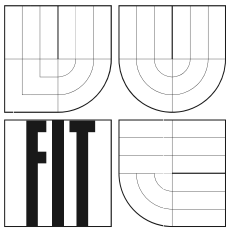
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Podpora beztřídního programování ve Squeak Smalltalku

Diplomová práce

2005 **Pavel Křivánek**

Podpora beztřídního programování ve Squeak Smalltalku

Odevzdáno na Fakultě informačních technologií Vysokého učení technického v Brně
dne 1. června 2005

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Vladimíra Janouška, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Pavel Křivánek
1. června 2005

Abstrakt

Cílem práce je navrhnout a implementovat sadu rozšíření prostředí Squeak Smalltalk, která do něj doplní podporu beztřídního programování. Popisuje podrobně základní koncepty této moderní objektové orientace ve formě, v jaké byla realizována v programovacím jazyce Self, a zkoumá dosavadní pokusy přenést beztřídní přístup do prostředí Squeak Smalltalk.

Tato práce porovnává jednotlivé možné implementace a na základě tohoto porovnání navrhuje řešení, které spočívá v rozšíření možností virtuálního stroje o podporu delegace a prototypů. Následně navrhuje podobu nového programovacího jazyka, jehož cílem je vhodným způsobem skloubit vlastnosti současných jazyků Self a Smalltalk-80 tak, aby dokázal efektivně využít stávající prostředí Squeak Smalltalk i provedené modifikace virtuálního stroje. Dále pro tento navržený jazyk implementuje funkční kompilátor provádějící překlad přímo do nativního bytekódu.

V závěru jsou zhodnoceny dosažené výsledky a případné nedostatky implementovaného řešení včetně návrhů na jeho další možné rozšiřování.

Klíčová slova

Objektově orientované jazyky, Smalltalk, Self, Marvin, beztřídní programování, prototypy, delegace, bytekód, virtuální stroj

Abstract

The goal of this diploma thesis is to implement a set of enhancements for the Squeak Smalltalk development environment which extends it with support of classless programming. It describes basic concepts of this modern object orientation, as realized in the Self programming language, and researches present attempts to bring a classless approach to Squeak Smalltalk. This thesis compares every possible implementation and, by virtue of this comparison, designs a solution which is based on an extension of the possibilities of the Squeak virtual machine with support of delegation and prototypes. Then it designs a new programming language which combines characteristics of Self and Smalltalk-80 to use the current Squeak Smalltalk environment and new virtual machine modifications effectively. This thesis implements a functional compiler for this new language with support of translation to the native bytecodes of Squeak. All results achieved, deficits and future development possibilities are summarized at the end.

Keywords

Object oriented programming, Smalltalk, Self, Marvin, classless programming, prototypes, delegation, bytecodes, virtual machine

Obsah

Seznam obrázků

Kapitola 1

Úvod

Squeak [7] je v současné době nejdynamičtěji se rozvíjející implementace jazyka Smalltalk-80. To, že je postaven nad tímto čistě objektově orientovaným jazykem, ale v žádném případě neznamená, že tomu tak bude vždy. Již v současné době se objevuje celá řada projektů, které umožňují nad Squeakem programovat pomocí jiných přístupů a jazyků, než je čistý Smalltalk. Nejviditelnější je projekt eToys, který je součástí squeakovského grafického uživatelského rozhraní Morphic a který poskytuje prostředí pro beztřídní vizualizované programování přizpůsobené pro výuku malých dětí. Jistý odklon od Smalltalku je patrný i u moderních velkých squeakovských projektů, jako je Croquet a Tweak [3]. K tomu lze přičíst několik samostatných implementací různých programovacích jazyků použitelných pod Squeakem (Lisp, Scheme či Prolog). Flexibilita samotného Smalltalku pak dále nahrává vzniku dalších nestandardních rozšíření, jako je AspectS [2].

Z řad uživatelů Squeaku občas dokonce zaznívá jisté rozčarování nad tím, že Squeak se stále drží "starého nudného Smalltalku-80". To může pro některé programátory přecházející z konvenčních jazyků znít celkem nepochopitelně, ale pravda je taková, že od počátku osmdesátých let bylo vytvořeno mnoho zajímavých konceptů beztřídním programováním založeným na prototypech počínaje a v tomto ohledu působí Squeak spíše konzervativně. Je to daň za komfort mít možnost vyjít na počátku vývoje z již funkčního řešení. Programátoři ve Squeaku se s tímto dědictvím musí v budoucnu umět vypořádat. Je sice fakt, že v porovnání s hlavním proudem vývoje programovacích prostředků působí Squeak velmi nekonvenčně, ale ustrnout na čtvrt století starých základech by se této platformě později mohlo vymstít.

Tato práce se snaží přispět k hledání cestiček, kudy by mohl směřovat budoucí vývoj Squeaku. Konkrétně se zabývá problematikou rozšíření schopností této platformy o přímou podporu beztřídního objektového paradigmatu, které bylo uvedeno v polovině osmdesátých let programovacím jazykem Self.

Při řešení tohoto úkolu byly stanoveny následující cíle:

- maximální přiblížení se schopnostem jazyka Self
- jednoduchá implementace
- snadná integrace se současným Squeakem
- rychlost výsledného řešení

Abychom pochopili vlastnosti beztřídních systémů, budeme se poměrně podrobně zabývat sémantikou jazyka Self. Jeho znalost není po čtenáři vyžadována. Nicméně tato práce předpokládá, že čtenář je alespoň obecně seznámen s principy a syntaxí jazyka Smalltalk-80 a orientuje se v problematice fungování virtuálních strojů.

V dalších částech této práce jsou popsány stávající pokusy implementovat beztřídní přístup na platformě Squeak Smalltalk. Jejich vyhodnocení následně přispívá do diskuse, jejímž cílem je vybrat nejlepší řešení vhodné pro další rozšiřování.

Zvolené řešení je dále podrobně popsáno jak po stránce návrhu tak implementace včetně popisu problematiky jejího napojení na infrastrukturu systému Squeak.

V závěru jsou zhodnoceny dosažené výsledky a nastíněny další možnosti vývoje tohoto projektu.

Tato diplomová práce nenavazuje na Ročníkový ani Semestrální projekt.

Kapitola 2

Třídní systém

Během sedmdesátých let minulého století probíhal ve výzkumném centru PARC firmy Xerox vývoj nového programovacího jazyka, který se pokusil významným způsobem rozšířit myšlenku využití objektů při programování nastíněnou v šedesátých letech simulačním jazykem Simula. Dalším velmi silným zdrojem inspirace pro jeho autory byl programovací jazyk Lisp, z něhož zpočátku výrazně čerpal i syntaktický návrh.

Smalltalk se stal prvním moderním čistě objektově orientovaným programovacím jazykem. Přišel s celou řadou nových konceptů, z nichž mnohé začínají do širší praxe pronikat až v posledních letech. Mezi základní kameny jeho návrhu patří zejména tyto vlastnosti:

1. vše včetně základních entit jako čísla, řetězce, booleovské hodnoty či třídy jsou objekty
2. jediný možný prostředek, jímž mezi sebou mohou objekty komunikovat, je zasílání zpráv
3. dynamická typová kontrola
4. inkrementální a explorativní programování (exploratory programming)

Smalltalk vychází z třídního systému, ve kterém jsou striktně dodržována následující pravidla:

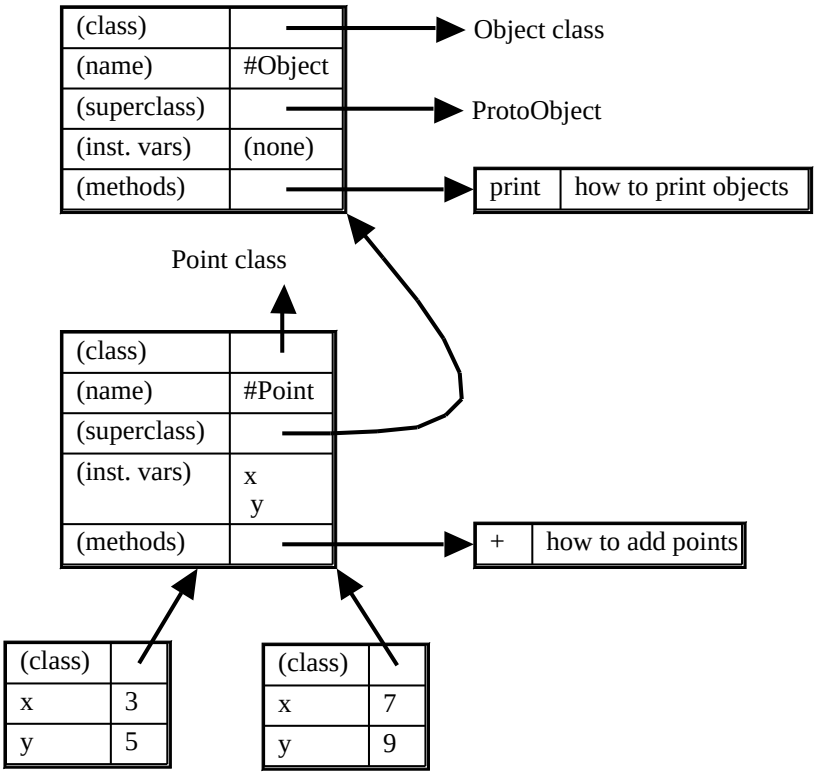
1. vše včetně tříd jsou objekty
2. každý objekt je instancí nějaké třídy
3. každá třída má svého předka

Na základě těchto postulátů je vybudována poměrně komplikovaná hierarchie využívající tříd a metatříd k tomu, aby programátor mohl snadno vytvářet objekty se sdíleným chováním. Složitost výsledné hierarchické struktury je dána mimo jiné i faktem, že se kvůli maximální obecnosti nevyhýbá ani cyklům.

Pokud je objektu zaslána zpráva, snaží se její selektor vyhledat ve třídě, jíž je objekt instancí. Pokud jej nalezne, vykoná příslušnou kompilovanou metodu v kontextu příjemce. V případě, že ve třídě není nalezena žádná metoda, jejíž jméno by odpovídalo selektoru zaslané zprávy, probíhá vyhledávání v předkovi prohledané třídy. Ve vyhledávání se pokračuje až do okamžiku, kdy je v hierarchii objeveno zacyklení.

Metatřídy jsou speciální třídy, které definují chování jiných tříd. Jejich existence plyne z pravidla, podle kterého mají všechny objekty přiřazenu třídu definující jejich vlastnosti, protože třídy jsou samozřejmě také objekty. S pomocí metatříd se implementují například konstruktory objektů.

Konkrétní podoba třídní hierarchie Smalltalku zde nebude diskutována. Předpokládá se, že čtenář je s ní alespoň obecně seznámen. Je popsána například v 5. části [1].



Obrázek 2.1: Třídní systém

Kapitola 3

Programovací jazyk Self

Self [5] je objektově orientovaný programovací jazyk s úzkou vazbou na své vývojové prostředí. Svojí syntaxí i sémantikou má velmi blízko k jazyku Smalltalk-80, ze kterého přímo vychází.

Stejně jako Smalltalk je i Self čistě objektově orientovaný dynamicky typovaný jazyk s inkrementálně budovaným prostředím s podporou explorativního programování. Self se ale od Smalltalku v některých směrech odlišuje:

- Self má jednodušší sémantiku a syntaxi než Smalltak. Jeho návrh je zaměřen na maximální jednoduchost a konkrétnost použitých konstrukcí. Díky tomu je více vlastností Selfu implementováno přímo v něm samém
- Self nemá žádné třídy. Namísto třídního systému používá konkrétnější objektový systém.
- vývojové prostředí Selfu je mnohem více zaměřeno na práci s existujícími objekty.

Programovací jazyk Self byl navržen v roce 1986 Davidem Ungarem a Randallem Smithem. Jeho první implementace byla vyvinuta roku 1987 na Stanfordské univerzitě. Zde sloužila jako podklad pro další výzkum, aby se následně roku 1990 dočkala veřejného vydání. Vývoje Selfu se v roce 1991 ujala firma Sun Microsystems Labs [8] a dále jej rozvinula [12].

V následujícím popisu se záměrně budeme snažit vyhnout v maximální možné míře syntaktické stránce Selfu. Ta bude diskutována později.

3.1 Objekty

Self rozlišuje dvě základní entity jazyka: objekty a sloty. Objekty jsou tvořeny kolekcí slotů a případným kódem, který má být v rámci objektu vykonán. Každý slot je pak tvořen jménem a referencí na jiný objekt. Tato jednoduchá struktura stačí, aby Self realizoval objektové paradigma v míře, jaká je známá ze Smalltalku a jiných objektově orientovaných programovacích jazyků. Navíc díky ní Self přináší další koncepty, které nejsou v jiných jazycích běžné (násobná a dynamická dědičnost).

Všechny objekty v Selfu mají sice stejnou strukturu, ale přesto lze mezi nimi rozlišit tři typy objektů podle toho, jakým způsobem se pracuje s asociovaným kódem:

1. datové (regulární) objekty. Tyto objekty nemají přiřazen žádný asociovaný kód
2. metody. Jsou to regulární objekty s přiřazeným kódem
3. bloky. Bloky jsou syntaktickou zkratkou pro dva regulární objekty, z nichž jeden má asociován kód

3.2 Sloty

Sloty jsou pojmenované reference na objekty. Rozlišuje se několik typů slotů:

1. Datové sloty určené pro čtení
2. Datové sloty určené pro čtení i zápis. Jedná se o syntaktickou zkratku pro dva sloty, z nichž jeden je určen pouze pro čtení a druhý pouze pro zápis. Oba referencují vždy stejný objekt
3. Sloty obsahující metody
4. Sloty formálních parametrů (argumentové sloty). Slouží k předávání parametrů metodám a blokům
5. Rodičovské sloty určené pro čtení. Slouží k delegování zpráv na jiné objekty
6. Rodičovské sloty určené pro čtení i zápis. Jedná se syntaktickou zkratku pro dva sloty, z nichž jeden je rodičovský slot určený pouze pro čtení a druhý pouze pro zápis. Oba referencují vždy stejného rodiče.

Dále lze identifikovat ještě další dva pomocné druhy slotů

7. Poznámkové sloty. Slouží k přiřazování poznámek k objektům a jiným slotům. Mohou obsahovat pouze řetězce a nemají přiřazeno žádné jméno. Pokud nebude řečeno jinak, dále uváděné informace o slotech se na tento typ nevztahují.
8. Rodičovské argumentové sloty. Nemají podporu v syntaxi jazyka. Slouží jako pomocné sloty při aktivaci metod.
9. Nepojmenované sloty. Nemají podporu v syntaxi jazyka. Slouží jako pomocné sloty při aktivaci bloků.
10. Nepojmenované rodičovské sloty. Nemají podporu v syntaxi jazyka. Slouží jako pomocné sloty při aktivaci bloků.

readOnlyParentSlot*	
readWriteParentSlot*	←
:parentArgumentSlot*	
(parentUnnamedSlot*)	
readOnlyDataSlot	
readWriteDataSlot	←
:argumentSlot	
(unnamedSlot)	
methodSlot	(↑ 3+4)
{ commentary slot }	

Obrázek 3.1: Objekt se všemi typy slotů

3.3 Idealizovaný bytekód

V rámci dalšího popisu programovacího jazyka Self budeme k demonstraci mechanismů, které využívá, používat idealizovaný bytekód, který bude obsahovat pouze pět instrukcí. Tento bytekód se neshoduje se skutečným bytekódem, jaký využívají implentace Selfu. Jedná se pouze o teoretickou konstrukci, která však přesně koresponduje se skutečným chováním selfovských programů. Byl vyvinut pro potřeby této práce a jinde v literatuře se neuvádí. Využití nachází především v přiložených grafech.

Instrukce idealizovaného bytekódu Selfu jsou:

pushSelf

- nemá žádný argument
- na vrchol zásobníku umístí objekt, v jehož kontextu je kód vykonáván
- alternativní označení: pushReceiver

send

- jako argument přijímá selektor volané zprávy
- provede volání metody v kontextu objektu, který je na vrcholu zásobníku
- ze zásobníku odebere vrchol a na jeho místo umístí výsledek volání

resend

- jako argument přijímá selektor volané metody
- provede volání metody v kontextu objektu, který je na vrcholu zásobníku
- ze zásobníku odebere vrchol a na jeho místo umístí výsledek volání metody
- vyhledávání slotu odpovídajícího selektoru zprávy začíná rodiči objektu, kterému patří metoda, v jejímž kódu je resend prováděn (method holder)

pop

- vyzvedne vrchol zásobníku a zahodí jej

returnTop

- návrat z metody. Za návratovou hodnotu se považuje aktuální vrchol zásobníku

3.4 Hledání slotu v objektu

Každý slot v objektu je identifikován svým jménem. Pokud je objektu zaslána zpráva, jsou postupně prohledávány všechny sloty v objektu a selektor zprávy porovnáván se jmény slotů.

Když je nalezena shoda selektoru zprávy a jména slotu, je v případě datových slotů, argumentových slotů a rodičovských slotů vrácena reference na odkazovaný objekt.

V případě, že nalezený slot obsahuje metodu, je tato metoda invokována a vrácena reference na objekt získaný jako návratová hodnota volané metody.

Do jména zprávy se nezapočítávají speciální znaky jako hvězdička nebo dvojtečka, které pomáhají na syntaktické úrovni jazyka rozlišovat jednotlivé typy slotů.

Self nijak nerozlišuje mezi přístupem k datovým, rodičovským a argumentovým slotům a slotům obsahujícím metody. To znamená, že v rámci rozhraní objektu mohou být tyto typy slotů libovolně zaměňovány.

