

def metoda_b(&block)
  metoda_a(&block)
end
```

V Ruby je legální předat metodě blok, i když ta ho nepoužije, tj. neobsahuje ani `yield`, ani jinou konstrukci pracující s blokem. V Pythonu se po překladu stane z bloku standardní argument, takže výsledný kód vypadá nějak takto:

Python

```
def metoda_a(self):
    pass

def metoda_b(self, block=None):
    metoda_a(block)
```

a v ten okamžik interpretr Pythonu vyhodí výjimku, protože metodě `metoda_a` předáváme o jeden parametr navíc, se kterým nepočítá. Řešení je možné dvojí:

  - Přidat parametr `&block` do metody `metoda_a`.
  - Odstranit parametr z volání metody `metoda_a`.

- Podpora pro *instanční proměnné objektu třídy* (viz kapitola o třídách, s. 53) není kompletní, momentálně to znamená na jednom místě nahradit:

  Ruby

  ```
  Repeater.count += 1
  ```

  za:

  Ruby

  ```
  Repeater.count = Repeater.count + 1
  ```

- Není implementováno `class_eval`, příslušný kód je přepsán ručně.
- Podpora vícevláknového zpracování zatím není implementována. Příslušný kód je přepsán.
- Také není implementována nahrada modulu `Matrix`, takže transformace založené na maticích aktuálně nejsou podporovány.

Implementace

Detailní popis změn (unifikované diffy) jsou v souboru `prawn-ruby/patches.txt`

6 Závěr

Cílem práce bylo porovnat možnosti jazyků Ruby a Python 3, vyvinout nástroj pro automatický překlad z Ruby do Pythonu a demonstrovat ho na překladu knihovny Prawn pro generování PDF souborů. To se ukázalo jako obří úkol, nicméně cílů se podařilo z větší části dosáhnout.

Přes polovinu této práce tvoří porovnání jednotlivých konstrukcí obou jazyků a možné varianty řešení jejich vzájemného převodu. I přes značný rozsah této části není jistě zcela vyčerpávající a bylo by možné na toto téma napsat ještě několikanásobně více. Přesto je popsána většina běžnějších konstrukcí, na které lze narazit při psaní „normálního“ kódu, tj. kódu, který se nesnaží využívat na maximum možnost metaprogramování.

Paralelně vznikl i překladač z Ruby do Pythonu 3. I přesto, že jsem si záměrně vybral jazyky, které k somě mají relativně blízko, to byl nesnadný úkol. Kromě očekávaných problémů plynoucích ze základních rozdílů mezi jazyky:

- měnitelné řetězce v Ruby a neměnitelné v Pythonu,
- neexistence přímého ekvivalentu bloků v Pythonu,
- při neuvedení explicitního `return` je návratovou hodnotou metody hodnota posledního vykonaného výrazu v Ruby, *None* v Pythonu,

se objevily i problémy překvapivější. Například fakt, že řetězce v Ruby mohou obsahovat libovolná binární data. Dále je to samozřejmě řada drobností, kdy se na první pohled zdánlivě podobné konstrukce chovají v detailech jinak. Například metoda `insert` pole vrací v Ruby toto pole, takže lze její volání řetězit, zatímco v Pythonu nevrací nic, a řetězit ji nelze. Nebo například:

```
"abc".split("")
```

rozdělí v Ruby řetězec na jednotlivá písmena, zatímco v Pythonu vyhodí výjimku *empty separator*.

Překladač vznikal paralelně s textem, tj. některé konstrukce jsem nejprve teoreticky popsal a pak implementoval, u jiných to bylo obráceně. Musím říci, že v těch případech, kdy jsem nejprve problém promyslel a popsal vznikal kvalitnější kód a také trvalejší. Tam, kde jsem se nejprve pustil do implementace jsem ji pak musel častěji upravovat, protože se objevila nějaká okrajová situace, se kterou jsem prve nepočítal. Proto jsem nakonec všechny situace nejprve aspoň v bodech popsal. Dodatečně jsem tak dal za pravdu Knuthovu *literárnímu programování* a podobným *documentation-first* přístupům. Dokonce si myslím, že kdybych nemusel psát i textovou část práce, že bych překladač asi ani nebyl schopen dotáhnout do použitelného stavu.

Také mi při vývoji hodně pomohlo grafické znázornění AST elementů, které tvoří přílohu A.

Co se týče knihovny Prawn, bylo dosaženo stavu, kdy do Pythonu přeložená verze dokáže generovat jednoduché PDF dokumenty, které mohou obsahovat obrázky. Zatím zdaleka nebyla otestována celá její funkčnost. Také její použití z Pythonu zatím není příliš přirozené, je nutné používat speciální třídu pro řetězce a podobně.

Jako překvapivý bonus jsem objevil pár drobných chyb v knihovně Prawn. Například třída `TTFunk::Table::Head` obsahuje atribut `checksum_adjustment`, ale příslušná proměnná má název `@check_sum_adjustment`, takže je interpretr Ruby nepropojí. Tyto chyby prověřím a případně pošlu autorům knihovny Prawn.

Seznam použitých zdrojů

FLANAGAN, David a MATSUMOTO, Yukihiro. 2008. *The Ruby programming language*. 1. vyd. O'Reilly. ISBN 978-0-596-51617-8.

FOWLER, Martin a PARSONS, Rebecca. 2011. *Domain-Specific Languages*. 1. vyd. Addison-Wesley. ISBN 03-217-1294-3.

HEROLD, Helmut. 2004. *Awk & sed: Příručka pro dávkové zpracování textu*. 1. vyd. Computer Press. ISBN 80-251-0309-9.

MOLHANEC, Michal. 2005. *Zvýrazňování syntaxe v XSLT*. Vysoká škola ekonomická. Vedoucí bakalářské práce: Ing. Jiří Kosek.

PARR, Terence. 2010. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. 1. vyd. The Pragmatic Programmers. Pragmatic Bookshelf. ISBN 19-343-5645-X.

PERROTTA, Paolo. 2011. *Metaprogramming Ruby: Program Like the Ruby Pros*. 1. vyd. The Pragmatic Programmers. Pragmatic Bookshelf. ISBN 19-343-5647-6.

PRECHELT, Lutz. 2000. *An empirical study of working speed differences between software engineers for various kinds of task*. Submission to IEEE Transactions on Software Engineering (nepublikováno). Dostupné také z: http://page.mi.fu-berlin.de/prechelt/Biblio/variance_tse2000.pdf

RAHIEN, Ayende. 2010. *DSLs in Boo: Domain-Specific Languages in .NET*. 1. vyd. Manning. ISBN 19-339-8860-6.

SHAUGHNESSY, Pat. 2014. *Ruby Under a Microscope: An Illustrated Guide to Ruby Internals*. 1. vyd. No Starch Press. ISBN 1-59327-527-7.

A Vstupní AST

Znázornění jednotlivých prvků Ruby AST na příkladech, tak jak ho generuje použitý parser.

A.1 Jednoduché literály

Příklad AST

