VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

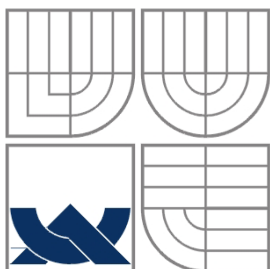
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

INTERPRET DYNAMICKÉHO PROGRAMOVACÍHO JAZYKA PRO VĚDECKÉ VÝPOČTY

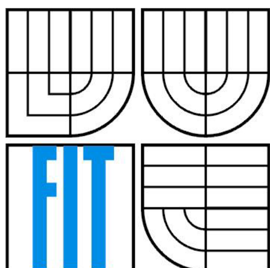
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. TOMÁŠ OCELÍK
AUTHOR

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

INTERPRET DYNAMICKÉHO PROGRAMOVACÍHO JAZYKA PRO VĚDECKÉ VÝPOČTY

INTERPRETER OF A DYNAMIC PROGRAMMING LANGUAGE FOR SCIENTIFIC COMPUTING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Tomáš Ocelík
AUTHOR

VEDOUCÍ PRÁCE Ing. Jan Kněžík
SUPERVISOR

BRNO 2012

Abstrakt

Práce se zabývá návrhem a popisem dynamického reflektivního jazyka, založeného na prototypování. Nejprve jsou vysvětleny principy typické pro tuto skupinu jazyků a jsou stručně popsáni známí představitelé. Dále je krátce pojednáno o jazycích pro matematické výpočty. Poté práce podrobně popisuje navržený programovací jazyk, jeho gramatiku a sémantiku. Jsou vysvětleny principy typové kontroly a dědičnosti. Je také ukázáno, jakým způsobem jsou implementovány základní řídící konstrukce známé z jiných jazyků. V další části je představen návrh virtuálního stroje pro vytvořený jazyk. Je vysvětlen použitý výpočetní model, organizace objektové paměti a interní reprezentace význačných struktur navrženého jazyka. Nakonec je rozebrána dynamická typová kontrola, překladač a způsob překladu typických konstrukcí do vnitřního kódu virtuálního stroje.

Abstract

The master‘s thesis deals with design of a dynamic reflective prototype-based language. First, basic principles of this language group are explained and well known representatives are described. Then languages for scientific computing are shortly discussed. Next section of the thesis describes in detail the proposed programming language, its grammar and semantics. Principles of type checking and inheritance are explained. Thesis also demonstrates implementation of basic control structures known from other languages. Next section shows design of virtual machine for the language described before. Section explains used computational model, organization of the object memory and internal representation of important structures of the designed language. Finally, dynamic type checking, compiler and compilation of typical structures to the virtual machine internal code are discussed.

Klíčová slova

Dynamický programovací jazyk, virtuální stroj, objektové orientované jazyky, jazyky založené na prototypech, paralelismus, dynamická typová kontrola, překladač.

Keywords

Dynamic programming language, virtual machine, object-oriented programming languages, prototype-based languages, parallelism, dynamic type checking, compiler.

Citace

Ocelík Tomáš: Interpret dynamického programovacího jazyka pro vědecké výpočty, diplomová práce, Brno, FIT VUT v Brně, 2012

INTERPRET DYNAMICKÉHO PROGRAMOVACÍHO JAZYKA PRO VĚDECKÉ VÝPOČTY

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Jana Kněžíka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Tomáš Ocelík
23. 5. 2012

Poděkování

Děkuji svému vedoucímu, panu Ing. Kněžíkovi za kritické připomínky týkající se zejména návrhu jazyka a architektury virtuálního stroje.

© Tomáš Ocelík, 2012
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah 1

1 Úvod 4

2 Objektově orientované jazyky 6

2.1 Základní koncepty OOP 6

2.1.1 Abstrakce 6

2.1.2 Zapouzdření 6

2.1.3 Dědičnost 7

2.1.4 Polymorfismus (mnohotvarost) 7

2.2 Třídně orientované jazyky 8

2.3 Prototypově orientované jazyky 9

2.3.1 Delegování 9

2.3.2 Rysy a prototypy 10

2.3.3 Rušení objektů 10

2.4 Existující objektově orientované jazyky 11

2.4.1 Smalltalk 11

2.4.2 Self 13

2.4.3 IO 15

3 Jazyky pro vědecké výpočty 17

3.1 Octave 17

3.1.1 Vektory a matice 17

3.2 Maxima 18

3.3 Sage 18

3.4 R 19

4 Návrh programovacího jazyka 20

4.1 Základní principy 20

4.1.1 Prostředí jazyka 20

4.1.2 Vznik a rušení objektů 21

4.1.3 Typy slotů 21

4.2 Definice navrženého jazyka 22

4.2.1 Literály a vestavěné typy 22

4.2.2 Zprávy 26

4.2.3 Dědičnost 29

4.2.4 Datové typy 29

4.2.5 Metody 30

4.2.6 Prototypové objekty ... 31
4.2.7 Koncept jmenných prostorů ... 32
4.2.8 Asynchronní zprávy ... 34
4.2.9 Gramatika jazyka ... 34
4.3 Implementace řídících konstrukcí ... 36
4.3.1 Větvení ... 36
4.3.2 Cykly ... 37
5 Návrh a realizace virtuálního stroje ... 39
5.1 Základní koncept ... 39
5.2 Objektová paměť ... 40
5.2.1 Implementace objektové paměti ... 40
5.2.2 Objektový model ... 40
5.2.3 Sloty ... 41
5.2.4 Tabulka slotů ... 43
5.2.5 Primitivy ... 43
5.2.6 Prototypové objekty pro vestavěné datové typy ... 44
5.2.7 Matice a pole ... 44
5.2.8 Klonování objektů ... 45
5.2.9 Perzistentní obraz ... 45
5.3 Interpret vnitřního kódu ... 47
5.3.1 Interní reprezentace metod a bloků ... 47
5.3.2 Běhová reprezentace metod a bloků kódu ... 48
5.3.3 Vytvoření nové metody ... 50
5.3.4 Vytvoření dvourozměrných struktur z literálů ... 51
5.3.5 Registry a zásobník ... 51
5.3.6 Popis základních instrukcí ... 52
5.3.7 Typy a volání metod z hlediska VM ... 53
5.3.8 Průběh interpretace ... 55
5.3.9 Realizace řídících struktur ... 57
5.4 Překladač ... 58
5.4.1 Překlad a interní reprezentace programu ... 59
5.4.2 Generátor bytekódu ... 60
5.4.3 Překlad literálů ... 62
5.4.4 Řízení překladu ... 62
5.5 Inicializace a interaktivní režim ... 63
6 Demonstrační příklady ... 65
6.1 Návrhové vzory ... 65

6.1.1 Singleton 65
6.1.2 Adaptér 65
6.1.3 Návštěvník 66
6.2 Ukázky algoritmů 67
6.2.1 Faktoriál 67
6.2.2 Výpočet kořenů kvadratické rovnice 67
7 Závěr 69

1 Úvod

Objektově orientované programování a objektově orientované jazyky představují moderní a stále se rozvíjející směr v oblasti programovacích jazyků. Objektové paradigma oproti předchozím paradigmatům přináší některé nesporné výhody a značně zjednodušuje návrh software. Na softwarový systém již nenahlížíme jako na soubor dat a kolekci funkcí, které s těmito daty operují, ale nahlížíme na něj jako na soubor objektů, které mají určitý stav a množinu operací, jež tento stav mohou měnit. Takové objekty pak mohou korespondovat s nějakými objekty reálného světa, čímž je návrh a implementace programu srozumitelnější lidskému chápání.

Nespornou výhodou objektů je také jejich znovu použitelnost. Objekt sloužící určitému účelu, který má dobře definované rozhraní, může být použit ve formě komponenty v jiném programu.

Koncept objektově orientovaného programování se objevil již v 60. letech u jazyka *Simula*, který byl určen primárně do oblasti simulací. Zde se poprvé objevily pojmy jako *třída* a *objekt*. Brzy se ukázalo, že tento přístup je použitelný obecně. Dalším jazykem z této skupiny byl *Smalltalk*, ve kterém byly poprvé použity pojmy *zasílání zpráv* nebo *dědičnost*. V 80. letech byl vyvinut jazyk *Self*, který vychází ze Smalltalku a jedná se o první jazyk založený na prototypech (*prototype-based*). Ve stejné době probíhal i vývoj jazyka C++, jenž se stal prvním skutečně rozšířeným objektově orientovaným jazykem. V devadesátých letech se pak objevil nyní již velmi populární jazyk *Java*, který vychází z C++ a ze Smalltalku a představuje jakýsi kompromis mezi dynamičností Smalltalku a syntaxí, inspirovanou jazykem C a zejména C++. Z ještě novějších jazyků můžeme zmínit jazyk *Io*, který vychází ze Selfu, vyznačuje se velmi jednoduchou syntaxí bez klíčových slov a velkou přenositelností.

Objektové paradigma se také velmi uplatnilo u moderních modelovacích technik jako je jazyk UML, který rovněž pracuje s pojmy objekt nebo třída, dědičnost či zasílání zpráv.

Cílem této práce bylo navrhnout reflexivní dynamický programovací jazyk, který by byl vhodný pro vědecké výpočty, poskytoval prostředky pro modelování paralelismu a prováděl dynamickou typovou kontrolu. Navrhnuli jsme jazyk, jenž je svou syntaxí velmi podobný jazyku C, ale ideově vychází z konceptů použitých v jazycích Self, Java nebo u výše zmíněného Io. Pro jazyk byl poté vytvořen návrh virtuálního stroje a interpret pro příkazovou řádku, který bude umět spuštěný program ukládat na disk do perzistentního image.

Práce nejprve představuje některé základní koncepty typické pro objektově orientované jazyky, vysvětluje rozdíl mezi prototypově založenými (*prototype-based*) a třídně založenými (*class-based*) jazyky. Jsou vysvětleny pojmy *dědičnost* nebo *polymorfismus*. U prototypově založených jazyků jsou dále podrobněji popsány jazyky *Smalltalk*, *Self* a *Io*, u kterých jsme se inspirovali.

Třetí kapitola stručně popisuje některé jazyky určené pro matematické výpočty. Jsou vysvětleny základy syntaxe a některá specifika, jimiž se tato skupina jazyků vyznačuje a jimiž jsme se inspirovali při návrhu vlastního jazyka.

Čtvrtá kapitola podrobně specifikuje navržený jazyk, jeho gramatiku, sémantiku. Jsou představeny základní principy jazyka, způsob vytváření nových objektů a práce s nimi. Je popsán koncept dědičnosti a typové kontroly. Kapitola dále popisuje základní výpočetní prostředí jazyka a implementace typických řídících konstrukcí známých z jiných jazyků.

Kapitola pět se zabývá popisem návrhu a implementace virtuálního stroje pro vytvořený jazyk. Ukazuje základní blokovou architekturu celého programu a popisuje jeho jednotlivé části a jejich účel. Je představen výpočetní model z pohledu jazyka a překladače. Dále je popisován návrh objektové paměti, způsob interpretace bytekódu, vyhodnocování výrazů, postup při volání metod či vestavěných primitiv.

Šestá kapitola na jednoduchých příkladech ukazuje možnosti jazyka a jeho schopnosti při implementaci návrhových vzorů.

Poslední, sedmá kapitola shrnuje výsledky práce a prezentuje možnosti dalšího rozšíření.

2 Objektově orientované jazyky

Objektově orientované jazyky vycházejí ze strukturálních jazyků. Na rozdíl od nich ale sdružují data a operace nad nimi do jednoho celku zvaného *objekt*. Data jsou pak chápána jako stav objektu a operace, nazývající se *metody,* tento stav nějakým způsobem mění.

Objekty spolu komunikují zasíláním zpráv, přičemž zpráva může nést argumenty. Objekt, který přijme zprávu, na ni může nějakým způsobem reagovat, pokud jí rozumí. Způsob reakce je však závislý čistě na tomto objektu a více objektů může na stejnou zprávu reagovat různě. Program pak chápeme jako systém komunikujících objektů, které spolupracují pro dosažení nějakého cíle. Následující kapitola popisuje základní koncepty objektově orientovaného programování.

2.1 Základní koncepty OOP

U objektového programování rozlišujeme čtyři základní koncepty. Jedná se o *abstrakci, zapouzdření, polymorfismus* a *dědičnost*. Jejich implementace závisí do značné míry na konkrétním jazyce.

2.1.1 Abstrakce

Abstrakce souvisí především s objektově orientovaným návrhem. Jak již bylo zmíněno v úvodu, objekty se dají použít k modelování reálného světa. Prakticky nikdy však nepotřebujeme modelovat všechny vlastnosti a stavy, kterých může reálný objekt nabývat, ale modelujeme pouze to, co je pro výpočet podstatné a ostatní zanedbáme. Objekt potom funguje jako jakási černá skříňka, která před okolím skrývá nepodstatné detaily své implementace a poskytuje směrem k okolí určité rozhraní. Uživatelé objektu potom nemusí mít žádné informace o tom, jakými algoritmy a datovými strukturami je činnost objektu realizována.

2.1.2 Zapouzdření

Jak již bylo řečeno, objekt nese určitý stav, vyjádřený hodnotami jeho atributů. Pro správné fungování programu je však vždy nutné, aby tyto interní atributy objektu byly konzistentní – aby byl objekt v nějakém korektním, dobře definován stavu. Musí se proto zajistit, aby atributy objektu nešly měnit kýmkoliv a jakýmkoliv způsobem, ale aby za jejich změnu zodpovídal pouze daný objekt. Okolí potom může stav objektu měnit pouze přes jeho rozhraní. Implementace takového rozhraní pak může zajistit, že vnitřní stav objektu se bude měnit pouze definovaným způsobem.

Koncept zapouzdření tedy napomáhá skrývat před okolím způsob reprezentace konkrétního stavu objektu a zároveň skrývat implementaci konkrétních algoritmů. Okolí vidí pouze definované veřejné rozhraní, přes které s objektem komunikuje. Toto do značné míry zjednodušuje implementaci nových vlastností nebo změnu stávajícího chování – stačí pouze dodržet rozhraní s okolím.

Implementaci je pak možné dokonce i zcela změnit tím, že se použitý objekt vymění za jiný, definující stejné rozhraní. Pokud má objekt rozhraní dobře specifikováno, je možné jej obvykle použít i v jiných programech. Koncept zapouzdření potom velmi dobře umožňuje splnit i princip znuvupoužitelnosti známý ze softwarového inženýrství.

Zapouzdření mívá obvykle podporu přímo v objektově orientovaných programovacích jazycích. Ty často obsahují nějaký mechanismus nastavení viditelnosti, pomocí kterého lze určit, které vlastnosti a metody budou přístupné vně objektu a budou tvořit jeho veřejné rozhraní.

2.1.3 Dědičnost

Dědičnost umožňuje implementovat *sdílené chování*. Objekt, který je potomkem nějakého jiného objektu, dědí všechny jeho vlastnosti, chování a přidává k němu svá vlastní specifika, případně předefinovává některé zděděné vlastnosti.

Dědičnost může být buď *jednoduchá*, kdy bude objekt mít maximálně jednoho předchůdce, nebo *vícenásobná*, má-li objekt více rodičů. U vícenásobné dědičnosti je však nutné řešit konflikty jmen, kdy dva různí rodiče implementují reakci na stejnou zprávu. Různé jazyky toto řeší různým způsobem.

Lze takto vytvářet celou hierarchii dědičnosti, kdy na jejím vrcholu bude nějaký obecný objekt, implementující základní chování společné pro všechny objekty a jeho potomci se budou tím více specializovat pro konkrétní oblast a účel, čím níže v hierarchii budou. Je zřejmé, že hierarchie dědičnosti tvoří buď strom v případě jednoduché dědičnosti, nebo acyklický orientovaný graf případě vícenásobné dědičnosti. Obecně platí, že dědičnost nemůže vytvářet cyklus a objekt tedy nemůže být sám sobě rodičem i potomkem.

Dědičnost také souvisí s datovým typem objektu. Objekt má kromě svého datového typu i datové typy všech svých předchůdců. Je tedy možné objekt jistého datového typu přiřadit i do proměnné, která má datový typ jeho předka. Pokud se však k tomuto objektu přistupuje prostřednictvím takové proměnné, jsou viditelná pouze rozhraní definovaná objektem odpovídajícím datovému typu proměnné a rozhraní definovaná jeho rodičovskými objekty (podle stejného principu). Je tomu tak právě proto, že objekt obsahuje jednak své rozhraní a dále rozhraní svých předchůdců. Pak je zajištěno, že budeme-li pracovat s objektem přes takovou proměnnou, bude dané sadě zpráv rozumět. Proměnná však zároveň neříká nic o konkrétním podtypu objektu a není tedy možné využívat specifické rozhraní dané konkrétním objektem – datovým typem.

2.1.4 Polymorfismus (mnohotvarost)

Polymorfismus souvisí s mechanismem zasílání zpráv. Jak již bylo řečeno, objekty spolu komunikují zasíláním zpráv a reakce objektu na danou zprávu je čistě v jeho režii. V mnoha případech máme jeden obecnější objekt, který rozumí jisté zprávě a dále pak několik jeho potomků, kteří na tuto

zprávu reagují obecně různě, a to způsobem pro ně specifickým. Můžeme zde využít skutečnosti, že objekt určitého datového typu je možné přiřadit do proměnné datového typu jeho předka (viz předchozí kapitola). Takto přiřazenému objektu můžeme zmíněnou zprávu poslat, aniž bychom řešili, který konkrétní objekt je aktuálně do proměnné přiřazen, jaký slot bude nalezen, a tedy jaká konkrétní akce se provede.

Jako ilustrační příklad můžeme uvést objekty modelující geometrické tvary, u kterých máme spočítat jejich obsah. Je zřejmé, že například obsah kruhu bude počítán jiným způsobem a z jiných parametrů, než obsah čtverce. Všechny geometrické tvary však mají společné vlastnosti (obsah, obvod, počet vrcholů a podobně), můžeme proto využít principu dědičnosti a mít jeden objekt modelující obecný geometrický tvar, který bude rozumět zprávě *spocti_obvod.* Potomci modelující konkrétní geometrické tvary pak budou na tuto zprávu reagovat jim odpovídajícím způsobem.

2.2 Třídně orientované jazyky

Často potřebujeme mít více objektů, které mají stejné chování – rozumí stejné množině zpráv, jež jsou i stejně implementovány. O takových objektech říkáme, že jsou stejné třídy. Odtud plyne název pro *třídně orientované jazyky.* Takový jazyk obsahuje speciální objekty – *třídy*. Třídy fungují jako jakési šablony pro vytváření nových objektů a obsahují ono sdílené chování a vlastnosti. Vytváření nových objektů obvykle probíhá pomocí speciální jazykové konstrukce. Tomuto procesu se říká *instanciace* a v různých jazycích je realizována různě. Objektům pak někdy říkáme *instance*.

Jako příklad zde můžeme uvést automobil, které má různé vlastnosti – objem motoru, maximální rychlost, počet sedadel nebo množství paliva v nádrži. Takový automobil bychom mohli v nějakém hypotetickém programu modelovat jako třídu, jejíž atributy by odpovídaly uvedeným vlastnostem. Jako její operace by se pak daly považovat například *nastartovat*, *zrychlit, natankovat* a podobně.

Konkrétní automobil, vyskytující se v reálném světě, by pak odpovídal konkrétnímu objektu (instanci) v paměti, přičemž jeho modelované vlastnosti by odpovídaly vlastnostem tohoto objektu.

Objekty vytvořené na základě jedné třídy tedy sdílejí společné chování implementované pomocí metod, jež jsou součástí definice třídy. Jednotlivé instance se pak liší pouze uloženými daty reprezentujícími stav daného objektu a jsou v paměti uložena jako součást takového objektu v podobě proměnných. Je rovněž možné specifikovat, že nějaká data budou přímo součástí třídy, protože modelují vlastnost, kterou mají všechny objekty dané třídy stejnou. Takovým proměnným říkáme *statické* a jejich změna u třídy je viditelná u všech objektů dané třídy.

Vztah dědičnosti u těchto jazyků není realizován mezi konkrétními objekty, ale mezi třídami. Typickými představiteli jazyků této kategorie jsou v úvodu zmíněné C++ nebo Java.

2.3 Prototypově orientované jazyky

Jazyky patřící do této skupiny nepoužívají žádný speciální typ objektu jako šablonu pro vytváření jiných objektů. Všechny objekty takového jazyka mají obvykle stejnou strukturu a obsahují typicky seznam dvojic „název – hodnota", kterým říkáme *sloty*. Název slotu odpovídá stejnojmenné zprávě a hodnotou je buď odkaz na jiný objekt, nebo metoda implementující chování. Každý objekt prototypově orientovaného jazyka tedy může obsahovat jak data, tak i chování. Této skupině jazyků také říkáme *čistě objektově orientované*.

Důležité je zde zmínit, že obsah každého slotu je možné za běhu měnit, což platí i pro sloty obsahující metody. Díky tomu je dosaženo dynamičnosti, která se u třídně orientovaných jazyků nevyskytuje a program napsaný pomocí prototypově orientovaného jazyka může za běhu modifikovat sám sebe ve smyslu změn svého kódu. Jazyky této skupiny jsou obvykle klasifikovány jako jazyky *dynamické*.

Kromě možnosti měnit jakýkoliv slot tyto jazyky poskytují také prostředky pro zkoumání slotů daného objektu a dalších jeho vlastností za běhu programu. Ten tedy může v konečném důsledku zkoumat a měnit svou vlastní strukturu. Takové jazyky označujeme jako *reflexivní*, přičemž rozlišujeme strukturální a výpočetní reflexi podle toho, zda je možné zkoumat pouze data nebo i výpočetní stav programu.

Nové objekty na rozdíl od třídně orientovaných jazyků nevznikají instanciací, ale *klonováním* existujících objektů. Takto vzniklý objekt je u některých jazyků přesnou kopií objektu původního.

Při zasílání zprávy určitému objektu se prohledávají sloty tohoto objektu a hledá se slot se stejným názvem. Pokud je nalezen, je podle typu slotu buď vrácen jeho obsah – reference na nějaký jiný objekt, nebo je provedena odpovídající metoda. Pokud slot nalezen nebyl, je v závislosti na jazyce buď oznámena chyba, nebo případně provedena nějaká předem definovaná akce.

2.3.1 Delegování

Protože prototypově orientované jazyky neobsahují třídy, nemají ani žádný explicitní způsob definice vztahu dědičnosti. Místo toho je zaveden obecnější koncept zvaný *delegování*, pomocí nějž lze dědičnost realizovat.

Každý objekt má speciální *rodičovský slot* obsahující referenci na jiný objekt, na který může *delegovat* zprávy, jimž sám nerozumí. Výše popsaný proces reakce na zprávu pak probíhá tak, že po té, co není nalezen slot odpovídající dané zprávě u cílového objektu, pokračuje hledání u objektu odkazovaného pomocí rodičovského slotu. Pokud není nalezen ani tam, pokračuje se přes jeho rodičovský slot stejným způsobem dále, dokud není odpovídající slot nalezen, nebo hledání dosáhne objektu, který má rodičovský slot prázdný. Pak je ohlášena chyba nebo je dána programátorovi možnost specifikovat, co se v takovém případě má stát.

Nalezený slot není vyhodnocen v kontextu svého objektu, ale v kontextu původního objektu, který byl cílem zprávy. Tím se dosáhne stejného efektu, jako u klasické dědičnosti u třídně orientovaných jazyků. Protože i rodičovské sloty je možné libovolně měnit (pokud nevznikne cyklus), lze takto dosáhnout jevu zvaného *dynamická dědičnost*.