Pokud není nalezena žádná shoda selektoru volané zprávy a slotu, provede se nejdříve pokus delegovat zprávu na objekty referencované rodičovskými sloty. V případě, že delegace na rodičovské sloty proběhne neúspěšně a tedy ani v rodičovských objektech není nalezen žádný slot odpovídající selektoru volané zprávy, je vyvolána výjimka informující o nenalezení slotu v objektu.

3.5 Delegace

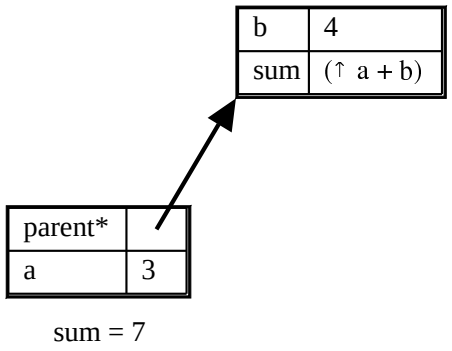
Pokud není v objektu, kterému je zaslána zpráva, nalezen žádný slot, jehož jméno odpovídá selektoru zprávy, provede se delegace na rodičovské objekty. To znamená, že virtuální stroj začne prohledávat postupně sloty objektů referencovaných rodičovskými sloty stejným způsobem, jaký je použit při hledání slotu u jednoho objektu.

Prohledávání objektů je prováděno rekurzivně, přičemž vyhledávací algoritmus povoluje cykly. Přitom je zaručeno, že žádný objekt nebude prohledán vícekrát.

Pokud je v delegovaném rodičovském objektu nebo posléze v některém z jeho rodičovských objektů nalezena shoda selektoru zprávy a jména slotu, je v případě datových slotů, argumentových slotů a rodičovských slotů vrácena reference na objekt odkazovaný nalezeným slotem.

V případě, že naležený slot obsahuje metodu, je tato metoda invokována a vrácena reference na objekt získaný jako návratová hodnota volané metody. Na rozdíl od běžného volání je ale tato metoda aktivována v kontextu objektu, v němž prohledávání začalo, tedy v objektu skutečného příjemce

Pakliže se při vyhledávání slotů ve více rodičovských objektech narazí na nejednoznačnosti, je vyvolána výjimka.



Obrázek 3.2: Delegace

Kromě této implicitní delegace má Self podporu ještě pro tzv. explicitní delegaci. Ta se liší v tom, že se přímo definuje objekt, na který má být delegace provedena. Tato konstrukce se využívá např. při řešení některých problémů s překrýváním jmenných prostorů. Nicméně, protože se nejedná o častou operaci, nemá přímou podporu v syntaxi jazyka a provádí se pomocí primitivních metod.

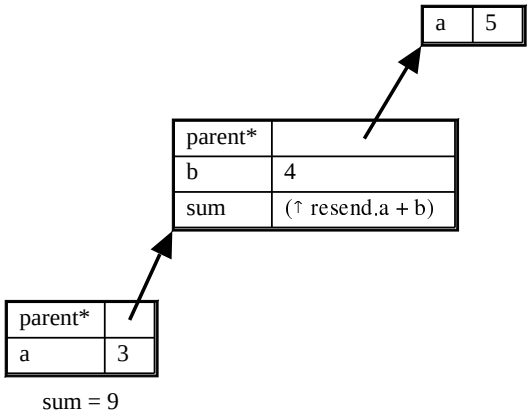
3.6 Resend

Resend je speciální typ volání, které se od běžného liší tím, že vyhledávání slotu odpovídajícího selektoru zasílané zprávy nezačíná v objektu, v jehož kontextu se kód provádí, ale od rodičů objektu, který svým slotem referencuje prováděnou metodu.

Reference na tento objekt (vlastník metody) musí být součástí metody. Typicky bývá obsažena v kolekci literálů. Vlastník metody většinou nebývá objekt, v jehož kontextu je metoda volána.

Resend se vždy aplikuje na objekt, v jehož kontextu probíhá volání metody, nikoliv přímo aktivace kódu. Rozdíl mezi těmito dvěma operacemi bude popsán později.

Vyhledávání slotů od rodičů vlastníka metody probíhá podle stejného algoritmu, který se používá pro delegaci.



Obrázek 3.3: Resend

3.7 Formální popis algoritmů

Tato kapitola popisuje algoritmy použité virtuálním strojem Selfu k vyhledávání slotů při zasílání a přeposílání zpráv. Převzato z [10] a [11] (přiblíženo syntaxi Smalltalku).

3.7.1 Vyhledávací algoritmus

$lookup(obj, sel, V)$

Vstup:	obj	prohledávaný objekt
	sel	selektor zprávy
	V	množina již prohledaných objektů (visited)
Výstup:	M	množina nalezených slotů (matching)

$obj \in V$
  ifTrue: [$M \leftarrow \emptyset$ **]** “detekce cyklu”
  ifFalse: [
    $M \leftarrow \{s \in obj \mid s.name = sel\}$ **.** “zkus lokální sloty”
    $M = \emptyset$ **ifTrue: [** $M \leftarrow parent\_lookup(obj, sel, V)$ **]].** “zkus rodičovské sloty”
$\uparrow M$

Jedná se o pomocný algoritmu použitý při zasílání zpráv a při přímém resendu. V definovaném objektu se pokusí nalézt slot a pokud se mu to nepodaří, prohledá objekty referencované rodičovskými sloty.

3.7.2 Hledání v rodiči

$parent\_lookup(obj, sel, V)$

Vstup:	obj	prohledávaný objekt
	sel	selektor zprávy
	V	množina již prohledaných objektů (visited)
Výstup:	M	množina nalezených slotů (matching)

$P \leftarrow \{s \in obj \mid s.isParent\}$ **.** "všichni rodiče"
$obj \in V$ **.**
$M \leftarrow \bigcup_{s \in P}(s.contents, sel, V \cup \{obj\})$ **.** "rekurzivně prohledej rodiče"
$\uparrow M$

Pomocný algoritmus, který u daného objektu prohledá rodičovské sloty a v objektech, které jsou jimi referencovány, se pokusí najít sloty odpovídající dodanému selektoru.

3.7.3 Zaslání zprávy

$send(rec, sel, args)$

Vstup:	rec	příjemce zprávy
	sel	selektor zprávy
	$args$	skutečné argumenty zprávy
Výstup:	res	výsledný objekt

$sel.beginsWithUndersore$ "selektor začíná podtržítkem"
  ifTrue: [$invoke\_primitive(rec, sel, args)$ **]].** "volání primitivy"
  ifFalse: [
    $M \leftarrow lookup(rec, sel, \emptyset)$ **.**
    $|M| = 0$ **ifTrue: [** $error$: $messagenotunderstood$ **].**
    $|M| = 1$ **ifTrue: [** $res \leftarrow eval(rec, M, args)$ **].**
    $|M| > 1$ **ifTrue: [** $error$: $ambiguousmessagesend$ **]].**
$\uparrow res$

Pokusí se objektu zaslat zprávu s dodanými argumenty. Rozlišuje primitivní a obyčejné zprávy. Pokud se nepodaří najít vhodný slot nebo pokud jich je nalezeno více, jsou vyvolány výjimky. V opačném případě se provede vyhodnocení slotu, tzn. vrácení jeho hodnoty u rodičovských a datových slotů respektive invokace metody referencované slotem.

3.7.4 Nepřímý resend

$resend(rec, smh, sel, args)$

Vstup:	*rec*	příjemce zprávy
	smh	vlastník metody
	sel	selektor zprávy
	args	skutečné argumenty zprávy
Výstup:	*res*	výsledný objekt

$M \leftarrow parent\_lookup(smh, sel, \emptyset)$ **.**
$|M| = 0$ **ifTrue: [** *error* : *messagenotunderstood* **].**
$|M| = 1$ **ifTrue: [** $res \leftarrow eval(rec, M, args)$ **].**
$|M| > 1$ **ifTrue: [** *error* : *ambiguousmessagesend* **]].**
$\uparrow res$

Provádí se při běžném přeposílání zprávy. Sloty se začínají vyhledávat od rodičů vlastníka metody, ve které je přeposílání použito.

3.7.5 Přímý resend

$undirected\_resend(rec, smh, del, sel, args)$

Vstup:	*rec*	příjemce zprávy
	smh	vlastník metody
	del	jméno delegovaného objektu
	sel	selektor zprávy
	args	skutečné argumenty zprávy
Výstup:	*res*	výsledný objekt

$D \leftarrow \{s \in smh \mid s.name = del\}$ **.** "najdi delegovaný objekt"
$|D| = 0$ **ifTrue: [** *error* : *missingdelegatee* **].**
$M \leftarrow lookup(smh.del, sel, \emptyset)$ **.**
$|M| = 0$ **ifTrue: [** *error* : *messagenotunderstood* **].**
$|M| = 1$ **ifTrue: [** $res \leftarrow eval(rec, M, args)$ **].**
$|M| > 1$ **ifTrue: [** *error* : *ambiguousmessagesend* **]].**
$\uparrow res$

Přeposílání zprávy na přímo specifikovaný objekt. Jedná se o pomocný algoritmus využívaný některými primitivními metodami Selfu. Nemá přímou oporu v syntaxi jazyka.

3.8 Aktivace metody

V Selfu jsou i metody běžné objekty. Na rozdíl od Smalltalku ale Self používá jejich konkrétnější reprezentaci, která je konzistentní s použitou sémantikou objektů. Při invokaci metod se tedy pracuje více na úrovní světa Selfu a tato operace je mnohem méně vnořena do útrob virtuálního stroje.

Objekt metody je tvořen rodičovským argumentovým slotem pojmenovaným `:self*` a dále množinou pojmenovaných slotů formálních parametrů. Objekt metody také obsahuje příslušný kód, který má být při zavolání metody proveden.

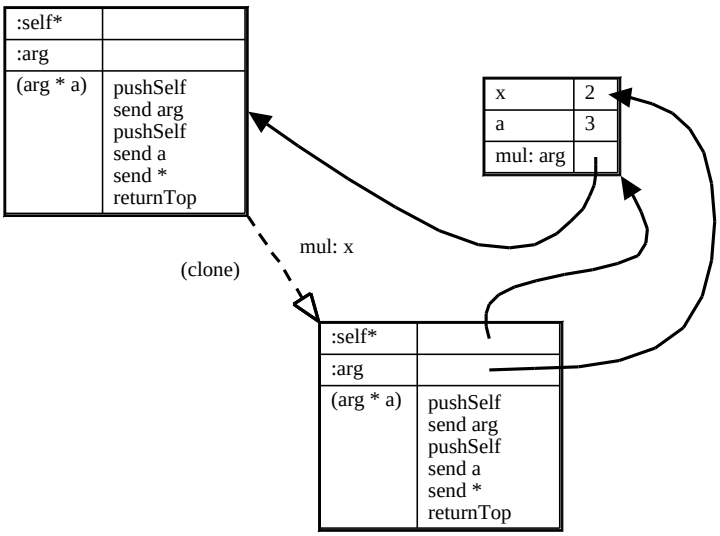
Pokud má být metoda aktivována v kontextu nějakého objektu, vytvoří virtuální stroj klon objektu metody a provede jeho inicializaci. Takto vytvořený objekt se nazývá aktivační objekt metody. Inicializace aktivačního objektu metody spočívá v tom, že do rodičovského argumentového slotu `:self*` je vložena reference na objekt, v jehož kontextu má být metoda provedena. Pokud metoda

obsahuje argumety, jsou příslušné argumentové sloty ze zásobníku postupně naplněny skutečnými parametry, s kterými byla metoda zavolána.

V posledním kroku je aktivován kód přiřazený metodě. V něm instrukce `pushSelf` idealizovaného bytekódu umístí na vrchol zásobníku přímo referenci na aktivační objekt metody. K argumentům metody se přistupuje naprosto shodně jako k běžným slotům – tedy voláním zprávy. Pokud se v metodě použije přístup k nějakému slotu, jehož jméno neodpovídá žádnému argumentu metody, je volání přeposláno pomocí rodičovského argumentového slotu `:self*` na objekt, v jehož kontextu je metoda prováděna a z něj pak případně na jeho rodiče.

Tímto způsobem je zajištěno, že při vykonávání metody jsou dodané argumenty upřednostňovány před sloty objektu s aktivním kontextem. Konstrukce `self` použitá v kódu metody je interpretována jako běžné zaslání zprávy v rámci aktivačního objektu metody, takže jeho sémantika se naprosto shoduje s běžnou smalltalkovskou praxí.

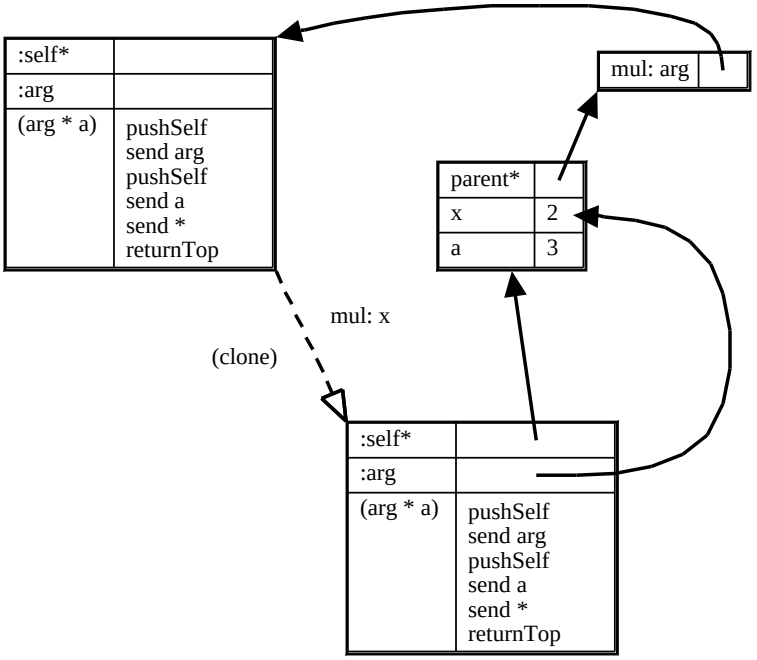
Aktivační objekt je při návratu z metody odstraněn garbage collectorem. Objekt metody se klonuje na aktivační objekt metody proto, aby jej mohlo využívat více procesů zároveň a byla možná rekurze.



Obrázek 3.4: Aktivace metody

Při aktivaci kódu se používá pouhé přeposílání zpráv rodičům. Přeposílání zpráv přes rodičovský argumentový slot `:self*` hraje stěžejní úlohu při implementaci delegace. Jestliže je totiž aktivována v rámci jedné metody metoda jiná, při vytvoření aktivačního objektu nové metody se do něj zkopíruje rodičovský argumentový slot `:self*` z původní metody, takže zasílání zpráv probíhá v kontextu stejného objektu, ať byla volaná metoda přímo součástí tohoto objektu nebo jeho libovolného rodiče.

Totéž platí, pokud je v rámci jedné metody rekurzivně zavolána tatáž metoda.



Obrázek 3.5: Aktivace metody při delegaci

3.9 Aktivace bloku

Bloky jsou ve skutečnosti syntaktickou zkratkou pro vytvoření dvou objektů:

- datový objekt bloku
- metoda bloku

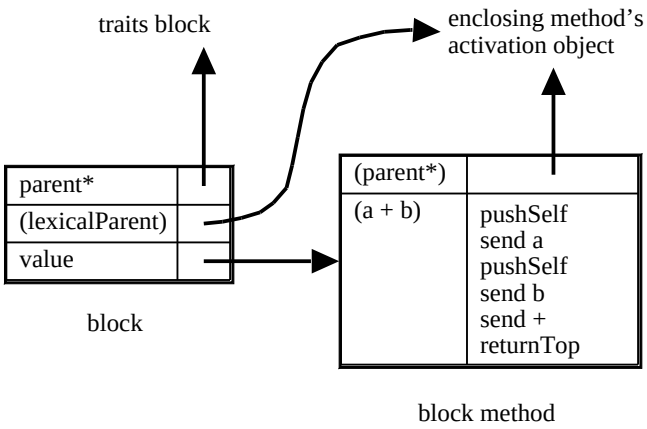
Datový objekt bloku obsahuje tři sloty. Prvním z nich je pojmenovaný rodičovský slot, který referencuje standardní objekt traits block zapouzdřující zprávy, kterým má každý blok rozumět.

Dalším slotem datového objektu bloku je nepojmenovaný slot, tzv. lexikální rodič. Ten se inicializuje při invokaci metody, která je vlastníkem bloku tak, že referencuje aktivační objekt metody. K tomuto slotu nelze pomocí standardních prostředků Selfu přistupovat.

Posledním slotem je aktivační metoda bloku, typicky pojmenovaný `value`, `value:` apod. Jméno této metody se určuje při vytváření bloku podle počtu argumetnů, který literál pro blok obsahuje. Tato aktivační metoda referencuje objekt metody bloku.

Metoda bloku je strukturou podobná běžnému objektu metody. Na rozdíl od ni ale nemá rodičovský argumentový slot `:self*`. Ten je v metodě bloku nahrazen nepojmenovaným rodičovským slotem. V okamžiku zavolání metody bloku, k čemuž dojde, když je bloku zaslána aktivační zpráva (např. `value`), je objekt metody bloku naklonován. Vytvořený klon se nazývá aktivační objekt bloku a podobně jako u aktivačního objektu metody jsou i jeho argumentové sloty inicializovány ze zásobníku skutečnými parametry.