```
nil
```

Příklad AST

```
true
```

Příklad AST

```
false
```


Příklad AST

```
self
```


Příklad AST

```
1
```

Příklad AST

```
1.1
```

Příklad AST

```
"a"
```

Příklad AST

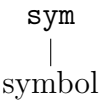
```
"aa"
```

Příklad AST

```
'aa'
```

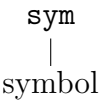
Příklad AST

```
:symbol
```



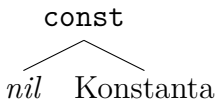
Příklad AST

```
:"symbol"
```



Příklad AST

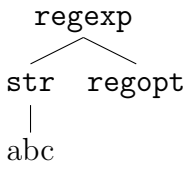
```
Konstanta
```



A.2 Regulární výrazy

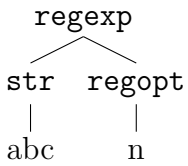
Příklad AST

```
/abc/
```



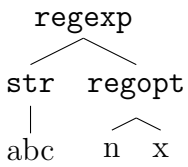
Příklad AST

```
/abc/n
```



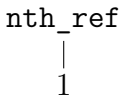
Příklad AST

```
/abc/nx
```



Příklad AST

```
$1
```



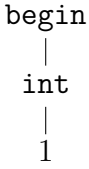
A.3 Závorky

Příklad AST

```
()
```

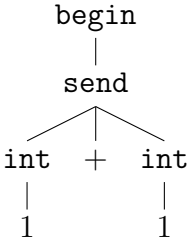
Příklad AST

```
(1)
```



Příklad AST

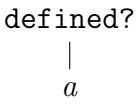
```
(1 + 1)
```



A.4 `defined?`

Příklad AST

```
defined? a
```



A.5 Operátory

Příklad AST

```
a and b
```

Příklad AST

```
a && b
```

Příklad AST

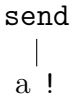
```
a or b
```

Příklad AST

```
a || b
```

Příklad AST

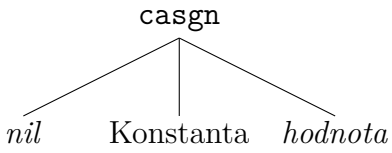
```
!a
```



A.6 Přiřazení konstantě

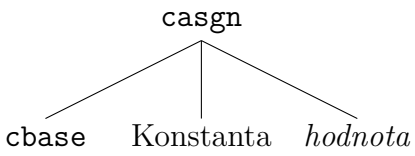
Příklad AST

```
Konstanta = hodnota
```



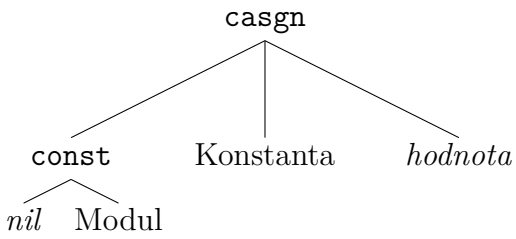
Příklad AST

```
::Konstanta = hodnota
```



Příklad AST

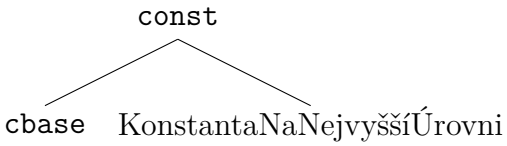
```
Modul::Konstanta = hodnota
```



A.7 Rozlišení názvů

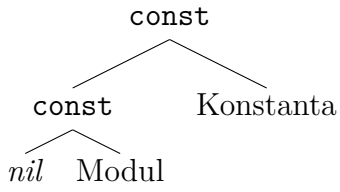
Příklad AST

```
::KonstantaNaNejvyššíÚrovni
```



Příklad AST

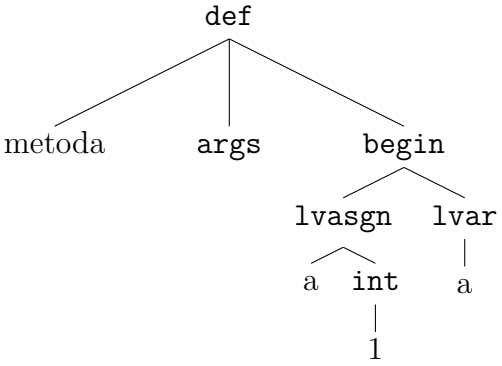
```
Modul::Konstanta
```



A.8 Proměnná

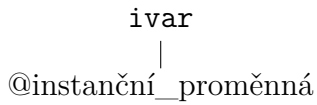
Příklad AST

```
def metoda
  a = 1
  a       # čtení lokální proměnné
end
```



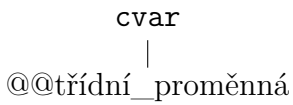
Příklad AST

```
@instanční_proměnná
```



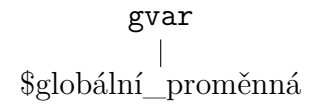
Příklad AST

```
@@třídní_proměnná
```



Příklad AST

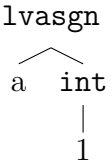
```
$globální_proměnná
```



A.9 Jednoduché přiřazení

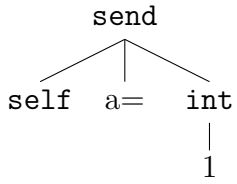
Příklad AST