2.3.2 Rysy a prototypy

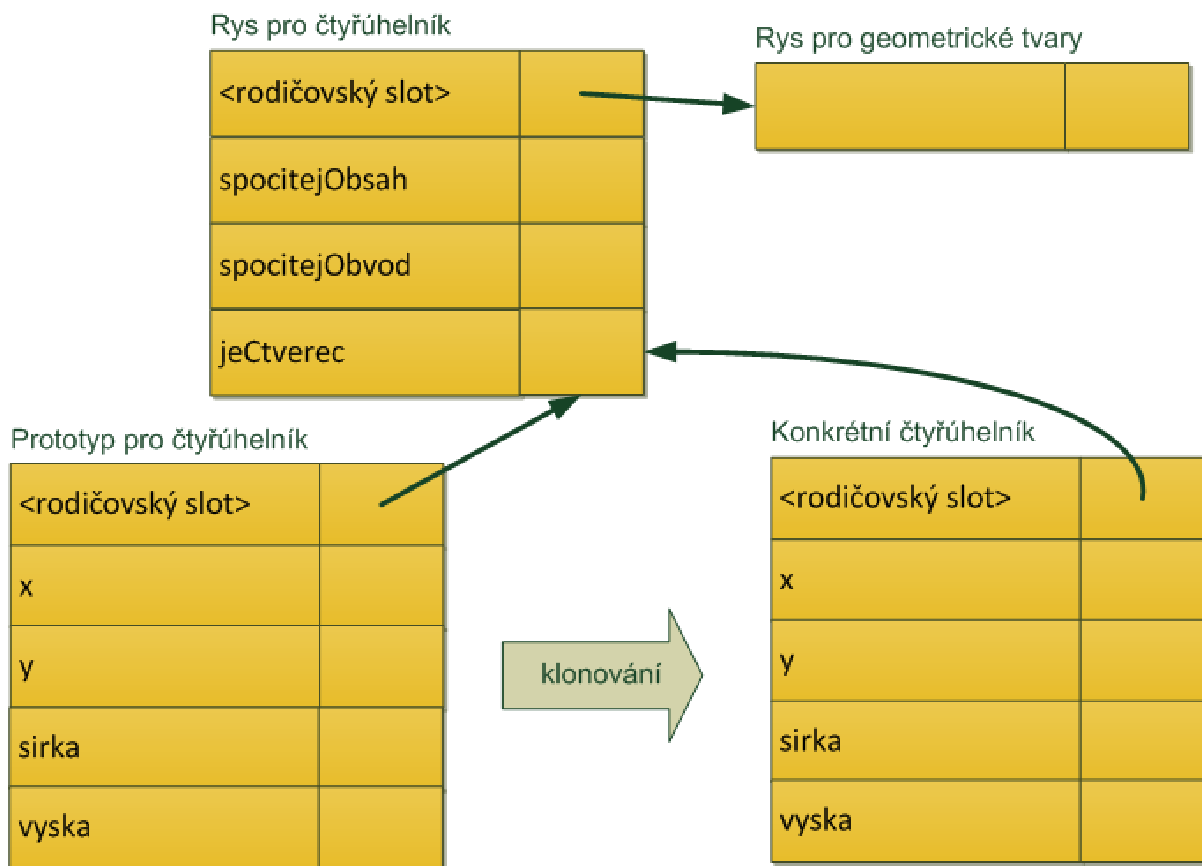
Rysy jsou obyčejné objekty, které však implementují sdílené chování určité skupiny objektů a na které jsou potom delegovány zprávy určené těmto objektům. Rys typicky obsahuje pouze sloty s metodami. Může však obsahovat i sloty s daty. Taková data pak budou díky delegování rovněž sdílena všemi objekty dané množiny. Provážeme-li objekty figurující jako rysy tak, že více specializovaný objekt bude mít ve svém rodičovském slotu odkaz na obecnější objekt, vytvoříme hierarchii dědičnosti rysů. Pokud jazyk dovoluje mít více rodičovských slotů u jednoho objektu, je možné vytvořit i vícenásobnou dědičnost. Rysy tedy fungují jako úplná náhrada za třídy z třídně orientovaných jazyků.

Nové objekty by mohly vznikat už klonováním rysů. Tím by se však ztratil efekt sdíleného chování. Naklonovaný objekt by byl přesnou kopií rysu a obsahoval by tedy i definice metod a společných vlastností. Kromě toho by takové řešení bylo náročné i na paměť. Spolu s rysem proto obvykle existuje ještě jeden nebo více objektů, které mají jako rodičovský slot nastaven objekt rysu a na rozdíl od něj obsahují pouze sloty, jež mají být pro každý objekt specifické. Takové objekty nazýváme *prototypy*. Objekty, které mají reprezentovat konkrétní objekty reálného světa, pak klonujeme právě z prototypů. Obrázek 2.1 ukazuje příklad rysu modelujícího čtyřúhelník, jeho prototyp a objekt, který představuje konkrétní čtyřúhelník.

Jazyky založené na prototypech bývají obvykle mnohem jednodušší než třídně orientované jazyky. Díky své dynamičnosti však poskytují širší možnosti než klasické třídně orientované jazyky. Následující kapitoly stručně pojednávají o jazyku Smalltalk, který jako první přišel s těmito koncepty. Dále je pak zmíněn Self, jenž je považován za první čistě objektově orientovaný jazyk a nakonec je popsán jazyk Io. Především s posledními dvěma jmenovanými je náš jazyk blízký příbuzný.

2.3.3 Rušení objektů

Jazyky založené na prototypech obvykle obsahují automatickou správu paměti pomocí tzv. *garbage collectoru*. Ten periodicky prohledává celou objektovou paměť nebo její část (tzv. *generační garbage collector*), vyhledává objekty, které už nejsou odnikud odkazovány, a odstraňuje je. Garbage collector se musí umět vypořádat i s cykly mezi objekty, které na sebe vzájemně odkazují, ale neexistuje na ně žádný odkaz zvenčí.



Obrázek 2.1: Příklad rysu, prototypu a objektu s konkrétními daty

2.4 Existující objektově orientované jazyky

2.4.1 Smalltalk

Smalltalk se nachází někde na pomezí mezi třídně a prototypově orientovanými jazyky. Jeho vznik se datuje do počátku 70. let do laboratoří firmy Xerox v Palo Alto. Od té doby bylo vytvořeno několik jeho implementací, z nichž nejznámější je Smalltalk-80, která vznikla počátkem 80. let.

Smalltalk však není pouze programovací jazyk, ale je to i sada objektů, poskytujících určitou funkcionalitu. Pod tímto názvem také můžeme chápat celé grafické vývojové prostředí implementované rovněž ve Smalltalku. Programátor zde má prostředky pro zkoumání jednotlivých objektů, implementace či změnu jejich metod, nebo například prostředky pro ladění. Protože je vše implementováno ve Smalltalku, může programátor dle vlastního uvážení všechny objekty prostředí měnit. V extrémním případě může celé vývojové prostředí i předefinovat.

Smalltalk neběží přímo na fyzickém hardware, ale ve virtuálním stroji. Ten na rozdíl například od Javy nepracuje s programem v podobě nějakého souboru, jenž je vždy spuštěn, ale pracuje s něčím, co je nazýváno *image* (obraz). Obraz obsahuje objekty, které byly uloženy v objektové paměti v okamžiku jeho vytvoření. Obraz potom neobsahuje pouze vlastní program, ale všechny

objekty, které se v paměti v tu chvíli nacházely a tedy i celé výše zmiňované vývojové prostředí včetně editorů. Při načtení obrazu pak program nezačíná od začátku, ale je obnoven stav, ve kterém program a celé vývojové prostředí bylo v okamžiku uložení image. Program tedy není samostatnou entitou, ale funguje jako rozšíření objektového prostředí, ve kterém byl vyvinut. Pro potřeby distribuce programů potom existují nástroje, které dokážou nepotřebné objekty z obrazu eliminovat.

Smalltalk má velmi jednoduchou syntaxi obsahující jen pět klíčových slov. První tři z toho jsou pseudoproměnné *true*, značící pravdu, *false,* značící nepravdu a *nil* je instance nedefinovaného objektu. Další dvě klíčová slova jsou *self* značící příjemce zprávy a *super* značící jeho rodičovský objekt. Tyto dvě hodnoty jsou dostupné v průběhu provádění metody a umožňují z ní přistupovat k objektu, jemuž byla zpráva zaslána.

Jsou definovány pouze čtyři jazykové konstrukce, a to zasílání zprávy objektu, návrat hodnoty z metody, přiřazení do proměnné pomocí operátoru „:="a vytváření některých objektů pomocí literálů (například čísel). Cykly a větvení jsou implementovány pomocí zpráv tak, jako ostatní operace.

2.4.1.1 Zasílání zpráv

Vše ve Smalltalku je chápáno jako objekt, a to včetně jednoduchých hodnot. Objekty spolu komunikují prostřednictvím zpráv, přičemž zpráva funguje jako požadavek na vykonání nějaké akce na straně příjemce. Odesilatel zprávy čeká po dobu jejího zpracování příjemcem na její výsledek. Důležité je zde říci, že konkrétní příjemce zprávy není určen při překladu, ale až za běhu. Díky tomu je možné dosáhnout polymorfismu.

Jazyk rozlišuje tři typy zpráv, které se liší svou syntaxí i způsobem vyhodnocení z hlediska priority. První skupinou jsou *unární* zprávy, což jsou zprávy, které nemají žádné argumenty. Příkladem může být „5 factorial", která počítá faktoriál čísla 5.

Další skupinou jsou *binární zprávy*, nesoucí jeden argument. Sem patří třeba zprávy pro aritmetické operace. Oproti zvyklostem z jiných jazyků je zde jiná priorita operátorů a výrazy se vyhodnocují zleva doprava. Prioritu operátorů pak musíme zajistit pomocí závorek. Příkladem může být výraz „3+4 * 5, který se ve skutečnosti vyhodnotí jako (3+4) * 5.

Poslední skupinou jsou zprávy s více argumenty (*keyword messages*). Tyto zprávy jsou identifikovány pomocí tzv. *selektoru*, který může být složen z více identifikátorů ukončených dvojtečkou, za něž se mohou zapisovat argumenty zprávy. Zaslání takové zprávy ukazuje obrázek 2.2.

```
aDictionary at: 'UK' put: 'United Kingdom'.
```

Obrázek 2.2: Příklad zprávy s více argumenty

Zpráva z obrázku má dva argumenty zapsané za identifikátory „at" a „put". Odpovídající metoda je pak identifikována jako „at:put:". Tento typ zpráv se rovněž používá pro implementaci řídících struktur, které potom vypadají srozumitelněji.

Unární zprávy mají nejvyšší prioritu, dále jsou vyhodnoceny binární zprávy a nakonec zprávy s více argumenty.

2.4.1.2 Bloky kódu

Bloky kódu obsahují příkazy jazyka zapsané do hranatých závorek „[„ a „]“. Blok potom funguje jako jakýkoliv jiný objekt. Na rozdíl od normálně zapsaných příkazů se ty v rámci bloku nevyhodnotí hned, ale jen pokud je blok vyhodnocen zasláním zprávy *value*. V tom případě je výsledkem vyhodnocení bloku výsledek jeho posledního příkazu. Obrázek 2.3 ukazuje příklad jednoduchého bloku a jeho vyhodnocení.

```
result := [counter := counter + 1] value.
```

Obrázek 2.3: Příklad vyhodnocení bloku

Bloky nacházejí využití u řídících struktur a umožňují programátorovi implementovat své vlastní řídící struktury. Obrázek 2.4 ukazuje způsob realizace větvení pomocí zprávy *ifTrue*:*ifFalse:*, nesoucí jako argumenty bloky kódu, jenž se vyhodnotí, je-li cílem zprávy *true* respektive *false*.

```
res := a > b
    ifTrue:[ 'vetsi' ]
    ifFalse:[ 'mensi nebo rovno' ]
```

Obrázek 2.4: Příklad realizace větvení

2.4.1.3 Třídy

Třídy v Smalltalku jsou obdobou tříd z třídně orientovaných jazyků popsaných dříve. Třída slouží jako šablona pro vytváření nových objektů a definuje jejich strukturu. Nový objekt potom vznikne instanciací, zasláním zprávy *new* zaslané dané třídě.

Třída také fyzicky obsahuje definované metody, čímž se šetří paměť. Pokud je objektu zaslána zpráva, která odpovídá metodě, je tato vyvolána ve skutečnosti v rámci třídy daného objektu (ale v kontextu příjemce zprávy).

Třídy rovněž slouží pro specifikování dědičnosti. Po zaslání je tedy nejprve vyhledávána odpovídající metoda v třídě cílového objektu. Pokud není nalezena, pokračuje se rodičovskou třídou a tak dále do nalezení zprávy nebo dosažení kořene stromu dědičnosti.

2.4.2 Self

Jazyk *Self* vychází ze Smalltalku a dále rozvádí a unifikuje jeho koncepty. Jedná se o skutečně první jazyk založený na prototypech. Zde bylo zavedeno, že objekt se skládá se slotů a byly zde také poprvé použity pojmy rys, prototyp nebo delegování. Obdobně jako Smalltalk i Self běží v rámci virtuálního stroje. Části programu se však za běhu mohou překládat do nativního strojového kódu, čímž se

dosáhne vyššího výkonu. I Self poskytuje grafické vývojové prostředí napsané v rovněž Selfu a stejně tak program je uchováván v podobě perzistentního image.

2.4.2.1 Objekty

Self rozlišuje datové a kódové objekty. Datové objekty nesou pouze data, zatímco kódové objekty obsahují i příkazy jazyka. Mezi kódové objekty patří metody a bloky kódu. Co se týče zpráv, obsahuje Self tři typy stejně jako Smalltalk. Rovněž jejich syntaxe a sémantika je stejná.

Objekty je možné vytvářet buď klonováním, kdy vznikne přesná kopie objektu, nebo zápisem pomocí literálů. Takové objekty pak vznikají v době překladu. Literál objektu se zapisuje do kulatých závorek. Jedinou výjimkou jsou bloky, jež se zapisují do hranatých závorek. Literál může obsahovat seznam slotů zapsaných mezi svislé čáry a oddělených tečkou. Rodičovské sloty jsou označeny hvězdičkou za názvem slotu. Za seznamem slotů mohou následovat příkazy jazyka oddělené rovněž tečkou. Zapsané příkazy jsou provedeny při pokusu o vyhodnocení objektu. Obrázek 2.5 ukazuje příklad literálu objektu, který obsahuje dva sloty a jeden příkaz.

```
(| sl1. sl2. | „Ahoj svete‟ printline )
```

Obrázek 2.5: Ukázka zápisu literálu objektu

Sloty v seznamu je možné inicializovat buď pomocí „=“, čímž vznikne slot pouze pro čtení (*read-only*), nebo šipkou „<-“, kdy vznikne přepisovatelný slot.

2.4.2.2 Organizace objektů

Objektová paměť Selfu (*Self world*) je organizována do stromové struktury s kořenem v objektu zvaném *Lobby*. Lobby funguje jako vstup do tohoto světa a má tři jmenné prostory implementované jako sloty. Ty obsahují objekty, které v sobě sdružují objekty odpovídajícího typu. Prvním slotem je *traits*, který obsahuje objekty implementující sdílené chování – rysy. Další skupinou jsou *globals*, kde jsou uloženy prototypy a další globální objekty. Poslední skupinou jsou pak *mixins*. Sem patří malé objekty bez přiřazených rysů.

2.4.2.3 Metody a bloky

Metody jako objekty nesoucí příkazy jazyka je možné definovat pomocí literálu z obrázku 2.5. Objekt metody vždy obsahuje implicitní slot „Self“, který je při zavolání metody nastaven na příjemce zprávy, jež tuto metodu vyvolala. Metoda může mít argumenty. Ty jsou specifikovány jako obyčejné sloty v seznamu slotů. Před jejich jménem je ale uvedena dvojtečka. Metody je možné přiřadit pouze do *read-only* slotů.

Bloky jsou speciálními typy kódových objektů, které podobně jako u Smalltalku nejsou provedeny při vyhodnocování, ale zasláním zprávy *value.*Příkazy bloku jsou potom vyhodnocovány v kontextu jeho aktivace. Objekt tedy může přistupovat k lokálním proměnným nadřazeného

kontextu. Bloky jsou v Selfu použity také pro implementaci všech řídících struktur. Obrázek 2.6 ukazuje příklad cyklu s podmínkou na začátku, který bude prováděn, dokud bude podmínka (udaná prvním blokem) vyhodnocena jako *true*.

```
[ cont ] whileTrue: [ ... ]
```

Obrázek 2.6: Ukázka cyklu s podmínkou na začátku

2.4.3 IO

Minimalistický jazyk IO vznikl v roce 2002. Jeho tvůrce se zaměřil na jednoduchost, snadnou rozšiřitelnost a vysoký výkon. IO vychází z celé řady jazyků, mezi nimiž jsou i funkcionální jazyky. Za všechny jmenujme *Smalltalk*, *Self* nebo *Lisp*. Zajímavostí jazyka je, že neobsahuje žádná klíčová slova a vše ve zdrojovém kódu kromě komentářů je chápáno jako zpráva. Vyhodnocením zprávy je pak získán odpovídající objekt.

Objekty vznikají klonováním. Na rozdíl od Selfu ale nevzniká úplná kopie objektu. Naklonovaný objekt neobsahuje žádný slot a jeho rodičovský slot je nastaven na původní objekt. Vzniklý objekt je tedy potomkem původního. Pokud programátor změní slot, který je dostupný v rodičovském objektu, vznikne u nového objektu kopie původního slotu s novou hodnotou. Tento jev se nazývá *diferenční dědičnost*. Programátor může také specifikovat metodu *init*, která bude zavolána bezprostředně po naklonování a umožňuje vytvořený objekt zinicializovat do výchozích hodnot.

IO rozlišuje dva typy zpráv – klasické zprávy, jež jsou specifikovány svým textovým jménem a operátory, které mají prioritu danou dle zvyklostí z jiných jazyků. Argumenty klasických zpráv se zapisují za zprávu do kulatých závorek a jsou odděleny čárkami. Argumentem může být jakýkoliv výraz. Argumenty jsou však na rozdíl od jiných jazyků vyhodnocovány až v příjemci zprávy. Obrázek 2.7 ukazuje příklad volání metody, která vrací maximum z předaných argumentů. Zároveň je ukázáno použití operátoru „:=“ pro vytváření nových slotů. Kromě toho existuje ještě operátor „=“ pro změnu existujícího slotu a operátor „::=“, který je podobný jako „:=“. Pro slot však vytvoří ještě *setter*.

```
h := max(4+ 5, 3 * 2)
```

Obrázek 2.7: Ukázka volání zpráv

2.4.3.1 Kódové objekty

Podobně jako Self, i IO obsahuje dva kódové objekty – metodu a blok.

Metody vznikají pomocí zprávy *method* a jsou chápány jako anonymní funkce. Název metody je dán názvem slotu, do kterého byla přiřazena. Zpráva *method* má proměnlivý počet argumentů. Nejprve jsou specifikovány názvy argumentů budoucí metody a jako poslední argument je předána

posloupnost příkazů jazyka, které budou tvořit tělo metody. Při zavolání metody potom vznikne objekt nesoucí konkrétní argumenty, který navíc obsahuje slot *self* odkazující na příjemce zprávy. Obrázek 2.8 ukazuje příklad vytvoření nové metody.

Bloky se vytvářejí pomocí zprávy *block*, která rovněž přebírá označení argumentů a posloupnost příkazů, jež budou vykonány při aktivaci bloku. Oproti Selfu je objekt spjat s kontextem, ve kterém vznikl namísto kontextu, ve kterém byl aktivován.

```
odecti:= method(a, b, a - b)
```

Obrázek 2.8: Ukázka vytvoření nové metody

3 Jazyky pro vědecké výpočty

Tato kapitola se stručně zabývá některými jazyky používanými pro vědecké výpočty. Ukazuje především základní konstrukce a specifické rysy, ze kterých jsme vycházeli při návrhu našeho jazyka.

3.1 Octave

Octave je jazyk a stejnojmenný program fungující v příkazové řádce, který je určen především pro matematické výpočty. Jazyk obsahuje vestavěné prostředky pro popis různých matematických operací a konstrukcí, například pro goniometrické funkce nebo matice. Program dale obsahuje funkce například pro řešení rovnic či kreslení grafů.

Proměnné jsou vytvářeny pouhým zapsáním jejich názvu následovaným například operátorem „=“ a matematickým výrazem, jehož výsledek bude do proměnné uložen. Program v Octave je možné také napsat v běžném textovém editoru a poté jej v Octave pouze provést. Syntaxe jazyka je téměř totožná se syntaxí pro program Matlab a platí, že program napsaný v Octave obvykle bez větších změn běží i v Matlabu.

3.1.1 Vektory a matice

Vektory a matice lze vytvářet zápisem jejich prvků do hranatých závorek. Jednotlivé řádky jsou odděleny středníky a prvky na jednom řádku mezerami. Zápis „[1 2 3]“ vytvoří řádkový vektor o třech prvcích. Kdybychom dali mezi prvky středníky, obdrželi bychom vektor sloupcový.

Mimo to lze vektory vytvářet také pomocí generátorů. Uživatel specifikuje dolní a horní mez, případně krok a Octave vygeneruje odpovídající vektor. Pokud nebyl krok zadán, předpokládá se implicitní krok 1. Příkaz „x = 2:2:6“ vytvoří vektor s prvky 2, 4, 6 a uloží jej do proměnné *x*.

Zápis „[1 2; 3 4]“ vytvoří matici o rozměrech 2×2. Matice je možné také skládat z jiných matic tak, že zdrojové matice zapíšeme jako prvky vznikající matice. Octave poskytuje rovněž užitečné funkce pro vytváření jednotkových matic zadaného rozměru, či například matic o zadaných rozměrech vyplněné samými nulami nebo jedničkami.

3.1.1.1 Operace

Standardní operace sčítání, odečítání a násobení jsou dostupné i pro matice. Operace jsou definovány obvyklým způsobem známým z matematiky. Kromě toho je možné provádět i operace nad jednotlivými prvky matice pomocí operátorů .* pro násobení, ./ pro dělení a .^ pro umocňování.

Prvky z vektoru nebo matice lze vybírat prostřednictvím indexování. Index se zapisuje do kulatých závorek a dimenze jsou odděleny čárkou. Indexem může být obecně matematický výraz nebo dvojtečka, vybírající všechny prvky dané dimenze. Pomocí dvojtečky je možné specifikovat

také interval prvků. Například zápis „v(k)" vybírá k-tý prvek vektoru *v*. Obdobně zápis „m(i, j)" vybírá prvek z i-tého řádku a j-tého sloupce matice *m*. Chceme-li vybrat prvky *m* až *n* vektoru *v*, použijeme zápis „v(m:n)". Opět obdobně v případě matice příkaz „m(:, c)" vybírá celý sloupec *c* matice *m* a příkaz „m(r, :)" vybere celý řádek *r*.

3.2 Maxima

Maxima je další nástroj pro matematické výpočty, implementovaný v LISPu. Jazyk podporuje matice, seznamy, či komplexní čísla. I zde jsou prostředky pro vykreslování grafů, řešení lineárních rovnic nebo pro diferenciální a integrální počet.

Proměnné opět vznikají prostým zápisem. Operátorem přiřazení je zde však dvojtečka. Kromě toho interaktivní režim označuje každý výsledek návěštím začínajícím znakem „%", následovaným číslem. V dalších příkazech se pak lze na předešlé výsledky odkazovat i pomocí těchto návěští.

Komplexní čísla se zapisují stejně jako obyčejná čísla. Číslo je ale násobeno konstantou „%i", jež je rovna odmocnině z -1. Výraz „3 + 4*%i" pak vyjadřuje komplexní číslo, jehož reálná část je 3 a imaginární část je 4.

Jazyk obsahuje rovněž seznamy založené na seznamech jazyka LISP. Dalším datovým typem jsou pole, která mohou být obecně mnohorozměrná. Pro přístup k prvkům pole nebo seznamu se používají hranaté závorky. Například příkaz „b[3] : 5;" přiřadí do třetího prvku seznamu hodnotu 5. Jazyk umožňuje rovněž vícenásobné přiřazení. Příkaz „[a,b,c] : [7,8,9]" vytvoří nový tříprvkový seznam a do prvků přiřadí odpovídající hodnoty. Přiřazení se děje paralelně.

Posledním zde popsaným typem jsou struktury, které slučují různá data do jednoho celku. Strukturu je možné definovat funkcí *defstruct*, která přijímá jeden argument popisující název struktury a názvy jejích prvků. Instanci struktury potom vytvoříme pomocí funkce *new*, které předáme název struktury. Obrázek 3.1 ukazuje příklady užití příkazů pro definici a vytvoření instance struktury. K jednotlivým prvkům struktury můžeme přistupovat pomocí operátoru „@" následovaným názvem prvku.

```
defstruct (myStruct (a, b, c));
new(myStruct);
```

Obrázek 3.1: Ukázka definice a vytvoření instance struktury

3.3 Sage

Sage je napsáno v Pythonu a slučuje dohromady přibližně 100 různých knihoven pro matematické výpočty. Díky tomu je Sage silný nástroj pro různé oblasti matematiky zahrnující algebru, teorii čísel, numerické metody, kombinatoriku, teorii grafů či kryptografii. Je určen především pro výuku

matematiky a výzkum. Program je možné používat prostřednictvím webového rozhraní nebo příkazového řádku.

Syntaxe Sage je téměř totožná se syntaxí jazyka Python. Základní matematické operace jako je přiřazení, porovnávání a číselné literály mají stejný tvar. Zrovna tak platí, že příkazy jsou spíše řádkově orientované a zanoření je realizováno pomocí odsazení. Rovněž uživatelské funkce jsou definovány pomocí klíčového slova *def* stejně jako v Pythonu. Uživatel také může vytvářet nové typy pomocí tříd.

3.4 R

Jazyk *R* a jemu odpovídající prostředí je zaměřeno především na statistickou analýzu dat a jejich zobrazení. R se dá snadno rozšiřovat pomocí balíků. Výpočetně kritické operace je možné implementovat v jazycích C/C++ či Fortran. Jazyk vychází z mnohem staršího jazyka *S*, ze kterého převzal jisté rysy objektově orientovaného programování.

Stejně jako výše uvedené jazyky, i R obsahuje přímou podporu pro matice, komplexní čísla, vektory a operace s nimi. Kromě toho jsou k dispozici seznamy a speciální struktury pro statistiku připomínající databázové tabulky (*data-frames*). Jedná se o strukturu podobou matici s tím rozdílem, že každý sloupec může mít jiný datový typ. Obrázek 3.2 ukazuje příklad vytvoření vektoru.

```
x <- c(10.5, 4.6, 3.3, 2.4, 1.7)
```

Obrázek 3.2: Příklad vytvoření vektoru

Šipka funguje jako operátor přiřazení a nastavuje výrazu vpravo jméno. Představený příkaz tedy vytvoří vektor (kolekci) pojmenovanou *x*. Další možností je vektor vygenerovat pomocí generátoru stejně jako u Octave. Dodejme, že prvky vektoru musí být vždy stejného typu.

