Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Virtuální stroj pro Lisp v jazyku Scala

Bc. Adam Červenka

Vedoucí práce: Ing. Jiří Daněček

4. května 2015

Poděkování

Chtěl bych poděkovat panu Jiřímu Daněčkovi za konzultace a vedení této práce. Panu Marcelu Hlopkovi za rady ohledně jazyka Scheme a architektury virtuálního stroje. A v neposlední řadě bych chtěl poděkovat všem, kteří mi pomáhali s korekturou textu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 4. května 2015 .....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2015 Adam Červenka. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Červenka, Adam. *Virtuální stroj pro Lisp v jazyku Scala.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Práce se zabývá implementací virtuálního stroje napsaného v jazyce Scala, který interpretuje kód jazyka Scheme. Dále pojednává obecně o virtuálních strojích a čtenáři přiblíží jazyk Scheme a Scala.

Klíčová slova Scheme, Lisp, Scala, virtuální stroj, interpret

Abstract

The work deals with implementation of virtual machine written in Scala, which interprets code of the language Scheme. Further it discusses the matter of virtual machines in general and introduces languages Scala and Scheme to the reader.

Keywords Scheme, Lisp, Scala, virtual machine, interpreter

Obsah

Úvod **1**

1 Virtuální stroje **3**
1.1 Abstrakce 3
1.2 Architektura počítače 5
1.3 Virtuální stroje 6
1.4 Virtuální stroje pro jazyky vyšší úrovně 7
1.5 Virtuální stroj pro Pascal P-code 8
1.6 Virtuální stroje pro vyšší objektově orientované jazyky 10
1.7 Java Virtual Machine 12
1.8 Instrukční sada JVM 13
1.9 Common Language Infrastructure 16

2 Scheme **17**
2.1 Lisp 17
2.2 Lambda Kalkul 17
2.3 Rozšířené dialekty jazyka Lisp 18
2.4 Scheme 19
2.5 Sémantika 20
2.6 Syntaxe 21
2.7 Jmenné konvence 21
2.8 Lexikální konvence 21
2.9 Primitivní výrazy 22
2.10 Odvozené výrazy 24
2.11 Typy hodnot 26

3 Scala **31**
3.1 Třídy 32
3.2 Singleton a companion objekt 33
3.3 Trait 34

3.4 Case třídy 35
3.5 Pattern matching 36

4 Architektura **39**
4.1 REPL 39
4.2 Parser a Evaluator 40
4.3 Syntaktický strom 40
4.4 Typy a jejich hierarchie 41
4.5 Environment 43

5 Implementace **45**
5.1 Parser 45
5.2 Typy 47
5.3 Environment 47
5.4 Evaluator 48
5.5 Nativní procedury 49

6 Testy **53**
6.1 Vývoj řízený testy 53
6.2 Testování pomocí knihovny ScalaTest 53
6.3 Testy dané standardem jazyka Scheme 54

7 Možná rozšíření **55**

Závěr **57**

Literatura **59**

A Seznam použitých zkratek **61**

B Obsah přiloženého CD **63**

Seznam obrázků

1.1 Rozhraní mezi vrstvami abstrakce 5
1.2 Procesní a systémový virtuální stroj 7
1.3 P-code VM paměťová architektura 8

4.1 Hlavní smyčka programu 39
4.2 Struktura tříd na nejvyšší úrovni 40
4.3 Stará verze syntaktického stromu 41
4.4 Nová verze syntaktického stromu 42
4.5 Hierarchie typů 43
4.6 Enviromenty 44

Seznam tabulek

1.1 Příklady některých instrukcí P-code VM 10

5.1 Scala Parser - základní kombinátory 45
5.2 Scala Parser - pokročilé kombinátory 46

Úvod

Tato diplomová práce se zabývá implementací virtuálního stroje napsaného v jazyce Scala, který interpretuje kód jazyka Scheme. Ke zvolení tohoto tématu mě vedl v první řadě můj zájem o jazyky Scala, Lisp a funkcionální paradigma obecně. V druhé řadě pak mé zkušenosti s implementací virtuálního stroje pro jazyk Java. Mojí vnitřní motivací byla především touha porozumět těmto jazykům a způsobu jakým fungují.

Práce začíná kapitolou o virtuálních strojích, ve které je popsáno čím jsou, jak se dělí a na jakém principu tyto stroje fungují. Následující kapitola čtenáře stručně seznámí s jazykem Lisp, popíše jeho vztah k lambda kalkulu, nastíní jeho různé dialekty a především blíže popíše dialekt Scheme a jeho standard. Scheme byl zvolen coby jazyk, který bude implementovaný virtuální stroj schopen vykonávat. Kapitola Scala se zabývá základními vlastnostmi a konstrukty jazyka, ve kterém bude virtuální stroj napsán. Zejména odlišnostmi od Javy. Návrh je kapitola, která nastíní celkovou architekturu aplikace a její hlavní datové struktury. Implementace se bude zabývat tím, jak se program vypořádává s konkrétními problémy a jak se ve Scale povedlo konkrétní věci vyřešit. Testování bylo při vývoji velice důležité, a proto je v této kapitole popsáno, jak probíhalo a jak konkrétně některé testy vypadají.

Kapitola 1

Virtuální stroje

Tato kapitola popisuje virtuální stroje, jejich rozdělení a vlastnosti. Dále se zabývá příklady několika virtuálních strojů. Konkrétně to jsou P-code VM, JVM a CLR. Tato kapitola čerpá především z knihy Virtual machines [5], kde lze nalézt mnoho dalších informací o virtuálních strojích.

1.1 Abstrakce

Počítače jsou jedny z nejsložitějších systémů, které člověk kdy vytvořil. Komplexita celého počítače dalece přesahuje lidské schopnosti. Lidský mozek není schopen pojmout najednou tak obrovské množství informací. Vypořádáváme se s tím pomocí abstrakce. Začíná to elektrony ve vodiči, které si předávají elektrický náboj. Pro zjednodušení se na ně díváme jako na proud, který se přelévá vodičem tam a zpět. Pomocí tranzistoru můžeme sestavit elektronické obvody, ve kterých proud plynule teče pomocí nám známých zákonů. Vyjádřit logickou funkci je však pomocí takového obvodu obtížné. Pokud ale vytvoříme další úroveň abstrakce a z tranzistorů sestavíme logická hradla, kde na každém vodiči určité napětí představuje hodnotu jedna nebo nula, stane se návrh obvodu řešícího logickou formuli snadným. Procesor, nejdůležitější část počítače, je velmi složitý elektronický obvod. Navenek se však jeví jako černá skřínka a jediné, co je třeba pro komunikaci s ním, je jeho ISA (instrukční sada). ISA je na rozhraní mezi softwarem a hardwarem. Nad ní je umístěn Operační systém, který poskytuje uživateli příjemné rozhraní pro ovládání počítače a také knihovny pro usnadnění práce aplikačních programů.

Tento princip je využit skoro všude. Jeho výhodou je, že lidem umožňuje přemýšlet čistě na úrovni abstrakce, která je zajímá, a ostatní nebrat v potaz. Pokud píšeme program, nemusí nás zajímat, jaké napětí bude na vodičích procesoru, na jakém procesoru běží a často dokonce ani na jakém operačním systému. Druhou a také velmi důležitou výhodou tohoto principu je, že nejsme závislí na tom, co konkrétně je na nižší úrovni abstrakce. Tyto úrovně jsou odděleny rozhraním a pokud systém v nižší úrovni toto rozhraní dodržuje

a poskytuje, nemusí nás zajímat, jak interně řeší vykonávání našich příkazů. Program, který je napsán pro určité rozhraní, bude fungovat na každém stroji, který toto rozhraní splňuje. Dobře definované rozhraní umožňuje, aby například vývojáři Microsoftu programovali operační systém nad rozhraním, konkrétně instrukční sadou IA-32 a zároveň v Intelu takový procesor vyráběli bez toho, aby mezi sebou museli jakkoliv komunikovat. Podobné je to s operačním systémem a rozhraním, které poskytuje aplikačním programům.

Jako nevýhodu lze vidět to, že program napsaný pro jedno rozhraní nebude fungovat na druhém. Program napsaný pro Windows nebude fungovat pod Linuxem. Program běžící nad instrukční sadou IA-32 nebude běžet nad ARM. Obecně je rozmanitost v rozhraních dobrá, podporuje inovace a brání stagnaci. Snižuje však přenositelnost softwaru, což je omezující obzvláště v době internetu, kdy je výhodné používat software svobodně jako data.

Abstrakce nad pamětí a I/O (vstupy a výstupy) odstranila většinu závislostí na konkrétním hardwaru, ale některé závislosti stále zůstávají. Mnoho operačních systému je vyvinuto pro konkrétní systémovou architekturu. Například pro jeden procesor nebo pro vícejádrový procesor se sdílenou pamětí. Běžně se předpokládá, že hardwarové zdroje systému jsou využívány jen jedním operačním systémem. Toto váže všechny hardwarové zdroje k jednomu systému. Omezuje to flexibilitu, bezpečnost a izolaci chyb. Obzvláště pokud je systém využíván více uživateli.

Virtualizace umožňuje výše zmíněná omezení minimalizovat a zvýšit flexibilitu. Virtualizovaný systém se navenek jeví stejně jako ten reálný. Tedy splňuje domluvené rozhraní, ale uvnitř všechny zdroje a příkazy z rozhraní mapuje na opravdový systém pod ním. V tomto aspektu je virtualizace podobná abstrakci, obě implementují rozhraní nad sebou a využívají nějaké jiné rozhraní. Rozdíl je v tom, že virtualizace nemusí nutně skrývat detaily. Úrovně detailu virtuálního rozhraní a toho reálného, které se nakonec použije, mohou být stejné. [5]

Jako příklad virtualizace si lze představit pevný disk. Pokud si uživatel chce disk rozdělit na více částí, je to možné, a tyto části se opravdu budou navenek jevit jako disky dva. Rozhraní těchto disků však bude identické s diskem původním. Úroveň abstrakce zůstává stejná. Jedná se tedy jen o virtualizaci, nikoliv o abstrakci.

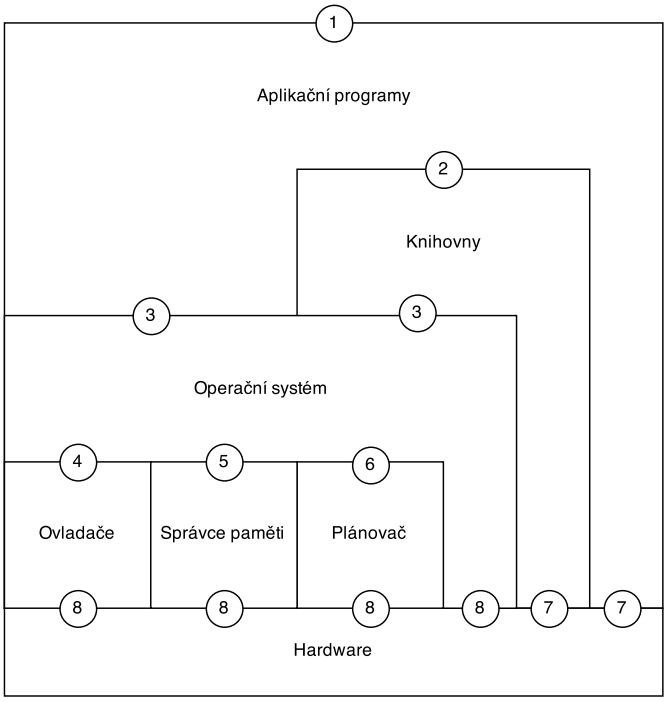
Virtualizace se nemusí omezovat jen na jednotlivou komponentu, jako je pevný disk, ale lze virtualizovat i celý stroj. Takový software se označuje pojmem virtuální stroj (VM - virtual machine). Takový virtuální stroj pak například umožňuje spuštění programu pro Windows, i když reálný operační systém je Mac.

Existuje mnoho druhů virtuálních strojů s mnoha účely. Virtuální stroj může běžet hned nad ISA a umožňovat tak běh dvou operačních systémů najednou. Může emulovat jednu ISA a ve skutečnosti používat úplně odlišnou. Virtuální stroj může být i aplikační program, který umožní spustit Linux přímo ve Windows. Vyšší programovací jazyky také často běží na nějakém vir-

tuálním stroji. Existuje virtuální stroj, jenž funguje pouze jako transparentní vrstva a provádí optimalizace kódu než se spustí na stroji opravdovém.

1.2 Architektura počítače

S virtuálními stroji souvisí architektura počítače. Jedna z věcí, která nás u nich nejvíce zajímá, je to, jak dobře implementují rozhraní dané právě architekturou. Architektura budovy určuje, jak se budova bude jevit jejím uživatelům a jaké možnosti jim poskytne. Nepopisuje už z jakých cihel bude postavena ani detaily elektrických rozvodů. Podobně i architektura počítačového systému popisuje, jak se systém jeví jeho uživatelům a jaké funkce jim poskytuje. Architektura už však neurčuje, jak mají tyto funkce být konkrétně implementovány. Každá architektura může mít mnoho implementací s rozdílnými charakteristikami. Například výkonnou implementaci a implementaci s nízkou spotřebou.



Obrázek 1.1: Rozhraní mezi vrstvami abstrakce

Na obrázku 1.2 je vidět typické rozdělení vrstev abstrakce. Každá z těchto vrstev má svoji architekturu a implementaci. Virtuální stroje se typicky vyskytují kolem hranice mezi hardwarem a softwarem, to jest ISA nebo výše. Dvě části ISA jsou důležité při specifikaci virtuálního stroje. První je část viditelná aplikačním programům (rozhraní č.7 na obrázku) neboli uživatelská

ISA. Druhá část je ISA, kterou používá operační systém ke správě zdrojů neboli systémová ISA. Ale operační systém může přistupovat k oběma (rozhraní č.8 na obrázku).

Další pro virtuální stroje důležitá rozhraní jsou ABI (application binary interface), které v sobě zahrnuje rozhraní 3 a 7 na obrázku, a API (application programming interface), které se skládá z rozhraní 2 a 7.

ABI je rozhraní, se kterým program komunikuje za běhu, tedy po tom, co je přeložen do strojového kódu. Definuje například, jak se předávají parametry funkcím nebo kde se bude nacházet návratová hodnota. Programy kompilované pro konkrétní ABI mohou běžet pouze na počítačích se stejnou ISA a operačním systémem.

API je rozhraní, se kterým se dostane do kontaktu i aplikační programátor. Patří sem standardní knihovny, které může program využít pro volání služeb operačního systému. Aplikaci napsanou pro určité API je možné zkompilovat pro jakýkoliv systém, který dané API implementuje.

1.3 Virtuální stroje

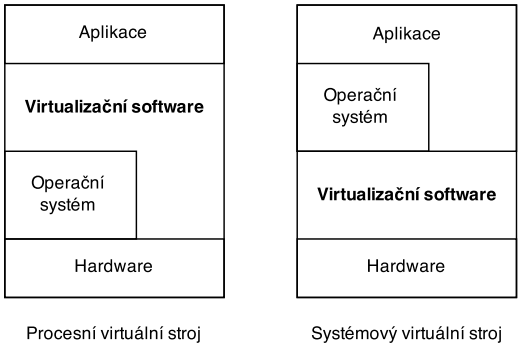
Stroj může v tomto kontextu znamenat více věcí. Pro proces vykonávající nějaký program je to jemu přidělená paměť spolu s registry a instrukcemi, které může provádět. Proces nemůže ke vstupům a výstupům přistupovat jinak než pomocí systému. Většinou proto používá nějaké knihovny, jejichž kód se spustí v rámci jeho vykonávání. Rozhraní mezi procesem a strojem představuje ABI.