Při klonování metody bloku je rovněž inicializován nepojmenovaný rodičovský slot tak, že referencuje aktivační objekt metody bloku, v níž je blok vytvořen. Jedná se tedy o stejný objekt, který je referencován lexikálním rodičem datového objektu bloku.



Obrázek 3.6: Struktura bloku

Při aktivaci bloku (zprávou `value`, `value:` apod.) se kód metody bloku provádí v kontextu aktivačního objektu bloku.

Touto poměrně komplikovanou strukturou je mimo jiné docíleno toho, že lokální proměnné definované uvnitř bloku a argumenty bloku mají přednost před lokálními sloty definovanými v rámci nadřazené metody.

Komplikovanější situace nastane v okamžiku, kdy se jeden nebo více bloků nachází uvnitř jiného bloku. V takovém případě lexikální rodiče datových objektů hlavního i dalších zanořených bloků referencují stále tentýž objekt – aktivační objekt mateřské metody. Naproti tomu nepojmenované rodičovské sloty aktivačních objektů bloků referencují vždy aktivační objekt nadřazeného bloku, takže vnořený blok může využívat lokálních slotů definovaných v rámci bloku nadřazeného.

Pro programátora je nepojmenovaný slot definující lexikálního rodiče datového objektu bloku nevýznamný a sama o sobě jeho přítomnost nevyúsťuje v žádnou sémantickou akci. Slouží jako pomocný slot virtuálního stroje. V důsledku toho například následující Selfovský kód

```
(| method = ( | slot = 42 | [ ] slot ) |) method
```

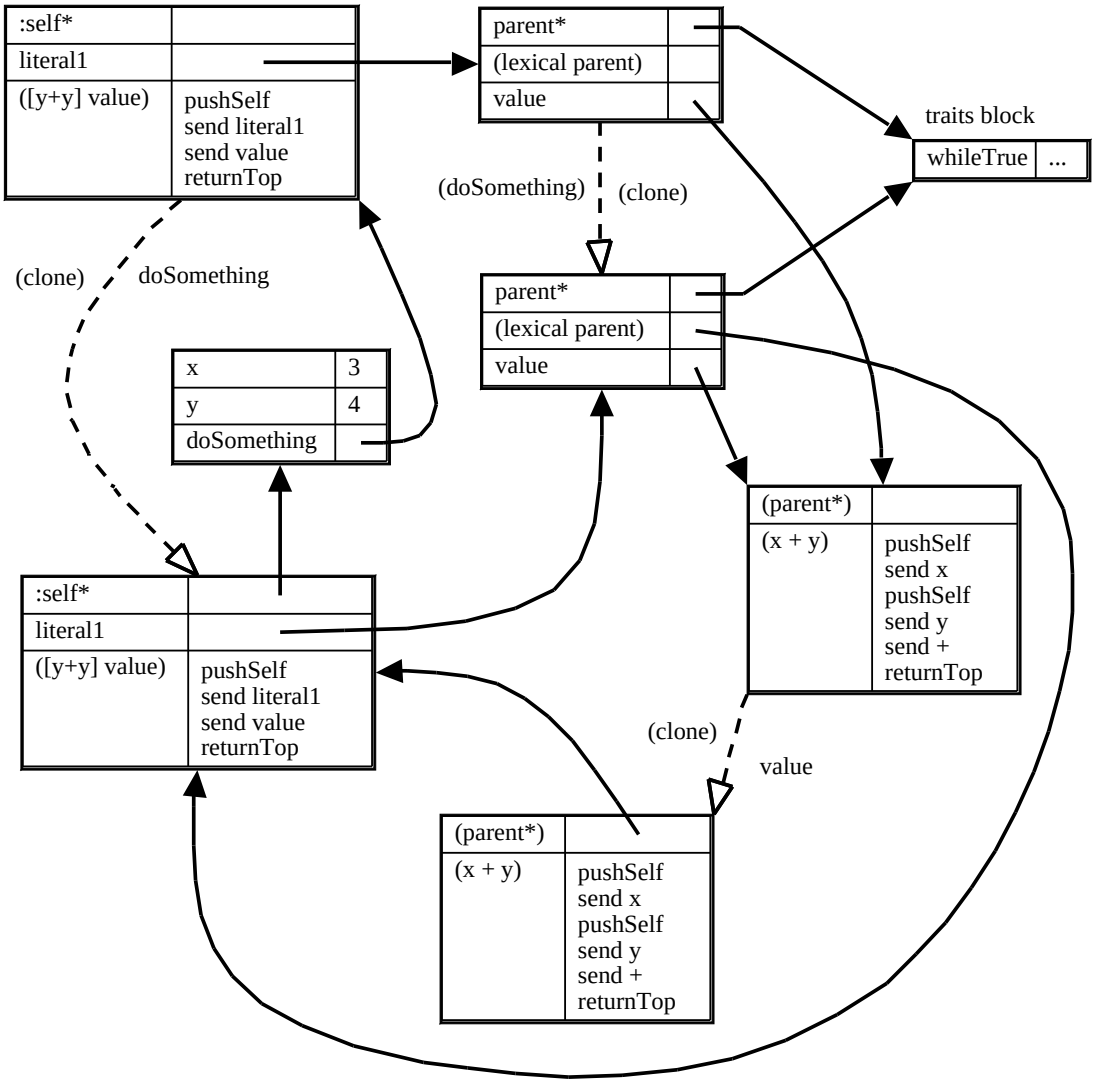
vyvolá výjimku informující o nenalezení slotu `slot`. Lexikální rodič není rodičovským slotem, přes který by se prováděla delegace.

V Selfu nelze volat blok, pokud jeho lexikální rodič již byl zrušen, tedy byla ukončena jeho nadřazená metoda.

3.10 Zapouzdření

Oproti běžným objektově orientovaným jazykům včetně Smalltalku používá Self extrémní přístup k zapouzdření dat objektu. Ve Smalltalku je stav objektu uchováván pomocí instančních proměnných, k nimž mají přístup pouze instanční metody objektu. Tento přístup není realizován zasláním zprávy, ale jsou pro něj vyhrazeny specializované instrukce bytekódu. Protože ve Smalltalku existuje pouze jediná možnost, jak může jeden objekt komunikovat s druhým, a to prostřednictvím zasílání zpráv, nemá žádný jiný objekt možnost k instančním proměnným objektu přistupovat a modifikovat je.

Naproti tomu Self žádné instanční proměnné nemá a k uchovávání stavu objektu slouží pouze sloty, k nimž se přistupuje pomocí zasílání zpráv. Objekt v Selfu tedy svoje data zapouzdřuje i



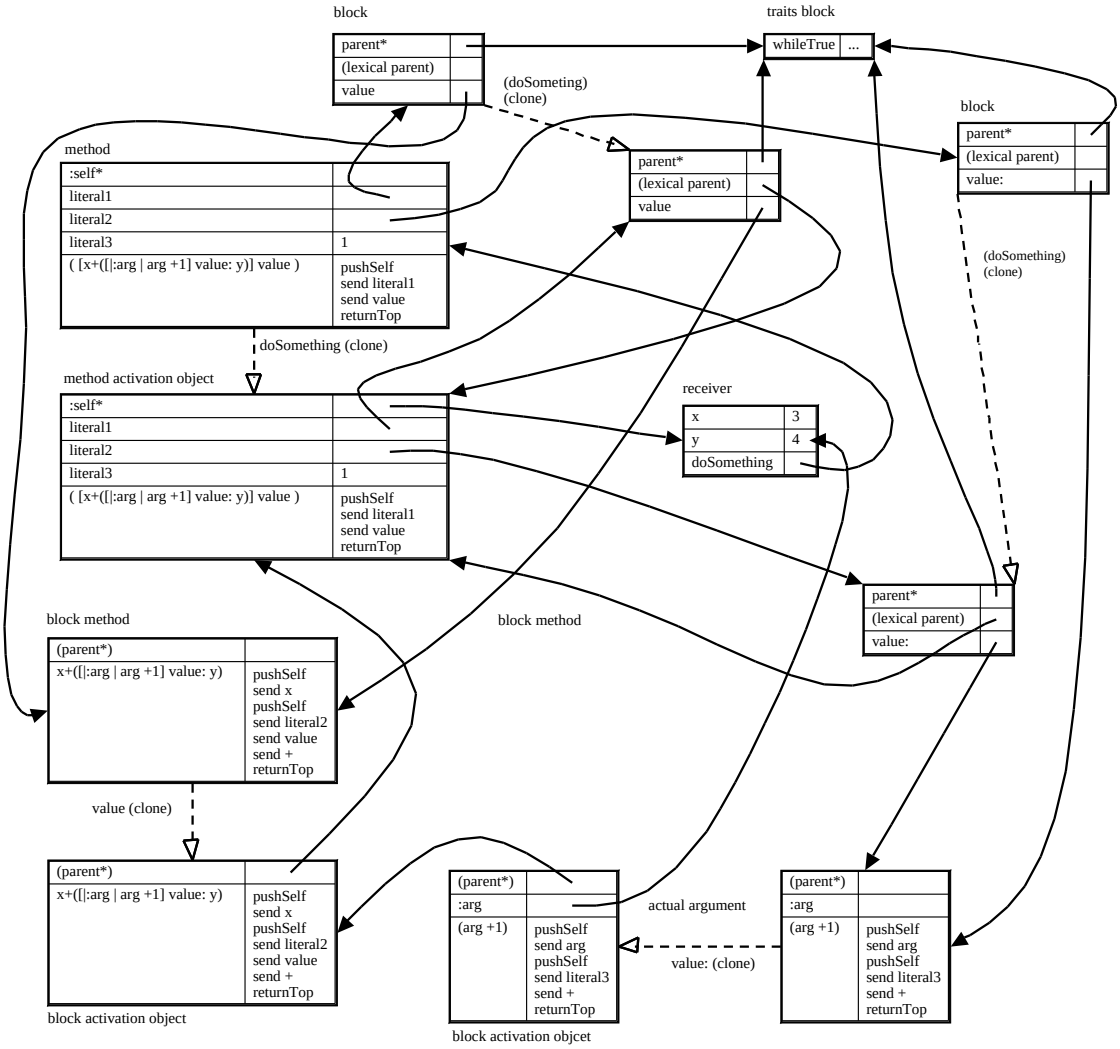
Obrázek 3.7: Aktivace bloku

před sebou samým. Čistota této konstrukce významným způsobem rozhraní objektu zobecňuje a zjednodušuje případný refactoring. Lze libovolně zaměňovat datové sloty za metody a zpět bez toho, aby se rozhraní objektu jakkoliv změnilo.

Stejně jako Smalltalk nepoužívá Self žádné rozlišení slotů na privátní, chráněné a veřejné, jako je tomu například u C++. To může být poměrně diskutabilní rys těchto dvou jazyků, ale je nutno uznat, že díky tomu je programátor mnohem méně svazován, výrazně se zjednodušuje refactoring a i tyto jazyky samy jsou jednodušší a čistší.

U Smalltalku lze vidět jistý náznak ochrany rozhraní v přítomnosti instančních proměnných, ke kterým má přístup pouze objekt sám. Self opustil i toto, což mu přineslo výše uvedené výhody, ale na druhou stranu se svému okolí více odkrývá. Z programátorské praxe ve Smalltalku lze ale vyvodit následující závěry:

- většina objektů své instanční proměnné svému okolí stejně zpřístupňuje pomocí jedné nebo



Obrázek 3.8: Aktivace zanořeného bloku

dvou tzv. přistupujících metod. Ty mají většinou velmi jednoduchý charakter. V těchto případech probíhá přístup Selfu k datům efektivněji, protože Smalltalk musí nejdříve vyhledat příslušnou metodu, spustit ji a v ní provést zápis do instanční proměnné s patřičným indexem.

- při návrhu objektu se velmi často stejně kód přepisuje tak, aby i sám objekt přistupoval ke svým datům pomocí přistupujících metod. Tento refactoring je poměrně nepříjemný.
- konstruktory objektů (jeho třídní metody) nemohou k instančním proměnným přímo přistupovat, což si často zbytečně vynucuje vytvářet speciální instanční metody pro inicializaci.

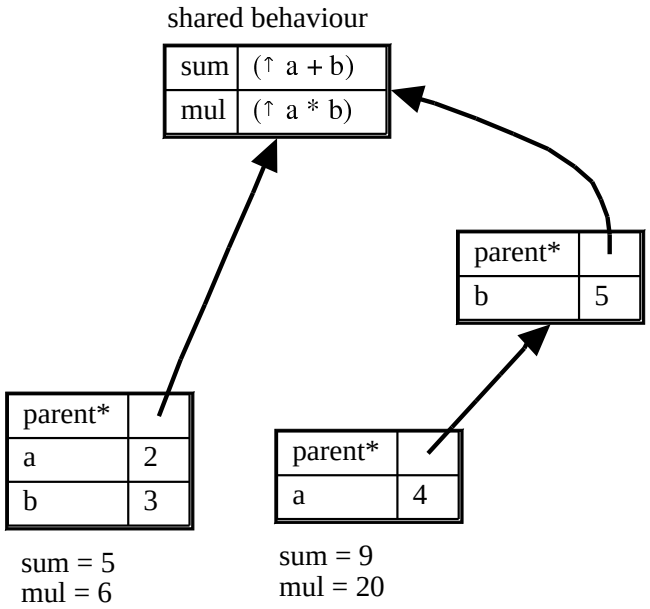
Na druhou stranu může být přímý přístup k instančním proměnným ve Smalltalku rychlejší, což ale lze setřít vhodnými optimalizacemi na úrovni virtuálního stroje.

Samozřejmě jak v Selfu tak ve Smalltalku může objekt své rozhraní lépe ochránit, pokud to skutečně potřebuje, ale za cenu menší efektivity zpracování.

3.11 Simulace třídní hierarchie

Přestože Self ze svého návrhu úplně odstranil pojem třída, neznamená to, že by nebyl schopen implementovat koncept sdílení chování se stejnými schopnostmi, jaké má třídní systém Smalltalku. Naopak se ukazuje, že vyjadřovací síla Selfu je v tomto směru podstatně větší než ta, jaké kdy dosáhl Smalltalk.

To, jak Self implementuje sdílené chování, je velice prosté. Pokud více objektů referencuje přes některý svůj rodičovský slot jeden objekt, pak sdílí jeho rozhraní. To samozřejmě platí i v případě, že vazba na rodiče probíhá přes jednoho nebo více dalších rodičů.



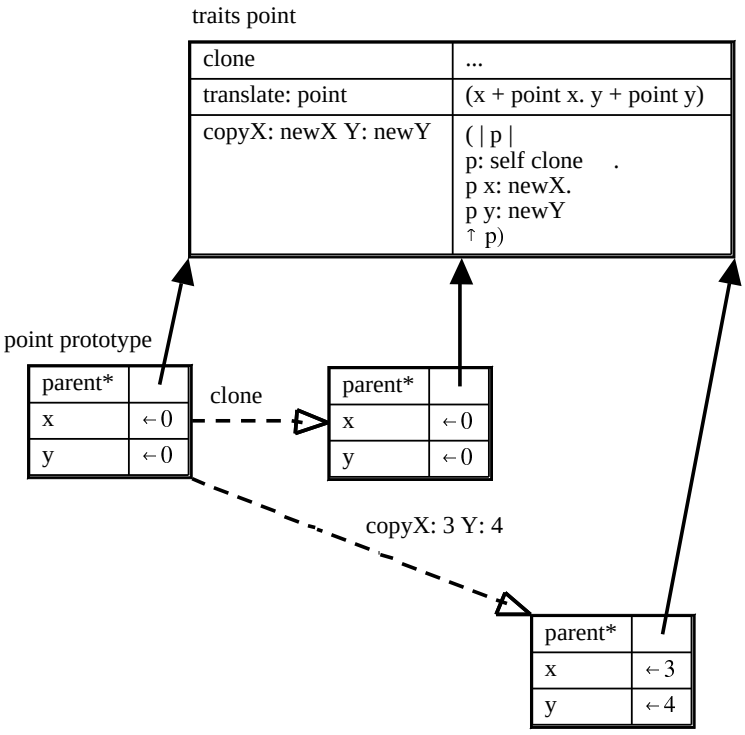
Obrázek 3.9: Objekty se sdíleným chováním

K plnohodnotné náhradě třídního systému si ale s pouhou delegací nevystačíme a musíme zavést o něco složitější strukturu.

Objekty kromě toho, že sdílí určitou část rozhraní (chování), musí obsahovat i data. Od těchto dat se také většinou očekává, že budou inicializována na nějaké standardní hodnoty. Tato data přirozeně nemohou být přímo součástí sdíleného rozhraní objektů, protože by je sdílely všechny objekty provádějící na rozhraní delegaci. Proto by objekty měly samy o sobě obsahovat datové sloty kódující jejich současný stav. Objekty se sdíleným chováním by neměly být konstruovány tak, aby obsahovaly ekvivalenty datových slotů, protože by snadno mohlo dojít k mylné interpretaci. Tento objekt se sdíleným chováním se označuje jako trait (rys).

Naskýtá se otázka, jak zajistit, aby objekty se stejným sdíleným chováním, tedy ekvivalenty instanci téže třídy ze Smalltalku, měly stejné i celé kompletní rozhraní a shodně inicializované datové sloty. Nejjednodušší cesta, jak toho docílit, je objekt simulující třídu a obsahující tedy sdílené chování nejčastěji ve formě množiny slotů s metodami, doplnit o objekt, který bude reprezentovat standardně inicializovanou instanci simulované třídy, tzv. prototyp. Skutečné instance pak vzniknou naklonováním nebo zkopírováním prototypu. Odtud se přeneseně beztřídní objekty nazývají prototypy.