K prvkům vektoru je možné přistupovat pomocí výrazu zapsaného v hranatých závorkách za jménem vektoru. Výraz se vyhodnocuje jako další vektor sloužící k indexaci. Podle jeho typu jsou pak vybrány prvky prvně jmenovaného vektoru. Obrázek 3.3 ukazuje dva příklady výběru z pole. V druhém případě je pro výběr použit vektor vytvořený pomocí generátoru.

```
y <- x[!is.na(x)]   # y bude obsahovat neprázdné prvky z x
x[1:10]             # vybere prvky 1 až 10 z x
```

Obrázek 3.3: Ukázka indexace vektoru

R operuje s entitami, nazývajícími se objekty. Vektory jsou pak brány jako *atomické* objekty. Naopak seznamy jsou *rekurzivní objekty*, protože mohou obsahovat prvky různých typů, tedy i jiné seznamy. Objekt může mít pojmenované atributy. Ty jsou přístupné funkcí *attr*, která je zároveň umožňuje i nastavovat. Objekty jsou zařazeny do tříd. Třída ovlivňuje chování tzv. *generických funkcí,* což je jakási obdoba polymorfismu.

4 Návrh programovacího jazyka

Jak již bylo řečeno v úvodu, při návrhu našeho jazyka jsme se inspirovali především jazykem Io a v některých ohledech jazyky Self, Java nebo Smalltalk. Obdobně jako Io, ani navržený jazyk neobsahuje žádná klíčová slova a veškeré řídící struktury jsou realizovány pomocí zpráv a bloků kódu.

Oproti výše zmíněným jazykům je zavedena dynamická typová kontrola pro sloty a argumenty metod. Snažili jsme se také navrhnout nativní prostředky pro základní matematické struktury, jako jsou matice či komplexní čísla a zprostředkovat operace nad nimi. Jazyk dále obsahuje přímou podporu pro paralelismus pomocí asynchronních zpráv.

4.1 Základní principy

Obdobně jako u jiných jazyků této kategorie, objekt je složen ze slotů. Ty mohou být různých typů (viz níže) a každému slotu odpovídá nějaká zpráva. Taková zpráva je pak dále v této práci často označována jako *název slotu*. Oproti jiným jazykům mohou být některé sloty základních objektů určeny pouze ke čtení. Jedná se o sloty obsahující základní funkcionalitu a pokus o změnu takového slotu vyvolá chybu.

Objekt dále mimo jiné obsahuje odkaz na svého rodiče, čímž je umožněna jednoduchá dědičnost a tato informace se rovněž využívá při typové kontrole, což bude rozebráno dále.

S objekty je manipulováno čistě pomocí referencí, obdobně jako u Javy. Potom existuje také speciální reference *Null*, která je obdobou konstanty NULL z jazyka C nebo z Javy a používá se k vyjádření neplatné reference. O rušení nevyužitých objektů se stará jednoduchý *garbage collector*.

4.1.1 Prostředí jazyka

Navržený jazyk obsahuje předpřipravený globální strom objektů plnících základní funkce a umožňujících interakci programu s okolím. Jsou zde například umístěny i prototypové objekty pro *vestavěné datové typy*. Kořenem tohoto stromu je objekt *Lobby*, který funguje jako vstupní bod do celého objektového prostředí. Objekty, které vznikly na globální úrovni, případně v interaktivním režimu jsou potom sloty Lobby. Tento objekt také obsahuje slot odpovídající zprávě „Lobby" a jeho hodnotou je reference na Lobby. Díky tomu je možné objekt explicitně adresovat při zasílání zpráv.

Z hlediska dědičnosti jsou všechny objekty přímí nebo nepřímí potomci objektu *Object*, který implementuje základní funkcionalitu, zejména klonování a manipulaci se sloty. Tento objekt je jediný, který nemá rodiče. Protože se uplatňuje pouze jednoduchá dědičnost a zároveň platí, že každý objekt musí mít definovaného předka, vytváří hierarchie dědičnosti jedinou stromovou strukturu s kořenem v objektu Object.

4.1.2 Vznik a rušení objektů

Nové objekty mohou z pohledu jazyka vznikat výhradně dvěma způsoby. Prvním způsobem je *klonování* nějakého existujícího objektu. Nově vzniklý objekt má jako rodiče nastaven objekt, ze kterého byl naklonován. Uplatňuje se zde *diferenční dědičnost*, takže nově vzniklý objekt nemá žádný vlastní slot a využívá všechny sloty objektu původního. Toto opatření snižuje paměťové nároky a zároveň zefektivňuje vytváření nových objektů, které jsou na začátku velmi malé. Programátor poté může za jistých podmínek překrýt slot rodičovského objektu a vytvořit jeho novou implementaci v takto naklonovaném objektu.

Druhým způsobem je definice objektu pomocí *literálu*. Toto je vůbec nejčastější způsob vytváření objektů, kterým lze vytvořit základní typy objektů, například řetězce, čísla nebo bloky kódu. Vzniklý objekt má stejnou podobu, jako kdyby byl naklonován dle prvního způsobu ze svého prototypu. Jedinou výjimkou jsou čísla a pravdivostní hodnoty, které mají specifické chování. Tyto případy budou popsány dále.

Objekty není možné explicitně rušit. Je tomu tak proto, že objekt může být odkazován z mnoha míst (viz například sekce o datových typech) a při přístupu k němu by pak mohlo dojít k chybě, protože by již neexistoval. Místo toho budou pomocí garbage collectoru rušeny pouze ty objekty, u kterých je jisté, že už nejsou odnikud odkazovány. Podobně je tomu například u jazyka Java. Na rozdíl od Javy bude garbage collector velmi jednoduchý a nebude objekty v objektové paměti prohledávat podle generací.

4.1.3 Typy slotů

Každý slot obecně obsahuje buď referenci na nějaký objekt, nebo přímo hodnotu, pokud se jedná o dříve zmíněné vestavěné typy. Sloty mohou být z pohledu jazyka dvojího typu:

- **Datový slot** – výsledkem přístupu k němu je reference na objekt, který je v něm obsažen, případně přímo hodnota, pokud se jedná o vestavěný typ, například celé číslo. Obsah datového slotu je možné přímo přiřadit do jiného datového slotu.
- **Kódový slot** – obsahuje standardní metodu, implementovanou pomocí prostředků jazyka, nebo *primitivu*. Primitiva je speciální typ metody, která je vestavěna přímo do virtuálního stroje a poskytuje základní funkcionalitu. Při přístupu ke kódovému slotu je metoda provedena a poté je vrácen výsledek jejího vyhodnocení, což může být opět buď reference na nějaký objekt, nebo hodnota. Kódové sloty tedy nelze přímo měnit, nebo jejich obsahu zasílat zprávu – při přístupu k nim je totiž daná metoda provedena a dále se pracuje až s jejím výsledkem.

Důležité je také zmínit, že stejně jako objekty, i všechny sloty mají specifikován svůj datový typ, který bude dynamicky kontrolován při přiřazování hodnoty do slotu. Problematika typové kontroly bude probrána dále.

Jak již bylo řečeno výše, některé sloty mohou být pouze pro čtení. Ty jsou dány virtuálním strojem a nelze je z úrovně jazyka nijak měnit ani překrývat v potomcích, protože obsahují základní funkcionalitu jazyka. Může se jednat například objekty reprezentující vestavěné datové typy, objekt Lobby či většinu základních primitiv (viz výše). Pokus o překrytí nebo změnu takového slotu skončí chybou virtuálního stroje a ukončením provádění dané posloupnosti příkazů.

4.2 Definice navrženého jazyka

Tato kapitola se snaží o formálnější popis gramatické, sémantické a lexikální stránky jazyka. Dále jsou vysvětleny mechanismy dědičnosti a typové kontroly. Popisovaný jazyk rozlišuje velikost písmen, mezery, prázdné řádky nejsou významné.

4.2.1 Literály a vestavěné typy

Jak již bylo zmíněno dříve, literály slouží k přímé definici základních typů objektů. Takto zapsané objekty jsou vytvořeny buď v době překladu, nebo během provádění programu. Z pohledu programátora se však takto vytvořený objekt vždy bude jevit, jako by byl naklonován z prototypového objektu pro daný typ.

Výjimkou jsou pouze čísla a logické hodnoty. Ty jsou uloženy přímo jako hodnota slotu a za odpovídající *prototypový objekt* jsou dočasně nahrazeny jen v okamžiku, kdy je obsah takového slotu cílem nějaké zprávy. Vestavěné typy hodnot budou mít také z důvodů optimalizace rychlosti podporu základních operací ze strany virtuálního stroje, což bude podrobněji rozebráno v části zabývající se jeho návrhem. Prototypové objekty mají svá určitá specifika, která budou rozebrána dále. Následuje popis jednotlivých literálů.

4.2.1.1 Textový řetězec

Textový řetězec se zapisuje do uvozovek a může obsahovat tisknutelné znaky. Některé netisknutelné znaky je možné zapsat pomocí *escape sekvencí*. Escape sekvence začíná zpětným lomítkem, které je následováno jedním z povolených znaků. Mezi dovolené escape sekvence patří „\n“ pro označení konce řádku, „\t“ pro tabulátor, \“ pro uvozovky a „\\“ pro zpětné lomítko. Řetězcové literály jsou vytvářeny jako objekty, které jsou potomky objektu *String*. Řetězce jsou vytvářeny již v době překladu. Za běhu programu je pak při přístupu k literálu řetězce tento objekt transparentně nakopírován virtuálním strojem, aby se zajistila neměnnost původního řetězce - například pokud se literál řetězce vyskytuje v těle cyklu a zároveň se zde mění.

4.2.1.2 Číselné literály

Čísla zapisujeme do zdrojového kódu přímo. Čísla patří mezi vestavěné typy a jsou tedy uložena přímo jako hodnoty slotu. Rozlišují se tyto čtyři druhy (datové typy) čísel:

- **Celé číslo:** Jeho velikost a rozsah odpovídá typu *long* z jazyka C++. Celá čísla je možné zapisovat také v osmičkové soustavě – číslo začíná nulou, nebo v šestnáctkové soustavě – číslo začíná znaky „0x“. Pokud je číslu zaslána zpráva, je dočasně nahrazeno prototypovým objektem *Number*.
- **Desetinné číslo:** Zapisuje se ve tvaru „celá část“, desetinná tečka a „desetinná část“. Velikost a rozsah odpovídá typu *float* z jazyka C++ pro danou platformu. Prototypovým objektem pro tento datový ten je objekt *FNumber*.
- **Komplexní celé číslo:** Zapisuje se ve tvaru „číslo“ a znak „i“. Například „3i“ nebo „-4.5i“. Ve výrazech je možné komplexní číslo kombinovat s jiným číslem a výsledkem je opět komplexní číslo. Například „3 + 2i“. Velikost a rozsah reálné i imaginární složky je stejný jako u obyčejného celého čísla. Komplexní celá čísla jsou reprezentována rovněž jako vestavěné typy, a jejich prototypovým objektem je *Complex*.
- **Komplexní desetinné číslo:** Platí zde obdobná pravidla zápisu jako pro desetinné číslo. Rozsah reálné a imaginární složky je stejný jako u desetinných reálných čísel, tj. odpovídá datovému typu *float* z jazyka C++. Desetinné komplexní číslo má odpovídající prototypový objekt *FComplex.*

4.2.1.3 Logické hodnoty

Logické hodnoty patří rovněž mezi vestavěné typy. Obdobně jako u jiných jazyků jsou k dispozici dvě hodnoty – *True* značící pravdu a *False* značící nepravdu. Prototypovým objektem je *Boolean*. Technicky jsou logické hodnoty reprezentovány jako proměnné typu *bool* jazyka C++.

4.2.1.4 Null

Jak již bylo zmíněno výše, *Null* je speciální hodnota slotu vyjadřující neplatnou referenci. Lze ji přiřadit do jakéhokoliv slotu (který není pouze pro čtení) bez ohledu na jeho datový typ. Null patří mezi vestavěné typy, nemá však žádný prototypový objekt a nelze jej tedy použít jako cíl žádné zprávy.

4.2.1.5 Matice

Představují matice v matematickém slova smyslu. Matice může obsahovat pouze objekty stejného datového typu. Typ obsažených objektů je dán datovým typem prvního vloženého prvku. Pokud datový typ z takového prvku nelze odvodit, je ohlášena chyba a provádění sekvence příkazů končí. Matice mohou být jedno nebo dvourozměrné a místo samotných objektů jsou uchovávány opět reference na ně. Prvkem matice může být samozřejmě i vestavěný typ, například Null. Obrázek 4.1 ukazuje příklad zápisu maticového literálu.

```
[1, 2; 3, 4];
```

Obrázek 4.1: Příklad matice 2 × 2 čísel

Matici je možné vytvořit tak, že mezi hranaté závorky „[„ a „]“ zapíšeme jednotlivé prvky oddělené čárkou. Řádky pak oddělujeme středníkem. Musí platit, že každý řádek obsahuje stejný počet prvků. Je také možné uvést prázdné závorky, což znamená prázdná matice.

Objekt reprezentující matici je potomkem objektu *Matrix*. Přesná syntaxe zápisu literálu matice je uvedena dále, v kapitole zabývající se gramatikou jazyka.

4.2.1.6 Pole

Pole je velmi podobné matici. Hlavní rozdíl je v tom, že jeho prvky mohou být objekty různého typu. Obdobně jako matice, i pole mohou být jedno nebo dvourozměrná. Vícerozměrná pole můžeme vytvořit tak, že jako prvky nějakého pole budou objekty, které opět reprezentují pole.

Literál pro pole vytvoříme tak, že mezi dvojice znaků „[|“ a „|]“ zapíšeme jednotlivé prvky oddělené čárkou. Řádky, obdobně jako u matice, oddělujeme středníkem a i zde platí, že jednotlivé řádky musí mít stejný počet prvků. Objekt reprezentující pole je potomkem objektu *Array*. Obrázek 4.2 ukazuje příklad zápisu pole.

```
[| „ABC“, Null; 4, 2.1i |];
```

Obrázek 4.2: Příklad pole 2 × 2 různých prvků

4.2.1.7 Selektor

Selektor slouží k výběru prvků z polí, matic, případně i řetězců. Lze jej však nadefinovat pro jakýkoliv objekt (viz dále). Selektor nemůže existovat samostatně, ale píše se za literál (například řetězec), případně za zprávu, která zpřístupňuje objekt definující selektor (například matice). Selektor se zapisuje do hranatých závorek „[„ a „]“ a může vybírat z jedné, nebo ze dvou dimenzí, jejichž zápis je oddělen čárkou. První zapsaná dimenze vybírá u matic a polí řádky, druhá, která je nepovinná, vybírá sloupce.

Jako výraz určující dimenzi může být jakýkoliv výraz, jehož výsledek je typu Number. Lze rovněž místo výrazu použít dvojtečku pro výběr všech řádků (sloupců). Lze též specifikovat interval řádků (sloupců) tak, že před a za dvojtečku uvedeme matematické výrazy určující minimální a maximální index výběru.

Typ výsledku vyhodnocení selektoru se liší podle toho, zda byl v konečném důsledku vybrán pouze jeden prvek, nebo více. Pokud byl vybrán jediný prvek, je vrácen přímo ten. Pokud bylo vybráno více prvků, je vrácena nová složená struktura. Případné reference na objekty jsou však nakopírovány z té původní (nedochází ke klonování uložených objektů) a změna přes vrácenou strukturu tedy ovlivňuje původní objekt.

Řetězce mají u návratové hodnoty selektoru výjimku, kdy je vždy vrácen nový řetězec, protože jazyk neobsahuje žádný datový typ pro jeden znak. Dodejme také, že u řetězců je relevantní pouze

první dimenze, protože jsou jednorozměrné. Specifikace druhé dimenze jiné než pomocí dvojtečky (která je implicitní) vede k běhové chybě.

Tabulka 4.1 ukazuje příklady různých selektorů a připojuje k nim komentář.

Selektor (bez uvedené předcházející zprávy)	**Význam**
[1]	Vybírá prvek s indexem 1 z první dimenze.
[:]	Vybírá všechny prvky první dimenze.
[3:5]	Vybírá prvky s indexem od 3 do 5 (včetně) z první dimenze. Výsledná struktura má 3 prvky.
[1,3:5]	Vybírá prvek s indexem 1 z první dimenze a prvky 3 až 5 z druhé dimenze. Výsledná struktura má 3 prvky.
[2:4, 1:3]	Vybírá prvky 2 až 4 z první dimenze a 1 až 3 z druhé. Výsledná struktura má 3 řádky a 3 sloupce.
[:, :]	Vybírá všechny prvky dvourozměrné struktury. I zde je však vrácena kopie (pokud výsledek není pouze jeden prvek, viz výše).

Tabulka 4.1: Příklady různých selektorů

Selektor je reprezentován objektem, jenž je potomkem objektu *Selector*. Konkrétné parametry selektoru je možné specifikovat pouze výše popsaným literálem, nikoliv změnou slotů objektu selektoru. Dále u tzv. *operátorových zpráv* bude popsán způsob, jak lze objekt selektoru získat a využít k implementaci do vlastních objektů.

4.2.1.8 Blok kódu

Blok kódu, jak už název napovídá, nese posloupnost příkazů jazyka. V přeložené podobě pak nese odpovídající posloupnost *instrukcí* virtuálního stroje. Jedná se v podstatě o základní stavební kámen všech programů, protože veškeré příkazy jazyka musí být součástí nějakého bloku. Příkazy, které jsou zapsány na globální úrovni (tj. ne v rámci nějaké metody) jsou pak chápány jako součást globálního bloku kódu, který se vyhodnocuje v kontextu Lobby a který se začne provádět po spuštění programu.

Blok kódu se zapisuje mezi složené závorky „{„ a „}“ a jednotlivé příkazy jsou odděleny středníkem. Je rovněž možné zapsat prázdné závorky, které vyjadřují prázdný blok. Je také možné do sebe bloky zanořovat. Blok kódu je potomkem objektu *Block*.

Příkazy se v rámci jednoho bloku vyhodnocují směrem od prvního k poslednímu. Jako výsledek vyhodnocení bloku je pak chápán výsledek jeho posledního provedeného příkazu. V případě prázdného bloku je výsledkem hodnota Null.

Lokální proměnné vytvořené v rámci určitého bloku jsou jeho sloty. V rámci zanořeného bloku je možné přistupovat k proměnným (slotům) vytvořeným v rámci vnějšího bloku. Jako vnější blok je

chápán blok lexikálně nadřazený danému literálu bloku. Tento princip je podrobně vysvětlen dále, v části pojednávající o *jmenných prostorech* a o *implicitním příjemci zprávy*. Zde je také popsán algoritmus vyhledávání odpovídajícího slotu. Zavedli jsme konvenci, že jména lokálních slotů budou začínat malým písmenem.

Blok má význam především jako parametr některých zpráv implementujících řídící struktury, například cykly. Je však možné blok kódu zapsat i jako samostatný příkaz. To ale nemá moc smysl, protože takto zapsaný blok je při provádění vyhodnocen jako reference na něj. Blok totiž, ač nese kód, patří mezi datové objekty. Z toho také plyne, že stejně jako jakýkoliv jiný datový objekt, i takto vytvořený blok lze přiřadit do nějakého slotu. Zde je však potřeba mít na paměti, že blok má vždy dán svůj vnější blok a tedy i lexikální kontext místem zápisu literálu, nikoliv místem použití. Podrobnosti budou rozebrány dále.

4.2.2 Zprávy

Zprávy tvoří jádro celého jazyka. Každá zpráva má svého příjemce a výsledek, což je zase nějaký objekt nebo hodnota. Zpráva navíc může mít libovolný počet argumentů. Pro syntaxi zápisu identifikátoru zprávy platí obdobná pravidla, jako pro identifikátory z jiných jazyků. Identifikátor začíná písmenem nebo podtržítkem a dále může obsahovat libovolný počet velkých či malých písmen z první poloviny ASCII tabulky (tj. bez diakritiky), číslic a podtržítek.

Zprávy mohou být buď *synchronní*, nebo *asynchronní*. Při zaslání synchronní zprávy odesílatel čeká na vyhodnocení zprávy u příjemce a teprve pak se pokračuje v dalším výpočtu. Jedná se o standardní zprávy popisované v této kapitole. Asynchronní zprávy se vyhodnocují v samostatném vlákně a jako jejich dočasný výsledek je vrácen objekt typu *Future,* reprezentující budoucí skutečný výsledek zprávy. Asynchronní zprávy budou popsány v dalších částech textu.

Pokud má zpráva dány argumenty, zapisují se do jednoduchých závorek „(“ a „)“ za zprávu. Je dovoleno použít i prázdné závorky, pokud zpráva žádné argumenty nemá. Sémantický význam je pak stejný, jako by žádné závorky uvedeny nebyly. Považujeme za konvenci uvádět závorky vždy, když dané zprávě odpovídá metoda. Argumenty jsou vyhodnocovány na straně příjemce postupně zleva doprava.

Jako argument může být zapsán jakýkoliv výraz jazyka, například blok kódu. Pokud je nějaký blok posledním argumentem zprávy, je možné jej uvést za uzavírající závorku. Případně, pokud zpráva obsahuje jediný argument, a to blok kódu, je možné jej uvést přímo za identifikátor zprávy bez závorek pro argumenty. Toho se využívá zejména u řídících struktur, jejichž zápis je pak velmi podobný řídícím strukturám známým z jazyků C nebo Java.

U argumentů je rozhodující pořadí. Je to podobné, jako například u funkcí jazyka C. Všechny argumenty mají také specifikovaný datový typ, který je ověřován oproti typu skutečně předaných objektů při vyvolání metody. Mechanismy typové kontroly jsou popsány dále.

Důležité je zde také zmínit, že zprávy je možné řetězit pomocí operátoru tečky „.“. Řetězec zpráv se pak vyhodnocuje zleva doprava a zpráva, která je zapsána vpravo od tečky je zaslána objektu, který je výsledkem vyhodnocené zprávy zapsané vlevo od tečky.

Zprávu nelze zasílat objektu Null. Pokud se například v průběhu vyhodnocování řetězce zpráv některá zpráva (kromě poslední) vyhodnotí na Null, je při zaslání následující zprávy z řetězce ohlášena chyba a provádění sekvence příkazů končí. Obrázek 4.3 ukazuje příklad nějakého řetězce zpráv. Je zde také ukázáno použití selektoru, jehož indexem je výsledek jiné zprávy.

```
Lobby.a[m];
```

Obrázek 4.3: Ukázka řetězce zpráv a použití selektoru

4.2.2.1 Implicitní a explicitní příjemce

Jazyk rozlišuje *implicitního* a *explicitního* příjemce zprávy. Explicitní příjemce je použit vždy u zprávy, která je součástí nějakého řetězce zpráv a není zapsána jako první – víme, že zprávu posíláme objektu, který je výsledkem předchozí vyhodnocené zprávy. Explicitní příjemce zprávy je použit i v případě, kdy je zpráva zaslána nějakému literálu. To je možné například u řetězců, čísel a bloků. Maticím a polím nelze posílat zprávu přímo, ale je nutné je nejprve přiřadit do nějaké lokální proměnné v rámci samostatného příkazu.

Implicitní příjemce se používá ve všech ostatních případech. Typickým příkladem je samostatně zapsaná zpráva v rámci nějakého výrazu, nebo první zpráva z řetězce zpráv, pokud není poslána literálu. Implicitního příjemce má například i zpráva „m“ z obrázku 4.3.

Smyslem implicitního příjemce je zejména zjednodušit zápis přístupu k lokálním proměnným nadřazeného bloku – nepotřebujeme jej do zápisu nijak uvádět. Implicitní příjemce je podrobně popsán dále v kapitole o jmenných prostorech. Tam je také uveden algoritmus vyhledání implicitního příjemce zprávy.

4.2.2.2 Operátorové zprávy

Jazyk chápe operátory také jako zprávy. Standardní zprávy se vyhodnocují postupně zleva doprava. To je ale krajně nevhodné pro matematické výrazy. U Smalltalku se tento problém řeší vhodným závorkováním. Naším cílem bylo vytvořit jazyk vhodný pro vědecké výpočty. Proto bylo potřeba chování operátorů specifikovat tak, aby byla tvorba matematických výrazů přirozená.

Inspirovali jsme se jazykem Io a zavedli tzv. *operátorové zprávy*. Je dána pevná množina operátorů a jejich pevná priorita, která je nastavena dle běžných zvyklostí. Tyto operátory jsou však jen jakousi syntaktickou zkratkou pro standardní zprávy. Například binární operátor „+“ odpovídá zprávě „add(op)“, kde „op“ je druhý operand za zprávou. Výraz „a+4“ se potom vyhodnotí jako „a.add(4)“.

Matematické výrazy pak lze zapsat dvěma způsoby. Buď pomocí standardních zpráv – je ale potřeba se postarat o správné pořadí zápisu jednolitých zpráv, anebo pomocí operátorů s tím, že překladač správnou posloupnost zpráv vygeneruje sám.

Tímto je zároveň programátorovi umožněno si nadefinovat standardní operátory i pro své vlastní uživatelské datové typy a implementovat tím jakési jednoduché přetěžování operátorů, známé z jazyka C++. Je však potřeba dobře promyslet, jaký typ bude brán jako argument nebo argumenty, protože zde probíhá standardní typová kontrola stejně jako u běžných zpráv. Tabulka 4.2 ukazuje všechny operátorové zprávy a jejich ekvivalentní standardní zprávy.