```
a = 1       # lokální proměnná
```



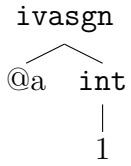
Příklad AST

```
self.a = 1  # instanční proměnná jako metoda self.a=(1)
```



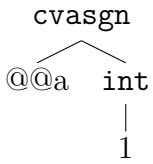
Příklad AST

```
@a = 1      # instanční proměnná
```



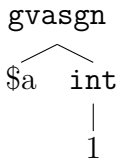
Příklad AST

```
@@a = 1     # třídní proměnná
```



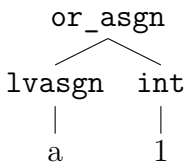
Příklad AST

```
$a = 1      # globální proměnná
```



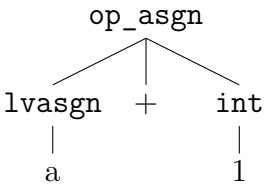
Příklad AST

```
a ||= 1     # kombinované přiřazení
```



Příklad AST

```
a += 1      # přiřazení s operátorem
```



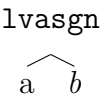
A.10 Vícenásobné přiřazení

masgn ... Multiple Assignment.
mlhs ... Multiple Assignment Left Hand Side, levá strana vícenásobného přiřazení.

Na začátku jsou pro porovnání AST podstromy jednoduchého přiřazení:

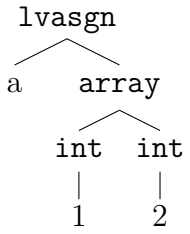
Příklad AST

```
a = b
```



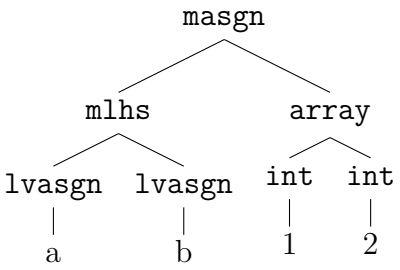
Příklad AST

```
a = [1, 2]
```



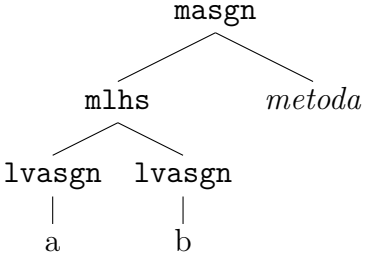
Příklad AST

```
a, b = 1, 2
```



Příklad AST

```
a, b = metoda
```



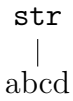
Uzly ve větvi `mlhs` jsou vlastně „nekompletní“ přiřazení. Zkompletují se dosazením příslušného prvku z pole na pravé straně. Základní pravidla pro vícenásobné přiřazení jsou stejná v Ruby i v Pythonu, takže je výhodné zachovat formu vícenásobného přiřazení. Alternativně bychom mohli nahradit vícenásobné přiřazení sérií jednoduchých, ale museli bychom použít dočasné proměnné, abychom byli schopni vyřešit situace jako je prohození proměnných na místě:

```
a, b = b, a
```

A.11 Řetězce a symboly s vloženými výrazy

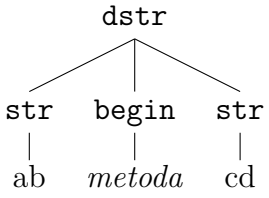
Příklad AST

```
"abcd"
```



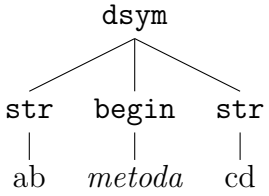
Příklad AST

```
"ab #{metoda} cd"
```



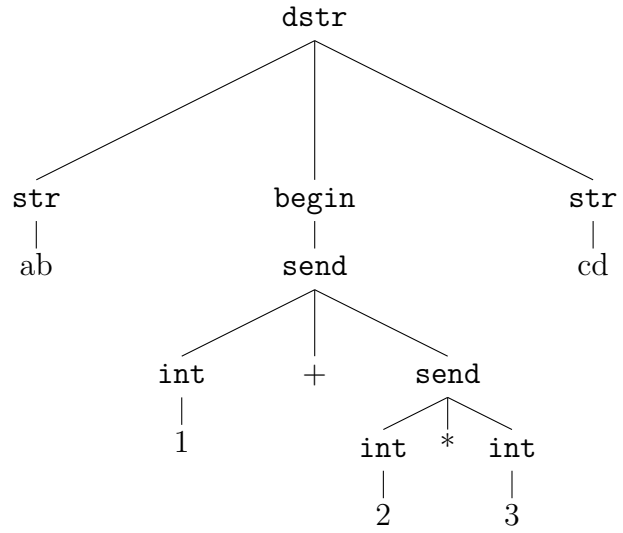
Příklad AST

```
:"ab #{metoda} cd"
```



Příklad AST

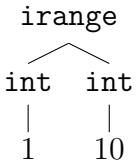
```
"ab #{1 + 2 * 3} cd"
```



A.12 Rozsahy

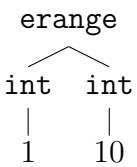
Příklad AST

```
1..10   # inkluzivní
```



Příklad AST

```
1...10  # exkluzivní
```



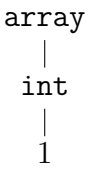
A.13 Pole

Příklad AST

```
[]
```

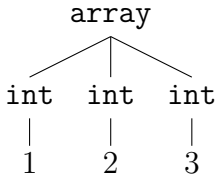
Příklad AST

```
[1]
```



Příklad AST

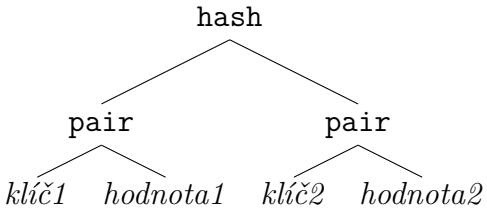
```
[1, 2, 3]
```



A.14 Slovníky

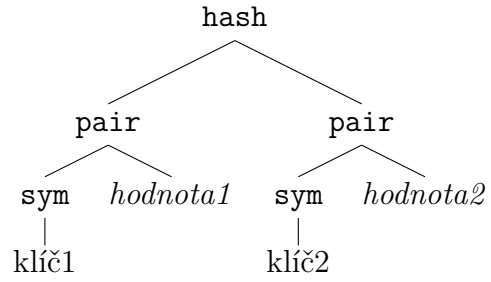
Příklad AST