Prototyp by neměl obsahovat žádné metody sdíleného chování. Tím, že se sdílené chování



Obrázek 3.10: Rys a prototyp

deleguje na rys, ušetří se sloty v instančních objektech (a tedy i paměť) a společné chování podobných objektů lze modifikovat na jednom místě.

K vytváření instancí se nevolají metody (konstruktory), které by byly obsaženy v objektu rysu, ale nechá se vytvořit kopie prototypu. Ve výsledku je tato simulace třídní struktury jednodušší a pochopitelnější než ta, kterou používá Smalltalk, protože se můžeme zcela obejít bez metatříd.

Prototyp a množina objektů, které vzniknou jeho naklonováním, tzn. že mají shodnou strukturu a liší se pouze v aktuálních hodnotách slotů, se nazývá clone family. Pokud nějaký objekt z této rodiny změní své rozhraní (např. si přidá či odebere sloty), struktura ostatních objektů zůstane nedotčena.

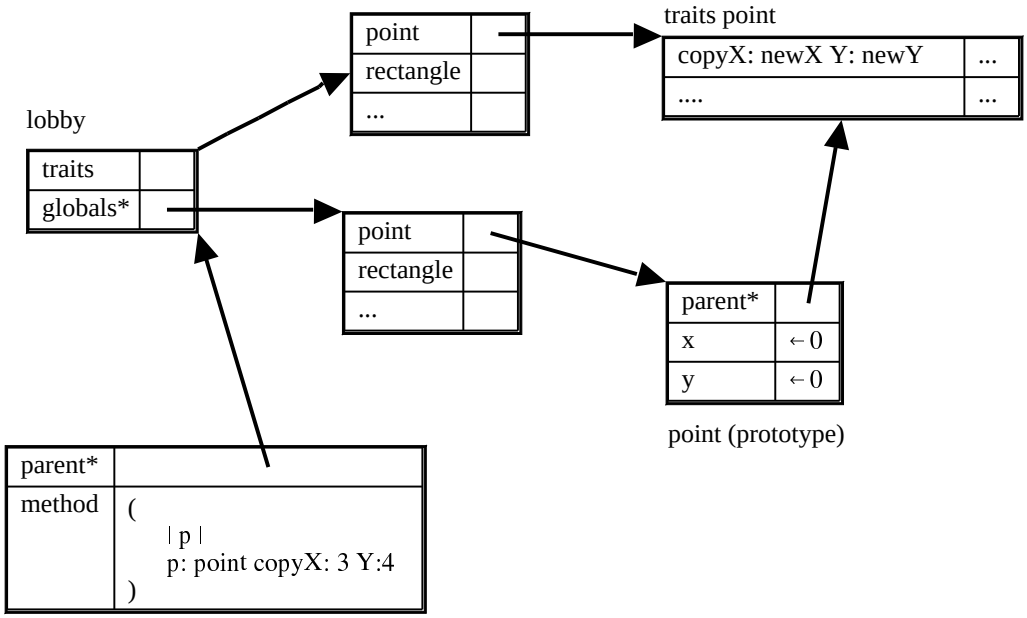
Klonování objektů je na úrovni implementace virtuálního stroje optimalizováno pomocí tzv. map. Ty definují společnou strukturu a konstantní data naklonovaných objektů, takže stejné nebo podobné objekty jsou paměťově reprezentovány velmi úsporně.

3.12 Jmenné prostory

Podobně jako Smalltalk-80 ani Self nemá jmenné prostory. V jeho případě je ale tato absence pouze formální, protože každý objekt si ve skutečnosti definuje svůj vlastní jmenný prostor. Objekty zasílají implicitně zprávy samy sobě. Prázdný objekt tedy nerozumí vůbec žádným zprávám.

Žádná přímá varianta globálních objektů tak, jak je zná Smalltalk, v Selfu neexistuje. Lze se bez nich jednoduše obejít pomocí delegace tak, že všechny objekty (nebo alespoň většina) budou přímo či nepřímo delegovat objekt, který bude obsahovat sloty s globálními objekty.

Předefinovat nějaký globální objekt lze pak jednoduše tím, že slot stejného jména umístíme tak,



Obrázek 3.11: Přístup k prototypům a rysům

aby byl při provádění algoritmu vyhledávání slotů nalezen dříve, než původní. Musíme to ale udělat tak, aby při prohledávání slotů nemohlo dojít k nejednoznečné interpretaci.

V Selfu lze (minimálně teoreticky) pomocí těchto jednoduchých prostředků udělat něco, co ve světě Smalltalku nemá obdobu – vytvořit modulární image a pracovat s více oddělenými uživateli. Vlastně se jedná o jistou obdobu chrootování známého ze světa Unixu.

Stačí, aby pro nový program byla vytvořena kopie globálních objektů, které ke své práci potřebuje, a zapojena do jeho objektové struktury. Takový program pak může pracovat zcela nezávisle na ostatních, ale přitom nad ním může mít supervizor stále kontrolu. Je výhodné, když programu nemusí být kopírovány všechny globální objekty, ale pouze moduly, které potřebuje a o které si sám řekne.

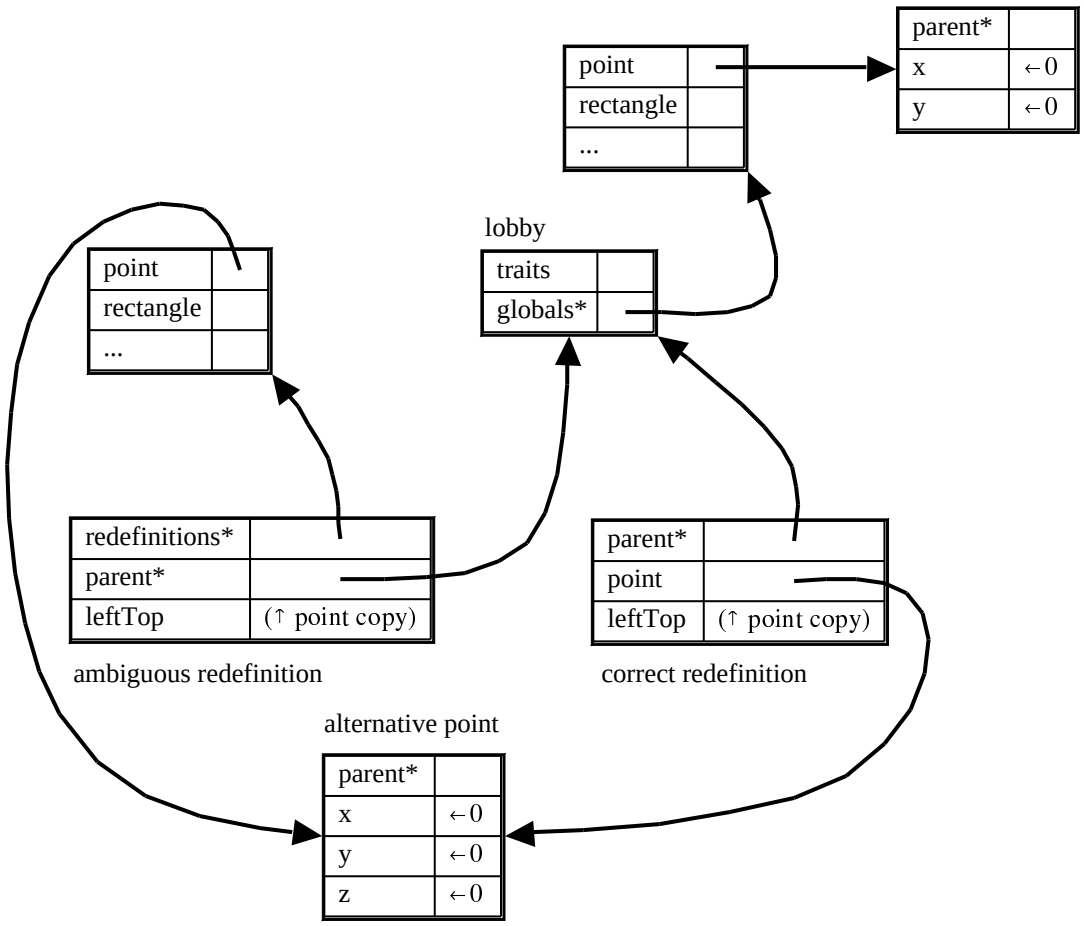
V případě, že program pracuje nekorektně, může si poškodit pouze vlastní objektovou strukturu, což ale na ostatní programy nebude mít žádný vliv.

3.13 Hierarchie rysů, násobná dědičnost a mixins

Při použití konceptu trait–prototyp se musíme umět vypořádat s problematikou třídní hierarchie. Pokud totiž vytváříme prototyp, jehož rodičem bude rys s prototypem s datovými sloty, musí tyto sloty obsahovat i vytvářený prototyp, aby jeho výsledné rozhraní bylo kompletní.

S velkými problémy se setkáme, pokud budeme chtít měnit strukturu prototypu a zároveň budeme vyžadovat, aby se provedené změny dotkly i objektů, které jsme od něj naklonovali, a objektů naklonovaných od odvozených prototypů. S velmi podobným problémem se vypořádává i samotný Smalltalk při změně struktury třídy, který ho dokáže efektivně řešit pouze s masivní podporou virtuálního stroje. V Selfu se tento problém v základní sémantice neřeší.

Selfovské objekty nejsou omezeny v počtu rodičovských slotů, takže pro ně nepředstavuje žádný problém kombinovat více sdílených rozhraní. Rovněž zde musíme u prototypů skloubit sloty

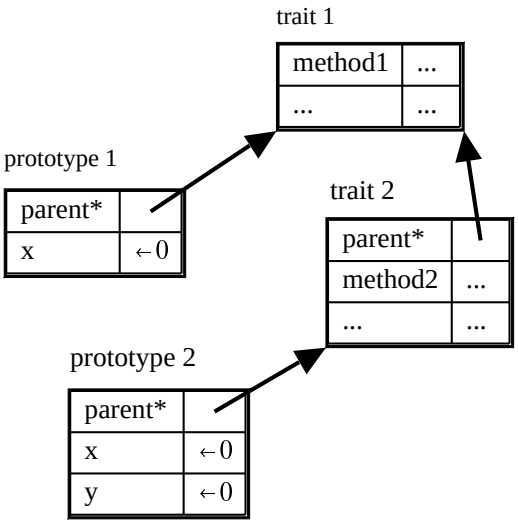


Obrázek 3.12: Nejednoznačné a jednoznačné předefinování globálního objektu

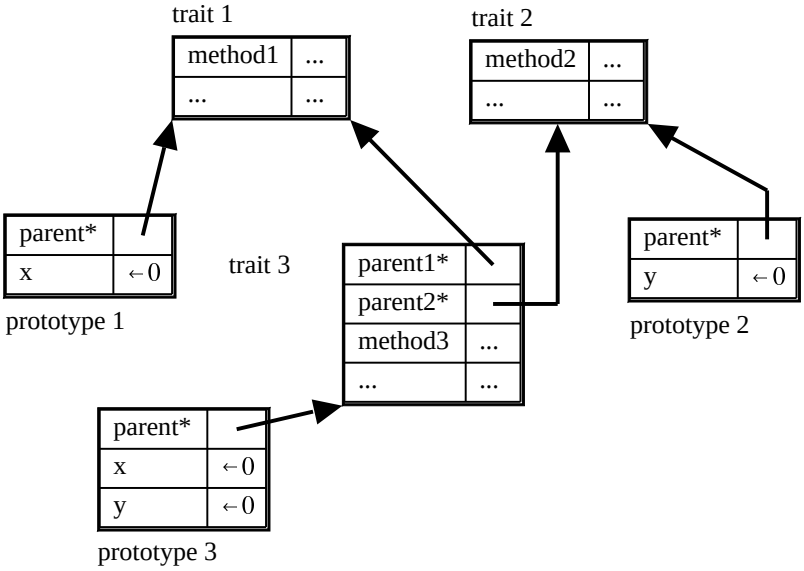
z více rodičovských prototypů, aby instance měly úplné rozhraní.

V praxi se snažíme násobné dědičnosti raději vyhnout. V objektově orientovaných jazycích se často nahrazuje čistším konceptem rozhraní. V Selfu se používají tzv. mixins.

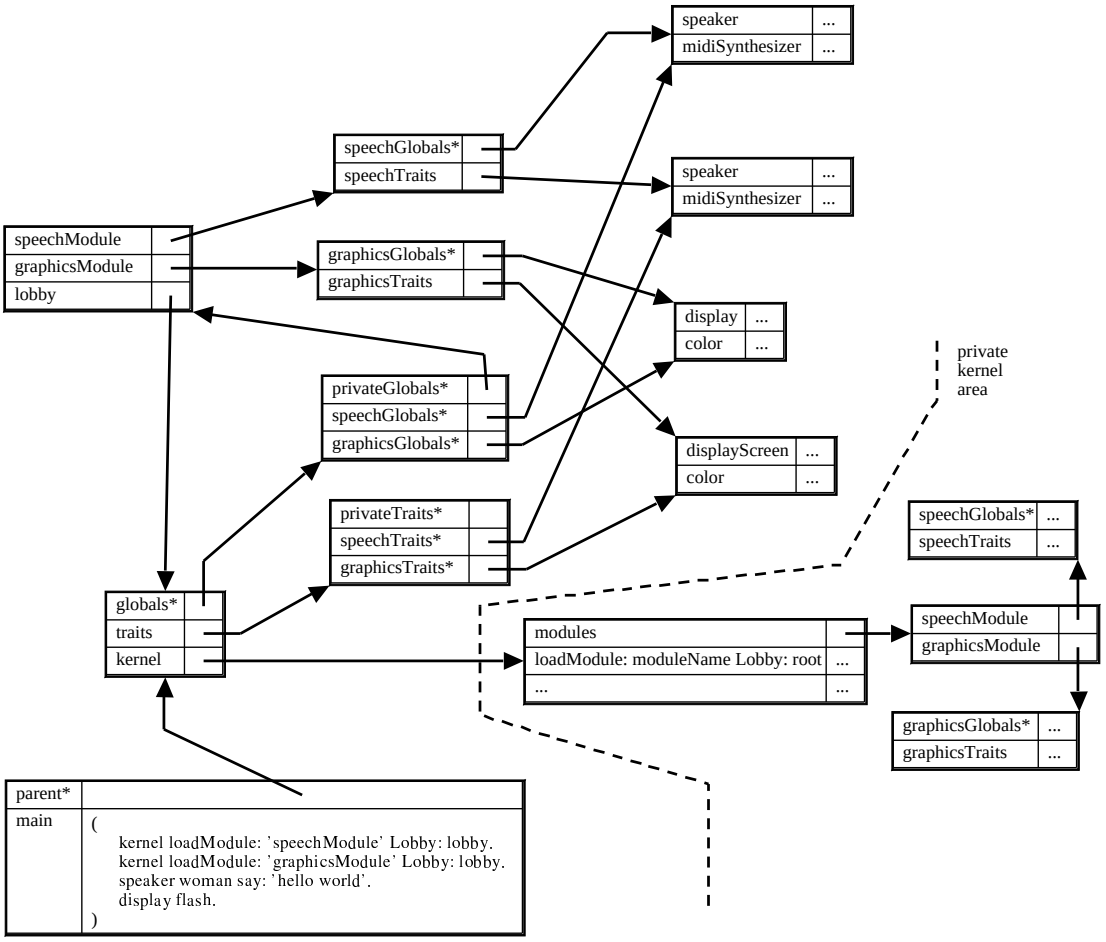
Mixins jsou velmi jednoduché objekty, které nemají žádné rodičovské ani datové sloty. Slouží jako nositele určitého typu chování, tedy jako množina několika málo metod. Mixins se pak mohou dodávat jako rodiče do různých rysů zároveň, čímž se zlepšuje obecnost a tvárnost vzniklé hierarchie.



Obrázek 3.13: Dědičnost a prototypy



Obrázek 3.14: Násobná dědičnost a prototypy



Obrázek 3.15: Chráněný modulární systém pomocí prototypů

Kapitola 4

Analýza současných přístupů

V této kapitole provedeme analýzu současných přístupů, které se používají k zavedení podpory prototypů do smalltalkovských prostředí.

4.1 Od základů přebudovaný systém

Koncepčně nejčistší varianta je začít od nuly a vybudovat nový systém, který spojuje ty nejlepší vlastnosti objektově orientovaných jazyků v jednom komplexním celku. Tento přístup má celou řadu výhod. V první řadě je velmi málo náchylný ke kompromisním řešením, které vždy vytvářejí limity snižující v dlouhodobém pohledu životaschopnost výsledku.

Další nesporná výhoda je otevřenost vůči novým nekonvenčním přístupům, které mohou otevřít zcela nové cesty vývoje tak, jak se to podařilo například samotnému Selfu.

Postup při vytváření nového sytému bývá následující:

1. analyzovat současné systémy a vytvořit nový jazyk integrující jejich nejlepší rysy
2. pokud výsledek navazuje na smalltalkovské systémy, pak vytvořit odpovídající virtuální stroj
3. vybudovat základní image
4. vytvořit cesty, jak pokud možno automaticky přenést softwarové vybavení ze stávajících systémů

Již existuje několik běžících projektů, které se o to snaží, ať už vycházejí ze světa Smalltalku nebo z jiných prostředí. Nových jazyků čerpajících ze Selfu je celá řada (OScheme, Proton, Plaid, Slate apod.). Některé z nich vypadají skutečně zajímavě a mohly by se ukázat jako životaschopné.