Operátorová zpráva s operandy	Standardní zpráva
a+b	a.add(b)
a-b	a.sub(b)
a*b	a.mul(b)
a/b	a.div(b)
a%b	a.modulo(b)
a<b	a.less(b)
a <= b	a.lesseq(b)
a>b	a.greater(b)
a >= b	a.greatereq(b)
a==b	a.equal(b)
a!=b	a.noteq(b)
a && b	a.and(b)
a || b	a.or(b)
! a	a.not()
a[ ]	a.selGet(<selector>)
A a := b	addSlot(„a“, A, b)
a = b	changeSlot(„a“, b)

Tabulka 4.2: Operátorové zprávy

Zprávy „==“ a „! =“ jsou definovány již pro objekt Object a porovnávají reference na objekty. V případě selektoru je nejprve vytvořen odpovídající objekt selektoru a ten je pak předán jako argument zprávy selGet. Výběr správných prvků dle typu selektoru je pak čistě v režii příjemce.

Zprávy vytvoření slotu a změna slotu („:=“ a „=“) není možné používat ve výrazech, ale jen jako samostatné příkazy.

4.2.3 Dědičnost

Jak již bylo zmíněno výše, jazyk umí jednoduchou dědičnost. Každý objekt má speciální *rodičovský slot*, který obsahuje referenci na rodičovský objekt. Při zasílání zprávy se pak nejprve prohledají sloty cílového objektu. Pokud není slot nalezen, pokračuje se u rodičovského objektu a tak dále, dokud se nedojde na vrchol hierarchie dědičnosti, objekt Object, který žádného rodiče nemá. V případě nenalezení odpovídajícího slotu je oznámena chyba a provádění sekvence příkazů končí. Objekt tedy rozumí zprávám, pro které má odpovídající sloty a zprávám, kterým odpovídají sloty rodičovských objektů. Tím je simulována dědičnost.

Rodičovský slot je možné programově nastavovat a objekt tedy může během svého života zcela měnit své chování. Při změně rodičovského slotu je teoreticky možné vytvořit cyklus v referencích. Jazyk, přesněji virtuální stroj však bude tyto cykly hledat a případně oznámí chybu. U některých typů objektů, například Matrix, Number či jiných základních objektů není možné rodiče měnit. Je tomu tak z čistě technických důvodů. Takové objekty totiž obsahují specifická data pro podporu daného datového typu. Více bude vysvětleno dále, v části zabývající se návrhem virtuálního stroje.

Jak již bylo zmíněno, uplatňuje se diferenční dědičnost. Každý objekt tedy obsahuje pouze nové sloty, nebo ty sloty předka, které byly změněny. V potomkovi je možné předefinovat (překrýt) zděděný slot operátorem vytváření slotu „:=“. Výjimkou jsou sloty, které jsou pouze pro čtení. Ty překrýt nelze.

Při překrývání existujících slotů je rozhodující pouze název slotu. Datový typ, ani počet a datové typy parametrů (pokud je to metoda) nejsou brány v úvahu.

4.2.4 Datové typy

Navržený jazyk je dynamicky typovaný. Cílem námi navržené typové kontroly je snaha v co největší míře předejít situaci, kdy se s nějakou zprávou předá jako argument špatný objekt, který nerozumí požadované množině zpráv. Zrovna tak se nám zdálo být vhodné mít možnost specifikovat typ u jednotlivých slotů nějakého objektu. Například omezit, že něco může být pouze celé číslo.

Při návrhu typové kontroly jsme vyšli z toho, že typ objektu je v podstatě dán množinou zpráv, které rozumí. Je to dáno tím, že veškeré manipulace s objektem se dějí právě pomocí zpráv. Například chceme pracovat s číslem, a proto očekáváme, že daný objekt rozumí zprávám pro aritmetické operace a porovnávání. Pokud chceme pracovat s objektem reprezentujícím soubor, očekáváme, že bude rozumět zprávám pro čtení, zápis či uzavření souboru.

Na základě dědičnosti, popsané v předchozí kapitole, pak nějaký objekt rozumí i zprávám, kterým rozumí jeho rodič. Jak již bylo řečeno, objekt může předefinovat reakci rodičovského objektu na danou zprávu tím, že pro ni nadefinuje vlastní slot s vlastní reakcí. Nelze však žádným způsobem zajistit, aby objekt nemohl reagovat na zprávu svého rodiče – pokud nenadefinuje příslušný slot, použije se slot u rodiče. Pomocí dědičnosti tedy můžeme zajistit, že objekt bude dané množině zpráv

skutečně rozumět – bude mít správný *datový typ*. Ještě jednou dodejme, že neplatnou referenci (Null), můžeme použít jako hodnotu jakéhokoliv datového typu a lze ji tedy dosadit místo jakéhokoliv argumentu nebo přiřadit do jakéhokoliv slotu. Null nelze použít pro specifikaci datového typu. Zprávou *isNull* objektu Block lze ověřit, zda slot obsahuje platná data nebo Null.

Datový typ určujeme *prototypovým objektem*, což může být jakýkoliv existující objekt. Jako prototypové objekty se hodí především objekty z globálního stromu objektů. Jinak by nemuselo být možné datový typ specifikovat, protože by požadovaný prototyp nebyl v daném kontextu dostupný. Formálněji řečeno je pak kontrolovaný objekt specifikovaného datového typu, pokud je příslušným prototypem, nějaký jeho předek je tímto prototypem, nebo je *Null*.

Výhodou tohoto přístupu je, že nemusíme specifikovat konkrétní datový typ, ale můžeme použít nějaký obecnější, pokud očekáváme objekty více typů. Pokud je nám jedno, jakého typu daný slot bude, můžeme jako jeho typ použít přímo objekt Object.

Výjimkou z tohoto pravidla jsou pouze vestavěné datové typy (číslo, desetinné číslo, logická hodnota a podobně), protože jsou uloženy přímo v daném slotu a nelze je vydávat za referenci na nějaký objekt. Hodnoty těch typů je možné uložit pouze do slotů, jejich datovým typem je odpovídající prototypový objekt.

4.2.5 Metody

Metoda patří mezi kódové objekty, z čehož plyne, že při přístupu k tomuto typu objektu virtuální stroj provede příkazy, ze kterých se metoda skládá. Metoda je za běhu reprezentována objektem, jenž je potomkem objektu *Method* a jenž vzniká v okamžiku zavolání metody. Sám objekt metody neobsahuje žádný kód, ale funguje pouze jako jakási obálka pro blok kódu uložený vevnitř.

Každý objekt metody obsahuje slot pojmenovaný „self", který nese referenci na objekt, jenž byl cílem zprávy, která tuto metodu vyvolala. Přes slot „self" tedy můžeme přistupovat ke slotům cílového objektu a měnit prostřednictvím metody jejich stav. Objekt metody dále obsahuje sloty odpovídající argumentům, které byly do metody předány spolu se zprávou. Pojmenování a datový typ těchto slotů jsou určeny při vytváření metody. Jak již bylo zmíněno u zpráv, typy a počet skutečně předaných argumentů se kontrolují při zavolání metody. V případě, že počet nebo datový typ některého argumentu nesouhlasí s definicí metody, je vyvolána chyba a provádění končí.

Objekt metody dále obsahuje referenci na kořenový blok kódu. Tento blok je specifikován při vytváření metody spolu s argumenty a jejím názvem. Protože k objektu metody nelze přímo přistoupit, není možné k němu přidávat žádné další sloty (kromě argumentů, které byly předány se zprávou). Je však možné přidat slot přímo k objektu *Method*.

Metody se vytvářejí pomocí zprávy „*method*", kterou přijímá již objekt Object. Do každého objektu tedy můžeme přidat nějakou metodu. Zpráva *method* má tři argumenty. Prvním je řetězec specifikující název metody. Formát řetězce s názvem metody je ověřován, aby vyhovoval pravidlům

pro identifikátory, popsaným v předchozích kapitolách práce. Druhým argumentem je pole obsahující na každém řádku dvojici - prototypový objekt reprezentující datový typ a řetězec specifikující název argumentu. Pokud metoda nemá mít žádné argumenty, uvede se místo pole hodnota Null. Posledním argumentem zprávy je blok kódu, který bude tvořit tělo metody. Zavedli jsme konvenci, že názvy metod budou začínat malým písmenem.

Obrázek 4.4 ukazuje příklad definice metody u nějakého objektu reprezentujícího automobil. Příklad využívá výše zmíněné skutečnosti, že pokud je posledním argumentem zprávy blok kódu, je možné jej zapsat za uzavírající závorku.

```
automobil.method(„natankovat", [| Number, „pocetLitru" |])
{
      self.pocetLitru = pocetLitru;
};
```

Obrázek 4.4: Příklad vytvoření metody

4.2.5.1 Vlastní implementace operátorových zpráv

Kapitola 4.2.2.2 popisovala operátorové zprávy umožňující intuitivní způsob zápisu matematických výrazů. Jak již bylo řečeno, tyto zprávy (operátory) se během překladu převádějí definovaným způsobem na standardní zprávy nebo řetězce zpráv.

Tato skutečnost mimo jiné programátorovi umožňuje dodefinovat implementaci standardních operátorů pro své vlastní objekty. Stačí k objektu přidat metodu s názvem daným tabulkou 4.2 a dodržet počet argumentů. Jejich datové typy si programátor volí. Výjimkou je pouze selektor, jemuž odpovídající zpráva selGet vždy nese jeden argument, a to objekt typu Selector, jenž obsahuje informace o specifikovaných souřadnicích.

Rozhodnutí, jaká bude návratová hodnota operátorové zprávy, závisí zcela na programátorovi. Ten musí zvážit, zda je v daném případě výhodnější vracet původní objekt s modifikovanou hodnotou, nebo objekt nový, který obsahuje výsledek aplikace operátoru.

4.2.6 Prototypové objekty

Tato kapitola se podrobněji zabývá prototypovými objekty pro čísla a logické hodnoty. Motivací pro jejich zavedení byla velká režie spojená s ukládáním jednoduchých hodnot pomocí objektů. Je zřejmé, že průměrný program obsahuje relativně velký počet číselných konstant a aritmetických operací. Pokud bychom na čísla nahlíželi stejným způsobem, jako například na řetězcové literály nebo bloky, které jsou reprezentovány pomocí objektů, podstatně bychom zkomplikovali provádění programu. Při každém zapsaném čísle by pak virtuální stroj musel vytvořit nový objekt pro daný literál. Navíc by se muselo při vyhodnocování výrazů řešit, jak a při jakých aritmetických operacích budou objekty vznikat a kdy ne. Mohlo by také docházet k situacím, kdy nějakou číselnou konstantu přiřadíme do jednoho slotu a z něj jej přiřadíme do dalšího. Za předpokladu, že by to byl objekt

a přiřazovala se pouze reference, by mohlo dojít k paradoxní situaci, kdy programátor změní hodnotu prvního ze slotů, ale zároveň se změní i číslo v druhém slotu. Toto chování příliš neodpovídá přirozenému přístupu k číselným hodnotám, který je obvyklý z jiných jazyků.

Rozhodli jsme se proto ukládat číselné hodnoty přímo do slotů a stanovit speciální prototypové objekty, které budou figurovat jednak jako datové typy a jednak budou obsahovat operace pro práci s těmito hodnotami. Pokud je pak například číselné hodnotě zaslána zpráva, virtuální stroj hodnotu dočasně nahradí prototypovým objektem, což je v tomto případě Number a slot odpovídající zprávě hledá u něj. Pokud je výsledkem operace opět vestavěná hodnota, je vrácena jako výsledek namísto prototypového objektu.

Prototypové objekty pro základní typy jsou singletony. Výsledkem jejich klonování proto nejsou nové objekty, ale stále původní klonovaný objekt. Klonování takových objektů totiž nemá žádný smysl. Pokud chce programátor dodefinovat další operace pro práci s čísly nebo s logickými hodnotami, může je dopsat přímo do prototypového objektu. Tím zajistí, že tato operace bude dostupná pro všechny hodnoty daného datového typu.

4.2.7 Koncept jmenných prostorů

Jak již bylo řečeno, bloky kódu jsou standardní datové objekty a lokální proměnné jsou vlastně sloty bloku, ve kterém byly vytvořeny. Dále bylo také řečeno, že z vnitřního bloku lze přímo přistupovat k proměnným *lexikálně nadřazeného bloku* – bloku, ve kterém je vnitřní blok zapsán ve zdrojovém kódu. Je to umožněno díky tomu, že každý blok obsahuje speciální anonymní slot, který ukazuje na lexikálně nadřazený blok. Tento slot je doplněn v době překladu, když se vytváří interní reprezentace bloku a za běhu se již nemění. Příklad takové situace ukazují obrázky 4.5 a 4.6.

Jak je patrné z obrázku 4.5, pro přístup k proměnné „b“ ze zanořeného bloku není nutná žádná speciální jazyková konstrukce. Je to umožněno díky mechanismu implicitního příjemce, který byl zmíněn již výše. Algoritmus implementující tento mechanismus prohledává tzv. *lokální jmenný prostor*. Lokální jmenný prostor je množina všech slotů aktuálně prováděného bloku, slotů jeho rodičovských objektů a slotů jeho lexikálně nadřazených bloků. Důležité je zde zmínit, že lexikálně nadřazeným objektem kořenového bloku nějaké metody je objekt reprezentující tuto metodu. Metoda tedy ohraničuje lokální jmenný prostor a zároveň do něj dodává slot „self“ a sloty odpovídající argumentům metody (zprávy).

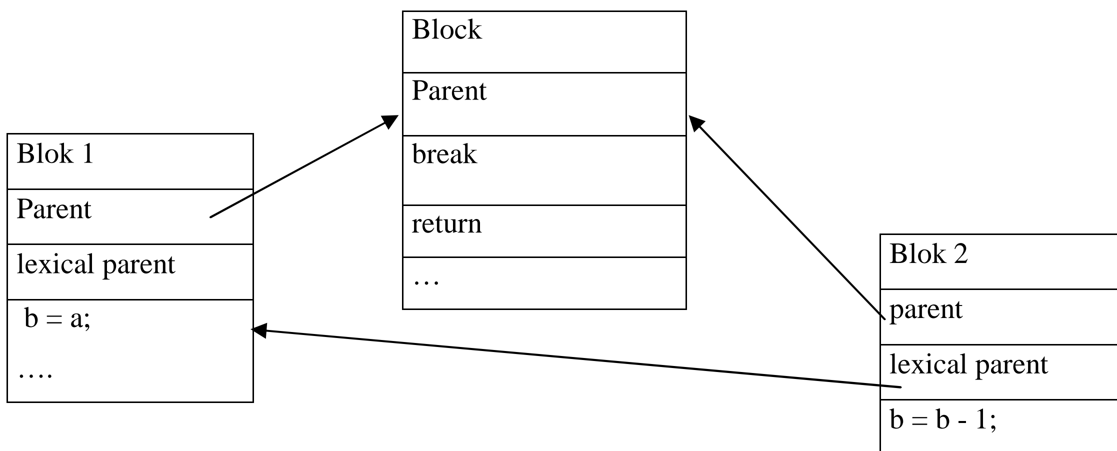
4.2.7.1 Lexikální kontext

Před provedením literálu bloku jeho běhová reprezentace, odpovídající objektu typu Block, nastaví svůj anonymní slot na běhovou reprezentaci lexikálně nadřazeného (tj. právě prováděného) bloku. Tato vazba existuje pouze tehdy, existuje-li tento nadřazený běhový blok. Jakmile zanikne, zanikne i vazba a zanořený blok dále existuje už bez nadřazeného kontextu.

Toto je podstatné, pokud programátor z nějakého důvodu přiřazuje literál bloku do proměnné. Z předchozího textu je zřejmé, že lexikálním kontextem takového bloku je blok, ve kterém byl literál zapsán. Nikoliv ten, ve kterém byl nebo je použit. Pokud nadřazený blok zanikne, dojde automaticky k tomu, že blok přiřazený do proměnné ztratí lexikální kontext a při vyhledávání implicitního příjemce jsou pak prohledávány pouze jeho lokální sloty. Pokud je původní nadřazený kontext opět obnoven (například v cyklu), je opět obnovena i zmiňovaná vazba.

```
{
	b = a;
	{
		b = b - 1;
	}.repeat();
};
```

Obrázek 4.5: Příklad zdrojového kódu se dvěma zanořenými bloky



Obrázek 4.6: Odpovídající objekty bloků kódu

Je zřejmé, že se lokální jmenný prostor neustále mění podle toho, jak postupuje provádění programu. Kromě lokálního existuje také *globální jmenný prostor*. Do něj spadají sloty objektu Lobby. Globální jmenný prostor je dostupný v každé části programu.

4.2.7.2 Algoritmus vyhledání implicitního příjemce

Obecně řečeno, algoritmus prohledává nejprve lokální jmenný prostor ve směru zevnitř ven a poté prohledává globální jmenný prostor. Prohledávání končí, jakmile je nalezen odpovídající slot, nebo jakmile je prohledán globální jmenný prostor a slot nalezen nebyl. V prohledávání se postupuje v následujícím pořadí:

- Sloty právě prováděného bloku.
- Postupně sloty rodičů právě prováděného bloku, stejně jako u explicitního příjemce.

- Postupně sloty lexikálně nadřazených bloků směrem zevnitř ven. Zde se již neprohledávají rodičovské objekty, protože nadřazené bloky mají stejného rodiče, jako aktuálně prováděný blok (objekt Block).
- Sloty právě prováděné metody, tedy argumenty a slot *self*
- Rodičovské sloty objektu právě prováděné metody, opět obdobně jako u explicitního příjemce.
- Globální jmenný prostor.

Toto pořadí zajišťuje, že sloty, které byly vytvořeny ve více zanořených blocích, překryjí sloty se stejným názvem, jež byly vytvořeny v méně zanořených blocích. Zároveň je zajištěno očekávané chování z hlediska dědičnosti.

4.2.8 Asynchronní zprávy

Jak bylo popsáno dříve, v kapitole pojednávající o zprávách, asynchronní zprávy jsou vyhodnoceny v samostatných vláknech. Jako asynchronní může být použita jakákoliv zpráva, a to tak, že se před její název napíše znak „@“. Zpráva pak místo své skutečné hodnoty vrátí objekt typu *Future* reprezentující budoucí skutečný výsledek. Objekt tento výsledek může získat zprávou *value* objektu Future. Pokud byl již výsledek mezitím spočten, je okamžitě vrácen. V opačném případě se volající pozastaví do doby, než bude výsledek k dispozici. Dodejme, že virtuální stroj na úrovni jazyka nezajišťuje žádný výlučný přístup ke sdíleným objektům a programátor sám musí toto zajistit pomocí mechanismů vzájemného vyloučení.

Každý objekt má vlastní frontu zpráv, do které se jeho asynchronní zprávy řadí. Zašleme-li těsně po sobě jednomu objektu dvě asynchronní zprávy, nejsou vyhodnoceny paralelně, ale druhá zpráva vyčkává ve frontě, než je vyhodnocení první zprávy dokončeno

Vzájemné vyloučení je možné realizovat pomocí objektů, jež jsou potomky objektu *Mutex*. I objekt samotný je možné použít pro vzájemné vyloučení, nicméně se to nedoporučuje. Místo toho je vhodnější tento objekt naklonovat a používat jeho potomky. Vlastní vzájemné vyloučení se pak realizuje zprávami *lock* a *unlock*.

4.2.9 Gramatika jazyka

Syntaxe je popsána pomocí EBNF[17]. Zápis obsahuje speciální terminální symboly *alpha*, *alpha-id* a *num*. Ty nejsou z úsporných důvodů vysvětleny pomocí EBNF, ale budou zde popsány slovně. Gramatika je potom zobrazena na obrázku 4.7.

- **alpha-id** je množina písmen malé a velké abecedy, které se nacházejí v první polovině znakové sady ASCII[19] – tj. znaky bez diakritiky.
- **alpha** je množina všech tisknutelných znaků sady ASCII (tj. přibližně od znaku 20_{hex}).

- **num** je množina všech číslic (ASCII znaky 30 hex až 39 hex) a dále pak písmen „a“ až „f“ nebo „A“ až „F“ přičemž písmena jsou aplikovatelná pouze pro pravidlo *number* a pouze, pokud odpovídající text začíná znaky „0x“.

```
program        = [stats];
stats          = [stats], stat;
stat           = „;“ | expr, („;“ | message-chain, decl)
               | message-chain, assign;
decl           = „:=“, (expr, | matrix-lit | array-lit), „;“;
assign         = „=“, (expr | matrix-lit | array-lit), „;“;
expr           = expr,
               („+“ | „-“ | „*“ | „/“ | „%“ | „<“ | „>“
               | „==“ | „!=“| „<=“ | „>=“| „&&“ | „||“), expr
               | „!“, expr
               | message-chain
               | „(“, expr, „)“, [(„.“, message-chain | „[“,
                 am-selector, „]“)]
               | block, [„.“, message-chain]
               | „(“, ( matrix-lit | array-lit), „)“;
message-chain = message, ([„.“, message-chain
               | „[“, am-selector, „]“, [„.“, message-chain]);
message        = msg-id-target, message-rest;
msg-id-target = msg-id | string | [-], number | fnumber | complex
               | fcomplex;
message-rest  = [„(“, „msg-arglist“, „)“], [block];
msg-arglist   = arg, { „,“, arg };
arg            = (expr | array-lit | matrix-lit);
block          = „{“, [stats], „}“;
matrix-lit     = „[“, [am-int] „]“;
array-lit      = „[|“, [ am-int ], „|]“;
am-in          = am-row, [ „;“, am-row];
am-row         = expr, { „,“, expr };
am-selector    = am-sel-in, [ „,“, am-sel-in ];
am-sel-in      = expr | „:“ | expr, „:“, expr;
msg-id         = [„@“], („_“ | „alpha-id“), {( „_“ | „alpha-id“ |
               „number“)};
string         = ‘“‘, { „alpha“ }, ‘“‘;
number         = [-], [„0“, [„x“]], „num“;
fnumber        = [-], „num“, „.“, „num“;
complex        = [-], „num“, „i“;
fcomplex       = [-], „num“, „.“, „num“, „i“;
```

Obrázek 4.7: Gramatika navrženého jazyka zapsaná v EBNF

4.2.9.1 Priorita operátorů

Priorita a asociativita operátorů (operátorových zpráv) je dána tabulkou 4.3 a vychází z běžných konvencí

Priorita	Operátory	Asociativita
1	()	Zleva doprava
2	[]	Zleva doprava
3	!	Zprava doleva
4	.	Zleva doprava
5	* / %	Zleva doprava
6	+ -	Zleva doprava
7	< <= => >	Zleva doprava
8	== !=	Zleva doprava
9	&&	Zleva doprava
10	||	Zleva doprava
11	:	-

Tabulka 4.3: Priorita a asociativita operátorů

4.3 Implementace řídících konstrukcí

Řídící struktury byly implementovány pomocí zpráv a bloků kódu a jejich syntaxe i sémantika je podobná řídícím strukturám známým z jiných jazyků. Jazyk zároveň programátorovi poskytuje prostředky pro vytváření vlastních řídících struktur pomocí mechanismu bloků a stávajících struktur.

4.3.1 Větvení

Větvení je realizováno pomocí zprávy *if* objektu Block a následně pomocí zpráv *elseif* a *else* objektu *Cond.* Je možné vyhodnocovat nejen podmínky s jednou větví, ale také podmínky s obecně více větvemi. Zpráva *if* očekává dva argumenty, a to objekt typu Boolean a blok kódu, který se provede, pokud je první argument roven True. Výsledkem je objekt typu *Cond*, který rozumí zprávám *elseif* nebo *else*. První zpráva má opět dva argumenty – objekt typu Boolean a blok kódu. Její sémantika je stejná jako u zprávy *if*. Zpráva *else* má pouze jeden argument, a to blok kódu. Ten se vyhodnotí, pokud výraz předaný zprávě *if* nebo výraz předaný poslední zprávě *elseif* byl roven False. Obrázek 4.8 ukazuje příklad takové konstrukce

```
if(a > b)
{
      „a je vetsi".println();
}.elseif(a == b)
{
      „jsou si rovny".println();
}.else
{
      „a je mensi".println();
};
```

Obrázek 4.8: Příklad větvení pomocí if-elseif-else

4.3.2 Cykly

4.3.2.1 Cyklus s podmínkou na začátku

Cyklus s podmínkou na začátku je realizován zprávou *while* objektu Block. Zpráva má dva argumenty typu blok kódu. Druhý blok se bude provádět tak dlouho, dokud bude výsledkem prvního bloku výraz True. Podmínka zde musí být na rozdíl od větvení předána také jako blok, protože bude metodou reagující na zprávu „while" prováděna opakovaně. Na obrázku 4.9 je příklad cyklu s podmínkou na začátku.

```
while({a != 5;})
{
      a = a – 1;
      if (a == 0) { break; };
};
```

Obrázek 4.9: Příklad cyklu s podmínkou na začátku

Provádění cyklu je možné ukončit zprávou *break* objektu Block. Ta ukončuje vždy nejvíce zanořený cyklus. Pokud byla zpráva break poslána bloku mimo cyklus, chová se stejně jako zpráva *return.* Ta je přijímána též blokem kódu, ukončuje právě prováděnou metodu a je ji možné zavolat v kterémkoliv bloku, a to i mimo cykly. Nese jeden argument, což je výraz, jehož výsledek bude vrácen jako výsledek metody.