Z pohledu operačního systému je celý systém spuštěn na stroji. Paměť, se kterou OS pracuje, je mu přidělena trvaleji a zatímco procesy začínají a končí, operační systém k ní má stále přístup. Rozhraním mezi operačním systémem a strojem je ISA.

Virtuální stroj je pak nejen software vykonávající virtualizaci, ale i skutečný hostitelský stroj, na kterém virtuální stroj běží. Samotný spuštěný proces/systém samozřejmě o hostitelském systému nic neví. Podle toho, jak rozdílně se jeví stroje procesu a operačnímu systému, lze rozdělit i virtuální stroje na procesní virtuální stroje a systémové virtuální stroje. [5]

Procesní virtuální stroj je často nazýván runtime software nebo jen runtime. Takový VM je spuštěn spolu s daným procesem, poskytuje mu prostředky pro komunikaci s I/O zařízeními a hardwarem celkově a když proces skončí, je ukončen i VM.

Naproti tomu systémový virtuální stroj poskytuje celé prostředí, na které může být nainstalován operační systém a běžet velmi dlouho. Na obrázku 1.2 je vidět systémový virtuální stroj, který emuluje jednu ISA na jiné. Jeho virtualizačnímu softwaru se také někdy říká VMM (Virtual Machine Monitor).



Obrázek 1.2: Rozdělení virtuálních strojů

1.4 Virtuální stroje pro jazyky vyšší úrovně

Virtuální stroje pro jazyky vyšší úrovně, zkráceně HLL VM, jsou procesní virtuální stroje, které slouží jako vrstva mezi operačním systémem a daným jazykem. Umožňují programům, aby nebyly závislé na konkrétním operačním systému a ISA. Bez virtuálního stroje by se program musel v lepším případě pro každý OS a ISA znovu zkompilovat. V horším případě by se musely přepisovat i knihovny nebo volání OS. Tyto stroje zpravidla nemají emulovat nějakou konkrétní architekturu, ale jsou navrženy tak, aby co nejlépe odpovídaly potřebám vyšších (často objektových) jazyků. Nicméně pro některé existují i jejich hardwarové implementace. Knihovny těchto strojů poskytují vyšší stupeň abstrakce než klasické systémové knihovny.

Ve zbytku tohoto textu, pokud zmíním virtuální stroj bez dalšího přívlastku, budu mít na mysli vždy procesní virtuální stroj tohoto typu.

Protože cílem VM je nezávislost na ISA, musí mít VM svoji vlastní virtuální instrukční sadu (V-ISA). Programy se při kompilaci přeloží do V-ISA a pak proběhne jejich distribuce. Na každém fyzickém stroji je pak virtuální stroj implementující danou V-ISA. Při interpretaci programu přeloží, respektive interpretuje, instrukce z V-ISA na instrukce ISA cílového počítače.

V-ISA neobsahuje pouze instrukce, ale i množství datových struktur a metadat, samotné instrukce už jsou většinou jednoduché.

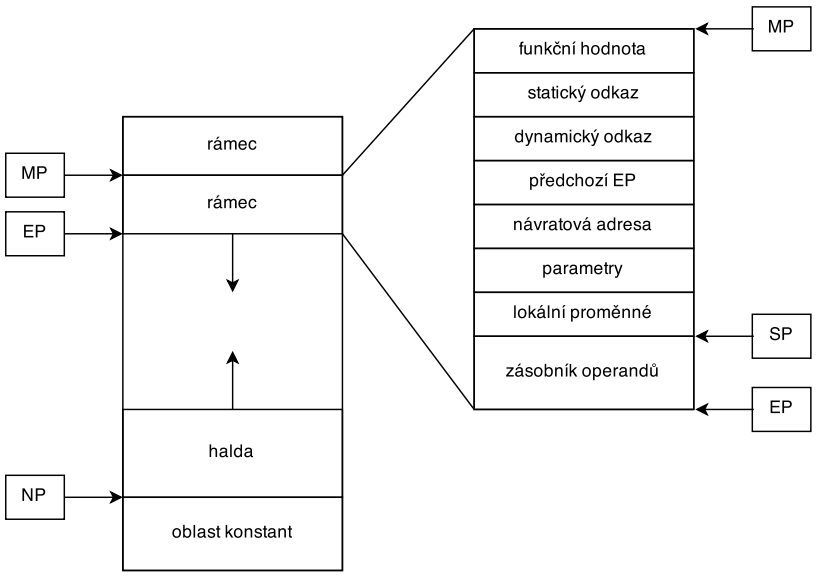
Na následujících stránkách bych rád popsal tři nejvýznamnější virtuální stroje. Prvním je virtuální stroj pro jazyk Pascal, který byl průkopníkem v této oblasti. Další je pak JVM (Java Virtual Machine), jehož implementaci jsem si sám vyzkoušel, a který je v současné době asi nejvýznamnějším virtuálním strojem. Posledním je CLI (Common Language Infrastructure) od Microsoftu, největší konkurent JVM. JVM a CLI jsou v mnoha ohledech velmi podobné virtuální stroje, proto se u CLI zaměřím především na to, v čem se od JVM liší.

1.5 Virtuální stroj pro Pascal P-code

P-code VM se zasloužil o rozšíření jazyka Pascal, protože umožnil snadnou přenositelnost kompilátoru Pascalu. Většinu základních konceptů tohoto VM převzaly i JVM a CLI. Základní myšlenka je taková: Kompilátor zkompiluje program v Pascalu a vygeneruje program v P-code. P-code je jednoduchá instrukční sada využívající zásobník. Program v P-code už se pak může pustit na P-code VM. Jediné, co je potřeba udělat, aby program v Pascalu mohl běžet všude, je implementace P-code VM pro každou požadovanou platformu.

Druhou velmi důležitou částí jsou standardní knihovny, které se starají o komunikaci s hostitelským strojem. Jedná se především o funkce pro čtení a zápis na standardní výstup nebo do souboru. Tyto nativní funkce jsou podporovány přímo virtuálním strojem.

Paměťová architektura P-code V-ISA se skládá z těchto částí: paměť pro instrukce programu, paměť pro konstanty (constant area), zásobník (frame stack) a halda pro alokaci dat. K paměti instrukcí se přistupuje pouze pomocí čítače instrukcí (PC) a je možné jen její čtení, nikoliv zápis. Konstanty jsou vygenerovány překladačem. Zásobník obsahuje rámce. Každý rámec odpovídá jednomu volání funkce. Zásobník pro operandy (operand stack) je součástí rámce. Halda a zásobník rostou v paměti proti sobě a pokud se potkají, vyvolá to výjimku.



Obrázek 1.3: P-code VM paměťová architektura

Rámec se skládá z následujících položek:

funkční hodnota návratová hodnota funkce, pokud něco vrací

statický odkaz Pascal umožňuje vnořování funkcí a vnitřní funkce vidí proměnné v jejich rodičích, k jejich nalezení slouží tato položka, která ukazuje na rodičovskou funkci

dynamický odkaz ukazuje na předchozí rámec, neboli hodnota MP pro předchozí rámec

předchozí EP jak je z názvu zřejmé, hodnota EP pro předchozí rámec

návratová adresa hodnota, na kterou by se měl nastavit PC při návratu z funkce

parametry parametry funkce

lokální proměnné obsahuje všechny lokální proměnné

zásobník operandů sloužící k vykonávání instrukcí p-code

P-code VM používá tyto registry:

- PC program counter - ukazuje na aktuálně vykonávanou instrukci
- SP stack pointer - ukazuje na vrchol zásobníku operandů
- MP mark stack pointer - ukazuje vrchol zásobníku rámců
- NP new pointer - vrchol haldy, zde se alokuje paměť pro nové proměnné
- EP extreme stack pointer - ukazuje konec aktuálního rámce

Maximální velikost zásobníku operandů je známá již za kompilace, tudíž je jasné, jak velký rámec musí být. Na jeho konec ukazuje EP. SP ukazuje na aktuální operand na vrchu zásobníku. Růst haldy se kontroluje vůči EP, není tedy potřeba kontrolovat kolize s haldou pokaždé, když se zvýší SP.

Instrukční sada se skládá z instrukcí, které mohou přidávat a odebírat operandy z vrcholu zásobníku, matematických instrukcí, logických instrukcí a dalších. Instrukce jsou typované, což znamená, že pro sčítání hodnot typu integer je potřeba použít instrukci adi, ale pro sčítání hodnot typu real je instrukce adr.

P-code VM se stal standardem pro ostatní virtuální stroje. Sdílejí spolu především tyto vlastnosti:

- Zásobníkově orientovaná instrukční sada. Je snazší pro ni napsat kompilátory a virtuální stroje. Zásobníková ISA také znamená menší binární programy.
- Paměť je rozdělena do buněk, ale velikost těchto buněk je záležitostí implementace, nikoliv ISA.

Tabulka 1.1: Příklady některých instrukcí P-code VM

Instrukce	Zásobník před	Zásobník po	Popis
adi	i1, i2	i1 + i2	sečte dvě hodnoty typu integer
adr	r1, r2	r1 + r2	sečte dvě hodnoty typu real
dvi	i1 i2	i1 / i2	dělení hodnot typu integer
inn	i1 s1	b1	přítomnost v množině
ldci	i1	i1	načte konstantu typu integer
not	b1	~b1	negace

- Paměť je rozdělena na zásobník a haldu, ale jejich rozsah není dán architekturou. Instrukce nikdy nepoužívají přímé adresy do paměti. Díky tomu je velikost paměti záležitostí implementace.

- Přístup k operačnímu systému je možný jen skrze standardní knihovny. Díky tomu by měl program být na operačním systému nezávislý. Na druhou stranu pak může být použita jen sada funkcí, kterou podporují všechny operační systémy.

1.6 Virtuální stroje pro vyšší objektově orientované jazyky

Model P-code VM usnadňující přenositelnost se osvědčil a moderní virtuální stroje na něm staví, ale kladou ještě další požadavky na to, co by měl virtuální stroj umět. Jedná se hlavně o bezpečnost a ochranu, podporu objektového paradigmatu, podporu internetu a výkon.

Terminologie není úplně jednoznačná co se týče rozlišení mezi implementací a definicí architektury virtuálních strojů. Java Virtual Machine mezi nimi nerozlišuje, termín JVM tak může být použit pro formát ClassFile souborů a instrukcí, stejně tak pro konkrétní implementaci, na které mohou být spuštěny. Microsoft common language infrastructure je architektura nebo funkční specifikace. Microsoft common language runtime (CLR) je konkrétní implementace vyvinutá Microsoftem. Pokud chceme mluvit jen o instrukční sadě Javy, nazývá se Java bajtkód, protože každý jeho opcode má velikost jednoho bajtu. Instrukční sada CLI se nazývá buď Microsoft intermediate language (MSIL), common intermediate language (CIL), nebo jen IL. Implementaci JVM spolu se standardními Java knihovnami (API) se říká java platforma. CLI implementace se standardními knihovnami se nazývá Microsoft NET framework. [5]

Požadavek bezpečnosti na virtuální stroj znamená, že program běžící na VM by neměl mít možnost přistupovat k jiným než jemu určeným datům. Samotný VM by také měl být chráněn před programem, který vykonává. Běžící program nesmí mít možnost jakkoliv zasahovat do vnitřních struktur virtuálního stroje. Některé populární jazyky vyšší úrovně jsou silně typované a

program musí přesně definovat rozsah, v jakém bude přistupovat k datům a jak bude kontrolovat tok programu. Tato informace je předána virtuálnímu stroji skrze V-ISA.

Přeložený Java program se skládá z binárních souborů, které mohou být uloženy lokálně nebo na vzdáleném úložišti. Každý takový soubor obsahuje na platformě nezávislý kód a metadata. VM obsahuje zavaděč (loader), který binární soubory umí načíst a ověřit, že jsou korektní a přeložit je do na platformě závislého formátu v paměti virtuálního stroje. V programu mohou být metody uživatelské, systémové a nativní. Uživatelské metody definuje uživatelský program. Systémové a nativní metody jsou součástí standardní knihovny. Loader a standardní knihovna jsou považovány za důvěryhodné. Nativní metody se používají především k přístupu k systému. Uživatel také může nadefinovat svoji nativní metodu, ale na vlastní nebezpečí, protože takové metody neprocházejí před spuštěním verifikací.

Další důvěryhodnou částí aplikace je emulátor, který emuluje vykonávání instrukcí V-ISA. Pokud by program chtěl provést nějakou potenciálně nebezpečnou akci, třeba přistoupit k souboru, standardní knihovna nejdříve požádá o povolení security manager, který rozhodne, jestli na takovou akci má program právo. Je důležité ověřit, že program nikdy neskočí mimo jemu přidělenou oblast paměti a stejně tak, že čtení a zápis se provede vždy korektně bez možnosti zásahu do jiných oblastí. JVM i CLI jsou silně typované a spoléhají ve velkém na kontrolu při zavádění. Díky tomu se za běhu musí provádět jen minimum kontrol.

Podporu objektově orientovaných jazyků tu nechci příliš rozebírat, toto paradigma je tak rozšířené, že předpokládám u čtenáře jeho znalost nebo schopnost si tyto informace snadno doplnit. Přesto zmíním, že jde především o podporu objektů, dědičnosti, polymorfizmu a skrývání dat.

Garbage collection je další součást moderních VM. Garbage collector (GC) se stará o to, aby objekty, které již nejsou dále používány, uvolnily místo v paměti pro objekty nové. Není to zadarmo a stojí to určitou část výkonu, ale díky tomu už nemohou nastat chyby, které se vyskytují při manuální správě paměti. Jde o čtení již smazaného objektu nebo o uvolnění paměti, která již jednou uvolněna byla. Jsou ale i situace, kdy je automatické uvolňování paměti rychlejší než manuální uvolňování paměti pro každý objekt zvlášť.

Protože binární soubory, části programu, mohou být umístěny na rozdílných místech na síti, snaží se moderní VM omezit počet dat, která musí být přenesena. Díky zásobníkové architektuře jsou soubory s kódem menší. Také se využívá lazy zavádění - binární soubory programu jsou načtené až ve chvíli, kdy je jich potřeba.

Poslední důležitá vlastnost moderních VM je snaha o co největší výkon. Je to velmi problematické, protože většina předchozích vlastností jde proti tomuto požadavku. Avšak existuje mnoho možností jak výkon zlepšit, jako jsou úpravy instrukční sady, překlad částí kódu do nativního jazyka a další pokročilejší metody.

1.7 Java Virtual Machine

1.7.1 Datové typy

Specifikace JVM stejně jako jazyk Java dělí datové typy na dvě skupiny: primitivní datové typy a reference. Reference může ukazovat na objekt nebo pole. Zatímco objekt se skládá z primitivních datových typů nebo referencí, pole je obsahuje. Primitivní datové typy v JVM jsou: byte, short, int, long, char, float, double, boolean a returnAddress. Tyto typy nejsou definované počtem bitů, který zabírají, ale rozsahem hodnot, kterých mohou nabývat. To dává implementaci svobodu v tom jak je bude ukládat. Referencí jsou tři druhy: reference na třídu, reference na rozhraní a reference na pole. Reference buďto ukazuje na nějaký objekt, anebo má hodnotu null. Jak přesně má být reference v JVM reprezentována, není určeno.