```
{klíč1 => hodnota1, klíč2 => hodnota2}
```



Příklad AST

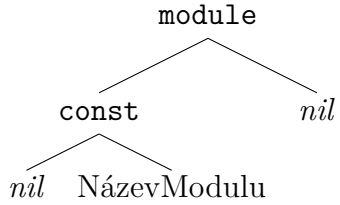
```
{klíč1: hodnota1, klíč2: hodnota2}
```



A.15 Moduly

Příklad AST

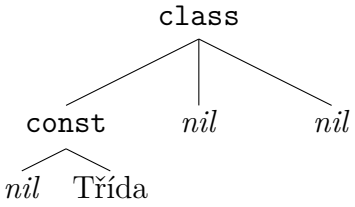
```
module NázevModulu
end
```



A.16 Třídy

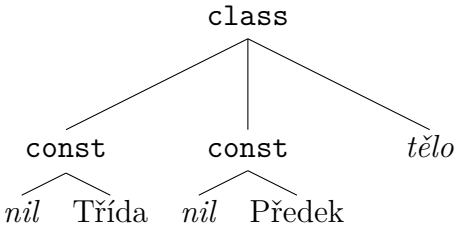
Příklad AST

```
class Třída
end
```



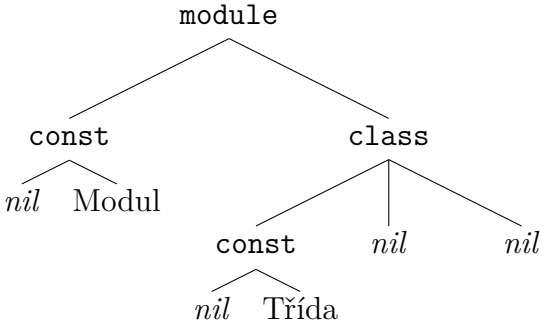
Příklad AST

```
class Třída < Předek
  tělo
end
```



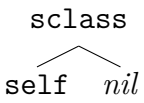
Příklad AST

```
module Modul
  class Třída
  end
end
```



Příklad AST

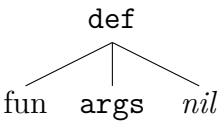
```
class << self
end
```



A.17 Deklarace metody

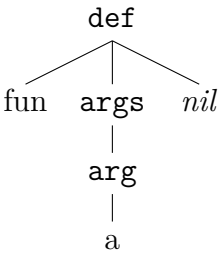
Příklad AST

```
def fun
end
```



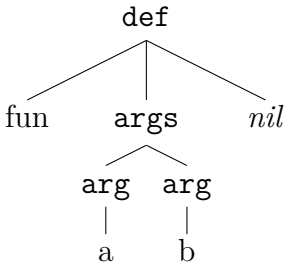
Příklad AST

```
def fun(a)
end
```



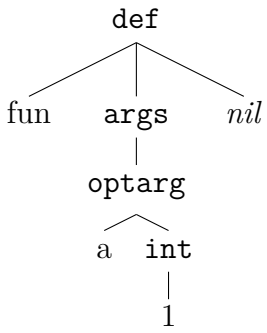
Příklad AST

```
def fun(a, b)
end
```



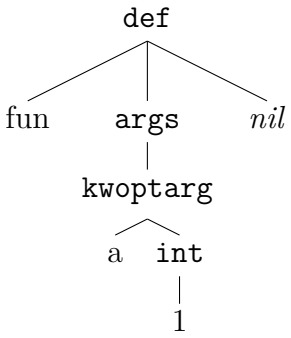
Příklad AST

```
def fun(a=1)
end
```



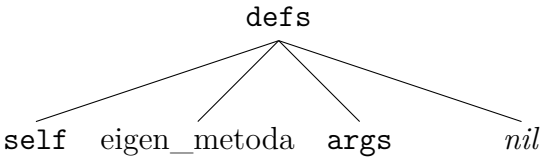
Příklad AST

```
def fun(a:1)
end
```



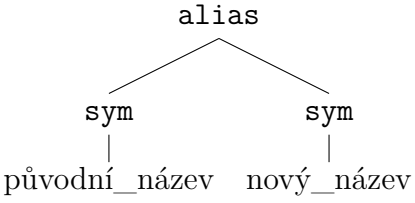
Příklad AST

```
def self.eigen_metoda
end
```



Příklad AST

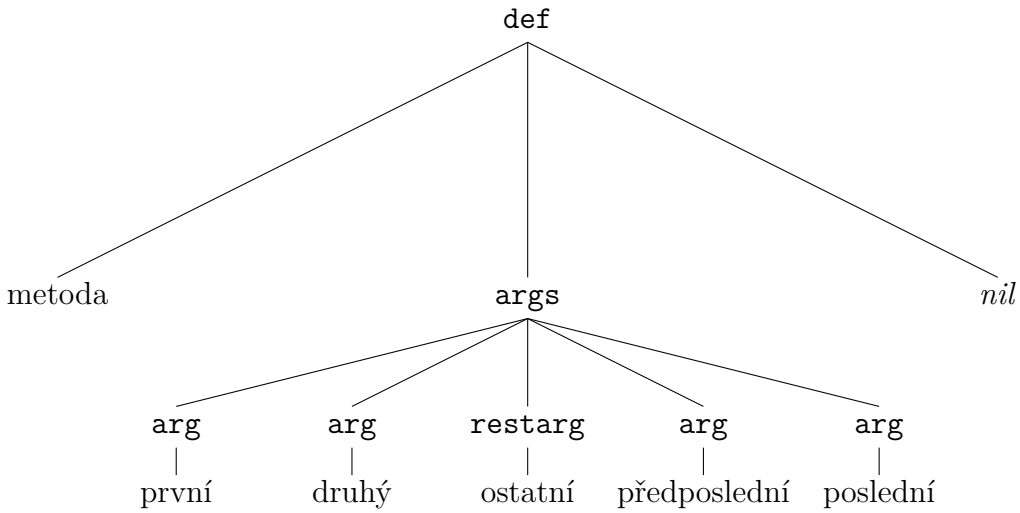
```
alias původní_název nový_název
```



A.18 Sběr parametrů

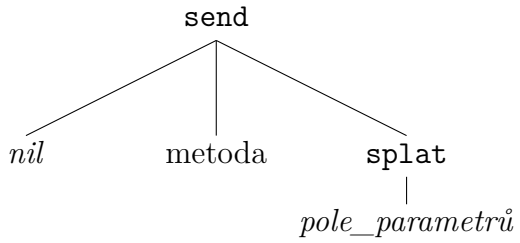
Příklad AST