4.3.2.2 Cyklus s podmínkou na konci

Cyklus s podmínkou na konci se realizuje pomocí zprávy „*until*" objektu Block. Zpráva má jeden argument, a to blok kódu s podmínkou. Blok, kterému byla zpráva poslána, se provádí tak dlouho, dokud je výsledkem vyhodnocení bloku předaného jako parametr objekt True. Obrázek 4.10 ukazuje příklad této konstrukce. I tento typ cyklu je možné přerušit zprávou „*break*".

```
{
     i.println();
     i = i + 1;
}.until({i < 10;});
```

Obrázek 4.10: Příklad cyklu s podmínkou na konci

4.3.2.3 Nekonečný cyklus

Poslední variantou je nekonečný cyklus. Je realizován zprávou „*repeat*“, která se zasílá bloku kódu. Tento typ cyklu je možné ukončit pouze zprávou break. Na obrázku 4.11 je příklad takového cyklu.

```
{
     a.read();
     if (a == 0) { break; };
}.repeat();
```

Obrázek 4.11: Příklad nekonečného cyklu

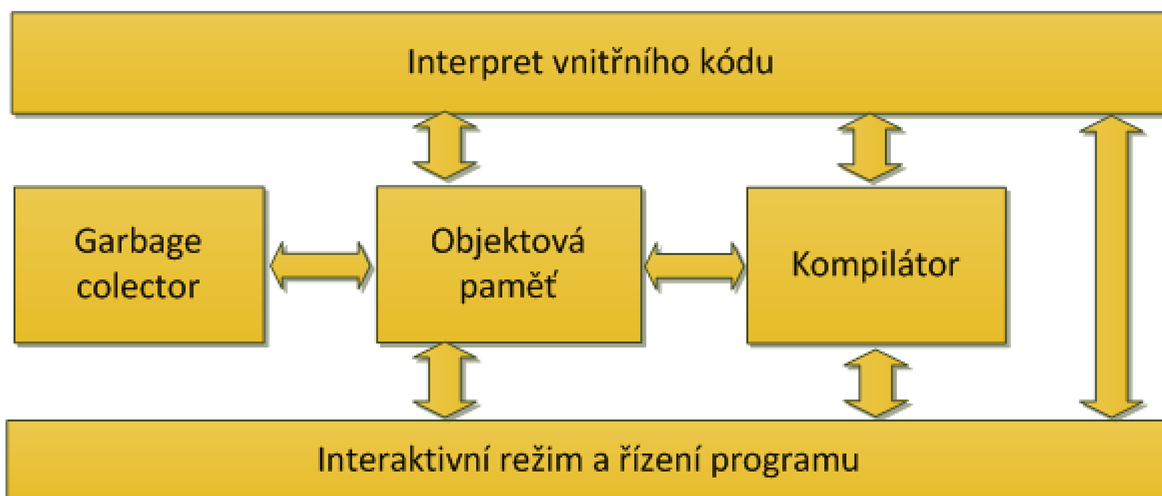
5 Návrh a realizace virtuálního stroje

Virtuální stroj je implementován v jazyce C++ s využitím principů objektově orientovaného programování. Jedná se o program bez grafického uživatelského rozhraní, který je ovládán přes terminál. Kromě interaktivního režimu, ve kterém uživatel s programem komunikuje stylem dotaz-odpověď, bude možné specifikovat textový soubor se zdrojovým kódem jako parametr při spuštění. Takový soubor pak bude načten a zpracován předtím, než se přejde do interaktivního režimu. Pokud v průběhu provádění souboru dojde k chybě, je provádění ukončeno a program přejde do interaktivního režimu okamžitě.

Je také možné specifikovat soubor s obrazem objektové paměti, který je načten místo výchozího objektového prostředí.

5.1 Základní koncept

Virtuální stroj se skládá z několika částí. První z nich je *překladač*, který na základě programu v navrženém jazyce generuje vnitřní kód. Druhou částí je samotný *interpret*, jenž umí tento vnitřní kód vykonávat. Další částí je *objektová paměť* zodpovídající za ukládání a správu objektů jazyka. Součástí této paměti je i jednoduchý *garbage collector*, který se bude starat o průběžné rušení nepotřebných objektů. V našem prototypovém řešení jsme garbage collector neimplementovali. Objektová paměť je také spoluodpovědná za načítání a ukládání *perzistentního obrazu* (image), který zachycuje celou objektovou paměť v době jeho uložení a umožňuje při příštím spuštění pokračovat od tohoto místa. Poslední částí je interaktivní interpret, který zajišťuje komunikaci s uživatelem. Obrázek 5.1 ukazuje blokové schéma celého systému a zachycuje základní souvislosti mezi jeho částmi.



Obrázek 5.1: Blokové schéma virtuálního stroje

5.2 Objektová paměť

Tato kapitola se zabývá návrhem a popisem implementace objektové paměti a způsobem reprezentace objektů jazyka.

Objektová paměť je zodpovědná za uchovávání objektů a spolupracuje s garbage collectorem na odstraňování již nepotřebných objektů. Virtuální stroj si za tímto účelem udržuje jeden velký seznam všech existujících objektů, který je pak periodicky prohledáván.

Jak již bylo několikrát zmíněno, s objekty se na úrovni jazyka pracuje pomocí referenci. Tyto reference jsou na úrovni virtuálního stroje implementovány pomocí klasických paměťových ukazatelů. Protože se umístění objektů v paměti během jejich života nemění, je jejich paměťová adresa zároveň využita jako jednoznačná identifikace.

5.2.1 Implementace objektové paměti

Objektová paměť je implementována třídou *CMemoryManager*. Tato třída obsahuje kolekci ukazatelů na všechny existující objekty. Kromě toho obsahuje ještě několik samostatných ukazatelů na různé význačné objekty, zejména objekt Lobby a jeho sloty. Tyto ukazatele jsou dostupné prostřednictvím metod třídy CMemoryManager pro ostatní části virtuálního stroje, například pro implementace primitiv. Pro některé z těchto objektů jsou navíc k dispozici metody zajišťující vytvoření nové instance objektu jazyka, jeho správné navázání do stromu dědičnosti a uložení do seznamu všech vytvořených objektů.

Třída je dále zodpovědná za inicializaci počátečního stromu objektů a slotů objektu Lobby. K tomu slouží metoda *initDefaultEnviroment*, která nejprve vytvoří základní objekty, uloží je do seznamu objektů a ukazatele na ně do samostatných ukazatelů na objekty. Dále zajistí inicializaci jejich základních slotů pomocí volání *initBasicSlots*. Nakonec jsou vytvořeny sloty objektu Lobby, které se naplní ukazateli na objekty vytvořené v předchozím kroku.

5.2.2 Objektový model

Základem pro reprezentaci objektů jazyka na úrovni virtuálního stroje je třída *CObjectBase*. Ta obsahuje funkcionalitu určenou pro všechny objekty uchovávané objektovou pamětí. Jedná se o ukazatel na předka a tabulku slotů, která bude popsána dále. Kromě toho obsahuje ještě logické příznaky určující, zda je možné objektu změnit předka nebo zda se jedná o singleton. Objekty označené jako singleton primitiva pro klonování nekopíruje. Toto je použito například u prototypových objektů nebo u Lobby.

Dalším důležitým atributem je typ třídy, který je dán výčtem. Ten určuje skutečnou třídu jazyka C++, jejíž instancí daný objekt je. Třída CObjectBase totiž není určena k přímé instanciaci. Místo toho se vytvářejí instance jejích potomků, které představují různé typy objektů. Například

existuje třída *CObject*, která představuje obecný objekt jazyka a jejími instancemi jsou objekt *Object* a jeho potomci. Tato třída je označena typem *OT_GENERIC*. Jiným příkladem může být třída *CNum* implementující funkcionalitu objektu *Number*. Ta má typ *OT_NUM*. Typ třídy se pak využívá pro bezpečné přetypování z CObjectBase, při zjednodušené typové kontrole u primitiv či u deserializace objektové paměti z perzistentního obrazu.

CObjectBase také implementuje základní primitivy, které jsou dostupné u všech objektů. Jedná se například o klonování nebo o manipulaci se sloty. Díky hierarchii dědičnosti v C++ není nutné tyto primitivy implementovat u potomků. Jakmile je například objektu String zaslána zpráva „addSlot“, je odpovídající slot nalezen u objektu Object, který ji přidal do své tabulky slotů. Objekt Object je instancí třídy CObject. Samotná primitiva pro přidání slotu je pak implementována u jejich společného předka, třídy CObjectBase a může být tedy bezpečně provedena.

Skutečnost, že různé objekty jsou na úrovni virtuálního stroje implementovány různými třídami, je také důvodem, proč nelze měnit rodičovský slot objektů libovolně, například z Object na Number. Takový objekt by pak byl sice tvořen instancí třídy CObject, ale fungovaly by pro něj i sloty, které jsou implementovány pouze pro třídu CNum. Při zavolání primitivy by pak nejspíš došlo k pádu programu z důvodu chybné práce s pamětí. Implementace změny rodičovského slotu musí tedy zajistit, že bude možné měnit rodiče pouze na objekt, který je z hlediska implementace ve virtuálním stroji typově ekvivalentní s původním rodičovským objektem. V praxi se to dá realizovat tak, že nový rodič musí být stejného typu (např. OT_GENERIC) jako původní rodič a tedy i jako je měněný objekt

5.2.3 Sloty

Sloty jsou implementovány jako instance třídy *CSlot*. Objekty této třídy se používají nejen pro uložení slotů v tabulce slotů, ale také pro realizaci registrů, či jiných struktur, které mají nést objekty jazyka. Dále je třída zodpovědná za většinu typových kontrol.

Samotná data jsou uložena ve standardním unionu jazyka C++. Struktura umožňuje uložit buď ukazatel na objekt nebo primitivu, celé nebo desetinné číslo, logickou hodnotu, nebo také komplexní číslo s reálnou i imaginární částí. Třída tedy rozlišuje několik typů slotů, jež jsou specifikovány výčtovým typem:

- **Primitiva** – obsahuje ukazatel na odpovídající metodu určité třídy, která tuto primitivu implementuje.
- **Ukazatel na objekt** – obsahem slotu je ukazatel na CObjectBase. Tento typ slotu je použit i v případě metody, kdy je odkazovaným objektem interní reprezentace metody (bude popsáno dále).
- **Hodnota vestavěného typu** – použije se v případě čísel a logických hodnot. Typ slotu dále specifikuje typ uložené hodnoty.

- **Null –** Hodnota Null je též realizována jako typ slotu.

Slot může být pojmenovaný. Pak kromě textové reprezentace jména nese dva hashe vytvořené z textového názvu. Ty jsou využity při vyhledávání v tabulce slotů k efektivnímu porovnávání. Dva hashe byly použity proto, aby se minimalizovala možnost kolize jmen u dvou různých zpráv. Jako hashovací funkce byly zvoleny *SuperFastHash* a *FNV-1*, jejichž volně šiřitelné implementace jsou běžně dostupné na internetu.

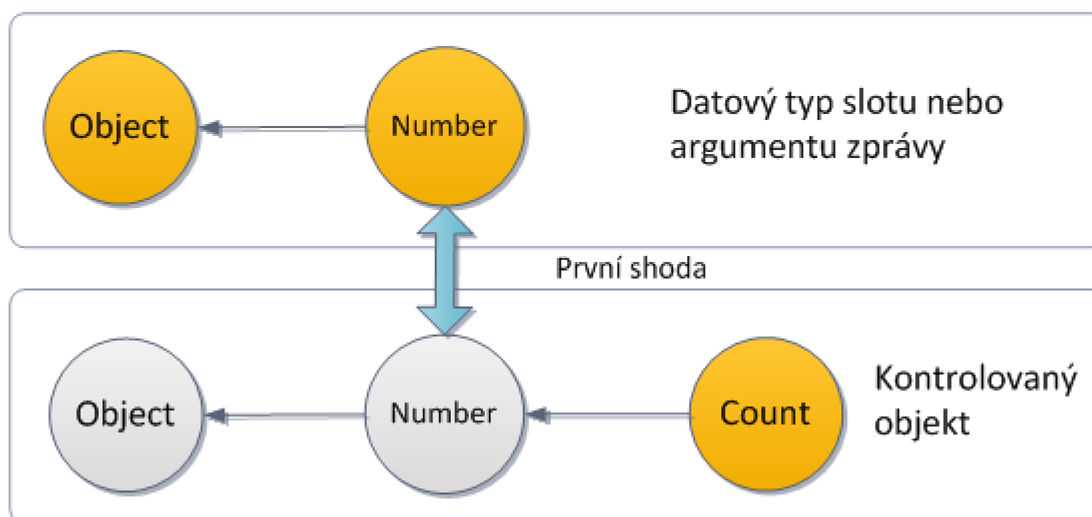
Slot také obsahuje dva příznaky udávající, zda je pouze pro čtení a zda je viditelný. Neviditelné sloty nejsou zahrnuty do seznamu slotů generovaných primitivou *listSlots* ani nejsou prohledávány při vyhledávání slotu v tabulce. Pojmenování neviditelných slotů obvykle začíná znakem „#“, aby se zajistila nekonfliktnost jména s případným jiným viditelným slotem.

5.2.3.1 Datový typ a typová kontrola

Slot může být typovaný. Pak kromě vlastního obsahu nese také ukazatel na objekt, který figuruje jako datový typ. Při přiřazování do typovaného slotu je potom vždy provedena i typová kontrola.

Jak již bylo řečeno výše, datový typ objektu je dán jeho pozicí ve stromové hierarchii dědičnosti. Typicky existuje nějaký prototypový objekt, který definuje operace pro daný typ, a objekty určené pro reprezentaci konkrétních věcí jsou potomky toho prototypového objektu.

Třída CSlot provádí typovou kontrolu podle typu předané hodnoty. Pokud je ukládanou hodnotou reference, je typová kontrola realizována jako jednoduché procházení objektů postupně od ověřovaného objektu, přes jeho rodičovský slot směrem ke kořenu stromu dědičnosti. Každý objekt je pak porovnáván na shodu s prototypovým objektem. Procházení končí, jakmile je nalezena shoda, nebo je dosaženo vrcholu stromu (objekt Object) bez jejího nalezení. Pak kontrola skončí neúspěchem a je ohlášena chyba nesouhlasu datových typů. Obrázek 5.2 schematicky zobrazuje průběh typové kontroly pro nějaký objekt „Count“.



Obrázek 5.2: Typová kontrola

Pokud je ukládanou hodnotou vestavěný typ, provádí se pouze ověření typu objektu, který je uložen jako datový typ. Například v případě celého čísla je zjišťováno, zda je datovým typem objekt typu OT_NUM, tedy objekt Number. Toto je také důvod, proč nelze vestavěné typy ukládat do slotů s datovým typem Object. Ten je totiž typu OT_GENERIC.

U hodnoty Null projde typová kontrola vždy s úspěchem. Sloty obsahující primitivy by neměly mít specifikovaný datový typ.

Datový typ slotu lze měnit. V tom případě se ale ověří, že objekt ve slotu obsažený, je i cílového datového typu. Důležité je zde zmínit, že kontrola datového typu u slotů je jednorázová. Může se tedy stát, že programátor přiřadí objekt nějakého typu do slotu stejného typu a následně změní rodiče objektu, čímž zároveň změní i jeho datový typ. Ten pak již nemusí být kompatibilní s typem slotu. Toto již třídou CSlot není ověřováno a programátor na sebe změnou typu objektu bere zodpovědnost za tuto akci stejně, jako je tomu i u přetypování v jazyce C. Ta samá situace může nastat, pokud je objekt odkazován z více míst. Do naší prototypové implementace virtuálního stroje nebyla zněna datového typu slotu zahrnuta.

5.2.4 Tabulka slotů

Tabulka slotů je realizována jako hashovací tabulka s dynamickou změnou velikosti. Při vyhledávání slotu se využije nejprve první hash, který se vydělí modulo velikostí tabulky. Tím získáme číslo pozice v tabulce. Tabulka umožňuje do jedné pozice uložit více položek ve formě krátkého lineárního seznamu. Ten se pak prochází sekvenčně a u každé položky se porovnají oba hashe. V případě nalezení shody je slot vrácen. V opačném případě je vráceno NULL.

Aby se prohledávání příliš nezpomalilo procházením dlouhého lineárního seznamu, jsou při každém prohledávání počítány kroky vedoucí k nalezení slotu a stanovuje se maximální hodnota ze všech prohledávání. Jakmile ta dosáhne určité hranice, je tabulka zvětšena. V tom případě je nutné všechny položky tabulky umístit znovu, aby korespondovaly s novým výsledkem dělení modulo prvního hashe. Domníváme se, že vzhledem k diferenční dědičnosti bude většina objektů obsahovat jen velmi málo slotů a výpočetní náročnost této operace bude tedy velmi malá.

5.2.5 Primitivy

Primitivy jsou implementovány jako standardní metody třídy CObjectBase a jejích potomků. Slot pak obsahuje ukazatel typu *VmPrimitive*, což je ukazatel na metodu třídy CObjectBase. Primitiva je volána se třemi parametry. Prvním parametrem je ukazatel na objekt třídy *CVmThread* představující aktuální stav vlákna virtuálního stroje, ze kterého byla primitiva zavolána. Druhým parametrem je ukazatel na objektovou paměť. Poslední parametr udává počet argumentů, se kterými byla primitiva zavolána. Kontrola správného počtu a typu argumentů je pak čistě v její režii. Tím je možné zajistit,

že primitiva může pracovat s různým počtem argumentů různých datových typů podle toho, jak je zrovna vyžadováno.

Návratovou hodnotou primitivy je výčtová hodnota typu *VmReturnValue*. Ta může být buď *RV_OK*, pokud je vše v pořádku, případně *RV_BREAK, RV_RETURN, RV_SAVE* nebo *RV_EXIT*. Význam posledních čtyř zmiňovaných návratových hodnot bude vysvětlen dále.

5.2.6 Prototypové objekty pro vestavěné datové typy

Třídy realizující operace nad základními datovými typy jsou koncipovány tak, aby je bylo možné „recyklovat" pro opakované použití u různých hodnot. Pokud je zpráva zaslána hodnotě vestavěného datového typu, virtuální stroj vybere odpovídající prototypový objekt, hodnotu do něj uloží a poté provede standardní vyhledání zprávy, jako u jiných objektů. Po provedení operace je výsledek uložen zpět místo dočasného objektu. Dočasný objekt je následně resetován do původního stavu, aby nebyly zanechány žádné vedlejší efekty v podobě uložené hodnoty. Přesný postup zasílání zpráv bude popsán dále.

Implementačně jsou takové třídy implementovány jako potomci třídy *CRecyclable*. Ta zajišťuje společné vlastnosti, například že objekty budou mít na pevno nastaveného předka, nebo že budou singletony.

Objektům Number, FNumber, Complex a FComplex odpovídají v tomto pořadí třídy *CNum*, *FNum*, *CComplex* a *CFComplex*. Objektu Boolean odpovídá třída CBoolean. Primitivy realizující aritmetické operace musí zajistit také implicitní konverzi mezi jednotlivými typy.

5.2.7 Matice a pole

Matice a pole jsou realizovány třídami *CMatrix* a *CArray*. Tyto třídy mají společného rodiče, a to třídu *C2dStructure*, která obsahuje většinu funkcionality pro oba objekty a umožňuje implementovat různé dvourozměrné struktury na ukládání prvků. Prvky jsou uloženy ve dvourozměrném poli realizovaném standardní kolekcí *vector* z knihovny STL. Prvek je uložen jako instance třídy CSlot, čímž je zajištěno, že matice (pole) může obsahovat jakoukoliv platnou hodnotu, kterou je možné uložit do jakéhokoliv jiného slotu. Navíc je v případě matic tímto zajištěna i standardní typová kontrola. Aby se zbytečně neplýtvalo pamětí, jsou ukazatele v prázdných pozicích dvourozměrné struktury nastaveny na NULL. Při přístupu k takové pozici, například pomocí selektoru, je vrácena hodnota Null. Naopak, je-li pomocí primitivy *setAt* nastavován obsah prázdné pozice, dojde automaticky k vytvoření nové instance třídy CSlot.

Matice upravuje rodičovskou třídou tak, že dodává další slot, který bude figurovat pro uložení datového typu. Zároveň je předefinována primitiva *setAt*. Ta při zavolání nejprve zkontroluje, zda již byl datový typ matice nastaven. Pokud ne, použije právě ukládanou hodnotu (předanou v podobě slotu) pro vytvoření instance třídy CSlot, která bude figurovat jako datový typ. Datovým typem pak

bude předek vkládaného objektu, pokud existuje. V případě vestavěných datových typů se pomocí ukazatele na objektovou paměť získá odpovídající prototypový objekt. Pokud je právě ukládanou hodnotou Null, je ohlášena chyba, protože z té nelze datový typ odvodit.

Vytvořený slot pro datový typ pak funguje jako šablona pro sloty nesoucí vlastní hodnoty matice. Při vkládání nového prvku je slot pro datový typ nakopírován, čímž dojde i ke kopii ukazatele na objekt figurující jako datový typ uvnitř slotu. Do takto vytvořeného slotu je poté nastavena vkládaná hodnota, čímž dojde ke standardní typové kontrole a k zajištění, že všechny prvky matice budou stejného datového typu.

Skutečnost, že slot reprezentující datový typ funguje jak šablona pro vytváření slotů pro hodnoty, je také důvodem, proč nelze pro první hodnotu matice použít Null. Pokud bychom vytvořili slot na základě hodnoty Null, dostali bychom netypovaný slot, do kterého je možné uložit cokoliv a nebyl tím splněn požadavek, že prvky matice musí být stejného datového typu. Ze stejného důvodu rovněž nelze odvozování datového typu v takové situaci přeskočit a ponechat „na později", až bude k dispozici vhodná vkládaná hodnota. Pak by byl sice zbytek matice typovaný, nicméně prvek, do kterého bylo vloženo Null by byl netypovaný. Jedinou možností by byla dodatečná typová kontrola takových slotů.

5.2.8 Klonování objektů

Klonování objektů je operace, která je specifická pro různé typy objektů. Abychom nemuseli implementovat primitivu pro všechny typy objektů, je implementována obecně v třídě CObjectBase. Zde je také zajištěno, že objekty s nastaveným příznakem singleton nebudou klonovány. Samotné klonování, které je specifické pro každý typ třídy, je realizováno pomocí virtuální metody *clone*. Ta je implementována u každé třídy a vytváří pomocí správce paměti novou instanci objektu. Instance je vytvořena za pomoci kopírovacího konstruktoru, jemuž je předán ukazatel *this*. Každý kopírovací konstruktor nejprve volá kopírovací konstruktor nadřazené třídy, až se takto dojde na kopírovací konstruktor CObjectBase. Ten kromě inicializace struktur nastaví rodičovský slot na objekt, ze kterého je kopírováno. Poté se v opačném pořadí provádí kód předchozích kopírovacích konstruktorů, až je vytvořena korektní kopie celého objektu. Objekt je poté primitivou pro klonování vrácen jako výsledek.

5.2.9 Perzistentní obraz

Perzistentní obraz ukládá stav objektové paměti v okamžiku jeho vytvoření. Obraz vznikne částečnou serializací jednotlivých objektů a jejich slotů. Aby se uspořilo místo a zároveň zjednodušila implementace, jsou ukládány pouze ty sloty, které nejsou označeny pouze pro čtení (například primitivy). Ukazatele na objekty uložené ve slotech jsou samozřejmě platné pouze za běhu programu. Při serializaci jsou proto překládány na číselné hodnoty pomocí třídy *CPtrToIdTranslator*. Tato třída

zajistí, že stejné ukazatele budou přeloženy na stejné identifikátory. Ukazatel NULL se pak přeloží jako identifikátor 0.

Soubor s obrazem je potom tvořen krátkou hlavičkou, posloupností uložených objektů a patičkou. Hlavička souboru obsahuje signaturu, což je posloupnost znaků „MPLUSIMG“. Následuje jeden rezervní bajt, který je vždy nastaven na nulu. Za tímto bajtem jsou vyskládány jednotlivé objekty. U každého objektu je nejprve uložen jeho typ a dále pak jeho číselný identifikátor. Poté následují samotná data objektu. Za posledním uloženým objektem je uveden typ OT_NA, který značí, že poslední načtený objekt byl i poslední uložený. Následují identifikátory význačných objektů, které mají být uloženy do samostatných ukazatelů ve správci paměti. Soubor je ukončen kontrolní posloupností znaků „MPLUSIMGEND“. Za proces serializace a deserializace je odpovědný správce paměti. Operace jsou pak volány prostřednictvím metod *serialize* a *deserialize* třídy *CMemoryManager*.

5.2.9.1 Serializace

Při serializaci je nejprve zapsána hlavička souboru. Poté se projde celý seznam objektů a pro každý se nejprve zapíše jeho typ a následně je vygenerováno jeho číselné označení, které se rovněž zapíše.

Serializace vlastních dat se liší pro každou třídu. Opět se však využívá dědičnosti. Každá třída implementuje virtuální metodu *serialize*, která přijímá ukazatel na výstupní soubor, ukazatel na správce paměti a dále ukazatel na objekt překládající ukazatele na číselná označení. Serialize pro danou třídu pak nejprve volá stejnou metodu svého předka a následně provede uložení svých vlastních dat. Ukládají se pouze ty informace, které jsou nutné k obnovení stavu objektu. Například třída CObjectBase neukládá informace z objektu tabulky slotů, ale pouze sloty samotné. Zde obecně platí, že pokud objekt obsahuje nějakou složenou strukturu (kromě jiného objektu), jsou její data uložena za data objektu.