1.7.2 Oblasti s daty

PC Stejně jakou u P-code VM ukazuje na právě vykonávanou instrukci.

JVM zásobník (Stack) Zásobník, který obsahuje rámce. K zásobníku není přestupováno přímo, a proto mohou být rámce uloženy i na haldě. Rámec je na zásobník vložen v okamžiku, kdy je volána funkce. V okamžiku návratu z funkce je z něj rámec naopak odebrán. Návratová hodnota funkce je poté vložena na zásobník operandů předchozího rámce.

Halda (Heap) Zde jsou alokovány všechny instance tříd a pole. Halda si musí automaticky spravovat a uvolňovat paměť, neboli musí být použit nějaký garbage collector. Není specifikováno jaký typ Gc má být použit, ani jak budou data přesně uložena v paměti.

Oblast metod (Method Area) Navzdory svému jménu jsou zde uložena data podle Javovských tříd. Oblast metod může být součástí haldy, ale nemusí. Ke každé třídě jsou zde uloženy veškeré informace včetně bajtkódů jejich metod. Třídy se načítají až ve chvíli, kdy jsou potřeba.

Constant pool Obsahuje konstanty pro třídu nebo rozhraní. Jsou to jak primitivní typy, tak i hodnoty typu String.

1.7.3 Rámec

Nový rámec je vložen na zásobník pokaždé, když je volána metoda a je ze zásobníku odebrán, když vykonávání metody skončí. Každý rámec má svoje vlastní pole s lokálními proměnnými, vlastní zásobník operandů a referenci na identifikátor třídy. Tento identifikátor je uložen na constant poolu a identifikuje třídu, s jejíž metodou je rámec svázán. Velikost zásobníku operandů a počet proměnných v constant poolu jsou vypočítány při kompilaci.

Lokální proměnné Není dáno kolik musí jedna lokální proměnná zabírat bajtů, ale představuje buňku, do které se musí vejít všechny datové typy, kromě typů double a long. Ty se musí vejít do dvou takových buněk.

Zásobník operandů Je jedinečný pro každý rámec a po vytvoření rámce je prázdný. Je to paměť, se kterou operuje většina instrukcí. Protože je JVM zásobníkový počítač, většina instrukcí načítá své operandy přímo z tohoto zásobníku. Existuje tak řada instrukcí, které na něj umí přidat lokální proměnnou nebo konstantu. A instrukce pro sčítání nebo pro volání dalších funkcí očekávají, že pro ně budou na zásobníku přichystané parametry.

Do každé buňky zásobníku operandů se musí vejít kterýkoliv podporovaný typ včetně typů long a double. Instrukce jsou typovány a není například možné vložit na zásobník dvakrát int a pak zavolat instrukci určenou pro long.

1.8 Instrukční sada JVM

Instrukce JVM je opcode velikosti jednoho bajtu a za ním následuje nula nebo více operandů. Tyto operandy představují argumenty nebo data pro instrukci. Při vykonávání metody pracuje JVM následujícím způsobem: Nejprve spočte novou hodnotu PC a načte opcode instrukce z pozice dané PC. Pokud daná instrukce má další operandy, JVM je načte. Poté je instrukce vykonána. Pokud JVM nedošla na konec bajtkódu, celý cyklus se opakuje od začátku. V opačném případě byla provedena celá metoda.

Drtivá většina instrukcí je typovaná a může pracovat jen s daným typem. Pro typy boolean, char, byte a short však platí, že na zásobníku operandů se rozšíří na int. Díky tomu se počet instrukcí drží na rozumné úrovni. Při ukládání do paměti jsou hodnoty přetypovány zpět na původní velikost. Pole bajtů tak pořád lze uložit o poznání úsporněji než pole hodnot typu int.

1.8.1 Instrukce pro načítání a ukládání dat

Tyto instrukce slouží pro přenos dat mezi lokálními proměnnými a zásobníkem operandů uvnitř rámce. Lze je rozdělit do několika skupin:

- Vložení lokální proměnné na zásobník operandů
- Uložení hodnoty ze zásobníku operandů do lokální proměnné
- Vložení konstanty na zásobník operandů
- Získání přístupu k více lokálním proměnným pomocí rozšíření indexu

Kromě těchto instrukcí mohou přidávat a odebírat hodnoty ze zásobníku operandů i instrukce manipulující s proměnnými objektu a polemi.

1.8.2 Aritmetické instrukce

Aritmetické instrukce pracují nad dvěma typy dat. Nad celými čísly a nad čísly s pohyblivou řádovou čárkou. Typicky je výsledek funkcí dvou parametrů. Parametry jsou načteny ze zásobníku operandů a výsledek je tam pak vložen. Jedná se o tyto skupiny instrukcí:

- Součet
- Rozdíl
- Násobek
- Podíl
- Zbytek po dělení
- Negace
- Bitový posun
- Bitová disjunkce
- Bitová konjunkce
- Bitová exkluzivní disjunkce
- Inkrementace lokální proměnné
- Porovnání

JVM nevyvolá výjimku v případě přetečení nebo podtečení při výpočtu.

1.8.3 Instrukce pro přetypování

Tyto instrukce slouží k přetypování všech číselných typů mezi sebou, nejjednodušeji je lze rozdělit na tyto dvě třídy:

- Rozšiřující konverze
- Zužující konverze

Při obou konverzích může dojít ke ztrátě nějaké informace jako je ztráta přesnosti nebo oříznutí nejnižších bitů. JVM však v těchto případech nevyvolá žádné výjimky.

1.8.4 Instrukce pro manipulaci s objekty

Instance tříd i pole jsou objekty, ale JVM k manipulaci s nimi používá odlišné instrukce:

- Vytvoření nového pole
- Vytvoření nové instance třídy
- Přístup k instančním a třídním proměnným
- Vložení prvku pole na zásobník operandů
- Uložení hodnoty ze zásobníku operandů do pole
- Vložení délky pole na zásobník operandů
- Kontrola, zda je objekt instancí nějakého typu nebo jeho potomkem

1.8.5 Instrukce pro práci se zásobníkem

Tyto instrukce manipulují s položkami na zásobníku bez ohledu na jejich typ. Jedná se o instrukce pro smazání hodnoty z vrchu zásobníku, duplikace hodnoty na vrchu nebo prohození dvou nejvyšších hodnot na zásobníku.

1.8.6 Instrukce ovlivňující tok programu

Tyto instrukce buď vždy, nebo pod nějakou podmínkou mění tok programu, a tedy hodnotu PC - jinak řečeno další instrukci, která se načte:

- Podmíněný skok
- Složený podmíněný skok
- Nepodmíněný skok

1.8.7 Instrukce pro volání metod a návrat z metod

- invokevirtual - zavolá metodu objektu, metoda se hledá podle typu objektu
- invokeinterface - volá metodu rozhraní, hledá implementaci metody v objektu
- invokespecial - volá speciální instanční metodu, například konstruktor nebo privátní metodu
- invokedynamic - instrukce není určená pro Javu, ale pro ostatní jazyky, umožňuje snadné použití duck typingu

1.9 Common Language Infrastructure

CLI podporuje objektově orientované výpočty stejně jako JVM, ale cíle tohoto virtuálního stroje jsou o něco širší než cíle JVM. Oba stroje jsou si velmi podobné v tom, že jejich běhové prostředí provádí kontroly programů a garbage collection. JVM byla navržena přímo pro jazyk Java, ale řada jazyků jde zkompilovat do binárních java souborů. Oproti tomu je CLI navržena tak, aby podporovala množství různých spolupracujících jazyků. Dokonce na ní mohou běžet i programy, které nejsou nebo nemohou být validovány zavaděčem. Hlavním rozdílem mezi JVM a CLI tedy je, že CLI je nezávislá nejen na hostitelském systému jako Java, ale též na jazyku vyšší úrovně.

Obdobou binárních Java souborů jsou na platformě .NET moduly. Obsahují jak metadata, tak kód ve formě MSIL (obdoba Java bajtkódu). Do těchto modulů lze zkompilovat programy v řadě jazyků. Některé z nich jsou: Java, C#, Visual Basic, .NET a C++. Moduly jsou při načítání validovány stejně jako v případě JVM, ale na rozdíl od něj lze tuto validaci vypnout.

V JVM lze volat nativní metody. Nativní kód se může dotazovat na různá data pomocí funkcí JVM. V CLI je spolupráce dotažena ještě dál, lze sdílet i datové struktury mezi různými jazyky. Má to však svou cenu. Může to vyžadovat velké změny implementace současných jazyků, jak se stalo třeba v případě Visual Basicu. Dochází i k dalším problémům, třeba v případě, kdy je jeden jazyk citlivý na velikost znaků a druhý nikoliv.

Instrukční sada MSIL, je bajtkódu velmi podobná. Kvůli podpoře jazyku bez verifikace však obsahuje řadu nekontrolovaných instrukcí pro alokování kusu paměti a přístup k nějaké její části nebo pro práci s ukazateli. Instrukce pro aritmetiku nejsou typované a CLI pozná až za běhu, jak instrukci provést podle toho, co se nalézá na zásobníku operandů.

Kapitola 2

Scheme

2.1 Lisp

Lisp je funkcionálně orientovaný programovací jazyk, respektive rodina jazyků. Jeho původní specifikace vznikla již v roce 1958. Existuje mnoho dialektů Lispu, které se od sebe více či méně liší. V dnešní době jsou nejrozšířenější dialekty Clojure, Common Lisp a Scheme.

Název Lisp je zkratka pro List processing. List je totiž základní datová struktura v Lispu. A sám zdrojový kód je též tvořen listy. Díky tomu, je možné pracovat s kódem jako s daty, a to dále usnadňuje psaní maker v Lispu.

Jak ještě bude podrobněji probráno, je Lisp rozšířením lambda kalkulu. Řada dnes v programování běžně rozšířených konceptů pochází právě z Lispu. Jedná se například o if-then-else konstrukt, funkce vyššího řádu, stromové datové struktury, dynamickou typovou kontrolu, Garbage collection a rekurzi.

2.2 Lambda Kalkul

Lambda kalkul je nejmenší univerzální programovací jazyk poskytující teoretický základ pro funkcionální programování. Alonzo Church ho objevil v roce 1936. Rok na to jeho student Alan Turing dokázal, že modely lambda kalkulu a Turingova stroje jsou ekvivalentní a popisují stejnou třídu funkcí.

Syntaxe lambda kalkulu:

```
<expression> = <name> | <function> | <application>
<expression> = ( <expression> )

<name> = constant | variable
<function> = lambda variable. <expression>
<application> = <expression> <expression>
```

Konstanty jsou čísla a nativní funkce, proměnné jsou identifikátory. Celý lambda kalkulus se skládá z definice funkcí a jejich následné aplikace. Vyhodnocení nějakého výrazu v lambda kalkulu tak spočívá v opakovaném dosazování parametrů do funkce, dokud se nedostaneme k výsledku.

```
((lambda x.xxx) 5)    =>    555
```

Výraz se skládá z definice anonymní funkce (jiné v klasickém lambda kalkulu nejsou) a aplikace této funkce na číslo pět. Výraz vyhodnotíme dosazením čísla za parametr x v těle funkce. Tělo se skládá ze tří za sebou následujících x tím pádem je výsledek 555.

Lisp je v podstatě jen rozšíření lambda kalkulu. Všechny základní koncepty jako lambda-funkce, proměnné, konstanty a aplikace, zůstávají, ale je k nim přidána další syntaxe a nativní funkce. Lisp navíc oproti lambda kalkulu umožňuje pojmenovávat funkce, používat podmínky, definovat funkce s více než jedním parametrem, používání datových struktur jako jsou vektory nebo listy a mnoho dalšího. Díky tomu je psaní v Lispu pohodlnější a lze jej prakticky použít pro programování a ne jen jako teoretický konstrukt.

Jednotná syntaxe Lispu umožňuje velmi snadno psát jeho další rozšíření. Stačí napsat funkci a definovat ji při spouštění Lispu. Na první pohled se to zdá jako výhoda, ale vedlo to ke vzniku mnoha různých verzí (dialektů) Lispu jako jsou: BBN-Lisp, Franz Lisp, Interlisp-10, Le-Lisp, Lisp 1.5, Lisp/370, Lisp Machine Lisp, Maclisp, NIL, Scheme, ZetaLisp a další. Taková roztříštěnost způsobuje velké problémy s přenosem zdrojových kódů z jednoho Lispu do druhého. Tento problém částečně vyřešil Common Lisp.

2.3 Rozšířené dialekty jazyka Lisp

2.3.1 Scheme

Dialekt vytvořený roku 1970 v laboratořích MIT. Vyznačuje se jednoduchým a minimalistickým designem. Díky tomu má pověst spíše akademického jazyka. Klade velký důraz na funkcionální aspekty jazyka a rekurzi.

Pro Scheme existuje několik verzí standardů nazvaných the Revisedn Report on the Algorithmic Language Scheme (RnRS). Nejrozšířenější používanou verzí je R5RS. Jednoduchost standardu má bohužel i své stinné stránky, konkrétně hlavně nepřenositelnost kódu mezi jednotlivými implementacemi.

2.3.2 Common Lisp

Common Lisp byl vyvinut s cílem sjednotit mnoho odlišných dialektů Lispu pod jeden standard. V tomto ohledu zastává opačný přístup oproti Scheme a snaží se do sebe zahrnout co nejvíce. Přichází tak sice o jednoduchost, ale získává kompatibilitu s mnoha dalšími dialekty. Díky této kompatibilitě

a množství knihoven, zahrnutých už ve standardu, je tento dialekt vhodný pro komerční použití. Common Lisp neprosazuje tak silně rekurzi a obsahuje i systém umožňující objektové programování Common Lisp Object System (CLOS).

2.3.3 Clojure

Clojure je dialekt Lispu kompilovaný do JVM bajtkódu. Díky tomu může využívat množství knihoven napsaných pro Javu. Jeho silnou stránkou je podpora vícevláknových aplikací a z toho vyplývající důraz na neměnné objekty.

2.3.4 Ostatní dialekty

Mezi ně patří především ELisp určený na psaní rozšíření pro textový editor EMacs, a AutoLisp určený na psaní rozšíření pro AutoCad. Přestože jsou oba dialekty rozšířené, jejich použití je omezené jen na jejich domovské prostředí. Navíc jsou oba poměrně zastaralé, co se návrhu jazyka týče.

2.4 Scheme

Jediné dva dialekty Lispu, se kterými jsem měl dosud zkušenosti, byly Common Lisp a Scheme. Syntaxe Scheme mi přišla přehlednější a jednodušší, a proto jsem se rozhodl v rámci této práce implementovat interpret Scheme. Nyní tento dialekt popíšu podrobněji než ostatní.

Pro jazyk Scheme existují dva standardy. První oficiální IEEE standard a druhý, který se nazývá Revisedn Report on the Algorithmic Language Scheme. N v názvu je číslo dané revize. Zkráceně se také píše jako RnRS. Tato práce se zabývá implementací druhého standardu ve verzi čtyři, tedy R4RS. Viz [2]. Pokud tedy budou další zmínky o standardu jazyka Scheme, je tím myšlen právě tento standard. A pokud bude zmíněna implementace, je tím myšlena implementace zmíněného standardu.