```
def metoda(první, druhý, *ostatní, předposlední, poslední)
end
```



Příklad AST

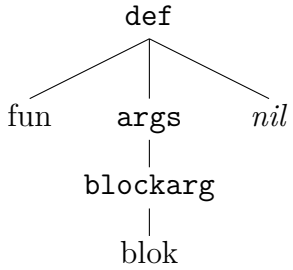
```
metoda(*pole_parametrů)
```



A.19 Parametr typu blok

Příklad AST

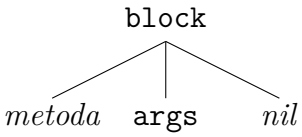
```
def fun(&blok)
end
```



A.20 Bloky

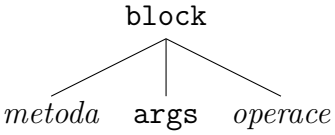
Příklad AST

```
metoda do
end
```



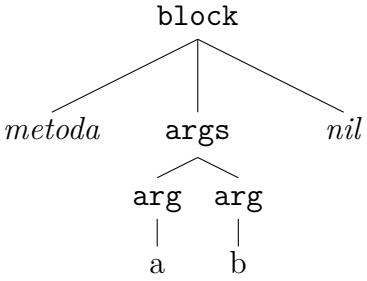
Příklad AST

```
metoda do
  operace
end
```



Příklad AST

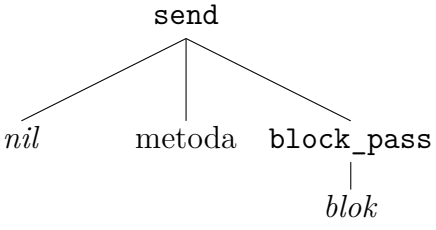
```
metoda do
  |a, b|
end
```



A.21 Předání bloku

Příklad AST

```
metoda(&blok)
```



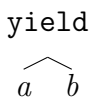
A.22 Volání bloku

Příklad AST

```
yield
```

Příklad AST

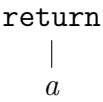
```
yield a, b
```



A.23 Return

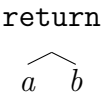
Příklad AST

```
return a
```



Příklad AST

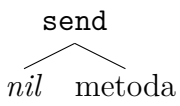
```
return a, b
```



A.24 Volání metody

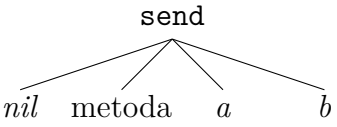
Příklad AST

```
metoda
```



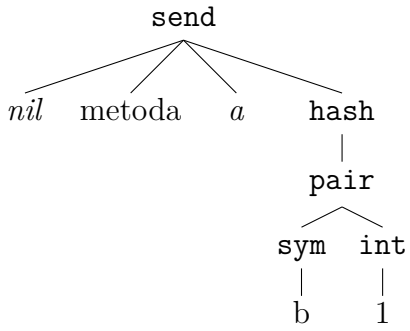
Příklad AST

```
metoda a, b
```



Příklad AST

```
metoda a, b:1
```



A.25 super

Příklad AST

```
super
```

```
zsuper
```

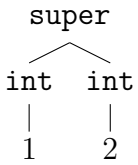
Příklad AST

```
super()
```

```
super
```

Příklad AST

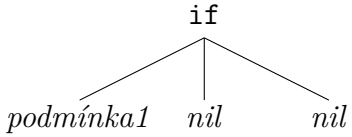
```
super(1, 2)
```



A.26 Podmínky

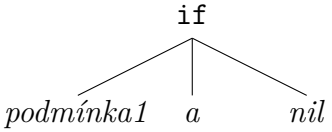
Příklad AST

```
if podmínka1
end
```



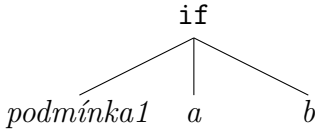
Příklad AST

```
if podmínka1
  a
end
```



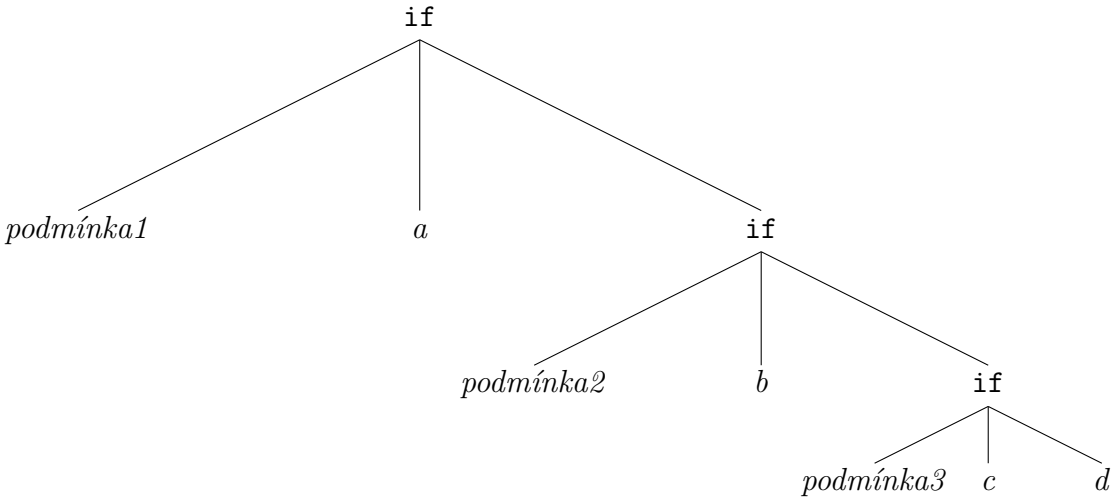
Příklad AST