Obzvláště projekt Slate [9] je mezi programátory ve Smalltalku poměrně dobře známý. Oproti Selfu přináší některé nové prvky, jako je například PMD (Prototypes with Multiple Dispatch), typovou inferenci apod. V dohledné budoucnosti se však nedá očekávat, že by Slate nabrala na významu. Její virtuální stroj byl původně napsán v Common Lispu, nyní je již přepsán do jazyka C. Pokud by jako výchozí platforma byl zvolen Squeak, velmi pravděpodobně by se Slate dostalo ze strany programátorů ve Smalltalku větší pozornosti, než je tomu nyní.

Budovat samostatný nový systém a pak do něj portovat částečně nebo zcela Squeakovskou image je pro realizaci nejproblematičtější řešení s nejasnou budoucností. V současné době není zveřejněn žádný projekt, který by se o něco takového pokoušel. Programátoři jsou tak nuceni si vybrat mezi poměrně konzervativním Squeakem nebo se vzdát drtivé většiny jeho výhod, oželet obrovské množství kódu, které pro něj bylo vytvořeno, a začít pracovat s některým sice nadějným, ale v praxi problematicky použitelným programovacím jazykem.

4.2 Implementace prototypování na úrovni image

Smalltalk se ukazuje jako velmi flexibilní vývojové prostředí. Jedním z důsledků této jeho příjemné vlastnosti je fakt, že podpora prototypů jde implementovat ve funkční formě bez razantních zásahů do stávajícího systému. Díky tomu se objevily tři implementace, které se Squeaku začleňují podporu prototypování [4]. Jsou to:

1. System-Prototypes.2 (Hans-Martin Mosner, 1998)
2. System-Prototypes.7 (Russell Allen, 2000)
3. Prototypes.1 (Russell Allen, 2005)

Na následujících odstavcích si tyto verze postupně přiblížíme.

4.2.1 System-Prototypes

Hans-Martin Mosner (System-Prototypes.2)

O první implementaci podpory prototypování pro Squeak se zasloužil Hans-Martin Mosner. Nedalo se hovořit o plné podpoře, protože prototypy zde nepodporovaly delegaci. Sdílené chování se implementovalo pomocí klonování objektů.

```
person := HmmProtoObject new.
person addSlot: 'name'.

parent := person clone.
parent addSlot: 'children'.

dad := parent clone.
dad name: 'Daddy'.
```

Aby podobné objekty se stejným chováním neimplementovaly zcela stejné metody zbytečně víckrát, takové objekty používaly pro definování svého chování společné instance třídy nazvané `ProtoBehavior`. Znamenalo to, že v okamžiku, kdy došlo k aktu klonování objektu, sdílel naklonovaný objekt s originálem stejné metody.

Teprve v okamžiku, kdy naklonovaný objekt přidáním nebo odebráním slotu změnil svoji strukturu, byl pro něj vytvořen nový objekt definující jeho chování.

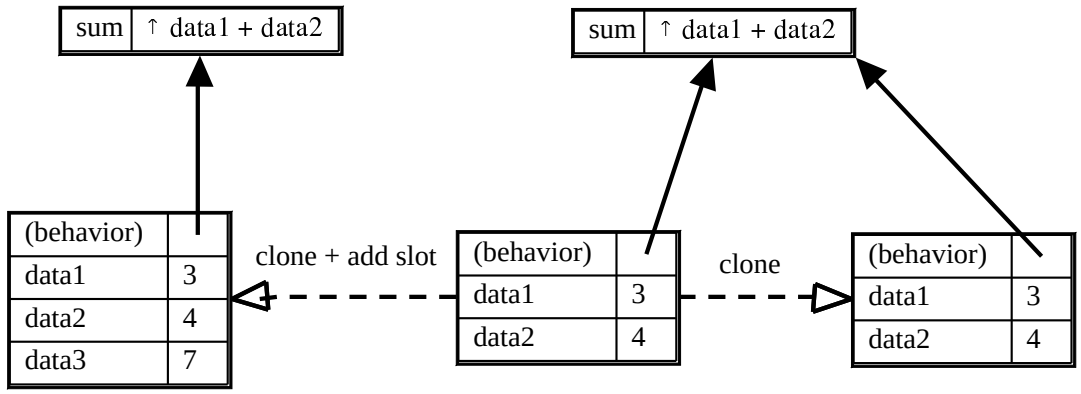
Mosnerova implementace již obsahovala vlastní modifikovaný inspektor přizpůsobený pro práci s prototypy. Další vlastnosti byly podobné jako u Allenovy verze, kterou si rozebereme podrobněji dále.

Russell Allen (System-Prototypes.7)

Na Mosnerův projekt později navázal Russell Allen, který stávající implementaci doplnil především o delegaci.

Každý prototyp je reprezentován dvěma objekty.:

1. instance třídy `PrototypeObject` (odvozená od třídy `Object`)
    - obsahuje kolekci datových slotů a množinu referencí na delegáty (rodičovské sloty)



Obrázek 4.1: Mosnerova implementace

2. instance třídy `PrototypeBehavior` (odvozená od třídy `Behavior`)
   - jedná se o objekty se sdíleným chováním
   - díky tomu, že svoje vlastnosti odvozuje od třídy `Behavior`, chová se podobně jako třída, to znamená, že obsahuje slovník implementovaných metod
   - prototypy se vytváří jako jejich instance, to znamená, že prototypy nejsou ve skutečnosti instancemi třídy `PrototypeObject`.
   - třída `PrototypeObject` je jim uměle dodána jako supertřída, takže prototypy rozumí zprávám, které jsou v ní implementovány

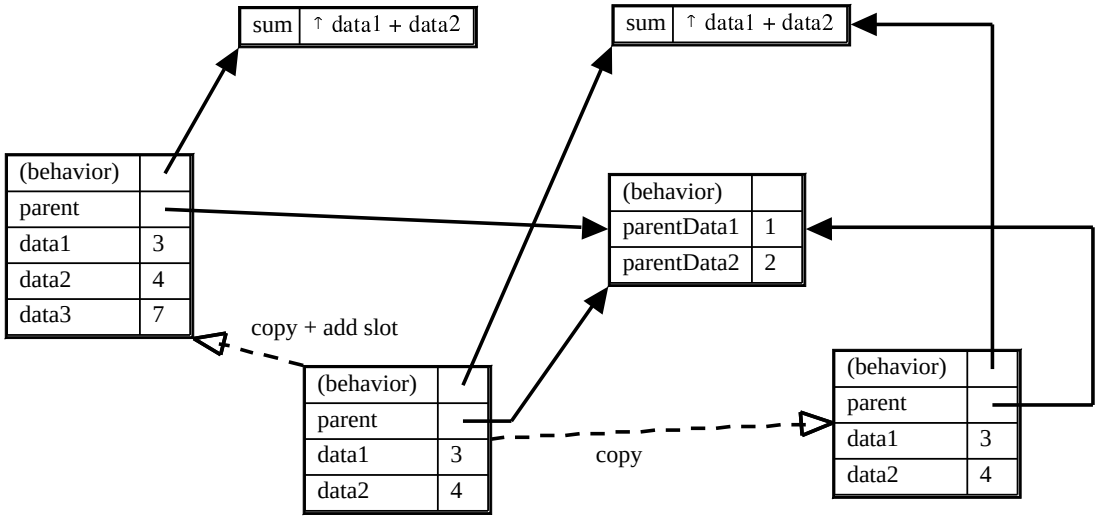
```
prototype := PrototypeObject new.
prototype class -> a descendent of PrototypeObject
prototype class class -> PrototypeBehavior
prototype class class superclass -> Behavior
prototype class superclass -> PrototypeObject
prototype class superclass superclass -> Object
```

Stejně jako u Mosnera, i v Allenově implementaci jsou objekty se sdíleným chováním (třída `PrototypeBehavior`) rozděleny na sdílené a soukromé, přičemž objekty se stejnou strukturou sdílí stejné chování.

Podobně jako v Selfu, i zde lze s prototypy komunikovat prostřednictvím zasílání zpráv. Prototoypové objekty nemají na rozdíl od svých smalltalkovských protějšků žádné instanční proměnné. Veškerý přístup k datům objektu je simulován pomocí automaticky generovaných metod, které přistupují ke kolekci slotů a na specifikovaný index ukládají data nebo tato data vrací. Proto také každý objekt s jinou strukturou slotů vyžaduje zvláštní instanci třídy `PrototypeBehavior`.

4.2.2 Prototypes

V dubnu roku 2005 vydal Russell Allen zcela novou značně přepracovanou verzi projektu SystemPrototypes. Z toho důvodu tuto verzi označil novým názvem. Bohužel pro účely této práce vyšel tento projekt příliš pozdě a do návrhu řešení podpory prototypů ve Squeaku, které je zde prezentováno, žádným způsobem nezasáhl.



Obrázek 4.2: Allenova implementace

Na druhou stranu je téměř jisté, že i při dřívějším vydání této verze by tuto práci žádným výrazným způsobem neovlivnil, protože implementace, pro kterou jsem se nakonec rozhodl, jde zcela odlišnou cestou a využívá jiné prostředky, byť za účelem dosáhnutí podobného cíle.

Projekt Prototypes svým způsobem zjednodušuje předchozí implementaci v tom smyslu, že nyní již k reprezentaci prototypu není potřeba mít dva objekty, ale stačí pouze jediný. K jeho vytvoření se sice využívá třída `Prototype`, ale instance této třídy vznikají podstatně komplikovanějším způsobem, než tomu bylo dříve.

1. vytvoří se instance třídy `Behavior`, která představuje rodiče specifikujícího chování prototypu
2. jako supertřída se tomuto rodiči určí přímo sama třída `Behavior`
3. vytvoří se prototyp jako instance rodiče
4. do prototypu se naklonují instanční proměnné rodiče, tedy supertřída, slovník metod a specifikace formátu objektu
5. rodič získá identitu prototypu
6. slovník metod prototypu se rozšíří o kopii slovníku metod třídy `Prototype`, taže prototyp může rozumět zprávám pro práci se sloty apod.

Výsledný objekt je tedy jakýsi hybrid mezi třídou a běžným objektem, vlastně objekt, který vniká jako instance sebe sama.

V mnoha ohledech došlo ke zefektivnění původní implementace. Například k uchovávání dat se využívají přímo instanční proměnné prototypu a nikoliv kolekce se sloty. Samozřejmě i zde představují zprávy jediný způsob, jakým objekt může komunikovat sám se sebou. Pro každý datový slot se vytváří dvojice metod. Jedna pro čtení a druhá pro zápis hodnoty do slotu prototypu.

Jistý ústupek byl podniknut v práci s delegací na více objektů. Protože nyní již objekty nepoužívají k delegaci protokol `doesNotUnderstand:` jako v předchozí verzi, ale pracují se standardní smalltalkovskou dědičností. Z toho důvodu je možné delegovat pouze na jediný další objekt. Jedná se o poměrně bolestivou daň efektivitě.

4.2.3 Metody

To, v čem si jsou všechny tři uvedené verze podobné, je práce s metodami. Ty se překládají ze zdrojových kódů na kompilované metody pomocí standardního squeakovského kompilátoru. Do Smalltalku ani jeho kompilátoru nebyly zaneseny žádné změny. To sebou přináší výhodu snadné integrace se Squeakem včetně možnosti využívat standardní debugger, ale na druhou stranu kód využívající prototypy trpí jedním zásadním nedostatkem výrazným způsobem snižujícím čitelnost vytvořených programů.

Ve Smalltalku jsou data objektu uchovávána ve formě instančních proměnných přístupných pomocí identifikátorů. Tyto instanční proměnné jsou ve výsledném bytekódu kompilovaných metod indexovány a přistupuje se k nim pouze prostřednictvím těchto indexů.

V prototypech nic jako instanční proměnné de facto neexistuje. Vnitřně se jich sice k uchovávání referencí na data stále využívá, na druhou stranu k nim ale prototypy nemají povolen běžnými prostředky přístup a ke svým datům musí přistupovat pouze prostřednictvím metod. To však ve Smalltalku znamená, že před každým voláním metody směřované na sebe musí objekt uvést pseudoproměnnou `self`. Její výskyty pak zabírají velkou část vytvořeného kódu.

4.2.4 Zhodnocení

Implementace prototypování pouze na úrovni image má mnoho výhod:

Snadnost implementace Doplnění image o prototypy je záležitostí pouhých několika málo poměrně jednoduchých tříd. Není nutné dělat žádné výrazné zásahy do zbytku systému a pokud ano, tak pouze ve smyslu rozšíření stávající implementace. Není potřeba žádnou část systému přepisovat, takže výsledná image si zachovává původní míru konzistence

Snadná nasaditelnost Rozšíření o prototypy lze snadno nahrát do jakékoliv existující image jako běžný program bez nutnosti dalších zásahů. Stále se využívá stejný nezměněný virtuální stroj.

Ladící prostředky Pro ladění programů je možné využít běžný squeakovský debugger, který není nutné nijak rozšiřovat.

Integrace se standardním Squeakem Vytvořené programy mohou využívat běžné smalltalkovské třídy a objekty a pracují s nimi zcela běžným způsobem včetně např. práce s bloky.

Na druhou stranu toto řešení není z celé řady důvodů úplně ideální a má několik nevýhod:

Neefektivní implementace V tomto ohledu udělala poslední verze poměrně velký krok kupředu. Dříve byl každý prototyp reprezentován celou řadou objektů, konkrétně prototypem, v některých případech samostatným sdíleným chováním, kolekcí slotů, slovníkem metod apod. V nejnovější verzi se podařilo značně omezit nároky na reprezentaci slotů, ale i tak nelze hodnotit paměťové nároky na jeden objekt jako právě malé. Každý, i prázdný prototyp, musí vytvářet celou řadu dalších objektů, jako například slovník metod.

Rychlost vykonávání kódu prototypů je sice v poslední verzi poměrně uspokojivá, ale bylo toho dosaženo především na úkor omezení delegace. Pokud potřebujeme využít násobnou

delegaci, musíme se spokojit s velice pomalou implementací pomocí `DNU` protokolu na úrovni image.

Vývojové nástroje Implementace Prototypes na praktické nasazení není zcela připravena. Pominneme-li některé chyby, zjistíme, že krom inspektoru navazujícího na selfovský outliner a jeho případného začlenění do debuggeru toho k dispozici mnoho nemáme. Neexistují nástroje pro export objektů a kódu mimo image ani pro jejich zpětné nahrání, nenalezneme žádnou podporu pro vyhledávání metod a podobných nástrojů, na které jsou programátoři ve Smalltalku zvyklí.

Rovněž práce se samotným inspektorem se nedá se Selfem srovnat. Pro vizualizaci referencí se nepoužívají žádné konektory, a tak při manipulaci s více objekty lze snadno ztratit přehled.

Kapitola 5

Volba řešení

V této kapitole provedeme diskusi dosavadních poznatků a zhodnotíme užitečnost a realizovatelnost jednotlivých variant.

5.1 Nový přebudovaný systém

Idea zahodit stávající implementace a pokusit se od základů vybudovat projekt znovu a lépe je vždy velmi lákavá. Bohužel je to jedna z hlavních příčin roztříštěnosti světa otevřeného softwaru a existence velkého množství nedokončených prací.

Z mnoha důvodů jsem se pro toto řešení nerozhodl. Hlavní dva důvody, proč je lepší se vyhnout tvorbě vlastního systému založeného na prototypování, jsou extrémní náročnost a zbytečnost takového projektu. Implementovat vlastní rychlý virtuální stroj s vyladěným garbage collectingem je netriviální úkol. Navíc pokud by se nepodařilo vytvořit jednoduchou automatickou cestu ke konverzi programů ze Squeaku, jen stěží by si takový systém hledal uživatele a velmi rychle by upadl. Mnohem zajímavější varianta je navázat na některý existující projekt vytvářející smalltalkovské prostředí založené na prototypech. Z nich nejdále pokročil projekt Slate.

5.2 Portace Squeaku do Selfu

Self splňuje skoro všechny koncepční požadavky na ideální programovací jazyk a systém. Teoreticky by pak stačilo vyvinout způsob, jak přenést squeakovský software do Selfu a tím vytvořit variantu Squeaku postavenou nad selfovským virtuálním strojem. Na úrovní image by největší rozdíl spočíval v přechodu programovacího jazyka ze Smalltalku na podobný Self.

Přechod Squeaku na selfovksou platformu ale bohužel není moc reálný. Proti němu stojí v první řadě velmi nejistá budoucnost Selfu, který se netěší u své mateřské firmy adekvátní podpoře. Tato platforma víceméně živoří na okraji zájmu a proměnila se v čistě akademickou záležitost. Navíc neexistuje její implementace pro operační systém Windows a virtuální stroj pro Linux je pro praktické použití příliš pomalý.

Nesmíme zapomínat ani na to, že samotným přechodem Squeaku na Self by se musel změnit způsob vytváření virtuálního stroje tak, aby jej bylo možné jej stejně jako ve stávajícím Squeaku vytvářet v jazyce použitém image. Vyjít ze squeakovského virtuálního stroje by bylo nesrovnatelně jednoduší než přizpůsobovat současný Self. Je mnohem praktičtější ohnout Squeak směrem k Selfu než se snažit portovat Squeak na Self. Pokračování ve vývoji Selfu a jeho vylepšování po vzoru Squeaku se bohužel zdá být již zbytečné.