Ve standardu jazyka Scheme jsou uvedeny všechny syntaktické konstrukty nativní funkce a další požadavky, které musí jeho implementace splňovat. Ne všechny zde uvedené položky jsou však povinné, některé představují jen doporučení, kterých se implementace nemusí držet.

Ve standardu jsou označeny situace, ve kterých implementace musí signalizovat chybu. Je však i mnoho situací, ve kterých je rozhodnutí zda pokračovat ve vykonávání programu, ponecháno na implementaci.

U některých výrazů standard deklaruje, že hodnota, kterou vracejí, není specifikována (unspecified). V takovém případě se výraz musí vyhodnotit a vrátit nějakou hodnotu bez toho, aby signalizoval chybu. Jaká konkrétní hodnota je vrácena, je záležitostí implementace.

V kódu jazyka Scheme se můžeme potkat nejčastěji s dvěma konstrukty, popsanými ve standardu. Prvním jsou výrazy, neboli syntaktické konstrukty,

které představují samotné jádro jazyka, částečně převzaté z lambda kalkulu. Výrazy se dělí na dvě podskupiny: primitivní výrazy a odvozené výrazy. Do primitivních patří například lambda výraz, který vrací anonymní proceduru nebo výraz pro volání procedury. Primitivní výrazy nelze nahradit jiným konstruktem jazyka. Odvozené výrazy však lze vyjádřit pomocí výrazů primitivních. Nejsou tedy nezbytně nutné, ale jsou zde, protože stejnou věc vyjadřují přehledněji a stručněji. Sem patří například let výraz, umožňující definici lokálních proměnných pro blok kódu.

Následující text si neklade za cíl suplovat roli standardu, jen ho čtenáři přiblížit. Nejsou zde tedy uvedeny všechny podrobnosti a řada věcí je vynechána, ale jsou zde zahrnuty ty nejpodstatnější věci. Celý standard je přístupný online [2].

2.5 Sémantika

Scheme používá statický (nebo také lexikální) scope. To znamená, že proměnná se hledá v kontextu místa, kde byla funkce obsahující tuto proměnnou definovaná. Naproti tomu, když je použit dynamický scope proměnná se hledá v aktuálním stavu programu. Záleží tak na tom, odkud byla funkce volána. Stejně jako Scheme i většina moderních jazyků používá statický scope.

Scheme je jazyk s dynamickou typovou kontrolou. Typy jsou přiřazeny hodnotám, nikoliv proměnným. Proměnná tedy může v průběhu času ukazovat na více hodnot rozdílných typů. Veškerá typová kontrola probíhá až za běhu.

Všechny objekty vytvořené při běhu programu existují po celou dobu běhu, žádný objekt není nikdy zničen. Implementace Scheme mají povoleno využít místo v paměti, pokud lze dokázat, že objekt, který ho zabírá, už nebude nikdy znovu použit.

Pokud posledním příkazem nějaká funkce volá sebe sama, dochází k rekurzi, při které si není třeba pamatovat stav funkce, všechno ostatní už bylo provedeno. Standardně se však na zásobník ukládají data o všech funkcích, ve kterých je program právě zanořen. Pokud jazyk podporuje koncovou rekurzi, znamená to, že v takové situaci pozná, že data není třeba držet a nevytváří zbytečné množství dat na zásobníku. Implementace Scheme jsou povinny implementovat koncovou rekurzi. Díky tomu je možné provádět rekurzivní výpočty s použitím konstantní paměti. Iteraci, tak lze popsat pomocí běžného volání funkce a ostatní speciální iterační konstrukty, slouží ve Scheme pouze jako syntaktický cukr.

Procedury jsou objekty. Mohou být dynamicky vytvářeny, uloženy do proměnné nebo vráceny jako výsledek jiné procedury.

Parametry procedur jsou ve Scheme vždy předávány hodnotou. Jsou tedy vyhodnoceny předtím, než se zavolá samotná procedura, bez ohledu na to, jestli je bude potřebovat nebo ne. Existují i jazyky, ve kterých se parametr nevyhodnotí pokud jej procedura nepotřebuje.

Ve Scheme je každé celé číslo zároveň racionální číslo. Každé racionální číslo je zároveň reálné číslo a každé reálné číslo je i komplexní číslo. Není tedy třeba rozlišovat mezi aritmetikou pro celá a reálná čísla.

2.6 Syntaxe

Stejně jako většina ostatních dialektů Lispu používá i Scheme prefixovou syntaxi s plným ozávorkováním. Sečtení dvou čísel je tak list, který obsahuje ona čísla a název funkce pro sečtení, tedy znak +. Protože se jedná o prefixovou notaci, je název funkce uveden jako první položka listu a parametry následují za ním.

```
(+ 42 6)
```

Tato syntaxe je použita jak pro program, tak pro data. Syntax pro popis dat je tedy vlastně podmnožinou syntaxe pro popis celého programu. Díky tomu je možné a snadné ve Scheme pracovat s daty a programem stejným způsobem. Při čtení kódu procedurou read je kód nejprve celý načten jako kdyby šlo o data, nikoliv kód.

2.7 Jmenné konvence

Predikáty jsou ve světě Lispu procedury, které vždy vracejí boolovskou hodnotu. Ve Scheme jméno predikátu standardně končí znakem otazníku „?“ například „boolean?“.

Jména procedur, které mění hodnotu v již dříve alokovaných místech končí vykřičníkem „!“ například „set!“. Říká se jim mutační procedury (mutation procedures). Návratová hodnota těchto procedur není specifikována.

2.8 Lexikální konvence

Jazyk Scheme není citlivý na velikost písmen. Identifikátory „Abc“, „aBc“ a „ABC“ jsou tak pro Scheme ekvivalentní. Jediná výjimka, kde Scheme rozlišuje velikost písmen, je uvnitř textových řetězců a znaků.

Identifikátory jsou v jazyce Scheme podobné jako v jiných jazycích. Úplná definice identifikátoru je záležitostí implementace, ale v zásadě platí, že jakýkoliv řetězec písmen, číslic a speciálních znaků, který nezačíná číslicí, je identifikátor. Navíc platí že „+“, „-“ a „...“ jsou také identifikátory.

Zmíněné speciální znaky, které mohou být v identifikátoru použity jako by šlo o písmena jsou následující znaky:

+ - . * / < = > ! ? : $

Identifikátory jsou v jazyku Scheme využity k několika účelům. Část identifikátorů je použita jako vyhrazená klíčová slova jazyka. Identifikátory, které klíčovým slovem nejsou, mohou být použity jako jména proměnných.

Následuje výčet klíčových slov oddělených čárkou: =>, do, or, and, else, quasiquote, begin, if, quote, case, lambda, set!, cond, let, unquote, define, let*, unquote-splicing, delay, letrec

Mezera, tabulátor, nový řádek a další bílé znaky jsou používány ke zpřehlednění kódu a oddělení lexikálních tokenů od sebe, jinak jsou ale v kódu bezvýznamné. Bílé znaky se mohou objevit mezi tokeny, ale nikoliv uprostřed jednoho tokenu. Výjimka, kdy je bílý znak významný, je, když je použit uprostřed textového řetězce.

Začátek komentáře se označuje středníkem, vše za středníkem až po znak nového řádku je obsahem komentáře. Pro jazyk Scheme je komentář neviditelný, ale mezera na konci se jeví jako bílý znak. Není tedy možné, aby byl komentář uprostřed tokenu.

2.9 Primitivní výrazy

2.9.1 Reference na proměnnou

```
<jméno proměnné>
```

Tento výraz je velmi jednoduchý, skládá se pouze z identifikátoru a proměnné. Při jeho vyhodnocení se vyhledá proměnná daného jména v současném environmentu a vrátí se hodnota, na kterou ukazuje. Případně dojde k chybě, pokud proměnná daného jména není v environmentu definována.

2.9.2 Literál

```
quote <výraz>
‘<výraz>
<konstanta>
```

Quote a ‘ jsou identické. quote <výraz> se vyhodnotí na <výraz>. Tento jednoduchý konstrukt umožňuje zadávat do programu například nějaká data. List obsahující data zadaný bez použití tohoto konstruktu se vyhodnotí jako volání funkce. V mnoha situacích to není žádoucí. Například pokud má být list předán parametrem nějaké proceduře. Díky quote toho lze dosáhnout.

```
'(1 2 3)
> (1 2 3)

'abc
> abc
```

Dalším literálem je přímo konstanta. Jedná se o čísla, boolovské hodnoty, textové řetězce a znaky. Tyto se vyhodnotí vždy pomocí identity a není tak třeba používat quote.

```
42
> 42

"ahoj"
> ahoj

#t
> #t
```

2.9.3 Volání procedury

`<operator> <operand>*`

Volání procedury je de facto aplikace z lambda kalkulu. Operátor i operandy jsou vyhodnoceny v nespecifikovaném pořadí. Operátor by měl vrátit hodnotu typu procedura, jinak dojde k chybě. Následně je procedura zavolána s vyhodnocenými parametry.

Procedury, kterým se říká nativní, jsou ve Scheme dostupné hned po spuštění, další nové procedury lze vytvářet pomocí lambda výrazu.

```
(+ 42 2)
> 44
```

Volání nativní procedury +, která sečte všechny své parametry.

2.9.4 Lambda výraz

`lambda <argumenty> <tělo>`

Lambda výraz vytvoří a vrátí novou proceduru. <argumenty> je list identifikátorů, každý z nich představuje jedno jméno argumentu, který bude procedura přijímat. Při vytváření si procedura zapamatuje aktuální environment.

Při volání pak procedura dostane již vyhodnocené argumenty (viz předchozí výraz). Vytvoří nový environment, který bude mít jako předka environment uložený při vyhodnocování lambda výrazu. Poté v nově vytvořeném environmentu definuje všechny argumenty s použitím jmen z <argumenty>. Nakonec se v nově vytvořeném environmentu vyhodnotí <tělo> procedury a výsledek posledního výrazu je návratová hodnota.

```
(lambda (x) (* x x))
>  procedura
```

```
((lambda (x) (* x x)) 2)
> 4
```

2.9.5 Podmínka

```
if <test> <tělo1> <tělo2>
if <test> <tělo>
```

Nejprve je vyhodnocen <test> pokud vrátí #t je vyhodnoceno <tělo1> a navrácena výsledná hodnota, v opačném případě je vyhodnoceno <tělo2> a navrácena výsledná hodnota. Pokud <tělo2> není zadáno, je vrácena nespecifikovaná hodnota.

```
(if (> 9 8) 'mensi 'vetsi)
> mensi
```

2.9.6 Přiřazení

```
set! <proměnná> <výraz>
```

Nejprve je vyhodnocen <výraz> a následně je uložen do proměnné. <proměnná> je název proměnné, která musí být definována v aktuálním environmentu nebo v některém z jeho předků. Výsledná hodnota není specifikována.

```
(set! foo 666)
```

2.10 Odvozené výrazy

Jak už bylo výše zmíněno, tyto výrazy lze nahradit použitím primitivních výrazů, ale jsou součástí standardu, protože jejich použití v kódu je přehlednější a jednodušší.

2.10.1 Podmíněné výrazy

```
cond ( <test> <výraz>* )+
cond ( <test> <výraz>* )+ ( else <výraz>+ )
```

Při vyhodnocování tohoto výrazu se postupně procházejí jednotlivé testy, které se vyhodnocují stejně jako v případě if výrazu. Pokud je test vyhodnocen kladně, provedou se všechny výrazy bezprostředně za ním a výsledek posledního z nich se vrátí jako výsledek cond výrazu. Místo posledního testu může

být uvedeno klíčové slovo else. To se vyhodnotí jako úspěšný test a zbytek vyhodnocení je stejný jako při běžném testu. Kód za else se tedy provede vždy, když všechny ostatní testy selžou.

```
(cond ((= 5 4) 'foo)
        ((= 4 4) 'bar))
> bar
```

```
case <klíč> ((<data>*) <výraz>+ )+
case <klíč> ((<data>*) <výraz>+ )+ (else <výraz>+ )
```

Nejprve je vyhodnocen klíč, jeho hodnota je pak porovnávána se všemi daty. Pokud s nějakými daty dojde ke shodě, tak se vyhodnotí všechny výrazy za nimi bezprostředně následující a hodnota posledního se vrátí jako výsledek case výrazu. Pokud se hodnota neshoduje s žádnými daty, vyhodnotí se výrazy za klíčovým slovem else stejným způsobem. Pokud else není přítomno, je výsledek case výrazu nespecifikovaný.

```
(case 'a
  ((a e i o~u) 'samohlaska)
  (else 'souhlaska))
> samohlaska
```

```
and <test>*
```

Vyhodnocuje se zleva doprava. Vyhodnocování skončí prvním testem, který se vyhodnotí jako #f. Ta samá hodnota se pak vrátí jako výsledek celého výrazu. Pokud se žádný z testů jako #f nevyhodnotí, je navrácena hodnota #t.

```
or <test>*
```

Vyhodnocuje se zleva doprava. Vyhodnocování skončí prvním testem, který se vyhodnotí jako #t. Ta samá hodnota se pak vrátí jako výsledek celého výrazu. Pokud se žádný z testů jako #t nevyhodnotí, je navrácena hodnota #f.

V případě těchto logických operátorů se tedy používá líné vyhodnocování.

2.10.2 Výrazy pro definici lokálních proměnných

Syntax těchto výrazů je identická, ale liší se především ve způsobu, jakým inicializují proměnné.

```
let ( (<proměnná> <inicializace>)*  ) <tělo>
```

Všechny inicializace jsou vyhodnoceny v nespecifikovaném pořadí v současném environmentu. Následně je vytvořen environment nový a jsou v něm definovány všechny proměnné s hodnotami vyhodnocených inicializací. Nakonec se

provedou všechny příkazy v těle s novým environmentem a výsledek posledního z nich je vrácen jako výsledek celého výrazu.

```
letrec ( (<proměnná> <inicializace>)*  ) <tělo>
```

Na rozdíl od předchozího výrazu se nejprve vytvoří nový environment a inicializace se vyhodnocují v něm. Definice proměnných a vyhodnocení už je stejný jako u let výrazu.

2.10.3 Sekvence

```
begin <výraz>+
```

Podobný výraz jako let, jen bez definic proměnných. Výrazy jsou vyhodnocovány zleva doprava a výsledek posledního výrazu je vrácen jako výsledek begin.

2.10.4 Quasiquote

```
quasiquote <výraz>
` <výraz>
```

Pokud se ve výrazu nevyskytne čárka „,“ je quasiquote identická s quote. Pokud se však čárka vyskytne, je výraz hned za čárkou normálně vyhodnocen a jeho výsledek je dosazen na původní místo čárky a výrazu bezprostředně za ní.

2.11 Typy hodnot

Standard jazyka Scheme specifikuje několik typů objektů. V této kapitole budou všechny stručně popsány. Ke každému typu existuje predikát - tedy procedura vracející boolovskou hodnotu, která určí zda její parametr je daného typu. Například procedura list? vrátí #t pouze pokud je její parametr opravdu listem. Následuje výčet typů:

boolean, pair, symbol, number, char, string, vector, procedure

2.11.1 Boolovské hodnoty

Boolovská hodnota je buď pravda nebo nepravda. Jejich reprezentace v jazyku Scheme se zapisuje jako #t a #f. Podmíněné výrazy však k boolovským hodnotám přistupují odlišně. #f berou vždy jako nepravdu, ovšem pravda je téměř vše ostatní (#t, list, vector, symbol, atd.).