Po zapsání všech objektů je vloženo označení OT_NA a uložena číselná označení význačných objektů. Nakonec je zapsán ukončující řetězec.

5.2.9.2 Deserializace

Deserializace probíhá ve dvou hlavních fázích. V první fázi je nejprve přečtena a ověřena hlavička. Následně se postupně načítají jednotlivé uložené objekty. Podle zaznamenaného typu se u odpovídající třídy zavolá statická metoda *newInstane*, jež vytvoří nový objekt požadovaného typu. Ukazatel na objekt se spolu s identifikátorem uloží do objektu třídy *CIdToPtrTranslator*, která zodpovídá za zpětný překlad číselných označení na ukazatele. Poté je u vytvořeného objektu zavolána metoda *deserialize*, která má podobné rozhraní, jako *serialize*. Opět se nejprve volá stejná metoda předka, která zajistí načtení jeho dat a potom se provede načtení a ověření vlastních dat objektu. Případné číselné identifikátory odpovídající ukazatelům jsou do objektu dočasně uloženy.

Takto je přečtena a ověřena ta část souboru, která obsahuje serializované objekty. V další fázi se pro každý objekt v paměti zavolá metoda *initPointers,* zodpovídající za obnovení ukazatelů na objekty z identifikátorů uložených v předchozí fázi. Opět se nejprve zavolá metoda předka a až potom je provedena inicializace vlastních ukazatelů. Metoda může kromě toho provádět další dodatečné nastavení hodnot a datových struktur. Tímto jsou obnoveny vazby mezi jednotlivými objekty.

Poslední operací je načtení identifikátorů význačných objektů, jejich překlad na ukazatele a uložení do správce paměti. Ověří se také správnost ukončujícího řetězce. Nakonec se do každého objektu přidají sloty s primitivami a vestavěnými objekty. Nastaví se rovněž sloty objektu Lobby. Tím je objektová paměť plně obnovena.

5.3 Interpret vnitřního kódu

Interpret bytekódu provádí interpretaci kódu vygenerovaného překladačem a je implementován třídou *CVmThread*. Virtuální stroj byl navržen jako zásobníkový s jedním obecným registrem a několika registry pro specifický účel. Tuto architekturu jsme zvolili pro její jednoduchost a relativně snadnou implementaci překladače.

5.3.1 Interní reprezentace metod a bloků

Interní reprezentace bloků a metod se používá pro zachycení „statických“ vlastností definovaných metod a bloků. Právě prováděná metoda nebo blok jsou navíc popisovány svou běhovou reprezentací, která bude popsána dále.

5.3.1.1 Reprezentace metody

Metoda je reprezentována objektem třídy *CInternalMethod*, která je potomkem třídy CObjectBase. Tento objekt není vytvořen při překladu, ale až primitivou *method* a je uložen do odpovídajícího slotu objektu, nad kterým byla primitiva provedena.

Objekt této třídy neobsahuje žádné vestavěné sloty, pouze sloty případných argumentů metody. Ty jsou vytvořeny jako typované, přičemž datovým typem je vždy objekt předaný jako datový typ argumentu a název slotu je nastaven podle názvu argumentu. Takto nachystané sloty pak slouží jako šablony pro vytváření slotů pro skutečné argumenty v okamžiku zavolání metody. Toto bude popsáno dále.

Nakonec interní reprezentace metody obsahuje ukazatel na interní reprezentaci předaného kořenového bloku kódu, který bude tvořit tělo metody.

5.3.1.2 Reprezentace bloku

Bloky kódu vznikají už při překladu ze zapsaných literálů. Blok je implementován jako instance třídy *CInternalBlock*, která je potomkem třídy CObjectBase. Instance nese přeložený bytekód jako

posloupnost struktur *VmInstruction*. Tyto struktury mohou obsahovat jak instrukce, tak i vestavěné datové typy. Mají tedy podobnou podobu jako struktura nesoucí obsah slotu.

Kromě přeloženého bytekódu obsahuje objekt také kolekci ukazatelů na lexikálně zanořené bloky a ostatní objekty, které byly ve zdrojovém zapsány v podobě literálů uvnitř tohoto bloku. Pro upřesnění dodejme, že odkazovány jsou pouze literály na nejméně zanořené úrovni. Strom zanořených bloků ve zdrojovém kódu pak koresponduje se stromem objektů třídy CInternalBlock v objektové paměti.

Toto řešení bylo zvoleno ze dvou důvodů. Prvním je především skutečnost, že bez seznamu zanořených bloků by tyto nebyly nijak provázány a při provádění programu by nebylo možné zjistit, do jakého lexikálního kontextu daný blok patří. Dalším důvodem byla potřeba zajistit, že interní bloky kódu a jiné literály budou odněkud dosažitelné a především budou takto svázány s interní reprezentací metody. Pokud by tomu tak nebylo, mohlo by se stát, že některé zanořené objekty by nebyly odnikud odkazovány. Garbage collector by je pak vyhodnotil jako nedosažitelné a z paměti by je odstranil.

Toto řešení také podporuje filozofii, že objekty v objektové paměti tvoří jeden velký strom. Metoda, která je uložena ve slotu nějakého objektu stromu, pak sama obsahuje strom objektů tvořících její tělo.

Dodejme, že interní bloky kódu neobsahují žádné sloty a z jazyka nejsou přímo dosažitelné. Při přístupu k literálu bloku totiž automaticky vzniká běhová reprezentace bloku, která je popsána dále.

5.3.2 Běhová reprezentace metod a bloků kódu

Běhová reprezentace vzniká pro každou interní metodu v okamžiku její aktivace. Smyslem běhové reprezentace je zachycovat data a vztahy specifické pro dané zavolání. Můžeme mít například metodu, která volá sama sebe rekurzivně. Pokud bychom pro ukládání metod používali pouze výše popsanou interní reprezentaci, nebylo by možné realizovat situaci, kdy byla daná metoda pokaždé zavolána s jinými hodnotami argumentů, což je v případě rekurze velmi obvyklý jev.

Obdobný problém nastává i u bloků kódu. V předchozí části práce bylo prezentováno, že lokální proměnné jsou realizovány jako sloty bloku. Pokud bychom tedy pro uložení bloku kódu použili pouze instance třídy CInternalBlock, nemohli bychom realizovat více oddělených běhů daného bloku (bytekódu) což by se projevilo opět například u výše zmíněného případu rekurze. Z tohoto důvodu jsme se tedy rozhodli realizovat obě jazykové konstrukce pomocí dvou objektů. Interní reprezentace pak nese statické informace, společné pro všechny instance a běhová reprezentace vyjadřuje jedno zavolání nebo provádění.

5.3.2.1 Reprezentace metody

Po nalezení slotu odpovídajícího zaslané zprávě je provedena akce v závislosti na typu slotu. Pokud virtuální stroj zjistí, že obsahem slotu je reference na objekt a objekt je typu OT_IMETHOD – jedná

se tedy o interní reprezentaci metody – není nalezený objekt uložen jako výsledek zprávy, ale místo toho je metoda provedena. V tuto chvíli virtuální stroj vytvoří nový objekt, jenž je potomkem objektu Method, který je dostupný jako slot Lobby. Vytvořený objekt bude sloužit jako běhová reprezentace právě zavolané metody. Objekt a jeho potomci jsou realizováni prostřednictvím třídy CMethod. K vytvořenému objektu jsou přidány sloty nesoucí konkrétní předané argumenty zprávy. Sloty jsou vytvářeny na základě těch z interní reprezentace metody obdobně, jako je tomu u nových prvků matic. Tímto je zajištěna typová kontrola u předaných argumentů. Dále je přidán slot *self*, jenž je nastaven na objekt příjemce zprávy. Objekt metody nakonec obsahuje také referenci na její interní reprezentaci a také ukazatel na běhovou reprezentaci kořenového bloku.

K objektu konkrétní metody nelze přímo přistoupit, lze pouze využívat jeho sloty, zejména slot self. Programátor nicméně může přidat nové sloty přímo do objektu Method a dodat tak do zavolaných metod vlastní funkcionalitu.

5.3.2.2 Reprezentace bloku

Prováděný blok je reprezentován objektem, jenž je potomkem objektu Block. Ten je na úrovni virtuálního stroje tvořen instancí třídy CBlock. Objekt implementuje primitivy realizující řídící konstrukce (například větvení) a dále pak primitivy pro zprávy *return*, *break*, *quit* a *exit,* realizující v tomto pořadí ukončení metody, ukončení cyklu a poslední dvě realizují ukončení programu. Mimo to je zde i primitiva *saveImage,* realizující uložení perzistentního obrazu. Ta má podobné chování jako zprávy exit a quit.

Běhový blok vznikne automaticky v okamžiku, kdy je při provádění bytekódu nalezena reference na literál bloku. Nově vytvořenému objektu je nastaven lexikální kontext na nadřazený (tj. právě prováděný) běhový blok. Zároveň se do vytvořeného bloku nastaví ukazatel na interní blok, nesoucí vlastní bytekód.

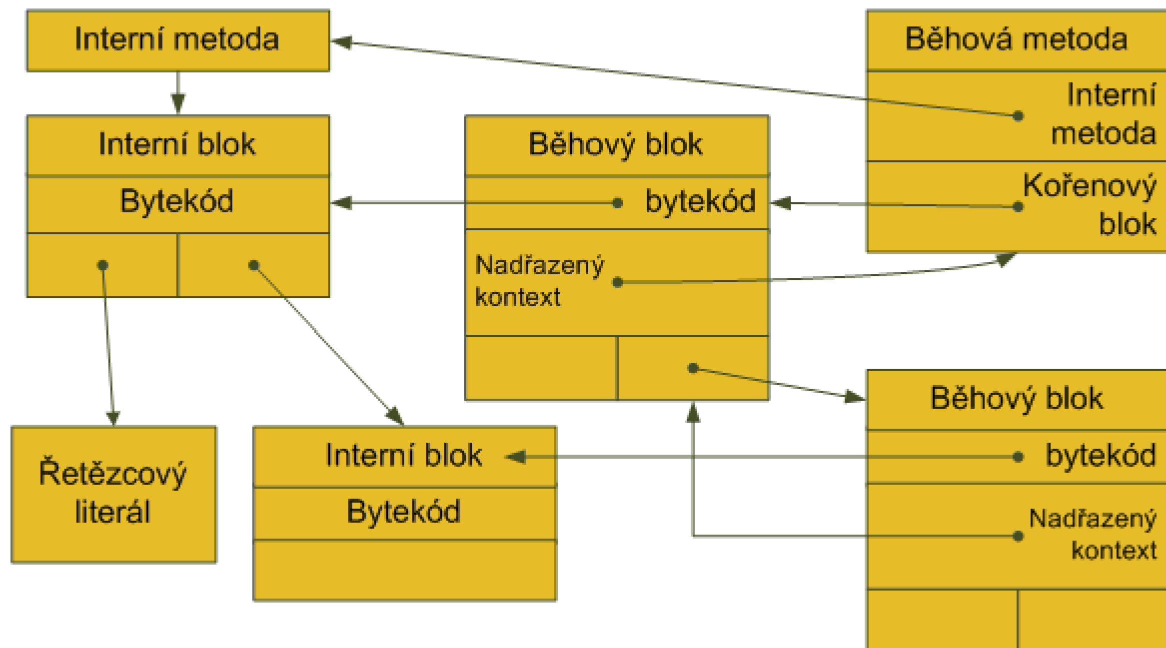
Poslední důležitou operací je příprava pole ukazatelů na zanořené literály. Pole vznikne na základě seznamu zanořených bloků, který je uložen u odkazované interní reprezentace. Všechny ukazatele takového pole jsou na začátku nastaveny na NULL, protože žádné zanořené běhové bloky doposud neexistují.

Blokové literály jsou v bytekódu odkazovány pomocí svého pořadového čísla. Toto číslo je odvozeno od pořadí bloků v seznamu zanořených literálů u interního bloku. A sem se literály dostávají v pořadí, v jakém byly překládány, tedy v pořadí zápisu ve zdrojovém kódu. Pořadové číslo také odpovídá indexu do pole zanořených běhových bloků u běhové reprezentace bloku.

Při nalezení reference na literál bloku je pak zmiňované pole zkontrolováno. Jestliže se na daném místě nachází NULL, znamená to, že běhový blok pro daný literál ještě nebyl vytvořen. V tom případě se vytvoří požadovaný blok způsobem popsaným výše. Pokud už na dané pozici nějaký ukazatel uložen je, znamená to, že běhový blok už vytvořen byl, například v předchozím kroku cyklu. Takový objekt pak stačí pouze *resetovat*, což v případě bloku znamená smazání jeho lokální tabulky

slotů, čím se smažou všechny lokální proměnné vytvořené v předchozím běhu. Tím se zajistí, že při opakovaném provádění bloku v cyklu není hlášena chyba při pokusu vytvořit lokální proměnnou. Kromě toho je ještě znovu nastaven ukazatel na právě prováděný blok, čímž se obnoví lexikální kontext.

Obrázek 5.3 ukazuje interní a běhové reprezentace bloků a metod pro nějakou hypotetickou metodu, jejíž kořenový blok obsahuje jeden zanořený blok a jeden řetězcový literál.



Obrázek 5.3: Souvislosti mezi jednotlivými objekty

Z výše uvedeného plyne důvod, proč nelze z jazyka nikdy přistoupit přímo k interní reprezentaci bloku. Odpovídající objekt není z jazyka dostupný (přesněji, je neviditelným slotem Lobby) a při přístupu k literálu je automaticky vytvořen běhový blok, který je provázán s nadřazeným blokem (lexikálním kontextem). Programátor tedy vždy pracuje pouze s běhovou reprezentací, které odpovídají potomci objektu Block.

5.3.3 Vytvoření nové metody

Jak bylo zmíněno v předchozím textu, nová metoda (její interní reprezentace) se vytváří až za běhu programu pomocí primitivy *method*, zatímco interní reprezentace bloku se vytváří již v době překladu. Primitiva prostřednictvím správce paměti vytvoří novou instanci třídy CInternalMethod a do této instance nastaví jednotlivé argumenty, pokud existují. Předaný blok byl předán ve své běhové podobě (viz předchozí kapitola) a je tedy provázán se svým lexikálním kontextem. V případě kořenového bloku však potřebujeme, aby byl svázán s nadřazenou metodou, nikoliv s kontextem, ve kterém byl zapsán jeho literál. Primitiva tedy z předaného běhového bloku vytáhne jeho interní reprezentaci a tu sváže s právě vytvářenou interní reprezentací metody.

5.3.4 Vytvoření dvourozměrných struktur z literálů

Obdobně jako metody, i pole a matice zapsané literálem se vytvářejí až za běhu programu. Toto řešení jsme zvolili pro jeho snadnou implementaci. Teoreticky by bylo možné objekty pro matice a pole vytvářet už v době překladu. Museli bychom však řešit situaci, kdy programátor zapíše prvek matice výrazem. Takový prvek by se musel vyhodnotit až v době běhu programu, v místě zápisu dvourozměrného literálu. Při překladu bychom pak museli rozlišovat, které prvky můžeme vytvořit již přímo v době překladu a které ne. Tím by se celá věc výrazněji zkomplikovala.

Vydali jsme se tedy cestou vytváření dvourozměrných struktur až za běhu. Jeho nevýhodou je o něco větší režie (vše se vždy musí vytvářet znovu), což může způsobovat jisté výkonnostní problémy, pokud by byl literál zapsán v cyklu. Zde však můžeme použít analogii s jazykem C, u kterého též není doporučeno vytvářet lokální proměnné v těle cyklu, i když překladač obvykle dokáže takovou proměnnou vytvářet ještě před vstupem do cyklu. Postup překladu dvourozměrných struktur bude popsán dále, v části zabývající se překladačem. Obdobný přístup byl zvolen také u *selektorů*.

5.3.5 Registry a zásobník

Virtuální stroj má několik registrů, z nichž některé jsou přímo modifikovatelné strojovým kódem a některé jsou dány virtuálním strojem a programově je měnit nelze, nebo jen nepřímo. Tyto registry jsou součástí třídy *CVmThread* a obsahují různé význačné hodnoty, které dohromady udávají výpočetní stav právě prováděného vlákna. Každé vlákno má vlastní instanci třídy CVmThread a tedy i vlastní sadu registrů.

Nejpoužívanějším je registr, který označujeme jako „RES“, *result*. U zasílání zpráv jeho obsah figuruje jako příjemce zprávy. Zároveň je do něj uložen i její výsledek. Obsahem tohoto registru mohou být jakákoliv data, která lze uložit do slotu každého objektu. Je to jediný registr, do kterého je možné přímo ukládat konkrétní hodnotu.

Dalšími registry jsou *Pointer to block* (značíme PB) a *Instruction pointer* (IP). Pointer to block obsahuje ukazatel na interní reprezentaci právě prováděného bloku kódu. Registr IP pak obsahuje číslo udávající pozici v rámci bytekódu tohoto bloku. Kombinace registrů PB a IP potom jednoznačně udává adresu právě prováděné instrukce. Registr IP se mění automaticky po každé instrukci o odpovídající počet položek *VmInstruction* (viz předchozí text). Registr PB se mění pouze při vstupu nebo výstupu ze standardní metody.

Další dvojící jsou registry PRM – *Pointer to Runtime method* a PRB – *Pointer to Runtime block*. PRM obsahuje ukazatel na běhovou reprezentaci aktuální metody. Obdobně registr PRB ukazuje na běhovou reprezentaci právě prováděného bloku.

Součástí virtuálního stroje je také zásobník, jehož jednotlivé položky, podobně jako registr RES, mohou nést stejná data, jako slot každého objektu. Zásobník je určen dvěma parametry, které

mohou mít charakter registrů. První parametr určuje adresu dna zásobníku. Vrchol zásobníku je pak dán druhým parametrem, vyjadřujícím počet obsazených položek zásobníku od jeho dna. Parametr určující počet položek v zásobníku lze měnit pouze nepřímo pomocí instrukcí pro manipulaci se zásobníkem. Parametr určující adresu dna zásobníku je čistě v režii virtuálního stroje a nelze jej měnit ani číst. Instrukční sada bude popsána v následující kapitole.

5.3.6 Popis základních instrukcí

Instrukce virtuálního stroje byly koncipovány na relativně vysoké úrovni abstrakce, čímž se usnadnil překlad poměrně abstraktního jazyka. Byly potom definovány instrukce pro zasílání zpráv, včetně zasílání asynchronních zpráv, práci se zásobníkem, s registrem RES. Nakonec byly implementovány některé speciální instrukce související s vytvářením objektů literálů matic a polí. Instrukce jsou z hlediska paralelního zpracování prováděny atomicky. Následuje popis implementovaných instrukcí:

- **MSG (message)** – zasílá zprávu objektu odkazovanému registrem RES. Výsledek zprávy je uložen opět v tom samém registru. Instrukce má dva parametry, a to označení zprávy, které se skládá ze dvou hashů a počet argumentů zprávy. Argumenty jsou uloženy postupně za sebou na zásobníku tak, že poslední argument tvoří jeho vrchol. Instrukce je implementována v několika verzích podle toho, jestli byly se zprávou předány nějaké argumenty a zda byla zaslána synchronně nebo asynchronně.
- **IMSG** – stejná instrukce jako MSG, ale zpráva je zaslána implicitnímu příjemci. Ten je vyhledán podle algoritmu, popsaného v kapitole 4.2.7.2. Výsledek je opět uložen do RES. I zde existuje několik verzí podle toho, zda zpráva nese argumenty, nebo zda byla zaslána jako asynchronní.
- **PUSH** – uloží obsah registru RES na vrchol zásobníku. Registr RES zůstane nezměněn.
- **POP** – z vrcholu zásobníku odstraní jednu položku. Instrukce může mít i argument, specifikující kolik položek se má ze zásobníku odstranit. Existují tedy dvě varianty. Registr RES zůstane nezměněn.
- **GET –** Do registru RES uloží ze zásobníku položku specifikovanou jako argument. Položka je udána číslem relativně od vrcholu zásobníku, přičemž 0 znamená vrchol zásobníku.
- **TOP –** Je obdobou instrukce GET, bere však vždy pouze položku z vrcholu zásobníku.
- **Sada instrukcí pro uložení hodnot vestavěných datových typů** – sem patří instrukce *SRES_NULL* ukládající do RES hodnotu Null, *SRES_NUM* s argumentem v podobě ukládaného celého čísla, podobně *SRES_FNUM* pro desetinné číslo, *SRES_COMPLEX* a *SRES_FCOMPLEX* pro celá a desetinná komplexní čísla.
- **RES_SLITERAL a SRES_BLITERAL –** ukládá do registru RES referenci na objekt reprezentující literál řetězce resp. bloku, který je specifikován svým pořadovým číslem v rámci nadřazeného (právě prováděného) bloku.

- **RES_NEW_ARR a SRES_NEW_MTX** – způsobí vytvoření nového objektu pole (matice) a uložení reference na něj do RES. Instrukce mají dva parametry, a to počet řádků a sloupců. Jejich využití je při vytváření objektů pro literály dvourozměrných struktur.
- **2D_SET_AT** – speciální instrukce, která umožňuje nastavit obsah prvku dvourozměrné struktury. Instrukce byla vytvořena pro snadné a rychlé nastavování prvků 2D struktur, které byly zapsány pomocí literálu. Podrobnosti budou osvětleny dále v části zabývající se překladačem. Instrukce má dva parametry, a to souřadnice řádku a sloupce. Hodnota prvku je brána z vrcholu zásobníku.
- **RES_NEW_AM_SEL –** způsobí vytvoření nového objektu selektoru a uloží referenci na něj do registru RES. Instrukce má dva parametry určující typ dimenzí selektoru (celá dimenze, index, interval). Na zásobníku je pak uložen odpovídající počet hodnot určujících souřadnice.

5.3.7 Typy a volání metod z hlediska VM

Jak je zřejmé z předchozího textu, virtuální stroj zná jednak standardní metodu, jejíž reprezentace byla popsána dříve, tak primitivu, která je implementována jako metoda třídy představující daný typ objektu. Z hlediska volací konvence pro oba typy platí, že příjemce je uložen v registru RES a argumenty jsou naskládány za sebou na zásobníku. Poslední argument je uložen na jeho vrcholu. Instrukce MSG nebo IMSG pak podle typu metody provede příslušnou akci. Za odstranění argumentů ze zásobníku je zodpovědný volající. Následuje podrobný popis jednotlivých typů metod.

5.3.7.1 Standardní metoda

Jedná se o metodu, která je definována programátorem v rámci jazyka pomocí zprávy *method*.

Vstup do metody

Vstup do metody je realizován metodou *performMethodEnter* třídy CVmThread. Přesný postup při vstupu do metody je následující:

- Vytvoří se běhová reprezentace pro volanou metodu a do ní se nastaví ukazatel na příslušnou interní reprezentaci volané metody. Jako slot „self“ – příjemce zprávy – se nastaví obsah registru RES.
- Do vytvořeného objektu se nastaví předané argumenty ze zásobníku. Objekt běhové metody vždy vezme další „prototypový“ slot z interní reprezentace a nakopíruje si jej. Poté do něj zkusí nastavit skutečný předaný argument, čím se provede jeho typová kontrola. Selhání typové kontroly vyhodí výjimku *CTypeCheckException*, která je zachycena až v interaktivním režimu (řízení VM), dojde tedy k přerušení běhu aktuální sekvence příkazů.
- Na zásobník se uloží v tomto pořadí registry IP, PRM a PRB.
- Nově vytvořený objekt metody se nastaví do PRM.

- Z interní reprezentace volané metody se vytáhne ukazatel na kořenový interní blok a na základě něj se vytvoří běhový blok. Interní a běhový blok se nastaví do registrů PB a PRB. Objektu běhového bloku se nastaví lexikální kontext na v předchozím kroku vytvořený objekt běhové reprezentace metody.
- Do registru RES se nastaví Null. Registr IP se nastaví na nulu.
- V dalším cyklu začne virtuální stroj provádět kód kořenového bloku zavolané metody.

Výstup z metody

Virtuální stroj přejde k procesu výstupu z bloku automaticky po dokončení poslední instrukce bloku. Kromě toho je možné proces výstupu vyvolat zprávami „break“, „continue“, „return“, „quit“ a „exit“, které jsou realizovány jako primitivy.