5.3 Portace Squeaku do Slate

Slate se v mnoha ohledech od standardního Smalltalku liší více než Self. Automatická konverze programů ze Squeaku neexistuje a takřka jistě existovat nebude. Virtuální stroj Slate zatím nedosáhl takové úrovně, aby byl schopen Squeak hostovat. Není to tedy vhodný kandidát na systém, který by mohl rozšířit schopnosti Squeaku o prototypy. Jeho cesta vede k vybudování samostatného systému, který sice značnou měrou bude ze Squeaku čerpat části kódu, ale zcela jistě jej nikdy úplně nenahradí. Pravděpodobnější je scénář, podle kterého Squeak převezme některé prvky Slate a adoptuje je pro svoje potřeby. Oba systémy se budou vzájemně inspirovat a doplňovat. Varianta navázat na projekt Slate se nekryje se zadáním této diplomové práce.

5.4 Podpora prototypování v image

Navázat na současnou implementaci podpory prototypování pouze s využitím prostředků standardního Smalltalku je, alespoň na první pohled, nejjednodušší varianta. V době určování směru, kterým se tato práce měla ubírat, nebyla ještě k dispozici nejnovější verze programového balíku Prototypes. V té době dostupná implementace byla poměrně hodně neefektivní.

Při vylepšení stávající implementace jsou potřeba vyřešit následující úkoly:

- vytvořit kvalitní outliner
- vytvořit další nástroje pro beztřídní programování spojené s outlinerem (vyhledávání slotů v hierarchii objektů apod.)
- zlepšit efektivnost implementace
- vyřešit export a import kódu
- vyřešit nadměrnost používání pseudoproměnné `self`

Outliner

Outliner je hlavní vývojový nástroj v pro beztřídní programování. Vytvořit modifikovaný inspektor v balíku System-Prototypes sice bylo velmi jednoduché, ale o skutečné náhradě outlineru hovořit nelze.

Forma, v jaké byl použit v Selfu, se až na drobné výhrady dá považovat za velmi zdařilou, takže nad jeho přibližným vzhledem a funkčností by nebylo třeba vést žádný dodatečný výzkum. Velmi vážně byla zvažována varianta nesoustředit se na uživatelské rozhraní Morphic, ale cílit implementaci na projekt Croquet, což by vedlo v rámci této diplomové práce k vytvoření trojrozměrné varianty outlineru. Především při práci více programátorů na jednom projektu by se mohlo jednat o skutečně přínosné řešení.

Morphicovská verze outlineru může s výhodou využívat kvalitně zpracovaných konektorů. Vzhledem k tomu, že Morphic byl na Squeak portován právě ze Selfu, jsou oulinery pro Morphic přirozeným konceptem (mnohem přirozenějším, než systémová okna, která se ve Squeaku primárně používají pro vývojové nástroje), neměla by jejich nová implementace v tomto prostředí být náročná.

Navazující export a import objektů je možné řešit například pomocí XML, k čemuž Squeak nabízí dostatečné prostředky.

Efektivnost implementace

Tu se podařilo částečně zlepšit samotnému původnímu autorovi a to pomocí velmi nekonvenčního a odvážného přístupu. Nakonec zvolené řešení je sice efektivnější než Allenovo, ale dosahuje toho díky razantnějším zásahům. Minimálně jako demonstrace flexibility Smalltalku je však Allenova práce skutečně velmi zajímavá.

Pseudoproměnná self

Problém nadměrného používání pseudoproměnné `self` byl jedním z hlavních důvodů, proč nakonec varianta rozšíření stávající implementace nebyla zvolena. Tento problém je pro použitelnost Prototypes klíčový.

Tuto potíž bohužel nelze vyřešit jednoduchým zásahem do kompilátoru. Vyžaduje si to příliš razantní zásah do gramatiky Smalltalku. Vzhledem k tomu, že kompilátor je ve Squeaku hodně optimalizován a vytvořen metodou rekurzivního sestupu, náklady na jeho úpravu by se minimálně blížily tvorbě vlastního kompilátoru. Kromě kompilátoru by se podobným způsobem musel upravit i dekompilátor a debugger.

5.5 Portace Selfu nebo Slate do Squeaku

Velice zajímavá varianta je portovat Self nebo Slate do Squeaku jako celek. Podobným způsobem je do Squeaku zapouzdřeno již několik jazyků, jako je Lisp, Scheme, Prolog apod. Důležité je vhodně určit míru separace hostitelského Squeaku od vnořeného systému.

Například pro Slate hrál hostitelský Common Lisp pouze úlohu dočasného virtuálního stroje. Podobně by i Squeak mohl hrát úlohu virtuálního stroje, který by využíval implementovaný garbage collecting, objektovou paměť a další již stabilní a otestované části Squeaku.

Velkou výhodou tohoto řešení je velká obecnost, malá náchylnost k omezujícím kompromisům a volné ruce při implementaci.

Nevýhodou je pak implementační náročnost a malá efektivnost výsledku. Například v případě Selfu by alespoň v počátečních neoptimalizovaných verzích měla být jeho implementace vytvořena s důsledným uplatněním všech principů, které Self využívá, jako je tvorba aktivačních objektů metod a bloků apod.

Výsledný projekt by pak sice měl obrovskou pedagogickou a demonstrační hodnotu, ale dosahoval by malého výkonu a v praxi by nebyl příliš použitelný.

Obrovský problém by například přestavovaly bloky a procesy, které jsou ve Squeaku značnou měrou optimalizovány. Tyto optimalizační postupy by se na vnořený beztřídní jazyk nedaly aplikovat bez silné a neefektivní mezivrstvy na úrovni image Smalltalku.

5.6 Zvolené řešení

Jak jsme si v minulé kapitole ukázali, variant, jimiž lze rozšířit Squeak o beztřídní programování na bázi prototypů, je celá řada. Ani jedno z nich není ideální ať již z koncepčních, implementačních nebo praktických důvodů.

Implementačně nejjednodušší je varianta rozšíření a vylepšení stávající implementace balíku Prototypes s použitím běžných prostředků smalltalkovské image. Náročnost tohoto úkolu ovšem dramaticky vzroste, pokud si jako cíl vytyčíme i odstranění přebytečných výskytů pseudoproměnné `self`.

Pedagogicky nejzajímavější řešení je naopak implementovat Self nebo Slate v prostředí Squeaku. Jako cesta k důslednému pochopení těchto jazyků a tvorby objektově orientovaných systémů obecně je tento přístup nepřekonatelný.

5.6.1 Obecná charakteristika

Po důkladném zvážení všech možných variant jsem se nakonec rozhodl pro vytvoření kompromisního řešení, které více či méně čerpá ze všech výše uvedených konceptů.

Jeho výchozím bodem je myšlenka, že pokud budeme chtít stávající implementaci prototypů na úrovni image zbavit nadměrného používání pseudoproměnné `self`, v podstatě se nevyhneme tvorbě vlastního kompilátoru. Když už jsme nuceni vytvořit si vlastní kompilátor Smalltalku s upravenou gramatikou, není těžké Smalltalk více modifikovat tak, aby lépe reflektoval nové požadavky, které na něj s přechodem na beztřídní systém budou kladeny. To v podstatě znamená jej přeformovat na Self.

V okamžiku, kdy vytváříme vlastně implementaci Selfu nad Squeakem, je z praktických důvodů velmi vhodné výsledek překladu neinterpretovat na úrovni image, ale zdrojové kódy kompilovat přímo do standardního squeakovského bytekódu a tím plně využít interpret virtuálního stroje.

Závěrečný krok je postaven na myšlence, že pokud již využíváme k interpretaci selfovského kódu přímo virtuální stroj Squeaku, můžeme tento virtuální stroj modifikovat tak, aby podporoval prototypy a práci s nimi.

To, jakým způsobem byly tyto myšlenky konkrétně realizovány a jaká rozhodnutí a důsledky to sebou přineslo, tvoří podstatu této práce.

Kapitola 6

Popis jazyka

Programovací jazyk Smalltak nebyl navržen jako beztřídní jazyk a proto je jeho použití v systémech rozšiřujících Squeak o podporu prototypů poněkud neohrabané a výsledný kód málo přehledný.

Zajímavé je, že ač se jedná o čistý objektově orientovaný programovací jazyk, nelze v něm v podstatě napsat literál pro vytvoření objektu, pro nějž již neexistuje jednoduchý literál (číslo, řetězec, pole apod.). Objekt lze vytvořit pouze na základě předpisu, na jehož základě se vytvoří objekt určité třídy a patřičně se inicializuje. To samozřejmě není literál, ale přímo zápis výpočetní sekvence, kterou vytvářený objekt může, ale také nemusí akceptovat. Jen stěží si lze představit, jak by vypadal například Lisp, pokud by neumožňoval zapisovat literálem seznamy.

To, že Smalltak nemá zápis objektů pomocí literálů, je částečně důsledkem toho, že objekt sám o sobě není nositelem svého chování. K tomuto účelu používá třídu. Pro tu ovšem ve Smalltalku neexistuje žádný pevně definovaný literální zápis. Proto textová forma smalltalkovských zdrojových kódů není přímou součástí specifikace jazyka a mimo zažitého zápisů s pomocí vykřičníků jako oddělovačů používají některé implementace pro přenos kódu XML.

V případě Smalltalku tento přístup nepředstavuje vážný handicap. Svým způsobem mu to dodává na čistotě, protože objekt je úplně zapouzdřená entita i pro svého tvůrce. Ten může maximálně určit, k jaké třídě bude vytvářený objekt náležet, ale jeho další obsah musí diskutovat přímo s ním.

U beztřídního programování by šel stejný princip použít také a současné implementace prototypů ve Squeaku jsou nuceny s objekty takto pracovat. Bohužel to dále snižuje jejich použitelnost, protože místo toho, abyste vytvořili objekt tak, že pomocí nějakých jednoduchých výrazových prostředků popíšete jeho strukturu, musíte napsat program, který ho postupně zkonstruuje. Jedná se o podobnou situaci jako při zavedení literálů pro výrazová pole do Squeaku.

Tím, že Self použil jednoduchou unifikovanou strukturu objektů tvořených sloty, zbavil objekty přímé závislosti na dalších objektech, čímž si výrazně usnadnil cestu k jejich zápisu pomocí literálů, což mu zpětně usnadňuje přenos objektů v textové formě.

Nutnost používat nadměrně pseudoslot `self` a nemožnost popisovat objekty pomocí literálů jsou dva hlavní důvody, proč je při zavádění beztřídního programování do prostředí Squeak vhodné upustit od programovacího jazyka Smalltalk.

Na druhou stranu se pro tento účel moc nehodí ani samotný Self. Aby mohly beztřídní objekty komunikovat s běžnými squeakovskými objekty, potřebují minimálně používat stejná pravidla pro jména selektorů zpráv. Ty se ale ve Smalltalku a v Selfu liší. Rovněž některé další charakteristiky Selfu a Smalltalku, jako je třeba implicitní vracení hodnot z metod, se výrazně liší. To by zcela jistě nebylo v situacích, kdy programátor potřebuje zároveň využívat jak Smalltalk, tak beztřídní programování, příliš vhodné a vedlo by to ke zbytečným chybám a zmatkům.

Z těchto důvodů jsem se rozhodl pro podporu beztřídního programování ve Squeaku vytvořit jazyk nový, který vhodným způsobem kombinuje vlastnosti Smalltalku a Selfu tak, aby jeho pou-

žívání bylo pro současné programátory ve Squeaku co nejpřirozenější a zároveň aby mohli snadno využívat vymoženosti, která do objektového programování přinesl Self.

Jedná se tedy o kompromisní řešení, které bylo zvoleno z čistě praktických důvodů. Mnoho změn, které do svého návrhu zahrnul Self a které Smalltalk svým způsobem vylepšují, bylo z nově navrhovaného jazyka vypuštěno právě kvůli interoperabilitě se Squeakem. V tomto ohledu se jedná o krok zpět, ale jsem přesvědčen, že v tomto případě se nejedná o nijak vážný problém, protože vyloučené změny jsou pouze kosmetického charakteru.

Vytvořený jazyk tvoří mezistupeň mezi Smalltalkem a Selfem a jako takový by bylo chybné označovat jej jedním či druhým jménem. Proto (společně s množinou modifikací squeakovského virtuálního stroje) dostal označení nové – Marvin.

V této kapitole budou diskutovány změny návrhu tohoto nově navrženého jazyka oproti původnímu Smalltalku a Selfu, takže zároveň se jedná o porovnání těchto výchozích jazyků po lexikální a syntaktické stránce. Proto nebylo uvedeno přímo v kapitole věnující se popisu Selfu.

6.1 Lexikální přehled

V lexikální analýze se Marvin snaží v maximální možné míře podobat squeakovskému dialektu Smalltalku, aby byla zachována maximální interoperabilita obou prostředí. Nicméně i od něj se v detailech odlišuje.

6.1.1 Identifikátory

Squeak Identifikátory ve Smalltalku vždy začínají písmenem a pokračují písmenem, číslicí nebo podtržítkem. Od verze 3.8 je možné v identifikátorech používat i písmena s diakritikou. Identifikátory mohou začínat velkými písmeny.

Self Na rozdíl od Smalltalku mohou v Selfu identifikátory začínat i podtržítkem. Takto zvolené identifikátory jsou vyhrazeny pro pojmenování primitivních metod. Národní znaky v identifikátorech nejsou povoleny.

Marvin Podobně jako Self, i Marvin umožňuje na začátku identifikátorů uvádět podtržítko. Umožňuje mu to absence přiřazovacího příkazu, pro nějž se ve Smalltalku používá právě podtržítko. Stejně jako v Selfu jsou tyto identifikátory vyhrazeny pro primitivní metody. Jejich přítomnost je ale pouze formální, protože v současné implementaci se nepředpokládá, že by z Marvina byly přímo volány primitivy a v tomto ohledu se plně spoléhá na přeposílání zpráv na běžné squeakovské objekty nebo na jejich přímé využití.

Kvůli interoperabilitě Squeaku a Marvina je velmi nevhodné tyto identifikátory běžně používat.

Na začátku identifikátorů je povoleno používat i velká písmena.

Marvin neumožňuje v identifikátorech používat národní znaky.

6.1.2 Unární, binární a slovní selektory

Squeak Pro unární selektory se používají přímo identifikátory.

Binární selektory mohou používat i znaky z horní části ASCII tabulky, přičemž se používá kódování MacRoman. Množina přípustných znaků je omezena pouze vyloučením nepřípustných kolizí a není nikde v literatuře přesně specifikována. Nicméně v praxi se používá pouze omezený rozsah znaků, který je víceméně shodný se selfovskými operátory.

Velikost písmen slovních selektorů není nijak omezena. Lze v nich využívat národní znaky.

Self Pro unární selektory se používají identifikátory. Jména unárních slotů ale nesmí začínat na podtržítko, které je vyhrazeno pro primitivní zprávy.

Binární selektory jsou tvořeny jedním či více speciálními znaky (+− <>? apod.). Ty jsou tvořeny znaky se základního rozsahu ASCII tabulky.

První část slovního selektoru začíná na malé písmeno. Každá další část slovního selektoru pak musí začínat na písmeno velké. To zlepšuje přehlednost výsledných programů.

Marvin Pro unární selektory se používají přímo identifikátory. Nelze v nich tedy používat národní znaky.

Pro binární selektory se používá přesně vymezený rozsah znaků vycházející ze Selfu.

Velikost písmen slovních selektorů není nijak omezena. Nelze v nich využívat národní znaky.

6.1.3 Čísla

Self Rozsah literálů pro celá čísla je omezen na velikost SmallInt. Na rozdíl od Smalltalku lze pro oddělovač základu (`r`) i oddělovač exponentu (`e`) používat malá i velká písmena.

Minimální velikost základu jsou 2, maximální velikost základu je 36.

Nemá literál pro decimální čísla s definovanou přesností.

Squeak Používá několik typů literálů pro zápis čísel. Umožňuje zapsat jak celá čísla (`42`) tak čísla reálná (`3.14`). Dále umožňuje zapisovat čísla s exponentem (`2e10`) a čísla o libovolném základě (`16rFF`). V nových verzích lze literálem zapsat například i decimální čísla s definovanou přesností (`3.14s2`).

Rozsah čísel zapisovaných literálem není nijak omezen.

Minimální velikost základu je 2, maximální velikost základu není omezena.

Marvin Literály pro čísla mají shodnou charakteristiku jako ve Squeaku, pouze současná verze neumožňuje zapisovat literály s definovanou přesností.

6.1.4 Řetězce

Squeak Řetězce se vkládají do apostrofů. Jejich obsah přesně odpovídá zápisu ve zdrojovém kódu včetně tabelátorů a odřádkování. Uvozovky se do řetězců vkládají zdvojením tohoto znaku.

Self Pro zápis řetězců používá zápis převzatý z jazyka C. Tzn. že se nové řádky zapisují pomocí zpětného lomítka náledovaného znakem `n` apod.

Marvin Zápis řetězcových literálů se shoduje se Squeakem.

6.1.5 Znaky

Squeak Literál pro znak se zapisuje ve Smalltalku jako znak dolaru následovaný libovolným (i bílým) znakem.

Self Znakové literály nezná. Nijak nerozlišuje mezi jednoznakovým řetězcem a znakem.

Marvin Používá znakové literály ve stejné formě jako Squeak. Proto narozdíl od Selfu z množiny znaků přípustných pro binární selektory vylučuje znak dolaru.

6.1.6 Pseudoproměnné

Jedná se o obdobu klíčových slov z jiných programovacích jazyků. V případě Selfu a Marvina se hodí spíše označení pseudosloty.

Squeak Používá konstanty `nil`, `true` a `false`. Jsou to reference na jedinečné instance (singletony) tříd `UndefinedObject`, `True` a `False`.

Dále používá pseudoproměnné `self` (referencuje příjemce zprávy) a `super` (referencuje příjemce zprávy v kontextu předka).