Většina konstruktů operujících s boolovskými hodnotami jsou ve Scheme speciální výrazy a ne nativní funkce. Důvodem je, že při volání funkce by se nejprve vyhodnotily všechny její parametry, což by například u výrazu if bylo špatně. Vyhodnotí se vždy jen jedna větev. Podobně je to i s výrazy and a or.

2.11.2 Páry

Pár je paměťová buňka obsahující dva objekty. Nazývají se (z historických důvodů) car a cdr. Někdy též head a rest. Pár lze vytvořit pomocí nativní procedury cons. Pomocí procedur car a cdr lze číst jeho obsah. Procedury set-car! a set-cdr! nám ho umožňují i měnit.

Pár se též dá zapsat jako „(car . cdr)“ tedy dva prvky v závorkách oddělené tečkou. Kód „'(1 . 2)“ vrátí to samé jako „(cons 1 2)“. Existují i další varianty funkcí car a cdr, například (caadr x) - tento zápis je jen zkráceným zápisem pro (car (car (cdr x))). Tyto procedury existují ve všech variantách až do hloubky čtyř. Je jich tedy 28.

2.11.3 Listy

List je základní struktura v Lispu. Je dokonce obsažena v jeho názvu a v listech je zapsán veškerý kód Lispu. List lze definovat následujícím způsobem:

- Prázdný list je list
- Pár jehož cdr je list je také list

Z toho plyne, že list je tvořen páry a zakončen prázdným listem. Jedná se vlastně o spojový seznam tvořený z párů, kde car obsahuje hodnotu a cdr další článek spojového seznamu nebo (pokud se jedná o poslední prvek) prázdný list. Následující dva zápisy listu jsou tedy ekvivalentní.

```
(1 2 3 4 5)
(1 . (2 . (3 . (4 . (5 . ())))))
```

Každý neprázdný list je tedy zároveň pár, opačně to však neplatí. Rovněž objekt, jenž byl v jednu chvíli list, jím v jinou chvíli už být nemusí, a to díky proceduře set-cdr!.

Prázdný list je speciální objekt délky nula, jenž není párem. Sekvence, která by jinak tvořila list, avšak není ukončena prázdným listem, ale jiným prvkem, se nazývá nepravý (improper) list.

2.11.4 Symboly

Symbol je podobný textovému řetězci v tom, že jde o nějakou sekvenci znaků. Pro jednu sekvenci se vytvoří maximálně jedna její reprezentace v paměti. Pokud se daná sekvence (jako symbol) vyskytne na jiném místě programu, už nebude zabírat žádné místo navíc, ale použije se stejný paměťový prostor. Tato vlastnost je výhodou ve chvíli, kdy je potřeba porovnat dva symboly. Stačí totiž porovnat jejich adresu v paměti, namísto náročného porovnávání textových řetězců.

Pro tyto vlastnosti je symbol v mnoha implementacích Scheme používán interně pro identifikátory. A i pravidla pro jeho zápis jsou stejná jako pro identifikátor.

2.11.5 Čísla

Ve Scheme existuje několik druhů čísel a platí mezi nimi vztah dědičnosti. Jedná se o následující typy:

number - complex - real - rational - integer

Platí tedy, že typ nalevo je zároveň všemi typy napravo. Tedy real je zároveň i complex a number. Pro všechny typy čísel existují predikáty number?, complex?, real?, atd.

Další dělení čísel je na čísla přesná (exact) a nepřesná (inexact). Nepřesná čísla jsou v počítači sice reprezentována nějakým číslem, ale víme, že to číslo odpovídá skutečnosti jen přibližně. Oproti tomu přesná čísla jsou vždy taková, jaká jsou uložena. Je jasné, že například délka listu tak musí být přesné číslo. Toto dělení je ortogonální (nezávisle) k rozdělení čísel podle typu. I pro něj existují predikáty exact? a inexact?.

Standard nevyžaduje podporu všech typů čísel výše zmíněných. Poskytuje řadu doporučení, ale jediné co opravdu vyžaduje, jsou celá čísla, která jsou nutná pro indexaci vektorů, jako délka vektoru, stringu a listu a pro další použití.

2.11.6 Znaky

Znaky jsou písmena, číslice, netištěné znaky a další speciální znaky jako diakritika. V Scheme se literál znaku zapisuje následujícím způsobem:

```
#\a, #\c, #\newline, #\space,...
```

Existují dva způsoby zápisu. Oba začínají znaky # a poté následuje buďto konkrétní znak anebo název znaku jako newline pro znak nového řádku, nebo space pro znak mezery. První způsob je citlivý na velikost znaků, druhý nikoliv.

2.11.7 Textové řetězce

Textový řetězec neboli string je sekvence znaků. Zapisuje se mezi dvojici uvozovek. Pokud chceme použít uvozovky uvnitř textového řetězce, musíme před nimi použít zpětné lomítko, aby bylo poznat, že se nejedná o konec řetězce, ale o zamýšlený znak. Zpětné lomítko se zapíše dvěma zpětnými lomítky. Následuje příklad:

```
"Rekl mi \"ahoj Pavle!\"."
```

Délka řetězce je rovna počtu znaků, jenž obsahuje, a je po vytvoření dále neměnná. Znaky jsou indexovány od nuly. První znak je tedy na pozici 0 a poslední na pozici délka - 1.

2.11.8 Vektory

Vektor je sekvence hodnot. Na rozdíl od listu umožňuje přistupovat ke svým jednotlivým prvkům efektivně pomocí indexu. Podobně jako u textového řetězce je i délka vektoru neměnné číslo, které je mu přiřazeno při vytvoření vektoru. Délka představuje počet prvků, které vektor obsahuje.

```
#('foo 42 "ahoj")
```

Tento kód vytvoří vektor délky tři obsahující symbol foo, číslo 42 a textový řetězec obsahující slovo ahoj.

2.11.9 Procedury

Procedury vznikají pomocí lambda výrazů. Řada procedur musí být v hlavním environmentu definována již po spuštění implementace Scheme. Takové procedury se označují jako nativní. Protože funkce jsou hodnoty stejně jako třeba listy, lze je používat stejným způsobem. Znamená to, že si funkce lze předávat parametrem nebo vracet jako hodnotu z jiné funkce.

```
((lambda (a b) (+ a b)) 42 5)
> 47
```

Kód z příkladu vytvoří pomocí lambda výrazu anonymní proceduru a následně ji zavolá s parametry 42 a 5.

Kapitola 3

Scala

Scala je relativně nový multiparadigmatický funkcionálně a objektově orientovaný jazyk nad platformou Java. Autorem Scaly je Martin Odersky, který dříve pracoval na překladači Javy. První verze jazyka vyšla v roce 2003. Ze začátku byla podporována i platforma .Net, ale v roce 2012 byla oficiální podpora zastavena. Scala umožňuje používat třídy napsané v Javě ve stejném projektu, dokonce i vzájemné reference z Javy do Scaly a zpět.

Filozofie Scaly staví na několika pilířích. Objektový přístup Scala umožňuje snadným členěním kódu na třídy, podporou polymorfizmu a mixinů, což je kompromis mezi vícenásobnou dědičností a strohými rozhraními bez možnosti definovat v nich implementaci. Potenciál DRY principu je tak možné ve Scale vyždímat opravdu do poslední kapky. Každá hodnota ve Scale je objekt. A to včetně hodnot jako Int nebo Boolean, které jsou v Javě primitivními typy. Funkce jsou také objekty[1]. Řada návrhových vzorů souvisejících s objektovým přístupem je podporována nativně.

Funkcionální stránka Scaly je neméně důležitá. Idea programu složeného z funkcí bez vedlejších efektů ve Scale není vynucována, ale jazyk k ní jeho uživatele navádí. Scala tak například klade velký důraz na neměnné objekty. Funkce jsou objekty, mohou tak být předávány parametrem nebo vráceny návratovou hodnotou. Podpora lambda výrazů umožňuje snadný zápis anonymních funkcí. Práci s neměnnými hodnotami dále usnadňuje to, že skoro vše ve Scale může vracet hodnotu. Je tak možné pracovat s if nebo match klauzulí jako s výrazem.

Scala se snaží tyto dva přístupy (funkcionální a objektový) sloučit dohromady v jeden organický celek. Scala má silný statický typový systém. To znamená, že typy jsou kontrolovány už při kompilaci a lze tak odhalit mnoho chyb, na které by se jinak přišlo až za běhu. Scala dokonce umožňuje polymorfizmus mezi generickými třídami.

Scala se přesto snaží v jednoduchosti, s jakou lze psát kód, přiblížit skrip-

[1] Nebo jsou na objekty snadno převoditelné.

tovacím jazykům jako jsou Ruby nebo Python. Například středník na konci řádku si Scala doplní až na pár nerozhodných případů sama a stejně je to i s typy - pokud lze typ z kontextu určit, není třeba ho v kódu definovat. Tento automatický typový systém je velmi pokročilý. I v případě, kdy je díky dědičnosti možno zvolit několik typů, kompilátor volí ten nejvíce specifický.

V dalších kapitolách budou popsány některé důležité konstrukty jazyka. Protože Scala a Java mají speciální vztah, bude zde často zmiňováno, jak se Scala od Javy odlišuje. Od čtenáře se tedy očekává, že je s Javou obeznámen alespoň na základní úrovni.

3.1 Třídy

Třídy nejsou ve své podstatě tolik odlišné od javovských, ale mnoho věcí je zjednodušeno a zautomatizováno ve snaze zabránit neustálému psaní téměř identického kódu jako jsou getry a setry.

```
class HelloClass {
  val hello = "string from class"
}
```

Tento kód definuje třídu s jednou neměnnou a bezparametrickým konstruktorem. Klíčové slovo val slouží k definici neměnných, do kterých lze přiřadit hodnotu jen jednou. Klasickou proměnnou lze definovat pomocí slova var. Třídu lze použít následujícím způsobem:

```
val foo = new HelloClass
println(foo.hello)
```

Tento kód vytiskne na standardní výstup "string from class". Závorky při volání konstruktoru jsou vynechány, stejně tak středníky na konci řádků. Scala si je doplní sama.

```
class HelloClass(name: String) {
  val hello = "hello"
  def sayHello():String = {
    hello + " " + name
  }
}
```

Na tomto příkladu už je o něco delší třída s konstruktorem, přijímajícím jeden parametr. Ve Scale je tělo primárního konstruktoru tvořeno tělem třídy a definice jeho parametrů se zapisuje bezprostředně za název třídy. Konstruktor z příkladu má jeden parametr typu String. Parametr lze definovat i s klíčovým slovem var nebo val. V takovém případě kompilátor automaticky vytvoří členskou proměnnou nebo neměnnou se stejným jménem uvnitř třídy.

Funkce sayHello() se definuje v rámci vykonávání konstruktoru. Protože funkce používá parametr konstruktoru name, musí si na tento parametr držet referenci. Taková funkce se nazývá closure.

Scala automaticky vrací poslední výraz v metodě, proto není třeba používat klíčové slovo return. Pokud se tělo metody vejde na jeden řádek, není potřeba uvádět obalovací závorky. Rovněž návratový typ metody si dokáže v tomto případě Scala zjistit sama. Definici třídy je tak možné zjednodušit následujícím způsobem:

```
class HelloClass(name: String) {
  val hello = "hello"
  def sayHello() = hello + " " + name
}
```

Následuje příklad použití třídy, který vypíše "hello Richard":

```
val foo = new HelloClass("Richard")
println(foo.sayHello())
```

3.2 Singleton a companion objekt

Scala nemá statické proměnné a metody, místo toho má singleton objekt. Singleton objekt se definuje stejně jako třídy, jen místo klíčového slova class použijeme slovo object. Takový objekt má pak vždy jen jednu instanci, proto singleton. Pokud existuje třída se stejným názvem jako objekt, říká se mu pak companion objekt, takový objekt může přistupovat ke členským datům dané třídy.

Protože je to přirozené místo, kde definovat metody, které by v Javě byly statické, dá se companion objekt využít i pro různé statické tovární metody.

```
abstract class WebPage(address:String){
  def nationality:String;
}
class CzechWebPage(address:String) extends WebPage(address){
  def nationality = "czech";
}
class GermanWebPage(address:String) extends WebPage(address){
  def nationality = "german";
}
object WebPage {
  def apply(address:String):WebPage = {
    if (address.endsWith(".cz")) new CzechWebPage(address);
    else if (address.endsWith(".de")) new GermanWebPage(address);
    else throw new UnsupportedOperationException
  }
}
```

Abstraktní třída WebPage definuje jednu metodu nationality vracející String. CzechWebPage a GermanWebPage tuto metodu implementují. Companion objekt obsahuje pouze jednu tovární metodu, která podle koncovky webové adresy vytvoří odpovídající třídu. Zde je třeba zmínit, že metoda apply má ve Scale speciální význam. K zavolání této metody stačí napsat za jméno objektu závorky a do nich parametry WebPage.apply("www.seznam.cz") je tedy to samé, co WebPage("www.seznam.cz"). Následuje příklad použití třídy:

```
val a = WebPage("www.seznam.cz")
println(a.nationality)
```

Tento kód vypíše “czech” na standardní výstup.

3.3 Trait

Trait je obdoba rozhraní v Javě. Hlavní rozdíl spočívá v tom, že v Javě (verze 7) mohou být v rozhraní metody, pouze definované. Trait umožňuje rovnou poskytnout i implementaci. Každá třída může mít jako předka jen jednu třídu, ale mnoho traitů. Pokud ale hovoříme o traitech, nejedná se o vícenásobnou dědičnost známou z jazyků jako je C++. Ve Scale se traity do tříd mixují. U vícenásobné dědičnosti nastává problém když dvě třídy mají stejnou metodu, pak totiž není jasné, která se má při volání použít. Scala tento nedostatek řeší linearizací traitů. Linearizace pomocí jednoznačně daných pravidel vytvoří lineární pořadí, ve kterém jsou traity poté volány. Proto jsou traity ve Scale stohovatelné.

```
abstract class Operation {
  def eval(x: Int):Int;
}
trait Twice extends Operation{
  abstract override def eval(x: Int):Int =
    {println("twice"); super.eval(2 * x) }
}
trait Logging extends Operation {
  abstract override def eval(x: Int):Int = {
    println("evaluating " + x)
    super.eval(x)
  }
}
class Identity extends Operation {
  def eval(x: Int):Int = {println("identity"); x }
}
class MyOperation extends Identity with Twice with Logging;
```

Příklad dvou stohovatelných traitů. Stohovatelnost je možná pouze tak, že traity rozšiřují nějakou abstraktní třídu, což je v tomto případě Operation.

Poté je možné, aby traity přepisovaly její metody a pomocí super volaly další implementace metody. Třída Identity pak poskytuje konkrétní implementaci metody eval.

```
val op = new MyOperation
val res = op.eval(5)
println(res)
```

vypíše:

```
evaluating 5
twice
identity
10
```

Z výpisu je zřejmé, v jakém pořadí se jednotlivé metody volají. Toto pořadí můžeme ovlivnit pořadím, v jakém mixujeme traity do třídy. Existují přesná pravidla linearizace, ale jejich popis je již nad rámec této práce.

S traity je možné pracovat jako s obyčejným rozhraním, které definuje jen signatury metod, ale stejně tak je možné je používat tak, jako na předchozím příkladu. Vždy je třeba zvážit, který přístup je za dané situace vhodnější.