```
if podmínka1
  a
else
  b
end
```



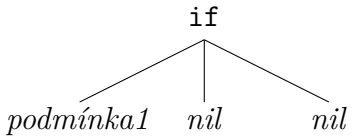
Příklad AST

```
if podmínka1
  a
elsif podmínka2
  b
elsif podmínka3
  c
else
  d
end
```



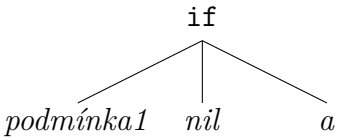
Příklad AST

```
unless podmínka1
end
```



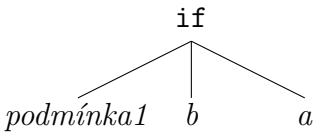
Příklad AST

```
unless podmínka1
  a
end
```



Příklad AST

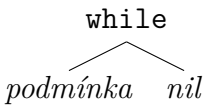
```
unless podmínka1
  a
else
  b
end
```



A.27 Cykly

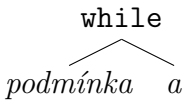
Příklad AST

```
while podmínka
end
```



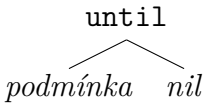
Příklad AST

```
while podmínka
  a
end
```



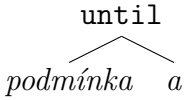
Příklad AST

```
until podmínka
end
```



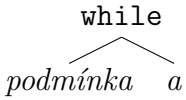
Příklad AST

```
until podmínka
  a
end
```



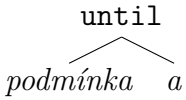
Příklad AST

```
a while podmínka
```



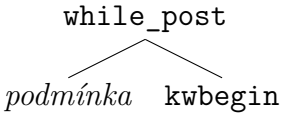
Příklad AST

```
a until podmínka
```



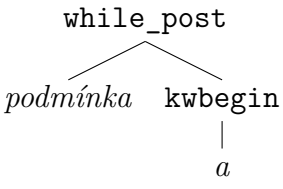
Příklad AST

```
begin
end while podmínka
```



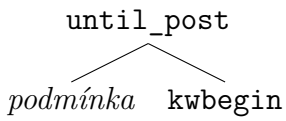
Příklad AST

```
begin
  a
end while podmínka
```



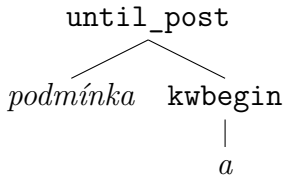
Příklad AST

```
begin
end until podmínka
```



Příklad AST

```
begin
  a
end until podmínka
```



A.28 break a next

Příklad AST

```
break
```

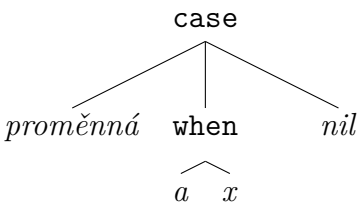
Příklad AST

```
next
```

A.29 Vícenásobný výběr

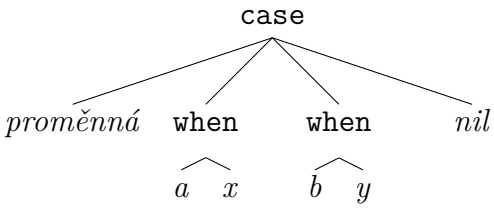
Příklad AST

```
case proměnná
  when a
    x
end
```



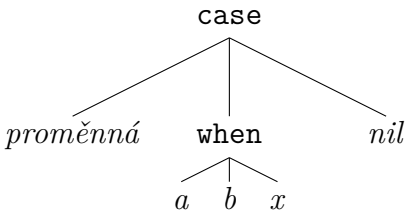
Příklad AST

```
case proměnná
  when a
    x
  when b
    y
end
```



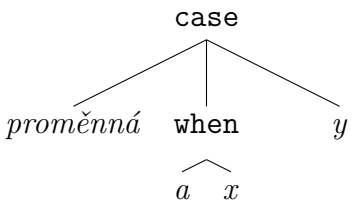
Příklad AST

```
case proměnná
  when a, b
    x
end
```



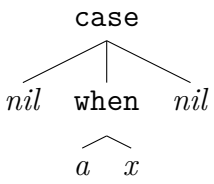
Příklad AST

```
case proměnná
  when a
    x
  else
    y
end
```



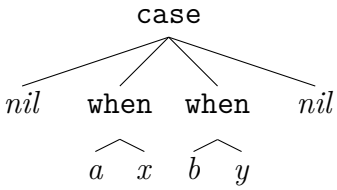
Příklad AST

```
case
  when a
    x
end
```



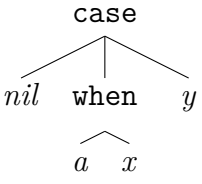
Příklad AST

```
case
  when a
    x
  when b
    y
end
```



Příklad AST

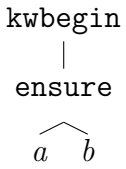
```
case
  when a
    x
  else
    y
end
```



A.30 Výjimky

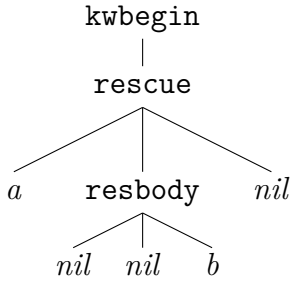
Příklad AST

```
begin
  a
ensure
  b
end
```



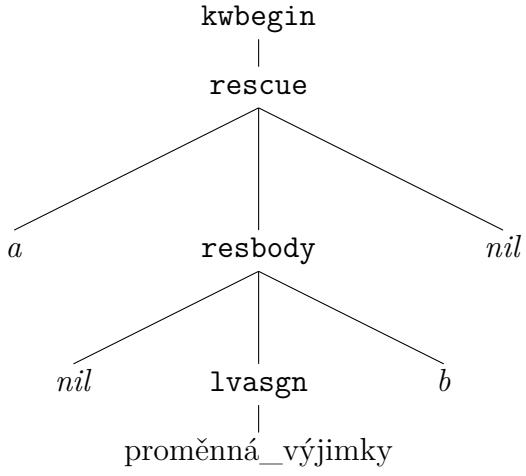
Příklad AST