Pseudoproměnná `thisContext` referencuje aktivní kontext, ve kterém je kód vykonáván.

Self Používá pouze pseudoslot `self`, který referencuje příjemce zprávy.

Pro referencování předka v kontextu rodiče používá klíčové slovo `resend` oddělené od zasílané zprávy tečkou.

Marvin Používá pseudosloty `self` (referencuje příjemce zprávy) a `resend` (referencuje příjemce zprávy v kontextu předka, není oddělen od zasílané zprávy tečkou).

Současný návrh jazyka nemá možnost referencovat aktivní kontext, začlenění pseudoslotu `thisContext` je plánováno do dalších verzí.

6.1.7 Symboly

Squeak Symboly jsou identifikátory nebo řetězce uvozené znakem mřížky.

Self Symboly nemá. Místo nich používá řetězce, přičemž stejné řetězce považuje za totožné objekty, což je přístup, který Smalltalk používá právě u symbolů.

Marvin Kvůli interoperabilitě se Squeakem symboly zachoval. Z tohoto důvodu není na rozdíl od Selfu znak mřížky povoleným znakem v binárních selektorech.

6.1.8 Komentáře

Squeak Smalltalk uzavírá komentáře do uvozovek. Zakomentovaný kód je překladačem ignorován a nahrazen jedním bílým znakem. Uvozovky se mohou vyskytovat v zápisu literálů pro řetězce. Smalltalk nemá jednořádkové komentáře.

Self Komentáře zpracovává stejně jako Smalltalk.

Marvin Umožňuje stejný přístup jako Squeak a Self. Navíc doplňuje lexikální analyzátor o možnost zpracovávat jednořádkové komentáře. Ty se zapisují jako dvě bezprostředně za sebou následující uvozovky. Zbytek řádku je pak ignorován.

Nemožnost zapisovat jednořádkové komentáře je ve Smalltalku i Selfu poměrně nepříjemná a je jednou z příčin, proč programátoři v těchto jazycích používají komentáře poměrně málo.

Marvinovo rozšíření přitom neznamená žádný zásadní zásah jazyka. Například ve standardní image Squeaku, což představuje přes půl milionu řádků kódů, se nenachází jediná situace, která by při zavedení jednořádkových komentářů ve formě, jakou používá Marvin, znamenala chybnou interpretaci stávajícího kódu.

6.2 Syntaktický přehled

Pakliže po lexikální stránce jsou si si Smalltalk, Self i Marvin téměř rovnocenné, v syntaxi lze zaznamenat rozdílů podstatně více. Při tom lze ve zkratce konstatovat, že Marvin je Smalltalk obohacený o zápis objektových literálů po vzoru Selfu a s možností používat implicitního příjemce zpráv.

Práce s objekty v Marvinovi je oproti Selfu více omezená a to v případech, kdy to bylo vhodné kvůli napojení Marvina na Squeakovský virtuální stroj. Ve skutečnosti není praktických omezení Marvina oproti Selfu mnoho, protože celou řadu omezení přidává Self až na úrovni implementace a v gramatice jazyka nejsou patrné. Na tyto případy během výkladu upozorníme. Ve výsledku je tak ve vyjadřovacích schopnostech pro programátora Marvin stejně bohatý jako Self a v některých konstrukcích jej i dokonce překonává.

6.2.1 Literály

Literály slouží k zápisu objektů, které konstruuje při překladu samotný překladač a které se jsou referencovány přímo kompilovanými metodami. Mohou to být jak velmi jednoduché objekty, jako jsou čísla, nebo i velmi složité objekty s definovaným chováním.

Základní literály jsou čísla, řetězce, znaky a symboly. Jejich podoba byla upřesněna v lexikálním přehledu.

Podobně jako Squeak, i Marvin umožňuje zapisovat pole literálů, tedy objekty tvořené literály. Pole literálů mohou být zanořená. Self k tomuto účelu používá přímo literály pro zápis objektů.

Squeak obsahuje proti Smalltaku-80 rozšíření spočívající v zavedení výrazových polí, tedy polí, jejichž obsah se tvoří až při vyhodnocení. V programátorské praxi se jedná o velmi užitečnou a vítanou konstrukci. Výrazová pole jsou optimalizovaně vytvářena přímo v bytekódu a virtuální stroj Squeaku obsahuje primitivy, které konstukci těchto polí urychlují.

Marvin v současné verzi výrazová pole nepodporuje. Do budoucích verzí se s nimi ovšem v návrhu jazyka počítá. Jsou jedním z důvodů, proč Marvin na rozdíl od Selfu nepodporuje poznámkové sloty, které ke svému zápisu využívají stejně jako výrazová pole složenou závorku.

Marvin samozřejmě umožňuje po vzoru Selfu zapisovat objekty i pomocí speciálních literálů. Ty budou diskutovány později.

6.2.2 Zprávy

Smalltalk i Self používají unární, binární a slovní zprávy. V některých aspektech jejich zápisu se ale liší. V první řadě se jedná o pravidla pro zápis selektorů, které byly diskutovány v lexikálním přehledu, tedy velikosti písmen a množiny použitelných znaků.

Další rozdíl spočívá v tom, že Self neumožňuje dělat sekvenci po sobě jdoucích rozdílných binárních selektorů. Na stejné úrovni mohou být pouze stejné binární selektory. Tím si přímo vynucuje po programátorovi, aby výrazy s různými operátory, u nichž se také předpokládají různé priority, explicitně závorkoval.

Ve Smalltalku přípustný výraz `1+2*3`, který vrací výsledek `9`, je v Selfu chybný. Musí se použít závorky, tedy `(1+2)*3`. Naproti tomu výraz `1+2+3` je v obou jazycích správný a vrací výsledek `6`. Self se takto snaží omezit zbytečné omyly ze strany programátorů. Závorkovaný zápis je součástí doporučované “štábní kultury” i ve Smalltalku a pro začátečníky je podstatně méně matoucí.

Podobný účel jako explicitní závorkování má i Selfem vynucovaný tvar unárních a slovních selektorů. Přispívají k omezení mylné interpretace výsledného kódu, protože je pak lépe vidět, kde začíná jeden selektor a končí druhý.

Například výraz `true ifTrue: [true] whileTrue: [halt]` je interpretován jako `true ifTrue: ([true] whileTrue: [halt])`. Smalltalk by se v tomto případě snažil provést zprávu se selektorem `ifTrue:whileTrue:`.

Marvin v obou případech volí smalltalkovský přístup. U explicitního závorkování binárních zpráv může teoreticky dojít v budoucích verzích ke změně na Selfovskou sémantiku, nicméně konvence pojmenování selektorů musí kvůli kompatibilitě s hostitelským prostředím zůstat nezměněny.

Oproti Smalltalku podporují Self i Marvin zprávy zasílané implicitnímu příjemci, to znamená, že se na začátku volání zprávy, kterou objekt posílá sám sobě, nemusí uvádět pseudoproměnná `self`. To ale neznamená, že by se jí tyto jazyky zcela zbavily.

Self používá pseudoslot `self` v případech, kdy je potřeba uložit na vrchol zásobníku příjemce, tedy například jako poslední výraz v metodě. Také se používá při zasílání zpráv objektu, pokud si z nějakého důvodu nepřejeme, aby jej nejdříve zpracoval aktivační objekt metody. Protože Marvin aktivační objekty nepoužívá, druhý důvod u něj nedává smysl.

6.2.3 Přeposílání zpráv

K přeposílání zpráv v příjemci v kontextu předka používá Smalltalk pseudoproměnnou `super`.

Self k tomuto účelu používá konstrukci, kdy se napíše klíčové slovo `resend` následované tečkou a selektorem přeposílané zprávy. Tečkou uprostřed zdůrazňuje, že se nejedná o přístup ke slotu, ale speciální režim zasílání následující zprávy.

Marvin používá pseudoslot `resend`, to znamená, že syntaxe je kombinací Smalltalku a Selfu. Na rozdíl od Selfu neodděluje klíčové slovo `resend` tečkou, což mu pomáhá udržet si jednoznačnou gramatiku (tečka se používá jako oddělovač výrazů).

Oproti smalltalkovskému `super` ale nelze `resend` v Marvinovi zapsat samostatně. Ve Smalltalku vloží samostatně použitý `super` na vrchol zásobníku totéž co pseudoproměnná `self`. V Marvinovi nelze klíčové slovo `resend` použít samostatně a musí za ním vždy následovat zaslání zprávy.

6.2.4 Sekvence

Sekvence je syntaktická zkratka použitá ve Smalltalku pro posílání několika zpráv postupně stejnému objektu. Tyto zprávy se od sebe oddělují středníkem. Tato konstrukce pomáhá zkrátit výsledný zápis programu.

Self ani Marvin tuto konstrukci nepodporují, nicméně její budoucí začlenění do gramatiky jazyka Marvin nelze vyloučit.

6.2.5 Návratové hodnoty

V tom, jaké implicitní hodnoty bloků a metod Self a Smalltalk vrací, se značně odlišují. Ve Smalltalku je implicitní hodnota metody vždy příjemce zprávy (`self`). Pokud objekt chce vrátit jinou hodnotu, musí tak učinit pomocí příkazu návratu hodnoty z funkce (znak stříšky).

U bloků je situace o něco komplikovanější. Pokud je blok prázdný, vrací se implicitně nedefinovaná hodnota (`nil`). Pokud blok obsahuje jeden nebo více výrazů, je vrácena hodnota vždy posledního z nich.

Příkaz návratu hodnoty z funkce smí být umístěn pouze na konci funkce nebo bloku. Nikdy za ním nesmí následovat další příkazy.

Oproti tomu Self u bloků i metod vrací implicitně výsledek posledního výrazu (tedy jako u bloků ve Smalltalku). Snaží se tak tyto dvě syntaktické konstrukce unifikovat. Prázdný blok vrací prázdný objekt, který neobsahuje žádné sloty.

Zajímavá je situace u metod. Pokud unární zpráva má prázdný obsah, je její zápis shodný s datovým slotem určeným pro čtení, který referencuje prázdný objekt. Sémantika je tedy shodná s prázdnými bloky. Naproti tomu pokud metoda přijímá alespoň jeden argument, což je případ binárních a slovních zpráv, nesmí být prázdná.

Marvin kvůli kompatibilitě používá smalltalkovskou sémantiku, tzn. že prázdné metody vrací příjemce (`self`) a bloky vrací výsledek posledního výrazu nebo nedefinovaný objekt (`nil`), pokud jsou prázdné.

Toto je jeden z nejproblematičtějších rysů jazyka Marvin. Ten totiž neobsahuje pseudoslot pro nedefinovanou hodnotu `nil` a tedy prázdný objekt žádný takový slot nezná. Vracet ho jako výsledek prázdného bloku tedy není právě ideální, protože není jasné, jak se k této hodnotě dospělo. Bylo by samozřejmě možné, aby místo přímého vrácení nedefinovaného objektu blok zaslal příjemci zprávu `nil` a získal tak požadovanou hodnotu. Zde by se ale setkal s problémy, pokud by objekt nebo některý z objektů, na který deleguje svoje zprávy, žádný slot neměl. Vyvolávat v takovém případě výjimky by nebylo vhodné.

Nejlogičtější varianta je vracet u prázdných bloků prázdný objekt tak, jako to dělá Self. Na druhou stranu Marvin potřebuje zachovat kompatibilitu chování bloků s hostitelským prostředím Squeaku. Pro použití `nil` také hovoří to, že se jedná o implicitní objekt, na který jsou v Marvinovi i v Selfu nastavovány hodnoty (prázdných) slotů, a v tomto případě se také jedná o hodnotu bez přímé vazby na konstruovaný objekt.

6.2.6 Sloty

Sloty jsou základní stavební prvek, ze kterého se v Marvinovi a Selfu tvoří objekty. Ve Smalltalku nemají přímou obdobu.

Datové sloty

Datové sloty se v Marvinovi zapisují podobně jako v Selfu. Tvoří je identifikátor zapsaný podle marvinovských pravidel následovaný rovnítkem (`=`) pro sloty určené pouze pro čtení nebo šipkou doleva ($<-$) pro sloty umožňující čtení i zápis. Za přiřazovacím znaménkem následuje výraz ohraničený do hranatých závorek.

V tomto se Marvin od Selfu odlišuje, protože Self u hodnot slotů žádné uvozující závorky nevyžaduje. Vyhýbá se tak nepříjemným konfliktům v gramatice. Na rozdíl od Selfu umožňuje Marvin vytvářet i prázdné metody, což je výhodné v situaci, kdy například potřebujeme mít v objektu existující metodu, ale doposud jsme nenapsali její obsah. Bez ohraničujících závorek by ale byl literál pro prázdnou metodu a literál pro prázdný objekt přiřazený do datového slotu naprosto shodný. V Selfu i Marvinovi jsou výrazy přiřazené do slotů vyhodnocovány již v době překladu. To znamená že, pokud by slot vypadal například jako `a = [self]`, nebude slot pojmenovaný `a` obsahovat referenci na objekt, jehož je slot součástí, jak by se mohlo na první pohled zdát, ale na slot, v jehož kontextu byl proveden překlad výrazu, jimž bývá standardně objekt lobby. Marvinova syntaxe se snaží tento odlišný charakter hodnot slotů zvýraznit pomocí hranatých závorek.

U hranatých závorek vyvstává otázka, zda se nebudou příliš plést z bloky. Bloky samozřejmě mohou být uváděny jako hodnoty datových slotů, pokud jsou uvedeny v ohraničujících závorkách.

```
(| data = [[|:a| a inspect]] |) value: 1
```

V tomto aspektu dosahuje Marvin lepších výsledků než Self, protože ten sice umožňuje do slotů vkládat bloky, ale v okamžiku, kdy je proveden pokus nějaký z nich vyhodnotit, Self ohlásí chybu konstatující, že jeho lexikální rodič (uzavírající aktivační metoda) již vypršel a blok nemůže být proto vyhodnocen. Jsou tedy pro praxi nepoužitelné.

Rodičovské sloty

Pro zápis rodičovských slotů platí podobná pravidla jako pro datové sloty. Od těch se odlišují tak, že bezprostředně za identitikátorem (jménem slotu) následuje znak pro hvězdičku, tedy jsou odlišeny stejně jako v Selfu. I rodičovské sloty mohou být určeny jen pro čtení nebo pro čtení i zápis.

Sloty metod

Pro sloty metod existují v Selfu dva přípustné zápisy. První má formu selektoru unární, binární nebo slovní zprávy, za nímž následuje rovnítko a objekt metody uzavřený do závorek. Pokud se jedná o binární nebo slovní zprávu, musí objekt metody obsahovat i patřičný počet argumentových slotů.

Druhý alternativní zápis se od prvního liší tím, že je možné zapsat jména argumentů přímo do hlavičky metody, tedy nalevo od rovnítka. Tato hlavička má pak naprosto stejný tvar, jako je tomu u Smalltalku. Objekt metody pak již žádné další argumenty nemá.

Marvin využívá pouze druhého přístupu, který je bližší programátorům ve Smalltalku. Pokud metoda obsahuje navíc argumentové sloty, jsou v současné implementaci ignorovány. V dalších verzích by jejich přítomnost měla být hlášena jako chyba.

6.2.7 Objekty

Objekty jsou v Marvinovi tvořeny jako množina slotů následovaná případným kódem. Celý objekt je ohraničen kulatými závorkami. Množina slotů náležících objektu je vložena mezi dva znaky roury (|) podobně, jako jsou ve Smalltalku ohraničeny dočasné proměnné. Jednotlivé sloty jsou pak od sebe odděleny tečkou. Marvin rozlišuje dva druhy objektů - regulární objekty a metody. Ty se od sebe odlišují přípustnou množinou typů slotů.

Regulární objekty

Regulární nebo-li datové objekty jsou objekty, jejichž součástí není žádný kód. To samo o sobě neznamená, že nemohou obsahovat metody. Přípustná množina typů slotů těchto objektů je následující:

- rodičovské sloty
- datové sloty
- unární, binární a slovní sloty metod

Nemají tedy možnost využívat argumentové sloty, což je logické, protože nemají kód, který by je mohl zpracovávat.

Metody

Metody jsou objekty se sloty a kódem. I v jejich případě nejsou přípustné všechny typy slotů. Jejich součástí mohou být:

- datové sloty
- argumentové sloty

Nemohou tedy mít rodičovské sloty a sloty zanořených metod.

Self

Self teoreticky nerozlišuje mezi regulárními objekty a metodami. Každý objekt může obsahovat libovolné typy slotů a kód. Jeho praktické implementace ale takto svobodné zdaleka nejsou.

Současná implementace Selfu rovněž odděluje regulární objekty od metod s tím, že argumentové sloty u regulárních objektů nepovoluje. Marvin tedy v tomto ohledu za možnostmi Selfu nijak nezaostává. U objektů metod nejsou v nových verzích Selfu povoleny vnořené metody, tedy ani zde Marvin nijak neztrácí.

Narozdíl od jazyka Marvin umožňuje Self mít v metodách rodičovské sloty. V tomto případě se ale jedná o konstrukci bez zjevného rozumného praktického využití. Takovéto rodičovské sloty se netvoří v objektu bloku (nelze je použít k rozšiřování schopností bloku), ale v aktivačním objektu bloku a to až za slotem delegujícího příjemce, takže rodičovské sloty metod nelze využít ani k překrývání definic. Následující příklad tedy vrátí jako výsledek číslo `56`.

```
(|a = (| b=56. p*=(|b=42|) | b |) a
```

To, že se i přes omezení použití různých typů slotů Marvin prakticky vyrovná Selfu, má velice významný dopad na jeho implementaci a především na napojení na virtuální stroj, jak bude diskutováno dále.