3.4 Case třídy

Case třída se liší od obyčejné třídy v tom, že překladač do ní automaticky doplní mnoho kódu, který by se jinak musel psát ručně. Ke každé case třídě je automaticky vytvořen companion objekt se statickou tovární metodou, takže je lze vytvářet zkráceným zápisem bez klíčového slova new. Dále překladač vytvoří přirozenou implementaci metod toString, equals a hashCode. Metoda equals porovnává celou strukturu do hloubky. Parametry konstruktoru jsou automaticky definované i jako členské neměnné.

Tyto třídy jsou nejsilnější v popisování stromových struktur a velmi dobře se s nimi pracuje také při pattern matchingu, o kterém ještě bude řeč.

```
abstract class Expr;
case class Or(l:Expr, r:Expr) extends Expr;
case class And(l:Expr, r:Expr) extends Expr;
case class Bool(b:Boolean) extends Expr;
```

Tento kód umožňuje vytvářet stromovou strukturu z logických výrazů and a or.

```
val expr = And(Bool(true), Or(Bool(false), Bool(true)));
println(expr)
```

Kód vytiskne na standardní výstup:

```
And(Bool(true), Or(Bool(false), Bool(true)))
```

3.5 Pattern matching

Pattern matching je ve Scale využíván na mnoha místech. Je to v jistém ohledu obdoba switch přepínače známého z Javy nebo C. Je mezi nimi však několik rozdílů. Match je výraz a vrací tedy hodnotu. Jednotlivé větve jsou nezávislé a JVM při vykonávání kódu nikdy nepropadne do další větve. A v neposlední řadě je match výraz v porovnávání hodnot mnohem silnější nástroj.

```
def basicMatch(s: String) = s~match {
  case "mrkev" => println("zelenina")
  case "jablko" => println("ovoce")
  case _       => println("neznam")
}
```

Příklad jednoduché funkce využívající match výraz. Výraz porovná String postupně a podle toho, jestli je stejný nebo ne, něco vytiskne. Podtržítko zde funguje jako výraz, který se rovná čemukoliv, takže pokud se nenajde shoda dřív, tato větev se provede. Match výraz funguje tak, že pokud se najde shoda, provede se daná větev (kód za =>) a jiná shoda už se nehledá.

Nyní představím vzory, které lze v tomto výrazu použít, některé z nich již byly použity v posledním příkladu.

3.5.1 Konstanta

Konstanta se bude shodovat jen sama se sebou. Jako konstanta může být použit jakýkoliv literál, val hodnota nebo singleton objekt.

```
def describe(x: Any) = x match {
  case 5      => "pet"
  case true   => "pravda"
  case "hello" => "ahoj!"
  case Nil    => "prazdny list"
}
```

3.5.2 Divoká karta

Podtržítko představuje divokou kartu, která se bude shodovat s čímkoliv. Může být též použita k označení proměnné v objektu, u které nám je jedno jakou má hodnotu, jak bude ještě ukázáno dále.

```
expr match {
  case "jedna" => 1
  case _      => 0
}
```

3.5.3 Proměnná

Pokud není proměnná blíže specifikována, dojde ke shodě vždy, stejně jako u divoké karty. Rozdíl je v tom, že match výraz nám za proměnnou dosadí onu pasující hodnotu a můžeme s ní dále pracovat.

```
expr match {
  case 0     => "nula"
  case nenula => "nenulova hodnota: "+ nenula
}
```

3.5.4 Konstruktor

Použití konstruktoru v match výrazu je silnou stránkou Scaly. Nejprve se zkontroluje, jestli má daný objekt stejný typ, jakého je konstruktor, poté se postupně kontroluje shoda parametrů v konstruktoru. Uvnitř těchto parametrů mohou být další konstruktory, pasování se tedy provádí do hloubky.

Tento způsob lze použít pro case třídy. Pro ostatní třídy je možné takového chování dosáhnout napsáním extraktoru. To je ale již nad rámec tohoto krátkého přehledu.

```
expr match {
  case And(Bool(false), _)  => println("lez!")
  case _                    =>
}
```

3.5.5 Sekvence

Sekvence jako je List a Array je možné testovat na shodu také. Používá se stejná syntax a lze kontrolovat libovolný počet prvků v listu.

```
expr match {
  case List(0, _, _) => println("tri prvky, za nulou")
  case _            =>
}
```

Někdy se hodí specifikovat list bez konkrétní délky, k tomu mohu použít _* jako poslední prvek ve vzoru. Shoda se najde pro libovolný počet prvků včetně žádného.

```
expr match {
  case List(0, _*) => println("zacina nulou, jeden a vice prvku")
  case _          =>
}
```

3.5.6 Typ

V match výrazu lze u proměnné specifikovat i typ. Následující příklad ukazuje jak.

```
def generalSize(x: Any) = x match {
  case s: String  => s.length
  case m: Map[_, _] => m.size
  case _           => -1
}
```

Následuje příklad funkce řešící boolovské výrazy zapsané pomocí výše uvedených tříd.

```
def eval(expr: Expr): Bool = expr match {
  case b: Bool           => b
  case And(Bool(false), _) => Bool(false)
  case And(_, Bool(false)) => Bool(false)
  case And(l, r)         => Bool(eval(l).b && eval(r).b)
  case Or(Bool(true), _) => Bool(true)
  case Or(_, Bool(true)) => Bool(true)
  case Or(l, r)          => Bool(eval(l).b || eval(r).b)
}
```

Hodnota typu Bool se vyhodnotí sama na sebe. Logické operátory And a Or se kontrolují, zda je není možné vyhodnotit bez dalšího zanořování se do rekurze. Pokud to nelze, zavolá se rekurzivně ta samá metoda na levý a pravý podstrom výrazu.

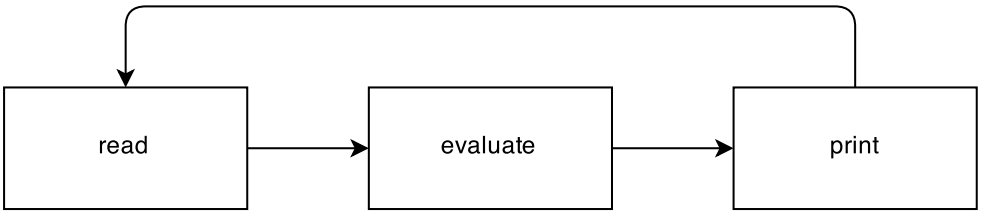
Kapitola 4

Architektura

Tato kapitola seznámí čtenáře s architekturou aplikace. Virtualní stroje jako je JVM nebo CLR pracují na o něco nižší úrovni než tato aplikace. Kód který vykonávají není přímo kód vyššího programovacího jazyka, ale jeho přeložená verze určená pro snadnější strojové zpracování. V tomto smyslu zastane tato aplikace roli překladače i virtuálního stroje. Nejprve kód převede do strojově snadno zpracovatelného tvaru, syntaktického stromu. A následně ho vykoná.

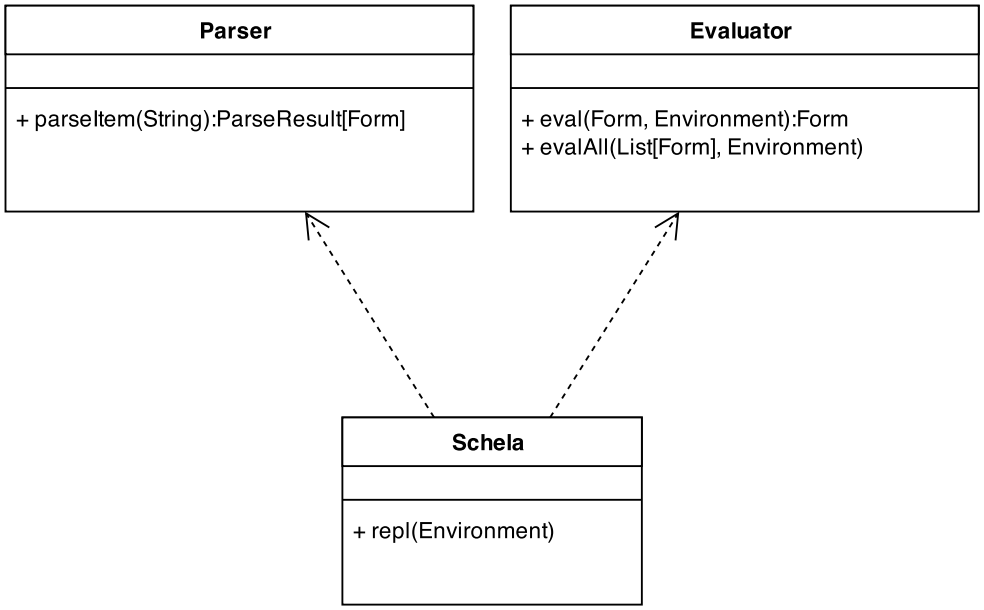
4.1 REPL

REPL je zkratka pro read-evaluate-print loop. Je to způsob, jakým v Lispu lze pracovat. Jedná se o interaktivní mód, ve kterém zadáme nějaký kód a rovnou dostaneme výsledek, ale program stále běží a v dalším příkazu tak můžeme použít třeba dříve definované proměnné a funkce. Spuštění celého zdrojového kódu je podobné, postupně se prochází jednotlivé příkazy a vyhodnocují se, jen výsledky se v průběhu netisknou na výstup.



Obrázek 4.1: Hlavní smyčka programu

Tato smyčka je tedy součástí i mojí aplikace. Program tedy nejprve provede syntaktickou analýzu zadaného příkazu, pokud vše proběhne bez chyb, příkaz se vyhodnotí a výsledná hodnota se vytiskne na standardní výstup.



Obrázek 4.2: Struktura tříd na nejvyšší úrovni

4.2 Parser a Evaluator

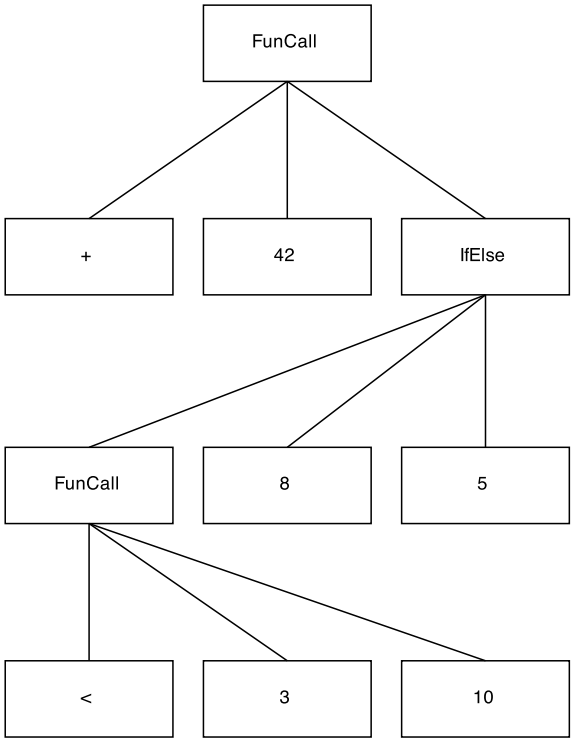
Na obrázku 4.2 je vidět základní členění funkcí v programu. Objekt Schela v sobě zahrnuje hlavní smyčku programu. Pokud uživatel zadá nějaký kód na vstupu, program jej načte a zavolá metodu parseItem. Parser provede syntaktickou analýzu a vrátí syntaktický strom daného kódu. V průběhu vývoje aplikace prošel tento syntaktický strom zásadní změnou, o které se ještě dále zmíním. Další na řadě je Evaluator, jehož funkce eval projde strom a vyhodnotí příkazy v něm obsažené.

Jak metoda repl, tak eval přijímají jako parametr environment. O tom, co to je environment a jaká je jeho architektura bude také pojednávat jedna z následujících kapitol.

4.3 Syntaktický strom

První verze syntaktického stromu byla složitější. Pro každou syntaktickou strukturu měl vlastní typ uzlu. Rozlišovalo se tedy, jestli jde například o definici funkce, definici proměnné, volání funkce, if konstrukt a tak dále. Syntaktický strom pro následující kód je ve své starší verzi vidět na obrázku 4.3 a v novější verzi na obrázku 4.4.

```
(+ 42 (if (< 3 10) 8 5))
```



Obrázek 4.3: Stará verze syntaktického stromu

V průběhu vývoje jsem zjistil, že například při použití quote se rozdíly mezi kódem a daty dosti stírají, a při syntaktické analýze kódu je obtížné určit, co bude v budoucnu třeba volání funkce, a co jen data.

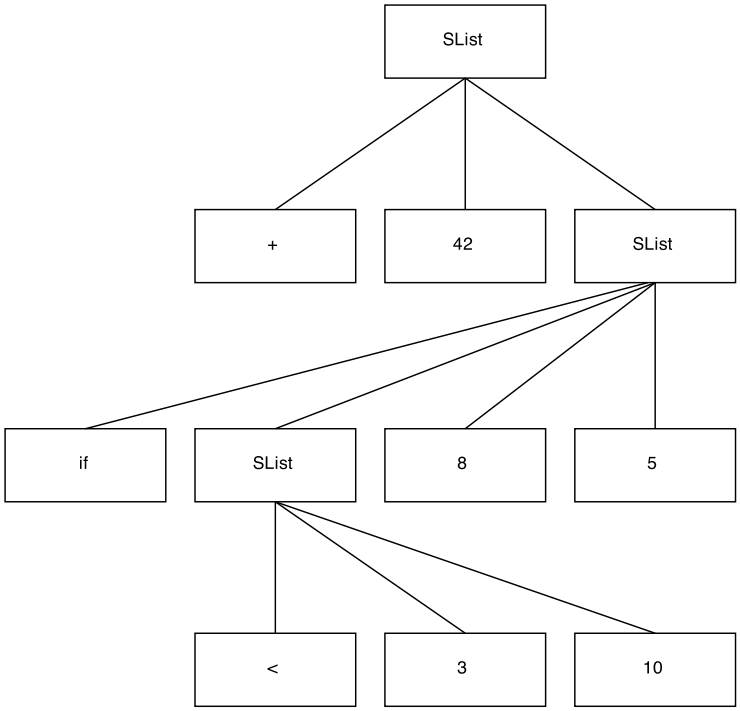
Rozhodl jsem se tedy syntaktický strom zjednodušit tak, aby používal jen základní struktury jazyka Scheme. Odpovědnost za určení, o jaký syntaktický konstrukt se jedná, jsem tak přesunul na Evaluator. Parser tak zkontroluje základní struktury kódu, jako jsou listy, symboly, textové řetězce a podobně. Například tvar if konstruktu, už ale musí zkontrolovat Evaluator.

Tato nová verze architektury syntaktického stromu je v souladu s filosofií Lispu, která umožňuje snadno zaměňovat kód a data, a pracovat s oběma, více či méně stejným způsobem.

Na obrázku 4.4 je vidět, že z nové verze stromu už není přímo poznat, zda se jedná o volání funkce nebo if. Strom je tvořen listy s nějakým obsahem, ale význam každého listu určí až Evaluator.