```
begin
  a
rescue
  b
end
```



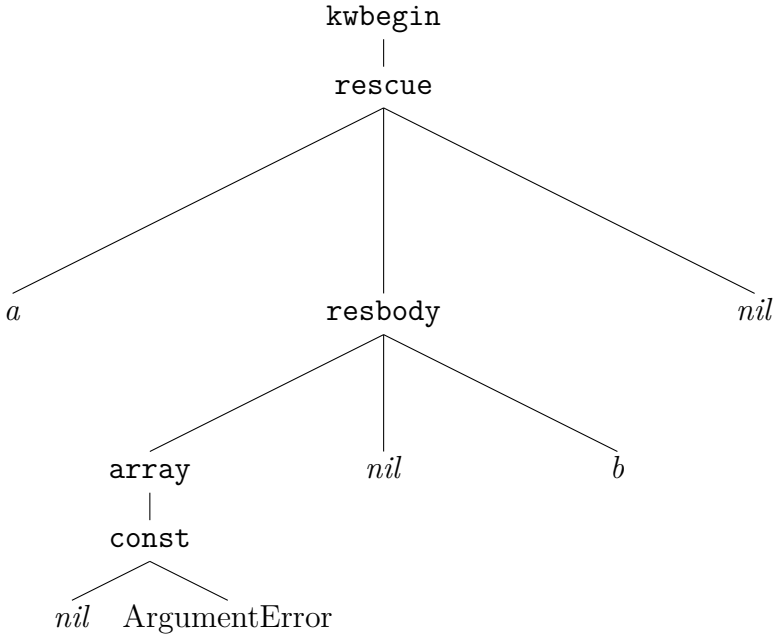
Příklad AST

```
begin
  a
rescue => proměnná_výjimky
  b
end
```



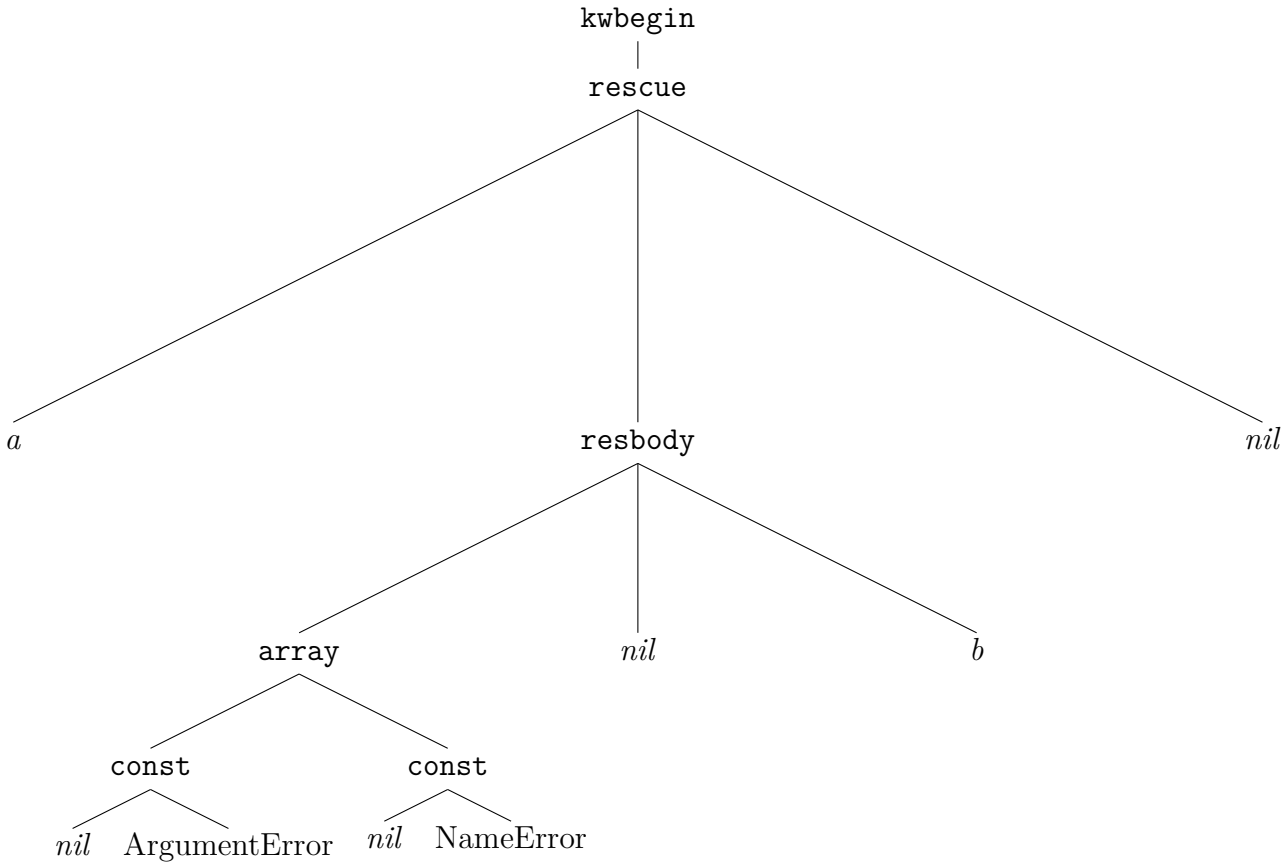
Příklad AST

```
begin
  a
rescue ArgumentError
  b
end
```



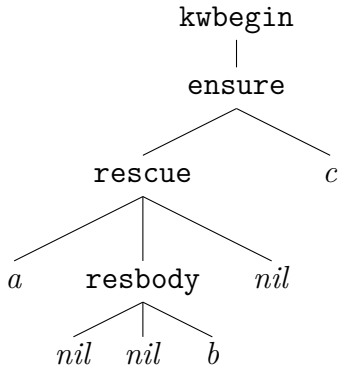
Příklad AST