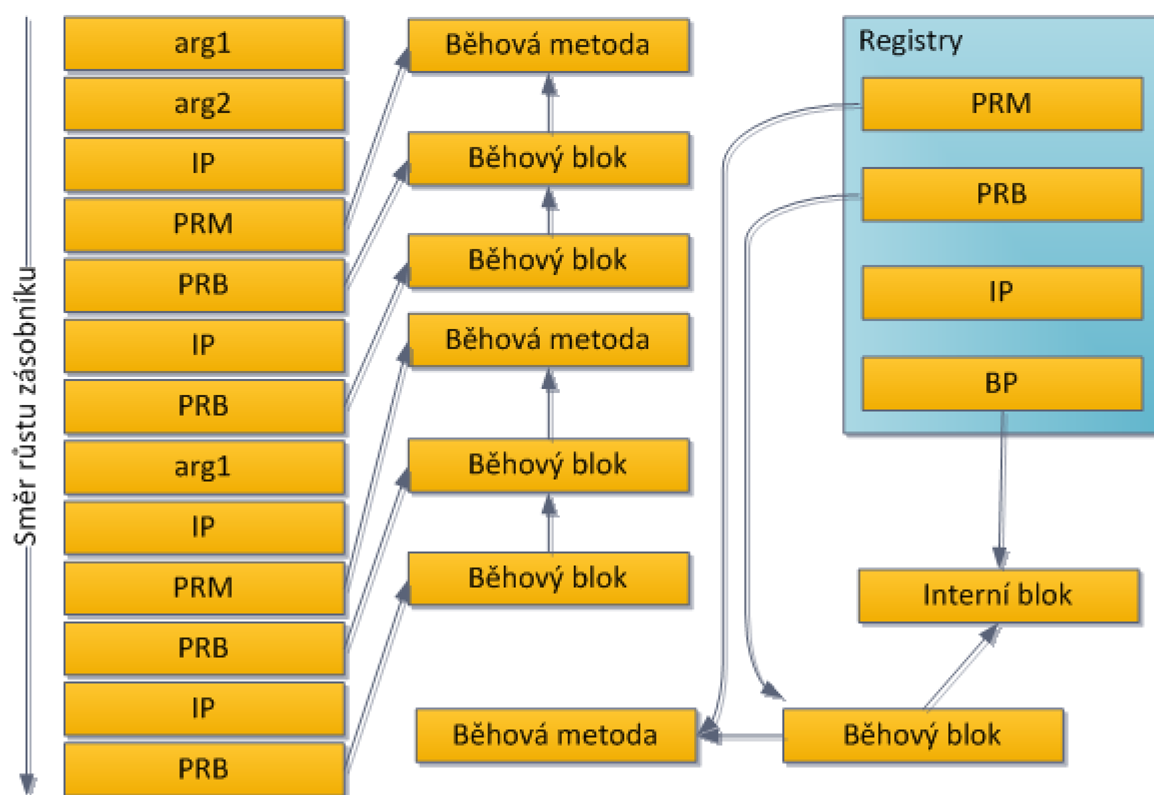
Standardní proces je realizován metodou *performMethodExit* třídy CVmThread a skládá se z těchto kroků:

- Hodnota registru RES je chápána jako výsledek bloku.
- U případných bloků, které byly uvnitř ukončovaného bloku vytvořeny pomocí literálů, se zruší vazba na lexikální kontext ukončovaného bloku. Tím je zajištěna ztráta lexikálního kontextu u zanořených blokových literálů, které byly předány mimo ukončovaný blok v podobě proměnných. V případě, že ukončovaný lexikální kontext začne opět existovat (pokud se nachází v cyklu), může být lexikální kontext opět obnoven (viz předchozí kapitoly o běhové reprezentaci bloků).
- Ze zásobníku se obnoví původní hodnoty registrů IP, PRM a PRB. Registr PB se nastaví na interní objekt, který je odkazován z běhového objektu, nově dosazeného do PRB.
- V dalším cyklu bude virtuální stroj pokračovat následující instrukci bloku, ze kterého byla metoda zavolána.

Kromě toho poskytuje třída CVmThread také metodu *execBlock*, která umožňuje v rámci aktuální metody a aktuálního cyklu provést zanořený blok, což se využívá u implementace řídících konstrukcí. Postup je pak podobný jako výše popsaný, akorát se nemanipuluje s žádnými objekty ani registry, které mají souvislost s metodami. Interně metoda *execBlock* zajišťuje jak vstup, tak výstup z bloku. Provedení kódu bloku pak není realizováno v rámci následujícího cyklu virtuálního stroje, ale zavoláním metody *runBlock* třídy CVmThread. Toto bude popsáno dále.

U provedení samostatného bloku se také uplatní dříve zmiňované pole zanořených literálů, díky kterému je po dokončení bloku zachován ukazatel na něj z nadřazeného běhového bloku. V případě cyklu pak není nutné v každém kroku vytvářet nový běhový blok, ale lze použít stávající.

Obrázek 5.4 zjednodušeně ukazuje situaci, kdy se provádí kód nějaké metody. Z obrázku je patrné, jak bude vypadat zásobník, registry a jak mezi sebou budou provázány běhové objekty. Zároveň je zde vidět, jak vypadá zásobník při provádění zanořeného objektu pomocí execBlock.



Obrázek 5.4: Organizace výpočetního prostředí při provádění nějaké standardní metody

5.3.7.2 Primitiva virtuálního stroje

V případě zavolání primitivy je celý proces výrazně jednodušší. Volací konvence je zde stejná jako v případě standardních metod. Příjemce zprávy a zároveň objekt, u kterého bude implementace primitivy zavolána, je odkazován registrem RES. Případné argumenty zprávy jsou uloženy na zásobníku.

Primitiva je zavolána jako metoda třídy jazyka C++. Virtuální stroj tedy neprovádí žádné uložení kontextu na zásobník tak, jak je to prováděno u standardních metod. Kontext je totiž uložen právě prostřednictvím zavolání primitivy a je tedy realizován reálným procesorem. Za ověření správnosti datových typů argumentů a jejich počtu pak zodpovídá implementace dané primitivy. Po dokončení primitivy je testována její návratová hodnota, která ovlivňuje další chování virtuálního stroje.

5.3.8 Průběh interpretace

Jak již bylo řečeno, za samotnou interpretaci bytekódu odpovídá třída CVmThread, reprezentující jedno vlákno virtuálního stroje. Interpretace sekvence přeložených příkazů (reprezentovaných pomocí interního bloku) začíná zavoláním metody *start* zmiňované třídy. Metoda vytvoří běhovou reprezentaci metody, která nebude svázána s žádnou interní reprezentací a dále vytvoří běhovou reprezentaci bloku nejvyšší úrovně a oba objekty mezi sebou prováže. Tyto dva objekty budou tvořit globální jmenný prostor. U běhového bloku je jeho vlastní tabulka slotů nahrazena ukazatelem na

tabulku slotů objektů Lobby. Tím se zajistí, že objekty vytvořené na globální úrovni budou sloty Lobby.

Poté jsou inicializovány registry do výchozích hodnot a je zavolána již dříve zmiňovaná metoda *runBlock*. Ta obsahuje dva zanořené cykly se switchem, jenž na základě aktuální instrukce provádí odpovídající operaci. Vnitřní cyklus obsluhuje jeden konkrétní blok a je přerušen buď doběhnutím na konec bytekódu nebo jednou z návratových hodnot primitivy – RV_RETURN, RV_BREAK, RV_EXIT a RV_SAVE. Pokud byl cyklus přerušen některou ze zmíněných hodnot, znamená to, že je požadován předčasný výstupu z bloku a je nutné uvolnit ze zásobníku všechny zbývající položky, které tam byly vloženy prováděným blokem a které by jinak byly uvolněny instrukcemi POP. Po přerušení vnitřního cyklu se ověřuje, zda je nadřazeným lexikálním kontextem právě ukončovaného bloku stejný objekt, jaký je odkazovaný registrem PRM. Pokud ano, znamená to, že je ukončován kořenový blok metody a je zavolána dříve zmiňovaná metoda *performMethodExit*. V další iteraci vnějšího cyklu pak pokračuje provádění bloku, ze kterého byla ukončená metoda zavolána.

Pokud nadřazeným lexikálním kontextem není metoda ale blok, znamená to, že bylo provádění bloku vyvoláno metodou *execBlock*, například jako tělo řídící struktury. V tom případě končí celá metoda *runBlock*. Jako návratová hodnota metody *runBlock* je potom vrácena ta, která přerušila cyklus. Volající této metody se pak na základě vrácené hodnoty může nějak zachovat, třeba přerušit provádění cyklu (ve smyslu řídící konstrukce jazyka), jež byly popsány dříve.

5.3.8.1 Vyhodnocení zpráv

Pokud byla zpráva zaslána hodnotě vestavěného typu, je v tuto chvíli nahrazena odpovídajícím prototypovým objektem. Dále je nalezen odpovídající slot. Způsob vyhledání takové slotu se liší podle toho, zda byla zpráva zaslána implicitnímu nebo explicitnímu příjemci zprávy. Oba případy byly popsány v předcházejících kapitolách.

Potom je zavolána metoda *evalueateMsg* třídy CVmThread, která v závislosti na typu nalezeného slotu provede odpovídající akci. V případě datového slotu je jeho hodnota uložena do RES. V případě standardní metody je zavoláno performMethodEnter, v případě primitivy je tato zavolána a vrácena její návratová hodnota. Pokud byl použit prototypový objekt, je po dokončení vyhodnocení zprávy resetován.

5.3.8.2 Vyhodnocení asynchronních zpráv

U asynchronních zpráv se využívá objekt třídy *CThreadManager*, který obsahuje informace o spuštěných vláknech a jejich přiřazení k objektům. Vlákna jsou realizována pomocí POSIXových volání.

Instrukce zavolá metodu *execParallel* správce vláken a předá jí označení asynchronně zasílané zprávy s případnými parametry a referencí na cílový objekt. Správce vláken zjistí, zda bylo pro tento

objekt vytvořeno vlákno a pokud ano, zařadí mu do fronty předanou zprávu. V opačném případě vytvoří novou instanci třídy CVmThread a přiřadí jí frontu zpráv. Následně je vytvořeno nové vlákno, které začíná statickou metodou *consumeParallel* u jmenované třídy. Metoda dostane referenci na vytvořený objekt reprezentující vlákno a informace o své frontě zpráv. Tato metoda funguje jako konzument fronty zpráv a provádí je pomocí volání *execMsg* třídy CVmThread. Výsledek pak uloží ke zprávě ve správci vláken. Volání *execParallel* mezitím vrací objekt *Future*, který nese ukazatel na strukturu ve frontě zpráv. Zasilatel asynchronní zprávy pak pomocí *Future* může získat její výsledek. Jakmile je fronta objektu prázdná, asociované vlákno končí.

5.3.8.3 Vyhodnocení řetězcových a blokových literálů

Při instrukci SRES_BLITERAL je z běhového bloku, odkazovaného pomocí registru PRB zjištěno, zda již existuje požadovaný běhový blok. Pokud ano, je resetován a jeho lexikální kontext je opět nastaven na objekt odkazovaný registrem PRB. Pokud ne, je vytvořen nový, jeho lexikální kontext nastaven na současný blok a zároveň je ukazatel na něj nastaven na odpovídající index pole literálů u současného bloku.

V případě řetězcového literálů (instrukce SRES_LITERAL) je vždy vytvořena kopie původního objektu, který vznikl při překladu. V obou případech je nakonec objekt uložen do registru RES.

5.3.9 Realizace řídících struktur

Jak již bylo řečeno, řídící struktury jsou realizovány pomocí primitiv. Virtuální stroj pouze poskytuje podporu ve formě reakce na návratové hodnoty primitiv.

5.3.9.1 Větvení

Větvení je realizováno objektem *Cond*, jenž je implementován pomocí třídy *CCond*. Tento objekt si uvnitř uchovává svůj stav, který zajišťuje, že nepůjde nad stejným objektem zavolat dvakrát za sebou *else*, ani po *else* zavolat například *elseif*. Po ověření korektního stavu objektu je testována hodnota podmínky a na základě toho se předaný blok buď provede voláním *execBlock*, nebo neprovede. Hodnota podmínky je do objektu uložena, protože bude využita případnou následující zprávou.

5.3.9.2 Cykly

U cyklů se musí opakovaně vyhodnocovat podmínka, což je také důvod, proč ta nemůže být předána jako matematický výraz, ale musí se jednat o blok kódu. V každém kroku cyklu je nejprve resetován blok, který tvoří jeho tělo a následně je pomocí *execBlock* vyhodnocena podmínka cyklu. Pokud byla výsledkem tohoto volání hodnota *True*, je pomocí stejného volání vyhodnoceno i tělo cyklu. V obou případech se kontroluje návratová hodnota tohoto volání. Pokud byla vrácena jiná hodnota než

RV_OK, vyhodnocování cyklu končí. Hodnota RV_BREAK je poté zaměněna za hodnotu RV_OK, čímž se zajistí, že případná zpráva *break* přeruší jen nejvíc zanořený cyklus.

5.3.9.3 Zprávy *break, return, exit, quit* a *saveImage*

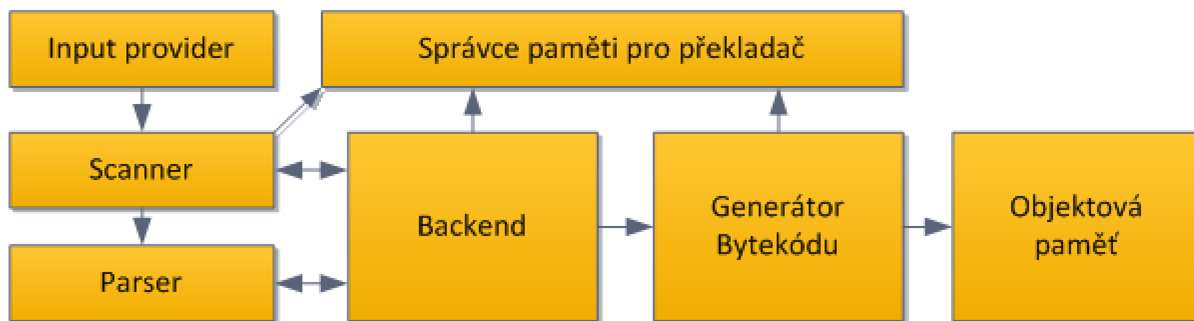
Tyto zprávy jsou realizovány jako velmi jednoduché primitivy, které do RES uloží předaný argument (kromě zprávy *break*) a jako svůj výsledek vrátí odpovídající návratovou hodnotu. Zpráva *break* vrací *RV_BREAK*, zpráva *return* vrací *RV_RETURN*, volání *exit* a *quit* vrací *RV_EXIT* a nakonec zpráva *saveImage* vrátí *RV_SAVE*. Tyto konstanty se pak v podobě návratových hodnot posílají dál až do místa, kde se nachází jejich ošetření. Jak již bylo zmíněno, takovým místem je buď metoda *runBlock*, případně primitivy realizující řídící struktury. Jedinými výjimkami jsou hodnoty *RV_EXIT* a *RV_SAVE*, které se dostanou až do hlavní třídy programu, případně do implementace interaktivního režimu, kde způsobí buď ukončení programu, nebo uložení perzistentního obrazu objektové paměti. Zde se také dostáváme k důvodu, proč se volání *saveImage* chová podobně jako volání *exit*.

Serializaci objektové paměti bychom mohli klidně vyvolat přímo z primitivy. Protože by k ní došlo uprostřed vykonávání programu, mohlo by se stát, že nějaký objekt by nebyl ve stavu, do kterého by šel snadno obnovit při následné deserializaci. Jako příklad uveďme běhový blok, jehož sloty se neustále mění tak, jak se mění stav jeho vykonávání. Při deserializaci bychom pak museli celý virtuální stroj uvést do stavu, ve kterém byl přesně v okamžik zavolání *saveImage*, jinak by uložené sloty nekorespondovaly s provedeným programem. Zvolili jsme proto jednodušší variantu, kdy zpráva způsobí ukončení provádění programu (případně návrat do interaktivního režimu) čímž se zajistí, že všechny objekty budou serializovány ve stavu, kdy jsou „v klidu“.

5.4 Překladač

Překladač zodpovídá za překlad programu, zapsaného v navrženém jazyce, do interpretovatelného bytekódu, kterému rozumí virtuální stroj. Má klasickou strukturu a skládá se z lexikálního a syntaktického analyzátoru a dále pak z generátoru vnitřního kódu. Pro usnadnění implementace rutinních částí překladače jsme použili volně dostupné nástroje. Pro tvorbu lexikálního analyzátoru byl použit nástroj *Flex* a syntaktický analyzátor byl vytvořen pomocí nástroje *Bison*. Obrázek 5.5 ukazuje blokovou strukturu celého překladače a jeho rozhraní s okolím.

Input provider poskytuje dalším částem překladače vstupní zdrojový kód. Tato část je implementována třídou *CInputProvider*, která má dva potomky. Třída *CFileInput* načítá zdrojový kód ze souboru určeného jeho názvem. Tato třída se použije pro načtení zdrojových kódů ze souborů, specifikovaných při spuštění virtuálního stroje. Dále třída *CStringInput* poskytuje zdrojový kód z předaného textového řetězce. Tato třída se využívá v interaktivním režimu.



Obrázek 5.5: Bloková struktura

Scanner respektive *parser* jsou implementovány pomocí výše zmíněných nástrojů a činnosti těchto bloků jsou obvyklé. Scanner je tedy zodpovědný za rozpoznávání a klasifikaci tokenů. Parser je pak zodpovědný za ověření správné syntaxe programu a vytvoření jeho interní reprezentace za pomoci bloku *Backend*. Program je interně reprezentován pomocí stromu objektů různých typů. *Interní reprezentace* programu bude popsána dále. Tyto čtyři doposud zmíněné bloky pracují současně na principu zřetězeného zpracování. Syntaktický a lexikální analyzátor jako jediné části programu nejsou implementovány jako třídy C++.

Po vytvoření interní reprezentace celého zdrojového kódu je proveden jeho překlad do bytekódu a vytvoření odpovídajících paměťových objektů, reprezentujících řetězcové a blokové literály. Za tuto fázi překladu je zodpovědný *generátor bytekódu*, implementovaný třídou *CCodeGenerator*. *Správce paměti pro překladač* se stará o uvolňování objektů dynamicky alokovaných během překladu různými částmi překladače.

5.4.1 Překlad a interní reprezentace programu

Motivací pro zavedení interní reprezentace bylo obtížné generování bytekódu a paměťových objektů přímo ze syntaktického analyzátoru. Dalším důvodem byla možnost dodatečných optimalizací zpracovaného zdrojového kódu.

Jak již bylo řečeno dříve, program je v interní reprezentaci tvořen stromem objektů. Tento strom je průběžně vytvářen syntaktickým analyzátorem za pomoci backendu.

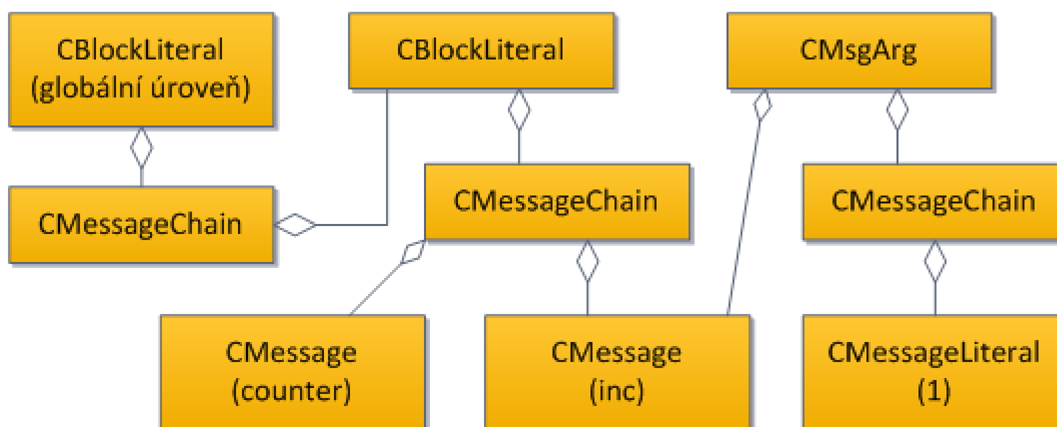
Literál bloku je reprezentován objektem třídy *CBlockLiteral*. Ten je ve zdrojovém kódu složen ze zpráv, přesněji řečeno z řetězců zpráv oddělených středníky. Jeden řetězec zpráv je modelován objektem třídy *CMessageChain*. Objekt reprezentující literál bloku pak obsahuje kolekci objektů třídy *CMessageChain*.

Řetězec zpráv je ve zdrojovém kódu složen z jednotlivých zpráv. V interní reprezentaci je řetězec složen z objektů třídy *CMessageLiteral* a jejích potomků. Jak je z názvu třídy patrné, objekt může představovat buď zprávu, nebo literál (například opět blok). Pokud se jedná o zprávu, je použita instance specializovanější třídy *CMessage*. Ta obsahuje informace o názvu zprávy a dále může obsahovat ukazatel na objekt třídy *CMsgArg*, jenž modeluje případné argumenty zprávy.

Jednotlivé argumenty jsou opět reprezentovány pomocí objektů třídy *CMessageChain*. Syntaxe totiž umožňuje jako argument použít jakýkoliv řetězec zpráv. Obrázek 5.6 obsahuje příklad jednoduchého zdrojového kódu a na obrázku 5.7 je ukázána jeho interní reprezentace. Předpokládejme, že tento literál bloku je zapsán na globální úrovni. Spojnice vyjadřují vztah agregace mezi objekty dle standardu UML.

```
{
      counter.inc(1);
};
```

Obrázek 5.6: Příklad zdrojového kódu



Obrázek 5.7: Interní stromová reprezentace příkladu z obrázku 5.6

5.4.1.1 Správce paměti

Aby se zajistilo spolehlivé uvolňování použité paměti, a to i v případě chyby při překladu, jsou veškeré dynamicky alokované struktury, včetně řetězců vzniklých v lexikálním analyzátoru, spravovány správcem paměti pro překladač. Ten je implementován jako jednoduchá třída *CCompilerMemoryMgr,* obsahující jeden velký seznam ukazatelů. Jednotlivé části překladače pak mohou do tohoto seznamu přidávat ukazatele na své alokované objekty. Při dokončení překladu, ať už úspěšném nebo ne, jsou všechny struktury odkazované ze seznamu uvolněny. Seznam je poté smazán.

5.4.2 Generátor bytekódu

Generátor bytekódu je volán až poté, co je úspěšně dokončen překlad a je sestaven strom interní reprezentace celého programu. Generátor nejprve vezme objekt reprezentující kořenový blok a v objektové paměti vytvoří odpovídající interní blok, instanci třídy *CInternalBlock.* Poté začne procházet jednotlivé řetězce zpráv a do vytvořené instance bloku ukládat odpovídající instrukce. Jakmile je nalezen nějaký jiný zanořený blok, je pro něj vytvořena další instance třídy *CInternalBlock*, která je ihned provázána se současným blokem. Generování pak pokračuje

v zanořeném bloku. Jakmile je zanořený blok vygenerován, je do jeho místa v nadřazeném bloku vložena reference na literál v podobě instrukce SRES_BLITERAL. Podrobnosti překladu literálů budou zmíněny v následujících kapitolách.

5.4.2.1 Postup překladu zpráv v jednotlivých situacích

Tato kapitola ukazuje všechny situace, které mohou nastat při zasílání zpráv a způsob, jakým bude daná sekvence zpráv přeložena do instrukcí vnitřního kódu. Znak „#“ zde znamená označení dané zprávy pomocí hashů.

Nejjednodušší situací je jediná zpráva, například „get“. Překladač v tomto případě vygeneruje jedinou instrukci „IMSG #get“. O něco složitější je situace, kdy je zpráva součástí nějakého řetězce zpráv. Například „get.toString“. Zde se vyjde z toho, že výsledek zprávy je uložen do registru RES. Sekvence se potom přeloží na dvě instrukce: „IMSG #get“ a následně „MSG #toString“. Takto lze řetězit libovolný počet zpráv.

Situace v předchozím odstavci se komplikuje, pokud má zpráva nějaké argumenty. V tom případě jsou nejprve vyhodnoceny ty, v pořadí zleva doprava. Jejich výsledky jsou uloženy na zásobník a následně je poslána samotná zpráva. Protože jsou argumenty také posloupnosti zpráv, proběhne jejich vyhodnocení úplně stejným způsobem. Nakonec je vždy potřeba uvolnit zásobník. Mějme následující příklad:

```
set(obj);
```

Tento příkaz se vyhodnotí na následující sekvenci instrukcí:

```
IMSG #obj
PUSH
IMSG #set, 1
POP
```

Situace je o něco složitější, pokud máme řetězec zpráv a v něm existují dvě po sobě jdoucí zprávy, které mají argumenty. V tom případě musíme před vyhodnocením argumentů druhé zprávy odložit výsledek první zprávy na zásobník, protože se při jejich vyhodnocení bude používat registr RES. Po vyhodnocení je do RES vrácen výsledek první zprávy pomocí instrukce GET s parametrem, odpovídajícím počtu položek od vrcholu zásobníku. Mějme následující příklad:

```
getProp(propName).convert(base);
```

Pro uvedený řetězec zpráv překladač vygeneruje následující posloupnost instrukcí:

```
IMSG #propName
PUSH
IMSG #getProp
POP
PUSH
IMSG #base
PUSH
GET 1
MSG #convert
POP 2
```

5.4.3 Překlad literálů

Z předchozích kapitol víme, že některé literály jsou vytvářeny již v době překladu a jiné vznikají až za běhu. Hodnoty vestavěných datových typů jsou pomocí dříve popsaných instrukcí uloženy přímo do registru RES. U blokových, respektive řetězcových literálů je použita dříve popsaná instrukce SRES_BLITERAL resp. SRES_SLITERAL. Spolu s tím je v objektové paměti vytvořen odpovídající objekt, který je ihned naplněn daty. Samotné znaky řetězce se tedy do bytekódu neukládají, ale jsou nakopírovány rovnou do odkazovaného objektu. Jako argument instrukce je použito pořadové číslo, které se dá odvodit od aktuálního počtu odkazovaných literálů v právě překládaném literálu bloku kódu.