4.4 Typy a jejich hierarchie

Hierarchie typů je nejlépe vidět na obrázku 4.5. Na samém vrcholu je trait Form. Od něj pak všichni ostatní dědí. Třídy vpravo a Quote představují

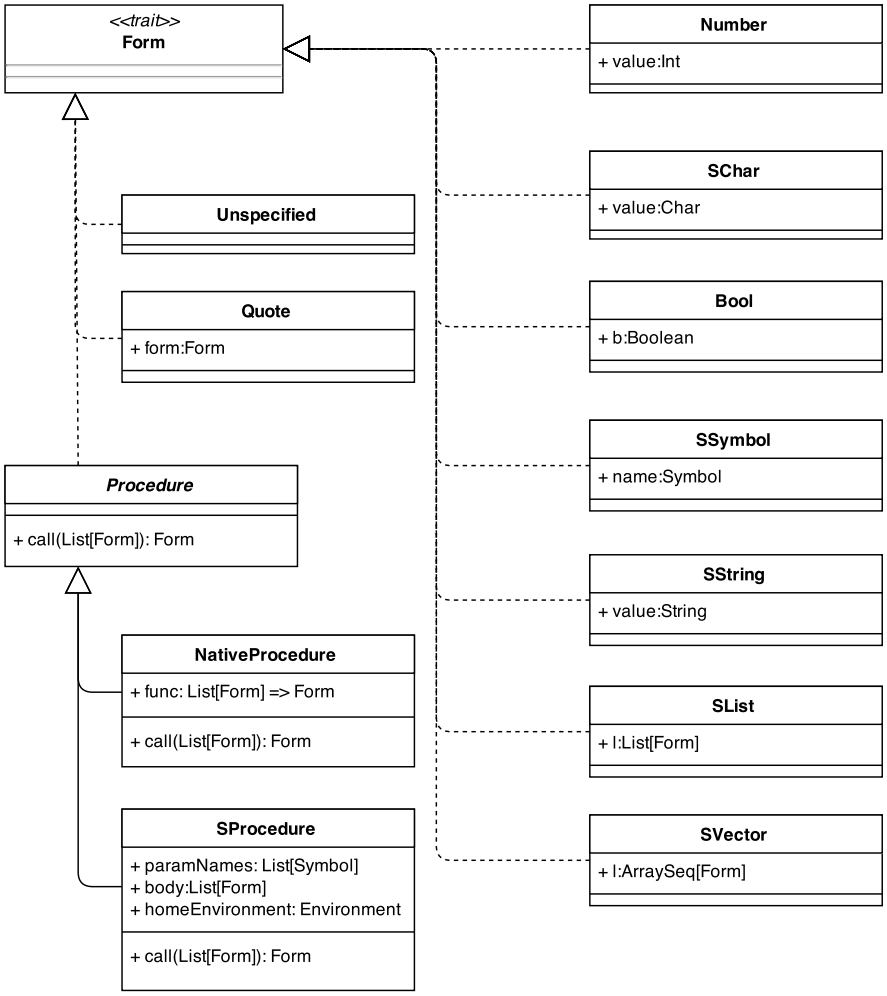


Obrázek 4.4: Nová verze syntaktického stromu

typy, které se mohou vyskytnout v syntaktickém stromu, který je výstupem z Parseru. Ostatní třídy se mohou vyskytnout jen jako výsledek vyhodnocení nějakého výrazu Evaluatorem. Quote by mohla být reprezentována listem jako třeba if, ale použít ji jako typ, je také korektní řešení. Třída Unspecified slouží jako hodnota, která je navrácena z procedur a syntaktických konstruktů „bez“ návratové hodnoty, jako je třeba definice nebo nastavení hodnoty do proměnné. Abstraktní třída Procedure představuje proceduru v již běžícím programu. Definuje jednu funkci call přijímající list hodnot typu Form. Tyto hodnoty představují parametry, se kterými je procedura volána. Třída má dva potomky SProcedure a NativeProcedure.

Třída SProcedure je procedura definovaná uživatelem programu. Vznikne vyhodnocením lambda výrazu. Třída má několik členských proměnných. První je list symbolů, představujících názvy parametrů procedury. Další členskou proměnnou je tělo funkce, uložené jako list hodnot Form. A poslední je domovský environment.

Třída NativeProcedure představuje nativní proceduru. Její jedinou členskou proměnou je funkce typu List[Form] => Form. To znamená funkce, přijímající jako parametr list hodnot typu Form a vracející typ Form. Je to přesně ta funkce, která se zavolá při volání nativní procedury. Třída NativeProcedure jí tedy jen obaluje.



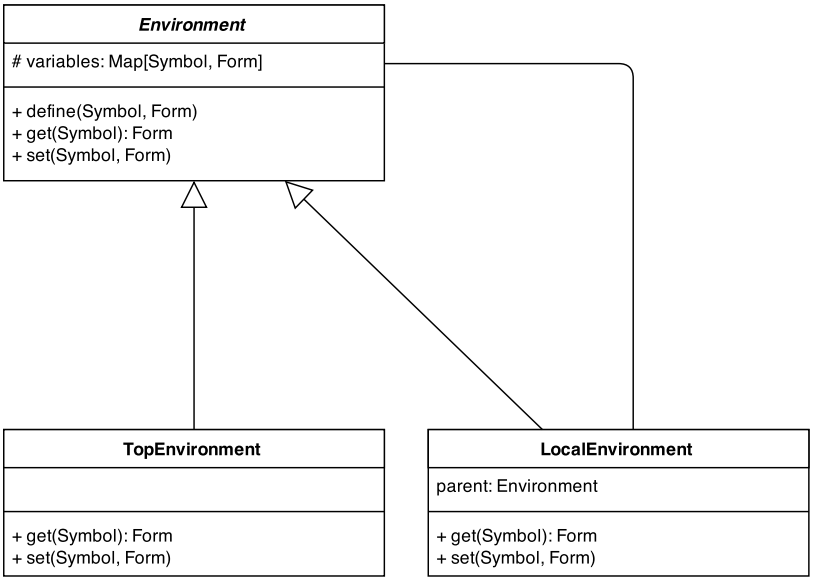
Obrázek 4.5: Hierarchie typů

4.5 Environment

Abstraktní třída Environment obsahuje mapu sloužící pro ukládání Lispových proměnných. Mapa jako klíč používá Symbol, a jako hodnotu Form. Tato mapa slouží jak pro definici proměnných, tak pro definici funkcí. Ve třídě jsou dále definované tři funkce, define, get a set. Jak je z jejich názvu zřejmé, používají se pro definici, čtení a zápis do paměťového místa, označeného daným symbolem. Od této abstraktní třídy dále dědí dvě třídy konkrétní, a sice TopEnvironment a LocalEnvironment.

TopEnvironment je, jak z názvu vyplývá, environment nejvyšší úrovně. Jsou v něm definovány všechny nativní funkce.

LocalEnvironment má jediný konstruktor s parametrem typu Environment. Tyto třídy tak tvoří jakýsi řetěz, kde na vrcholu je vždy TopEnvi-



Obrázek 4.6: Enviromenty

ronment a pod ním několik environmentů lokálních. Definice pak fungují vždy jen v rámci daného environmentu, ale metody get a set prochází celý řetězec nahoru, dokud nenaleznou definovanou proměnnou. Pokud ji nenaleznou ani v nejvyšším environmentu, tak se jedná o chybu.

Kapitola 5

Implementace

5.1 Parser

Pro syntaktickou a lexikální analýzu kódu je použita standardní Scala knihovna. Tato knihovna definuje interní doménově specifický jazyk, který se skládá ze speciálních funkcí, jež jsou ve Scale nazývány kombinátory. Kombinátor ve své podstatě kombinuje více jednodušších objektů do jednoho objektu složitého. Zde to jsou konkrétně pravidla gramatiky, jejichž kombinací vznikne kompletní definice bezkontextové gramatiky.

Jakým způsobem tento doménově specifický jazyk ve Scale funguje, už je nad rámec této práce. Bude zde tedy jen popsáno, jak kombinátory lze použít, a čeho tím lze dosáhnout, nikoliv jak Scala umožňuje takovýto jazyk vytvořit.

Následuje jednoduchý příklad demonstrující použití knihovny. Příklad obsahuje gramatická pravidla pro zápis boolovských hodnot a vektorů.

```
def hash = "#" ~ (vector | boolean)
def boolean = ("t" | "f")
def vector = "(" ~ rep(hash) ~ ")"
```

Každé pravidlo gramatiky je zapsáno jako funkce. Funkci pak lze použít v dalších pravidlech. Tabulka shrnuje kombinátory, které lze také použít pro zápis pravidel.

Tabulka 5.1: kombinátory

“text”	text - literál
“[abc]”.r	regulární výraz
A ~ B	B následuje po A
A | B	buďto A nebo B
rep(A)	opakovaný výskyt A

Pomocí těchto pravidel, lze snadno zapsat jakoukoliv bezkontextovou gramatiku. Výsledný parser skutečně dokáže ověřit, zda nějaký kód daná pravidla

splňuje. Co už ale nedokáže, je vytvořit pro něj použitelný syntaktický strom. Pro takový úkol je potřeba několik dalších funkcí.

Tabulka 5.2: pokročilé kombinátory

A <~ B	B následuje po A, vrať jen A
A ~> B	B následuje po A, vrať jen B
A ^^ B	nechť A vrací C pak B(C)

Pokud předchozí příklad rozšíříme o tyto operátory, lze již vytvořit kód, který převede vnější textovou reprezentaci na syntaktický strom, sestavený z námi zvolených objektů.

```
def hash = "#" ~> (vector | boolean)
def boolean = ("t" | "f") ^^ { case "t" => true; case "f" => false }
def vector = "(" ~> rep(hash) <~ ")"
```

Tento kód dokáže sestavit syntaktický strom reprezentující boolovské hodnoty a vektory pomocí Scalovských listů a objektů typu Boolean. Při syntaktické analýze boolovských hodnot, je použit operátor ^^ který aplikuje výsledek výrazu vlevo, jako parametr funkce vpravo. V příkladu je vpravo match výraz, který podle toho, o jaký se jedná textový řetězec, vrátí hodnotu true nebo false. Dalším, nově použitým operátorem, jsou vlnovkové šipky ~> a <~. Díky nim funkce vektor vrací jen výsledek kombinátoru rep. Obalující závorky se zahodí.

Následující kód používá objekt Parser pro syntaktickou a lexikální analýzu:

```
def code:Parser[Form] = (datum | quote | list | hash)
def datum = (number | string | symbol)
def hash = "#" ~> (vector | char | boolean)


def number:Parser[Number] = regex(new Regex("-?[0-9]+")) ^^
    (s~=> Number(s.toInt))

def boolean = ("t" | "f") ^^
    { case "t" => Bool(true); case "f" => Bool(false) }

def string = "\"" ~>
    """([^"\p{Cntrl}\\]|\\[\\'"bfnrt]|\\u[a-fA-F0-9]{4})*""".r <~
    "\"" ^^
    { case s~=> SString(s) }

def symbol = identifier ^^ {case v~=>
    SSymbol(Symbol(v.toLowerCase()))}

def identifier:Parser[String] = (identInitial | "+" | "-")
```

```
def identInitial =
    """[A-Za-z!$%&*/:<=>?~_^][A-Za-z!$%&*/:<=>?~_^0-9.+-]*""".r

def char = "\\" ~> """[^\s]+""".r ^^
    {case v~=> SChar.fromString(v)}

def list = "(" ~> listContent <~ ")" ^^ {case v~=> SList(v)}
def listContent:Parser[List[Form]] = rep(code) ^^
    (List() ++ _)

def quote = (quoteChar | quoteWord)
def quoteChar:Parser[Form] = "'" ~> code ^^ (s~=> Quote(s))
def quoteWord:Parser[Form] = "(" ~> "quote" ~> code <~ ")" ^^
    (s~=> Quote(s))

def vector = "(" ~> vectorContent <~ ")" ^^
    {case v~=> (SVector.constant(v))}
def vectorContent:Parser[ArraySeq[Form]] = rep(code) ^^
    (ArraySeq() ++ _)
```

5.2 Typy

Datové typy ze Scheme jsou reprezentovány pomocí case tříd. Díky tomu je lze snadno použít v match výrazu. Třídy obsahují pouze data, funkcionalita je minimální. Výjimkou je třída Procedure, ve které je definována metoda call, již její potomci implementují.

Většina typů ze Scheme, je reprezentována odpovídajícím typem ve Scale, obaleným case třídou. Například Scheme bool je vnitřně reprezentován jako case třída Bool, členskou proměnnou typu Boolean. Obalení case třídami zde umožňuje vytvoření syntaktického stromu.

5.3 Environment

Z Parseru zdrojový kód v podobě syntaktického stromu míří do Evaluatoru, který ho následně vykoná. K pochopení činnosti třídy Evaluator je však zapotřebí rozumět tomu, jak funguje environment. Proto je tato kapitola zařazena ještě před kapitolu o Evaluatoru.

Environment je třída zodpovědná za paměťové operace. Především se stará o vazby mezi jménem proměnné a hodnotou s ní svázanou. Tato vazba je implementována pomocí mapy, která používá jako klíč scalovský Symbol a její hodnoty jsou typu Form. Hlavní environment nebo jinak environment nejvyšší úrovně, představuje třída TopEnvironment. TopEnvironment je vytvořen při prvním spuštění smyčky repl a žádná jeho další instance se za běhu programu nevytvoří. LocalEnvironment je třída představující všechny ostatní environ-

menty v programu, kterých může být mnoho. Každý LocalEnvironment má referenci na environment vyšší úrovně. Na samotném vrcholu tohoto řetězu je tedy vždy TopEnvironment.

Kód ve Scheme se vyhodnocuje v kontextu aktuálního environmentu. Vyhodnocením některých příkazů se změní to, který environment je právě aktuální, ale vždy právě jeden aktuální je. V nejjednodušším případě existuje jen TopEnvironment, a ten zároveň je aktuálním environmentem.

Enviroment má tři základní operace: definice proměnné, čtení z proměnné a zápis do proměnné. Definice se provádí vždy v rámci aktuálního environmentu. Je chyba definovat dvakrát proměnnou se stejným jménem v rámci jednoho environmentu. Nastavení a získání hodnoty, s kterou je proměnná svázána, je složitější. Program nejprve hledá proměnnou daného jména v aktuálním environmentu. Pokud je úspěšný, provede požadovanou operaci. Pokud ne, postupuje řetězem environmentů nahoru až do environmentu nejvyššího. Pokud ani tam nenalezne hledanou proměnnou, dojde k chybě v programu.

5.4 Evaluator

Objekt Evaluator vyhodnocuje nebo jinak řečeno provádí kód. Nejdůležitější metoda v tomto objektu se nazývá eval a přijímá dva parametry. První z nich je hodnota typu Form neboli syntaktický strom. Druhým parametrem je environment, ve kterém se má kód provést.

Evaluator částečně přebírá roli syntaktického analyzátoru, protože syntaktický strom, který dostane z třídy Parser, se skládá jen z listů hodnot. Je na Evaluatoru, aby určil zda daný list je volání funkce nebo například podmíněný výraz. Evaluator tedy kromě provádění výrazů provádí i kontrolu jejich syntaxe. Ze standardu Evaluator implementuje primitivní a odvozené výrazy, tedy samotné jádro jazyka. Toto jádro však tvoří jen několik málo výrazů, mnohem větší část funkcionality jazyka je tvořena nativními procedurami. Evaluator však mezi nativními a uživatelem vytvořenými procedurami nerozlišuje, komunikuje s nimi jen pomocí rozhraní abstraktní třídy Procedure.