```
begin
  a
rescue ArgumentError, NameError
  b
end
```



Příklad AST

```
begin
  a
rescue
  b
ensure
  c
end
```



A.31 Rejstřík uzlů vstupního AST

alias, Deklarace metody, s. 112
and, Operátory, s. 103
arg, Deklarace metody, s. 112
args, Deklarace metody, s. 112
array, Pole, s. 109
begin, Závorky, s. 102
block, Bloky, s. 115
block_pass, Předání bloku, s. 115
blockarg, Parametr typu blok, s. 114
break, break a next, s. 121
case, Vícenásobný výběr, s. 121
casgn, Přiřazení konstantě, s. 104
cbase, Rozlišení názvů, s. 104
class, Třídy, s. 111
const, Jednoduché literály, s. 100
cvar, Proměnná, s. 105
cvasgn, Jednoduché přiřazení, s. 105
def, Deklarace metody, s. 112
defined?, `defined?`, s. 103
defs, Deklarace metody, s. 112
dstr, Řetězce a symboly s vloženými výrazy, s. 108
dsym, Řetězce a symboly s vloženými výrazy, s. 108
ensure, Výjimky, s. 123
erange, Rozsahy, s. 109
false, Jednoduché literály, s. 100
float, Jednoduché literály, s. 100
gvar, Proměnná, s. 105
gvasgn, Jednoduché přiřazení, s. 105
hash, Slovníky, s. 110
if, Podmínky, s. 117
int, Jednoduché literály, s. 100
irange, Rozsahy, s. 109
ivar, Proměnná, s. 105
ivasgn, Jednoduché přiřazení, s. 105
kwbegin, Cykly, s. 119
kwoptag, Deklarace metody, s. 112
lvar, Proměnná, s. 105
lvasgn, Jednoduché přiřazení, s. 105
masgn, Vícenásobné přiřazení, s. 107
mlhs, Vícenásobné přiřazení, s. 107
module, Moduly, s. 110
next, break a next, s. 121
nil, Jednoduché literály, s. 100
nth_ref, Regulární výrazy, s. 101
op_asgn, Jednoduché přiřazení, s. 105
optarg, Deklarace metody, s. 112
or, Operátory, s. 103
or_asgn, Jednoduché přiřazení, s. 105
pair, Slovníky, s. 110
regexp, Regulární výrazy, s. 101
regopt, Regulární výrazy, s. 101
resbody, Výjimky, s. 123
rescue, Výjimky, s. 123
restarg, Sběr parametrů, s. 114
return, Return, s. 116
sclass, Třídy, s. 111
self, Jednoduché literály, s. 100
send, Volání metody, s. 116
splat, Sběr parametrů, s. 114
str, Jednoduché literály, s. 100
super, super, s. 117
sym, Jednoduché literály, s. 100
true, Jednoduché literály, s. 100
until, Cykly, s. 119
when, Vícenásobný výběr, s. 121
while, Cykly, s. 119
yield, Volání bloku, s. 116
zsuper, super, s. 117

B Obsah doprovodného CD

B.1 Struktura

```
-+-[diplomová práce]
 |  +- diplomová práce.pdf     Tato práce v PDF
 |  +- prezentace.pptx         Prezentace k této práci
 |
 +-[prawn-ruby]                Knihovna Prawn upravená dle kap. 5.4
 |  +- patches.txt             Podrobný popis úprav
 |
 +-[rb2py]                     Vytvořený překladač
 |
 +-[Python35]                  Interpretr Pythonu pro Windows
 +-[Ruby22]                    Interpretr Ruby pro Windows
 |
 +-hello.bat                   Dávkový soubor pro Windows, kterým
 |                             lze spustit příklad
 |
 +-prawn.rb                    Skript spouštěný z hello.bat, který
 |                             přeloží knihovnu Prawn
 |
 +-hello.py                    Skript spouštěný z hello.bat, který
 |                             použije do Pythonu přeloženou
 |                             knihovnu Prawn pro vytvoření PDF souboru
 |
 +-hello.jpg                   Datové soubory, které příklad používá
 +-Helvetica.afm
```

B.2 Jak spustit příklad

Pro prostředí Windows je připravena ukázka. Používá přiložené interpretry Pythonu a Ruby, takže je není potřeba mít nainstalované.

Nejprve je nutné zkopírovat celý obsah CD do složky, do které lze zapisovat. Skript na začátku kontroluje možnost zápisu do aktuální složky a v případě, že zapisovat nelze, upozorní na tuto skutečnost a skončí.

Dále přejdeme do složky, kam jsme soubory nakopírovali a spustíme soubor:

```
hello.bat
```

Pakliže je vše v pořádku, spustí se nejprve překlad knihovny Prawn, resp. nejprve jejích částí PDF::Core a TTFunk, a pak i samotné knihovny Prawn. Během překladu se vypisují různé informace a varování překladače. Po překladu každé části počká skript na stisknutí klávesy.

Výsledkem překladu je vytvoření složky `prawn-python`, která obsahuje přeloženou knihovnu Prawn.

Proběhl-li překlad v pořádku, spustí se skript `hello.py`, který by měl vytvořit nový soubor `hello.pdf` obsahující text „Hello“ a obrázek.

B.3 Jak spustit příklad s lokálně nainstalovanými interpretery

Máme-li nainstalovány na počítači Ruby 2.2 (nebo odpovídající JRuby) a Python 3.5, můžeme spustit skripty přímo. Tento návod předpokládá, že v příkazovém řádku jsou na systémové cestě (`PATH`) dostupné spustitelné soubory interpretrů (příkazy `ruby`, resp. `python`) a jejich balíčkovací systémy (`gem`, resp. `pip`).

Pro `prawn.rb` je potřeba knihovna `parser` ve verzi 2.3.0.1. Poslední verze této knihovny mají bohužel problémy s literály, které nejsou platné v UTF-8 a které se na jednom místě knihovny Prawn vyskytují.

Instalaci můžeme provést na příkazovém řádku:

```
gem install parser -v 2.3.0.1
```

a pak spustit překlad knihovny Prawn:

```
ruby prawn.rb
```

Pro `hello.py` je potřeba knihovna `tzlocal`. Instalaci můžeme provést na příkazovém řádku:

```
pip install tzlocal
```

Následně pak vygenerování PDF souboru:

```
python hello.py
```