5.4.3.1 Překlad dvourozměrných struktur

Dvourozměrné struktury jsou vytvářeny až za běhu a v době překladu se tedy vygeneruje pouze posloupnost instrukcí pro vytvoření objektu. Odpovídající objekt je nejprve vytvořen instrukcí RES_NEW_ARR (RES_NEW_MTX) a nová struktura je odkazována registrem RES. Obsah RES je uložen na zásobník instrukcí PUSH a pokračuje se postupným vyhodnocením a ukládáním jednotlivých prvků. Každý prvek je vyhodnocen stejně, jako jakýkoliv jiný řetězec zpráv a výsledek je uložen na zásobník. Ze zásobníku se poté načte reference na vytvořenou strukturu a pomocí instrukce 2D_SET_AT se do něj vyhodnocený prvek uloží. Po vyhodnocení všech prvků se reference na vytvořený objekt ze zásobníku odstraní a zůstane pouze v registru RES, kde bude dostupná pro další vyhodnocování. Obrázky 5.8 a 5.9 ukazují překlad jednoduchého pole s prvky 2×2.

```
[| 1, „ab“; Null, Lobby |];
```

Obrázek 5.8: Příklad zápisu pole

5.4.4 Řízení překladu

Překladač jako celek je řízen instancí třídy *CCompiler*. Ta obsahuje metodu *compile,* zodpovídající za celý proces překladu zdrojového kódu do bytekódu. Metoda přijímá jeden argument, a to instanci

třídy *CInputProvider*, která budou sloužit jako zdroj dat pro daný překlad. Nejprve je inicializován lexikální analyzátor pomocí předaného poskytovatele a správce paměti pro překladač. Následně metoda inicializuje syntaktický analyzátor pomocí backendu. Poté je proveden překlad do interní reprezentace zavoláním funkce *doParse*. Pokud nastala v době překladu chyba, je vygenerována jedna výjimek. Při úspěšném překladu obsahuje backend ukazatel na interní reprezentaci bloku kódu globální úrovně zanoření. Ukazatel se předává do generátoru kódu prostřednictvím jeho metody *translateBlock*, jež zajistí vygenerování bytekódu a vytvoření potřebných objektů v objektové paměti. Opět, pokud dojde k chybě, je vygenerována výjimka. V případě úspěchu metoda vrací ukazatel na interní reprezentaci kořenového bloku. Tato je pak vrácena i metodou *compile* k dalšímu zpracování. Posledním krokem je uvolnění paměti alokované během překladu zavoláním metody *compilerFree* třídy *CCompilerMemoryMgr*.

```
RES_NEW_ARR
PUSH
SRES_NUM 1
PUSH
GET 1
2D_SET_AT 0 0
POP
SRES_SLITERAL 0
PUSH
GET 1
2D_SET_AT 0 1
POP
SRES_NULL
PUSH
GET 1
2D_SET_AT 1 0
POP
IMSG #Lobby
PUSH
GET 1
2D_SET_AT 1 1
POP
POP
```

Obrázek 5.9: Instrukce bytekódu pro příkazy z obrázku 5.8

5.5 Inicializace a interaktivní režim

Tato část programu zodpovídá za počáteční inicializaci všech jeho ostatních částí, za jeho řízení a za komunikaci s uživatelem. Inicializace a interaktivní režim jsou implementovány v třídě *CMplus*, jejíž instance je vytvořena ve funkci *main*. Po spuštění programu je nejprve inicializována objektová paměť – instance třídy *CMemoryManager*. Zde je inicializace jednoduchá a spočívá v nastaveních všech ukazatelů na význačné objekty na NULL.

V dalším kroku je vytvořena instance třídy *CCompiler*, která v sobě zapouzdřuje veškerou funkcionalitu týkající se překladače. Překladač v rámci svého spuštění vytvoří instanci třídy *CCodeGenerator*, která bude zajišťovat generování bytekódu po celou dobu běhu programu. Dále je vytvořena instance třídy *CCompilerMemoryMgr* zajišťující uvolňování paměti po dokončení překladu. Nakonec je vytvořen nový objekt třídy *CCompilerBackend*, který funguje jako podpora pro syntaktický analyzátor.

Po spuštění správce paměti a překladače je vytvořen objekt třídy *CVmThread*, který bude fungovat jako hlavní vlákno. Hlavní vlákno v dalších fázích zpracuje soubory se zdrojovým kódem, které byly při spuštění specifikovány parametrem „-f“ a provádí také příkazy, jež byly zadány v interaktivním režimu. Tímto je provedena základní inicializace celého programu.

Z funkce *main* je pak zavolána metoda *run* třídy *CMplus*. Zde jsou nejprve zpracovány argumenty zadané z příkazové řádky, případně vytištěna krátká nápověda k ovládání programu. Pokud byl specifikován soubor s perzistentním obrazem, je volána metoda *deserialize* nad objektem správce objektové paměti. Metoda zajistí vytvoření základního stromu objektů ze souboru. Následně je nad objektovou pamětí zavoláno *initSlots*, které do takto vytvořeného stromu dodá sloty pouze pro čtení, nesoucí základní primitivy. Pokud obraz specifikován nebyl, je vytvořeno výchozí prostředí zavoláním metody *initDefaultEnviroment,* opět nad objektem správce paměti. Vytvářený strom objektů je v tomto případě specifikován přímo v této metodě.

Pokud byly při spuštění specifikovány soubory se zdrojovým kódem, jsou v tuto chvíli přeloženy a zpracovány. Je vytvořena instance třídy *CFileInput*, do které jsou postupně nastavovány názvy jednotlivých souborů. Instance je potom předávána metodě *exec*, která zajistí překlad pomocí volání *compile* a následné provedení kódu voláním *start* třídy *CVmThread*. Na této úrovni jsou také zachytávány výjimky generované při chybách překladu nebo běhu programu. Pokud vznikla chyba, pokračuje se dalším specifikovaným souborem, případně program přejde do interaktivního režimu, jednalo-li se o poslední zpracovávaný soubor.

Po zpracování souborů je na standardní výstup vytištěn informační banner a program přejde do interaktivního režimu zavoláním metody *interaction*, kde čeká na vstup od uživatele. Ten je realizován metodou *readInput*, jež postupně načítá řádky ze standardního vstupu a skládá je do jednoho řetězce. Po načtení prázdného řádku je načítání ukončeno a doposud načtená data jsou nastavena do instance třídy *CStringInput*. Ta je předána do metody *exec,* zajišťující vyhodnocení.

Byla-li z *exec* vrácena hodnota RV_SAVE, je vyvolána serializace objektové paměti pomocí volání *serialize* u správce paměti.

Pokud byl načten konec souboru (CTRL+D), nebo byla z volání *exec* vrácena hodnota *RV_EXIT*, je interaktivní režim ukončen a nastavena návratová hodnota programu. Poté jsou postupně ukončovány jednotlivé jeho části. Nejprve je zrušen objekt hlavního vlákna, dále je zrušen objekt řídící překlad, čímž se zároveň uvolní všechny zbývající prostředky, alokované při překladu. Nakonec je zrušen správce objektové paměti, což zároveň způsobí i zrušení všech v ní uložených objektů.

6 Demonstrační příklady

Tato kapitola na jednoduchých příkladech ukazuje, jak je možné pomocí jazyka realizovat vybrané návrhové vzory. Jsou představeny vzory *jedináček* (singleton), *adaptér* a *návštěvník* (visitor). Dále ukázány dva jednoduché algoritmy. Jedná se o výpočet faktoriálu čísla a výpočet kořenů kvadratické rovnice.

6.1 Návrhové vzory

6.1.1 Singleton

Návrhový vzor singleton se používá tam, kde je potřeba zajistit existenci maximálně jednoho objektu pro konkrétní účel. Objekt potom nevytváříme přímo instanciací nebo klonováním, ale jinou metodou, která poprvé požadovaný objekt vytvoří a při dalších volání jej už jen vrací.

V příkladu se nejprve vytvoří prototypový naklonováním objektu Object. Následně je mu přidána metoda *getInstance*, která vytvořený objekt naklonuje, uloží do slotu a při dalších voláních ho už jen vrací. Singleton patří mezi vytvářecí návrhové vzory.

```
// vytvoreni prototypu
Object Singleton := Object.clone();
Singleton Singleton.objekt := Null;
Singleton.method("getInstance", Null)
{
     if(isNull(self.objekt))
     {
           self.objekt = self.clone();
     };
     return(self.objekt);
};
// vytvoreni objektu
Singleton s := Singleton.getInstance();
```

Příklad 6.1: Návrhový vzor singleton

6.1.2 Adaptér

Adaptér umožňuje komunikaci objektů s nekompatibilním rozhraním. Objekt potom překládá (adaptuje) rozhraní, používané jedním objektem, na rozhraní použité u druhého objektu. První objekt pak neposílá druhému objektu zprávu přímo, ale přes adaptér. Příklad předpokládá objekt *PrettyPrinter* s metodou *printPretty*. Adaptér pak obsahuje odkaz na zmíněný objekt a obaluje ho jiným rozhraním, které přijímá koordináty *x* a *y*. Adaptér patří mezi strukturální vzory.

```
Object PrettyPrinter := Object.clone();
Object Adapter := Object.clone();
PrettyPrinter.method("printPretty", [| String, "str" |])
{
     ("(" + str + ")").println();
};

PrettyPrinter Adapter.obj := PrettyPrinter;

Adapter.method("cordPrint",[| Number, "x"; Number, "y" |])
{
     "x:y = ".print();
     self.obj.printPretty(x.toString() + ":" + y.toString());
};
```

Příklad 6.2 Návrhový vzor adaptér

6.1.3 Návštěvník

Návštěvník patří mezi behaviorální vzory a umožňuje rozšiřovat funkcionalitu tříd bez jejich modifikace. Typickým použitím je situace, kdy máme sadu několika tříd, u kterých chceme mít možnost v budoucnu snadno přidávat další chování.

U takových základních tříd pak definujeme metodu *accept*, přijímající objekt s určitým rozhraním. Nové chování potom implementujeme v podobě tříd se stejným rozhraním a předáváme je pomocí *accept* do rozšiřované třídy. Předaný objekt obsahuje metodu *visit*, která přijímá objekt rozšiřované třídy a provádí nad ním požadovanou akci. Je zřejmé, že každé rozšíření pak bude implementováno vlastní třídou. Příklad ukazuje objekty geometrických tvarů, které mají společnou vlastnost, a to počet vrcholů. Dodávaná funkcionalita pak umožňuje počet vypsat.

```
// spolecny predek definujici rozhrani navstiveneho objektu
Object GShape := Object.clone();
Number GShape.nodes := 0;

// spolecny predek definujici rozhrani navstevnika
Object Visitor := Object.clone();
Visitor.method("visit", [| GShape, "s" |], {});

// metoda accept u navstiveneho
GShape.method("accept", [| Visitor, "v" |], {v.visit(self);});

// konkretni tvary
GShape triangle := GShape.clone();
Number triangle.nodes := 3;
GShape rectangle := GShape.clone();
Number rectangle.nodes := 4;

// konkretni navstevnik, tisknouci pocet vrcholu
Visitor printer := Visitor.clone();
printer.method("visit", [| GShape, "s" |])
{
    s.nodes.toString().println();
};
```

Příklad 6.3: Návrhový vzor visitor

6.2 Ukázky algoritmů

6.2.1 Faktoriál

Následující příklad ukazuje rekurzivní výpočet faktoriálu specifikovaného čísla.

```
method("faktorial", [| Number, "n" |])
{
     if(n == 0)
     {
           return(1);
     }.elseif(n < 0)
     {
           return(Null);
     }.else
     {
           return(n * faktorial(n - 1));
     };
};
```

Příklad 6.4: Výpočet faktoriálu

6.2.2 Výpočet kořenů kvadratické rovnice

Kořeny kvadratické rovnice $ax^2 + bx + c = 0$ získáme pomocí známého vzorce

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Pokud je výraz pod odmocninou (diskriminant) kladný, má rovnice dva kořeny. Pokud je nulový, má rovnice pouze řešení. Je-li diskriminant záporný, řešením jsou dva komplexně sdružené kořeny. Následující příklad implementuje zmíněný vzorec.

```
method("koreny", [| Number, "a"; Number, "b"; Number, "c" |])
{
     Number d := b * b - 4 * a * c;

     if (d == 0) {return( [| (b * -1) / (2 * a) |] ); };

     // dva koreny
     FComplex sq := d.sqrt();

     Array k := [| ((b * -1)+sq) / (2*a), ((b * -1)- sq) / (2*a) |];

     return(k);
};
```

Příklad 6.5: Výpočet kořenů kvadratické rovnice

7 Závěr

Práce nejprve představila základní principy fungování objektově orientovaných jazyků. Byly popsány třídně a prototypově založené jazyky. Protože z prototypově založených jazyků vychází i tato práce, byly rozebrány více a ukázali jsme jejich základní principy, použité u jazyků Smalltalk, Self a Io. Práce se rovněž stručně zabývala jazyky pro matematické výpočty, kde jsme čerpali inspiraci zejména pro některé aspekty syntaxe navrhovaného jazyka.

V další a jedné ze stěžejních částí práce, jsme se zabývali návrhem našeho jazyka. Byly zde představeny jeho základní principy, literály a jejich syntaxe a organizace objektového prostředí. Dále byla formálně specifikována gramatika jazyka pomocí EBNF. Byl popsán mechanismus použité typové kontroly, koncept operátorových zpráv, jmenných prostorů a dědičnost. Nakonec byly ukázány příklady standardních řídících konstrukcí.

Pátá část práce se zabývala návrhem samotného virtuálního stroje a popisem jeho prototypové implementace. Nejprve byla vysvětlena implementace objektové paměti a způsob realizace jednotlivých objektů jazyka. Zde bylo také ukázáno, jakým způsobem jsou implementovány primitivy a jako příklad je uveden popis realizace primitivy pro klonování. Nakonec je vysvětleno, jakým způsobem pracuje serializace a deserializace. Poté jsme se v této části zabývali interpretem bytekódu. Byla popsána interní a běhová reprezentace metod a bloků. Dále byly vysvětleny registry a užitý výpočetní model. Byla také představena použitá instrukční sada a volací konvence pro metody. Volání metod pak bylo podrobněji rozebráno a byly ukázány rozdíly mezi voláním primitiv a standardních metod. Kapitola také popisuje průběh vytváření metod a implementaci řídících struktur. Nakonec je popsána realizace asynchronních zpráv.

Poslední kapitola práce na jednoduchých příkladech zobrazila, jak lze pomocí navrženého jazyka realizovat vybrané návrhové vzory. Byly také ukázány dva příklady implementace jednoduchých algoritmů pro výpočet faktoriálu a kořenů kvadratické rovnice.

Prototypová implementace virtuálního stroje implementuje většinu funkcionality popsané v této práci. Některé méně podstatné části, například změna datového typu slotu či garbage collector implementovány nebyly. Rovněž sada operací pro jednotlivé objekty obsahuje jen základní činnosti. Pro reálné použití by se tedy vytvořený prototyp musel doplnit o průběžné uvolňování paměti a rovněž by se musela doplnit i sada dostupných operací, například o širší možnosti vstupu a výstupu. Nicméně, poslední část práce ukázala, že i námi implementovaný prototyp již umožňuje realizovat běžné algoritmy.

Jako rozšíření návrhu můžeme zmínit podporu pro další matematické struktury a spolu s tím i zjednodušení a rozšíření gramatiky jazyka. Užitečným rozšířením by bylo také lepší hlášení chyb. Ty jsou nyní hlášeny jen velmi stručně a u rozsáhlejšího programu je tak obtížné najít místo, kde se chyba nachází. Dále můžeme zmínit zavedení symbolů, podobných těm ze Smalltalku. Ty by se pak

daly použít při zápisu názvů doposud neexistujících slotů (například při jejich vytváření) místo současných řetězců. Lze se také zamyslet nad vícenásobnou dědičností. Zde by však byly zásahy do současného návrhu již poměrně rozsáhlé. Vyžadovalo by to přepracovat koncept vyhledávání slotu odpovídajícího dané zprávě a taky by bylo nutné přepracovat model datových typů a typové kontroly. Jako další možnost rozšíření se jeví také podpora nepřepisovatelných slotů i pro uživatele jazyka, ne jen pro virtuální stroj a možnost specifikovat soukromé sloty. Můžeme také zmínit jiný formát perzistentního obrazu, který by byl nezávislý na platformě počítače. Poslední a důležitou možností rozšíření by byly optimalizace při generování vnitřního kódu překladačem.

Literatura

[1] MITCHELL, John C. *Concepts in programming languages*. Cambridge : Cambridge University Press, c2003. 529 s. ISBN 05-217-8098-5.

[2] LIU, Yu-Cheng. *Smalltalk, objects and design*. Vyd. 1. New York : Excel, 1996. 289 s. ISBN 15-834-8490-6.

[3] SEBESTA, R. *Concepts of programming languages*. Massachusetts : Addison-Wesley, 1999. 670 s. ISBN 02-013-8596-1.

[4] HUNT, J. *Smalltalk and object orientation: An introduction*. Vyd. 1. London: Springer, 1997, 378 s. ISBN 35-407-6115-2

[5] Ungar D., Smith R. B. *Self: The Power of Simplicity*. In the Prodeedings of Conference on Object-Oriented Programming Systems, Languages, and Applications. 1987, Orlando FL, USA, p. 227-242

[6] ALLEN, Russell et al . The Self Handbook. *Self: the power of simplicity* [online]. Release 2. August 02, 2011 [cit. 2012-01-02]. Dostupné z: http://docs.selflanguage.org/4.4/

[7] Io Programming Guide. *Io* [online]. [cit. 2012-01-02]. Dostupné z: http://www.iolanguage.com/scm/io/docs/IoGuide.html

[8] LIBERTY, Jesse; POJSL, Josef; VORÁČEK, Karel. *Naučte se C++ za 21 dní*. 2. aktualiz. vyd. Brno : Computer Press, 2007. 796 s. ISBN 978-802-5115-831.

[9] KERNIGHAN, Brian; RITCHIE, Dennis M; ŠŤÁVA, Zbyněk. *Programovací jazyk C*. Vyd. 1. Brno : Computer Press, 2006. 286 s. ISBN 80-251-0897-X.

[10] HEROUT, Pavel. *Učebnice jazyka Java*. 1. vyd. České Budějovice: Kopp, 2001, 349 s. ISBN 80-723-2115-3.

[11] HONZÍK, Jan Maxmilián; HRUŠKA, Tomáš; MÁČEL, Michal. *Vybrané kapitoly z programovacích technik*. Vyd. 3. Brno : VUT v Brně, 1991. 218 s.

[12] Kolář, D.: *Principy programovacích jazyků a objektově orientovaného programování - III*, studijní opora pro kombinované studium, modul IPP III, VUT FIT, 2006

[13] MEDUNA, Alexander. *Automata and languages : theory and applications*. London : Springer, 2000. 916 s. ISBN 18-523-3074-0.

[14] Češka, M., Ježek, K., Melichar B., Richta K.: *Konstrukce překladačů*, Praha, CZ, ČVUT, 1999, 636 s., ISBN 80-01-02028-2

[15] Matlab : Documentation. THE MATHWORKS, INC. *MathWorks* [online]. [cit. 2012-01-02]. Dostupné z: http://www.mathworks.com/help/techdoc/

[16] EATON, John W. et al . Octave Documentation. *GNU Octave* [online]. 3.4.0. 1996, 1997, 1999, 2000, 2001, 2002, 2005, 2006, 20 [cit. 2012-01-02]. Dostupné z: http://www.gnu.org/software/octave/docs.html

[17] ISO/IEC 14977 : 1996(E). *Information technology. Syntactic Metalanguage. Extended BNF*. ISO, [15 February 1998]. Dostupné z: http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf

[18] BARNEY, Blaise . POSIX Threads Programming. LAWRENCE LIVERMORE NATIONAL LABORATORY. *POSIX Threads Programming* [online]. Wed, 24 Aug 2011 [cit. 2012-01-03]. Dostupné z: https://computing.llnl.gov/tutorials/pthreads/

[19] CERF, Vint . Request for Comments: 20 . *ASCII format for Network Interchange*. IETF, October 16, 1969. IETF Documents. Dostupné z: http://tools.ietf.org/html/rfc20

[20] PAXSON, Vern, Will ESTES a John MILLAWAY. The flex Manual. *Flex: The Fast Lexical Analyzer* [online]. version 2.5.35. 10 September 2007 [cit. 2012-01-08]. Dostupné z: http://flex.sourceforge.net/manual/

[21] Bison 2.5. FREE SOFTWARE FOUNDATION, Inc. *Bison - GNU parser generator* [online]. 14 May 2011 [cit. 2012-01-08].

[22] JONES, Meilir. *Základy objektově orientovaného návrhu v UML*. Vyd. 1. Praha: Grada, 2001, 367 s. ISBN 80-247-0210-x.

[23] PEZZÈ, Mauro a Michal YOUNG. *Software testing and analysis: process, principles, and techniques*. Hoboken: John Wiley & Sons, 2008, 488 s. ISBN 978-0-471-45593-6.

[24] PECINOVSKÝ, Rudolf. *Návrhové vzory*. Vyd. 1. Brno: Computer Press, 2007, 527 s. ISBN 978-80-251-1582-4.

[25] JOSUTTIS, M. *C++ standardní knihovna a STL*: *kompletní průvodce*. Vyd. 1. Brno: CP Books, 2005, 743 s. ISBN 80-251-0511-1.

[26] MCCARTHY, J., R BRAYTON, D EDWARDS, P FOX, L HODES, D LUCKHAM, K MALING a D PARK. *LISP I Programmers Manual*. [online]. 1960 [cit. 2012-05-20]. Dostupné z: http://history.siam.org/sup/Fox_1960_LISP.pdf

[27] Maxima 5.27.0 Manual. *Maxima, a Computer Algebra System* [online]. 2012 [cit. 2012-05-20]. Dostupné z: http://maxima.sourceforge.net/docs/manual/en/maxima.html

[28] Python v2.7.3 documentation. *Python Programming Language: Official Website* [online]. 2012 [cit. 2012-05-20]. Dostupné z: http://docs.python.org/

[29] Sage Documentation v5.0. *Sage Open Source Mathematics Software* [online]. 2012 [cit. 2012-05-20]. Dostupné z: http://www.sagemath.org/doc/index.html

[30] An Introduction to R. *The R Project for Statistical Computing* [online]. 2.15.0. 2012, 2012-03-30 [cit. 2012-05-21]. Dostupné z: http://www.r-project.org/

[31] NOLL, Landon Curt. FNV Hash. *Landon Curt Noll* [online]. 2012, 2012/05/14 [cit. 2012-05-21]. Dostupné z: http://www.isthe.com/chongo/tech/comp/fnv/index.html

[32] HSIEH, Paul. Hash functions. *Hash functions* [online]. 2004 [cit. 2012-05-21]. Dostupné z: http://www.azillionmonkeys.com/qed/hash.html

Seznam příloh

1) Způsob ovládání programu
2) Obsah přiloženého CD

Příloha 1 - Popis ovládání programu

Program může při spuštění přijímat následující parametry:

- -h tiskne nápovědu k programu.
- -i specifikuje soubor s přeloženým perzistentním obrazem. Může být specifikován pouze jeden. Pokud obraz nebyl udán, je vytvořena výchozí sada objektů.
- -f specifikuje soubory se zdrojovými kódy, jež budou při spuštění načteny a zpracovány. Pokud byl specifikován i obraz, je načten nejprve ten.

Po spuštění program vypíše vyzývací řádek, který začíná znaky „>>>“. Příkazy se zadávají za něj a odesílají se enterem. Program předpokládá víceřádkový vstup. Po zadání prvního řádku je prompt změněn na znaky „…“, což znamená, že zadaný řádek bude připojen k předchozímu. Příkazy jsou provedeny po odeslání prázdného řádku. Výsledek je poté vypsán pod příkazy, za značku „<<<“.

Datové sloty se vytvářejí udáním datového typu, názvu slotu a specifikováním jeho hodnoty za operátor „:=“.

```
String Object.bb := „bb“;
```

Již vytvořený slot je možné změnit operátorem „=“.

```
Object.bb = „cc“;
```

Pole a matice můžeme vytvářet pomocí literálu. Syntaxe je v obou případech téměř stejná a liší se jen použitými závorkami. Matice mají závorky „[„ a „]“, kdežto pole mají „[|“ a „|]“. Prvky na řádku oddělujeme čárkami a řádky pak středníky.

```
Array ar := [| Object, „ddd“; Null, 5 |];
Matrix m := [2, 3; 5, 6];
```

Metody se vytvářejí zprávou *method*. Jako argumenty se jí předají název metody v podobě řetězce, pole obsahující informace o argumentech, nebo Null, pokud žádné argumenty mít nebude. Posledním parametrem je blok kódu.

```
Lobby.method(„bbPrint", Null, {Object.bb.println();});
Lobby.method(„myPrint", [| String, „prefix“ |],
      { (prefix + bb).println; return(bb + „!“)});
```

Asynchronní zprávy posíláme pomocí znaku „@“. K výsledku pak přistupujeme přes objekt Future.

```
Future f := Lobby.@myPrint(„async: “);
f.value.toString.println;
```

Příloha 2 – Obsah přiloženého CD

Adresář *src* – Zdrojové kódy celého programu

Adresář *text* – Elektronická verze textové části práce.

Adresář *bin* – Spustitelný program v 64-bitové verzi

Adresář *examples* – Příklady z kapitoly 6.