5.4.1 Vytvoření procedury

Při vytváření i volání procedur dochází k manipulacím s environmentem. Procedura jako taková nikdy nevznikne v Parseru. Ke vzniknu instance třídy Procedure dojde až při vyhodnocování syntaktického stromu Evaluatorem. Procedury mohou vzniknout buď vyhodnocením lambda výrazu nebo provedením kódu pro definici procedur. To druhé je však jen syntaktický cukr a interně se použije stejný kód jako pro prvé.

Instance třídy SProcedure si v okamžiku svého vytvořeni uloží tři věci. List s názvy svých parametrů, tělo procedury a jako třetí domovský environment; tedy ten, jenž byl aktuální v okamžiku vytvoření procedury. Domovský environment je důležitý zejména pro správnou funkci closures.

5.4.2 Volání procedury - aplikace

Aplikace vypadá v syntaktickém stromu jako jednoduchý list. Na prvním místě by měl být výraz, který vrátí proceduru, a zbytek listu obsahuje parametry pro její volání.

Nejprve se vyhodnotí všechny parametry. Poté se vyhodnotí první prvek listu, a pokud výsledek není procedura, vyvolá se výjimka. Následuje volání procedury pomocí funkce call s již vyhodnocenými parametry. V tomto místě se rozchází procedury nativní a uživatelem definované. Nativním procedurám bude věnovaná speciální kapitola, takže zbytek podkapitoly se bude věnovat jen procedurám definovaným uživatelem. Taková procedura nejprve zkontroluje, jestli odpovídá počet parametrů, a následně vytvoří novou instanci třídy LocalEnvironment. Jako environment vyšší úrovně je použit domovský environment uložený v okamžiku vytváření procedury. V nově vzniklém environmentu se definují všechny parametry procedury stejně jako by to byly normální proměnné a přiřadí se jim hodnoty, získané při vyhodnocování aplikace.

Posledním krokem volání procedury je vyhodnocení jejího těla v nově vzniklém environmentu a vrácení výsledku posledního výrazu jako výsledku celého volání.

5.4.3 Bloky let, let* a letrec

Let konstrukt poskytuje sémantiku vytváření nového environmentu, známou z volání procedur, ale bez nutnosti nějaké procedury vytvářet. Konstrukty let* a letrec jsou však lehce odlišné. Let příkaz vytvoří nový environment a pak zleva doprava vyhodnocuje všechny výrazy v původním environmentu. Výsledek každého výrazu je uložen do nového environmentu pod požadovaným názvem.

Příkazy let* a letrec jsou implementovány stejnou funkcí. Jediný rozdíl od let zde spočívá v tom, že vyhodnocení každého výrazu je provedeno pod nově definovaným environmentem. Stejný environment, v kterém je definovaná proměnná, je tedy použit i pro její vyhodnocení, díky tomu je možné, aby ostatní výrazy používaly ke svému výpočtu již vyhodnocené proměnné.

Nakonec je tělo bloku vyhodnoceno pod novým environmentem. Výsledek posledního výrazu v bloku je výsledkem celého bloku.

5.5 Nativní procedury

Při vytváření nejvyššího environmentu jsou do něj přidány všechny definované nativní procedury. Díky tomu jsou tyto procedury přístupné všude v kódu. Pokud si tedy uživatel nezadefinuje proceduru vlastní, se stejným jménem.

```
case class NativeProcedure(func: List[Form] => Form)
    extends Procedure {
  def call(params: List[Form]) = func(params);
}
```

Tato case třída slouží jako typ pro nativní procedury. Z kódu je vidět, že má jednu členskou proměnnou func, která je scalovskou funkcí přijímající parametr typu List[Form], a vracející hodnotu typu Form. Při volání procedury se pak zavolá tato vnitřní funkce.

Pro definování nativních procedur tak není potřeba vytvářet žádné nové třídy. Stačí vytvořit Scala funkci typu List[Form] => Form, dosadit jí do nové instance NativeProcedure, a tuto instanci vložit spolu s jejím jménem do nejvyššího environmentu.

Nativní funkce jsou definovány v balíčku cz.cvut.fit.cerveada.schela.natives. Funkce jsou rozděleny stejně jako ve standardu a pro každou skupinu je v balíčku jiný Scala object. Každý object obsahuje mapu typu Map[String, List[Form] => Form] pojmenovanou natives. Funkce je pak možné zapsat následujícím elegantním kódem.

```
// definice nativni procedury "not"
natives("not") = (params: List[Form]) => params match {
  case Bool(b) :: Nil => Bool(!b)
  case a :: Nil      => throw new UnexpectedType(a, Bool(true))
  case l             =>
    throw new UnexpectedNumberOfArguments(l.size, 1)
}
```

Tento kód vytvoří anonymní funkci a uloží ji do mapy natives s klíčem "not". Celá funkce je tvořena jedním match výrazem nad svými parametry. Pokud list parametrů obsahuje pouze instance case třídy Bool, dojde k vytvoření nové instance, která obsahuje negaci členské proměnné instance předchozí. Pokud jsou parametry jiné, dojde k chybě buď v důsledku neodpovídajícího typu parametru nebo v důsledku jejich počtu.

```
private type procedureType = List[Form] => Form
private type inMap =
    scala.collection.mutable.Map[String,procedureType]
private type outMap =
    scala.collection.mutable.Map[Symbol,NativeProcedure]

private def addVariables(map: inMap) {
  variables ++= toSymbolMap(map)
}
private def toSymbolMap(map: inMap): outMap = {
  map.map { case (k, v) => Symbol(k) -> NativeProcedure(v) }
}
```

Výpis z kódu ukazuje funkci pro přidání nativních funkcí do environmentu. Variables je mapa s definicemi proměnných v nejvyšším environmentu, jako klíč však nemá String, ale Symbol a jako hodnoty nemá funkce, ale instance NativeProcedure. Proto je zde funkce toSymbolMap, která umí převést mapu do správného tvaru. String je použit kvůli jednoduššímu zápisu nativních funkcí.

Následující příklad demonstruje možnosti Scaly na nativních funkcích pro porovnávání čísel:

```
natives("=") = comparatorCall((a, b) => a == b)
natives("<") = comparatorCall((a, b) => a < b)
natives(">") = comparatorCall((a, b) => a > b)
natives("<=") = comparatorCall((a, b) => a <= b)
natives(">=") = comparatorCall((a, b) => a >= b)
def comparatorCall(fun: (Int, Int) => Boolean)(params: List[Form]):
    Bool = {
  if (params.size < 2)
    throw new UnexpectedNumberOfArguments(params.size, 2)

  val values = params.map {
    case Number(v) =>
v~case t       => throw new UnexpectedType(t, Number(0));
  }
  val result = values.sliding(2).forall {
    case a :: b :: Nil => fun(a, b)
  }
  Bool(result)
}
```

Pro pochopení kódu je nejprve potřeba vysvětlit pojem currying. Je to transformace funkce, při které za některé parametry dosadíme hodnoty, a vznikne tak funkce nová s méně parametry. Ve Scale toto lze provést a funkce comparatorCall to umožňuje díky rozdělení parametrů do oddělených závorek. Při přidávání funkce do mapy je pak místo prvního parametru dosazen příslušný komparátor, a tím vznikne funkce v mapou požadovaném tvaru.

Tělo funkce provede nejprve kontrolu počtu argumentů a rozbalí instance case třídy Number. Tím vznikne list hodnot typu Integer. Funkce sliding(2) vybere pomocí posuvného okýnka vždy dva za sebou následující prvky. Například z (A, B, C) tak vznikne ((A, B), (B, C)). Funkce forall zase ověří, zda nějaký predikát platí pro všechny prvky kolekce. Zde se konečně uplatní funkce fun, na každé dva vybrané prvky. Celý výsledek je pak obalen do instance case třídy Bool a vrácen jako výsledek procedury.

Kapitola 6

Testy

6.1 Vývoj řízený testy

Anglicky též Test-driven development (TDD), je metodika pro vývoj softwaru s pomocí automatických jednotkových testů. Vývojový cyklus začíná specifikací požadavků a napsáním testu, který ověřuje, že program dané požadavky splňuje. Díky tomuto kroku vývojář musí porozumět požadavkům ještě předtím, než je začne vůbec implementovat. Následně vývojář zkontroluje, že testy neproběhnou úspěšně. Kód, který testují, ještě neexistuje, pokud by tedy proběhly úspěšně, je jasné, že se někde stala chyba. Dalším krokem je napsání kódu, který splňuje testy. V tuto chvíli program funguje jak má a je možné kód dále upravovat za účelem vyšší čitelnosti či omezení duplicit. Díky testům můžeme po každé změně ověřit, že kód stále funguje jak má.

Při vývoji tohoto virtuálního stroje byla použita primárně tato metodika. Ovšem s několika modifikacemi. Testy nelze označit za striktně jednotkové. Testovaný kód je zapsán v kódu jazyka Scheme a proto v rámci testu vždy probíhá i syntaktická analýza kódu a další operace s tím spojené. Je tomu tak proto, že ruční vytvoření syntaktického stromu by si vyžádalo mnohem delší kód testů a výsledné testy by ztratily svou stručnost a eleganci. Testy jsou tedy spíše než na jednu třídu jazyka Scala zaměřeny na jeden prvek jazyka Scheme. Například test nativní procedury not nebo test konstruktu if.

Pro každý syntaktický konstrukt jazyka, a pro každou nativní proceduru, byl nejprve napsán test, a až následně implementace. Program také prošel v průběhu vývoje řadou změn a díky testům bylo možné takové změny provádět bez velkého rizika zanesení nových chyb do kódu.

6.2 Testování pomocí knihovny ScalaTest

Knihovna ScalaTest je určena pro psaní testu v jazyce Scala. Definuje vlastní doménově specifický jazyk (DSL) pro psaní testů. Tento jazyk umožňuje psát kód způsobem podobným přirozenému jazyku.

```
abstract class UnitSpec extends FlatSpec with Matchers with
  OptionValues with Inside with Inspectors {

  def eval(code:String, environment:Environment) = {
    val syntacticTree = Parser.parseItem(code).get
    Evaluator.eval(syntacticTree, environment)
  }

  def resultOf(code:String, env:Environment) = eval(code, env)
}
```

Abstraktní třída UnitSpec slouží jako potomek pro všechny třídy s testy. Definuje metodu eval, která provede syntaktickou analýzu kódu a následně kód vykoná. Výsledný výraz je návratovou hodnotou metody. Tato metoda slouží pro snadný zápis testů. Metoda resultOf je pouze alias metody eval.

Třídy s testy jsou členěny identicky jako třídy s implementacemi nativních metod. Tedy podle rozdělení ze standardu jazyka Scheme.

```
"A lcm" should "return the least common multiple of two or more
    integers" in {

  val env = new TopEnvironment()

  an[UnexpectedNumberOfArguments] should be thrownBy eval("(lcm)",
      env)
  an[UnexpectedNumberOfArguments] should be thrownBy eval("(lcm 6)",
      env)
  an[UnexpectedType] should be thrownBy eval("(lcm 8 'a 5)", env)

  resultOf("(lcm 56 42)", env) should be(Number(168))
  resultOf("(lcm 666 128)", env) should be(Number(42624))
  resultOf("(lcm 1024 997)", env) should be(Number(1020928))
  resultOf("(lcm 36 27 45 81 )", env) should be(Number(1620))
}
```

Kód z příkladu ukazuje test nativní procedury lcm. Nejprve je testováno chování v případě chyby, tedy že procedura vyvolá výjimku. Následně se testuje, zda procedura vrací správné výsledky.

6.3 Testy dané standardem jazyka Scheme

Standard jazyka Scheme často uvádí spolu s definicí procedur a struktur i příklady jejich použití. Tyto příklady jsou často zaměřené na různé mezní případy. Pokud to bylo možné, byly ukázky kódu ze standardu použity jako testy. Soulad implementace jazyka s normou je tedy ověřen i mnoha automatickými testy.

Kapitola 7

Možná rozšíření

Množství testů vytváří ideální podmínky pro další rozšiřování a refaktorizaci programu. Pokud se budeme držet použitého standardu jazyka Scheme, je možné program dále rozšiřovat o nativní procedury, které nebyly implementovány. Jedná se například o procedury pro kontinuace, některé procedury pro listy, textové řetězce a několik dalších procedur. Dalším vhodným rozšířením by byla dle standardu korektní implementace koncové rekurze. Standard dále specifikuje několik prvků jazyka, které nejsou povinné. Jedná se o čísla s plovoucí řádovou čárkou, velká celá čísla a také o podporu maker.

Další možné rozšíření spočívá v převodu programu na vyšší verzi standardu. V současné době je poslední ratifikovanou verzí standardu verze sedmá. Za zvážení stojí i možnost přidání nativní podpory pro objektové paradigma.

Závěr

Rešerše virtuálních strojů a dialektů jazyka Lisp byla provedena. Jako vhodný byl zvolen jazyk Scheme a jeho standard ve verzi čtyři. Podařilo se implementovat virtuální stroj, interpretující jazyk Scheme, v souladu se zvoleným standardem. Program byl průběžně testován automatickými testy. Zadání práce bylo splněno.

Díky této práci jsem získal řadu zkušeností. Rešerše virtuálních strojů mi pomohla uvědomit si širší souvislosti a porovnat stroje pro platformy Java, .Net a jazyk Pascal. Pozitivní zkušenost mám s vývojem v jazyku Scala, ve kterém bych rád v budoucnu napsal i větší projekt. Jazyk Scheme jsem díky studiu standardu a samotné implementaci poznal skutečně zblízka. Celkově přínos práce hodnotím kladně.

Literatura

[1] Pascal Implementation. [online]. [cit. 2015-01-19]. Dostupné z: `http://homepages.cwi.nl/~steven/pascal/book/pascalimplementation.html`

[2] Revised(4) Report on the Algorithmic Language Scheme. [online]. [cit. 2015-03-01]. Dostupné z: `http://www.cs.cmu.edu/Groups/AI/html/r4rs/r4rs_toc.html`

[3] Lindholm, T.: *The Java virtual machine specification*. Java se 7 edition / vydání, ISBN 01-332-6044-5.

[4] Odersky, L. S. M.: *Programming in Scala*. Walnut Creek, Calif: Artima, druhé vydání, 2010, ISBN 09-815-3164-4.

[5] Smith, J.: *Virtual machines*. Amsterdam: Elsevier, 2005, ISBN 15-586-0910-5.

[6] Sperber, M.: *Revised (6) report on the algorithmic language scheme*. New York: Cambridge University Press, c2009, ISBN 05-211-9399-0.

Příloha **A**

Seznam použitých zkratek

ABI Application binary interface

API Application programming interface

CLI Common Language Infrastructure

CLR Common Language Runtime

DRY Don't repeat yourself

DSL Domain-specific language

HLL VM High Level Language Virtual Machine

I/O Input/Output

ISA Instruction Set Architecture

JVM Java Virtual Machine

MSIL Microsoft Intermediate Language

REPL Read–eval–print loop

TDD Test-driven development

VM Virtual machine

Příloha **B**

Obsah přiloženého CD

```
readme.txt ..................................... stručný popis obsahu CD
exe ....................... adresář se spustitelnou formou implementace
src
  impl ..................................... zdrojové kódy implementace
  thesis ...................... zdrojová forma práce ve formátu LaTeX
text ..................................................... text práce
  thesis.pdf ........................... text práce ve formátu PDF
```