6.2.8 Poznámkové sloty

Poznámkové sloty slouží v Selfu k tomu, aby objektu mohl být přiřazen krátký slovní popis, který bude jeho účel dokumentovat. Self rovněž umožňuje komentovat skupiny slotů a to i zanořeně. To znamená, že součástí jedné okomentované skupiny mohou být další okomentované podskupiny slotů.

První návrhy gramatiky jazyka Marivin poznámkové sloty obsahovaly. Nakonec od nich bylo upuštěno z několika důvodů:

- pro funkčnost nejsou nijak důležité. Lze je velice snadno nahradit pomocí speciálního protokolu (tzn. určité standardní množiny zpráv, která bude vracet dokumentaci k objektu) i bez toho, aby byly přímo přítomny v gramatice jazyka. Podobně pracuje i Smalltalk s kategoriemi tříd a metod, případně s komentáři tříd. To také usnadňuje reflexivitu jazyka
- pokud nejsou komentáře objektů a slotů jejich přímou součástí, mohou být umístěny mimo ně a tím usnadnit například proces ořezávání image pro praktické nasazení.
- formát komentování objektů a slotů při jejich odstranění z definice jazyka není pevně stanoven a může tak být snadno modifikován, což je obecnější řešení
- při zvoleném způsobu implementace by se poměrně těžce k objektům a slotům přičleňovaly

Kapitola 7

Napojení na virtuální stroj

Programovací jazyk Marvin je navržen tak, aby co nejefektivněji přinesl ideu beztřídního programování na půdu virtuálního stroje Squeaku. Návrh gramatiky tohoto jazyka šel ruku v ruce s implementačními požadavky. Přesto se jeho vyjadřovací schopnosti prakticky shodují s možnostmi Selfu. Od něj se liší pouze méně restriktivními pravidly pro práci se selektory metod, absencí komentářových slotů a zákazem používat rodičovské sloty v objektech metod a bloků.

V této kapitole si popíšeme, jaké důvody vedly k těmto omezením a nastíníme si hlavní ideje použité při implementaci.

7.1 Rozšíření virtuálního stroje

Pro účely efektivní implementace beztřídního programování byl virtuální stroj Squeaku rozšířen o několik nových vlastností. Prozatím pomineme technické detaily, ty budou rozebrány v části zabývající se implementací. Pro následující výklad nejsou nezbytné

Provedené změny jsou následující:

1. virtuální stroj může pracovat se speciálním druhem objektů (tzv. prototypy), které jsou schopny obsahovat sloty. Tyto objekty mají podporu rodičovských slotů pro čtení i zápis, datových slotů pro čtení i zápis a slotů s metodami. Každý slot je identifikován svým jménem

2. mechanismus zasílání zpráv byl ve virtuálním stroji upraven tak, aby pro prototypy platil jiný, režim, při němž se využívá delegace

3. mechanismus zasílání zpráv v kontextu předka byl ve virtuálním stroji upraven tak, aby pro prototypy platil jiným režim, při němž se využívá delegace v kontextu předka.

Všechny další úpravy virtuálního stroje mají přímou souvislost s těmito třemi rozšířeními.

7.2 Základní literály

Literály jsou v Marvinovi navrženy tak, aby se na jejich místě mohly použít přímo běžné smalltalkovské objekty a jejich zápis byl shodný se stadnardním Smalltalkem. Jedná se konkrétně o čísla, řetězce, znaky, symboly a pole. Tím, že se využijí přímo běžné squeakovské objekty, se výrazně zlepší interoperabilita mezi Marvinem na straně jedné s squeakovským Smalltalkem na straně druhé. Marvin tak nemá potřebu vytvářet robustní objektovou základnu, protože může využívat tu, která je již implementována a odladěna přímo ve Squeaku.

Protože, jak si později ukážeme, lze při delegaci mezi prototypy do jisté míry využívat i běžné smalltalkovské objekty, nepřichází o možnost dále rozšiřovat vlastnosti literálů. Pokud se to ale ukáže účelné, může být kdykoliv kompilátor upraven tak, aby jako základní literály generoval přímo prototypy a například po vzoru Selfu velmi elegantně unifikoval řetězce, znaky a symboly.

7.3 Literály pro regulární objekty

Regulární objekty se vytváří z literálů tak, že se v k tomu uzpůsobeném virtuálním stroji prototyp naplní přímo sloty zapsanými v literálu. Rodičovské a datové sloty se inicializují na nedefinovanou hodnotu nebo referencují výsledek vyhodnocení výrazu obsaženého na pravé straně definice slotu.

Pro sloty s metodami se pak tyto metody pomocí speciálního kompilátoru přeloží na metody a reference na tyto kompilované metody (běžné instance třídy `CompiledMethod`) se uloží do slotů.

Přestože Marvin neobsahuje komentářové sloty, lze je snadno nahradit běžnými datovými sloty nebo využít indexových vlastností prototypů (viz dále) a spojit objekt s komentářem v jeden kompaktní celek. Obzvláště u objektů je ale vhodné využít první variantu, tedy využít datové sloty. Ty lze totiž velmi snadno nahradit za metody a popis objektu tak může být generován dynamicky, jak to vidíme ve Smallltalku v metodách `printOn:`.

7.4 Metody

Velká přednost jazyka Marvin spočívá v tom, že se veškeré metody kompilují přímo do nativního bytekódu Squeaku a s přeloženým kódem se pracuje jako s běžnými instancemi třídy `CompiledMethod`. Výsledný kód se proto provádí velmi rychle a prototypy se nechovají jako cizorodý prvek, ale pouze jako víceméně běžné objekty, které se od těch ostatních odlišují jiným principem zpracování zasílaných zpráv.

Úpravy virtuálního stroje jsou dokonce navrženy tak, aby na místě kompilovaných metod přeložených kompilátorem Marvna šlo použít běžné squeakovské kompilované metody ze tříd. Jediný problém, na který při tom lze narazit, je umístění odkazu na vlastníka metody do literálů, pokud tato metoda používá volání zpráv v kontextu předka (`super`).

Tím, že Marvin používá pro práci s metodami přímo kompilované metody Squeaku, nemá možnost využít mechanismu aktivačních objektů, se kterým pracuje Self. I přesto lze u metod dosáhnout požadovaného chování včetně použití slotů, které jsou součástí objektu metody tak, jak jej známe ze Selfu.

Metody mohou v Marvinovi obsahovat pouze dva typy slotů:

Argumentové sloty Ty se u metod nezapisují přímo, ale generují se po vzoru Smalltaku z hlaviček metod. Ve Squeaku se s argumenty zpráv pracuje tak, že na začátku každé metody se skutečné parametry zpráv vyzvednou ze zásobníku a uloží do dočasných lokálních proměnných. Poté již bytekód Squeaku nijak nerozlišuje mezi tím, co byla dočasná proměnná a co argument metody. Pouze kompilátor Smalltalku zabraňuje tomu, aby programátor reference dočasných proměnných argumentů přepisoval. Marvin argumentové sloty simuluje lokálními proměnnými.

Datové sloty Datové sloty jsou rovněž simulovány pomocí lokálních proměnných, takže přístup k nim je velmi rychlý.

7.4.1 Simulace slotů

Při kompilaci metody se analyzují lokální datové a argumentové sloty metody a pro každý se vytvoří lokální proměnná. Pokud se jedná o datový slot inicializovaný na počáteční hodnotu, uchovává se tato hodnota v literálech kompilované metody a na začátku provádění metody je do patřičné lokální proměnné načtena.

Lokální sloty metod odpovídají slotům aktivačních objektů v Selfu, takže jsou prohledávány dříve než sloty příjemce. Zároveň platí, že jediná možnost, jak v Selfu může aktivační objekt přistoupit, je poslat zprávu sám sobě. Proto může Marvin přístup k těmto slotům simulovat tak, že pro každou zprávu, která je v kódu metody zaslána implicitnímu příjemci, porovná její selektor s názvy lokálních slotů a pokud najde shodu, přeloží přístup k objektu jako přímý přístup k dočasné proměnné. Pokud shodu nenajde, přeloží ji jako běžné volání zprávy příjemci.

Z navázání metod přímo na kompilované metody Squeaku jasně plynou také důvody, proč jsou u těchto objektů zakázány rodičovské a poznámkové sloty. Reprezentace metod by se při použití těchto druhů slotů zkomplikovala, což by se odrazilo na výkonu a kompatibilitě se Squeakem.

7.4.2 Bloky

Bloky se zpracovávají velmi podobně jako metody. I tak ale existují mezi prací s metodami a bloky rozdíly.

Z jazykové stránky je důležité, že u bloků lze uvádět argumentové sloty. U metod nejsou nutné, protože je specifikuje již hlavička metody a Marvin na rozdíl od Selfu používá pouze tento jediný zápis slotů s metodami.

Další velmi podstatná změna je v tom, že bloky se nereprezentují v kompilovaných metodách jako literály, ač jimi formálně jsou. Pro konstrukci bloků se využívá speciální optimalizovaná konstrukce zapsaná v bytekódu používající skokové instrukce.

U bloků je nutné si uvědomit, že jejich lokální proměnné nejsou součástí jich samotných, ale vytvářejí se v jejich lexikálním rodiči, tedy metodě, která je obaluje.

7.5 Resend

Marvin využívá k provádění zpráv v kontextu rodičovského objektu stejné bytekódy, které Smalltalk používá při volání v kontextu rodičovské třídy (`super`). O správnou interpretaci se starají modifikace uvnitř virtuálního stroje. Podobně jako samotný Squeak doplňuje i Marvin na poslední pozici literálů v kompilované metodě referenci na objekt, jemuž překládaná metoda patří. V případě Smalltalku se takto samozřejmě referencuje třída.

Kapitola 8

Modifikace virtuálního stroje

Ač modifikace virtuálního stroje jde ruku v ruce s dalšími změnami provedenými na úrovni image, mohou být na nich v případě potřeby prakticky zcela nezávislé. Tvoří samostatný celek a mohou být bez větší námahy použity pro obměněnou implementaci beztřídního programování po Allenově vzoru, pokud se k takovému kroku někdo v budoucnu rozhodne. Svým způsobem se jedná o převedení dosavadních snah o prototypizaci Squeaku z úrovně image přímo na úroveň virtuálního stroje.

I výsledné použití a charakteristiky jsou původnímu balíku Prototypes podobné. Vnitřní reprezentace prototypů ale pracuje na zcela jiných principech.

V této části popíšeme zásahy do virtuálního stroje Squeaku, které do něj doplňují podporu prototypů a delegace.

8.1 Virtuální stroj

Virtuální stroj Squeaku je vytvořen velmi zajímavým způsobem. Namísto toho, aby byl jeho kód napsán v nějakém standardním přenositelném programovacím jazyce, jako je C nebo C++, je jeho velká část napsána přímo ve Smalltalku. Teprve vytvořené smalltalkovské zdrojové kódy se automaticky přetransformují na program interpretu napsaný v jazyce C. Zdrojový kód v C je pak přeložen přímo do nativního strojového kódu hostitelské platformy. Tato vlastnost je pro Squeak specifická a stojí velkou měrou za tím, že se dnes jedná o velmi úspěšnou implementaci Smalltalku-80.

Při programování interpretu virtuálního stroje přímo ve Smalltalku se nepoužívá Smalltalk ve své čisté podobě. Byla z něj vybrána jen podmnožina přípustných operací a dokonce přišel i o podporu objektů. Zjednodušeně lze říct, že Smalltalk byl takto degradován na přímou obdobu jazyka C s jinou syntaxí. Ač se to může jevit na první pohled jako neracionální operace, ve skutečnosti má velmi dobré důvody.

Díky tomu, že zdrojové kódy jsou napsány přímo ve Smalltalku, není třeba přímo provádět překlad do strojového kódu, ale naprogramovaný interpret se může nechat pracovat přímo ve squeakovské image. Proto mohou programátoři virtuálního stroje bez potíží používat všech výhod, které jim Smalltalk při vývoji aplikací nabízí počínaje kvalitním laděním a konče příjemným inkrementálním prostředím. Programátoři se tak mimo jiné vyhnou zdlouhavému překladu a získají nad běžícím virtuálním strojem kontrolu, o níž by si v případě C/C++ mohli nechat jen zdát.

K tomu je třeba ještě přičíst fakt, že vývoj Squeaku začal v polovině devadesátých let minulého století, kdy možnosti dostupných vývojových nástrojů a především jejich ladící možnosti ještě zcela nedosahovaly dnešní úrovně, což bylo pro autory Squeaku zhýčkané integrovanými vývojovými

prostředími Smalltalků jistě těžko snesitelné.

Samozřejmě některé platformně závislé části bylo nutné stále psát v čistém C, ale jejich procentuální podíl byl velmi malý.

Rovněž při přípravě této práce byly veškeré modifikace nejdříve odsimulovány a až teprve poté byly výsledné modifikace zahrnuty do překladu skutečného virtuálního stroje. Nikdy nebylo nutné vytvářet či modifikovat zdrojové kódy napsané v jazyce C.

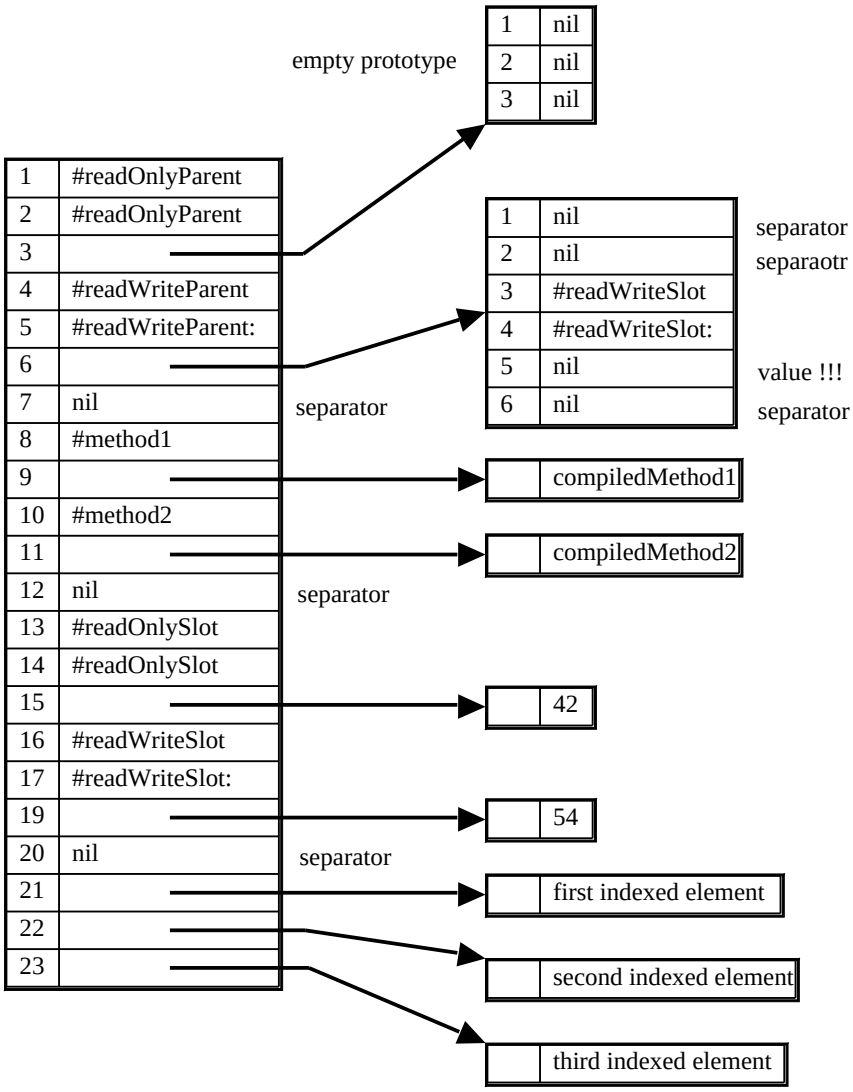
8.2 Prototypy

Prototypy jsou ve virtuálním stroji reprezentovány jako samostatný druh objektů, přičemž platí, že vycházejí z indexovaných objektů, jako jsou například pole (`Array`).

Z pohledu Smalltalku se jedná o instance specifické třídy (`MarvinPrototype`). Pro tyto instance modifikovaný virtuální stroj provádí jiný mechanismus zpracování zpráv, než je tomu u běžných smalltalkovských objektů. Tato třída je virtuálnímu stroji specifikována pomocí volání speciálních pro tento účel vytvořených primitiv.

Vnitřní reprezentace prototypu je následující:

- prototyp je tvořen indexovaným objektem pevné velikosti, přičemž každý prototyp je tvořen čtyřmi sekcemi
- každá sekce je oddělena referencí na nedefinovaný objekt (`nil`), který zde hraje roli separátoru
- první sekci tvoří množina rodičovských slotů. Každý slot je reprezentován třemi prvky pole prototypu. První prvek z trojice je reference na symbol se selektorem zprávy pro čtení hodnoty z rodičovského slotu. Druhý prvek je selektor pro zápis hodnoty do rodičovského slotu. V případě, že je slot určen pouze pro čtení, je hodnota druhého prvku trojice stejná jako první. Třetí prvek trojice je reference na hodnotu rodičovského slotu. Tou může být i objekt `nil`
- pokud objekt obsahuje více rodičovských slotů, trojice jeho položek následuje bezprostředně za trojicí předcházejícího rodiče. Sekce rodičovských slotů je ukončena separátorem (`nil`)
- pokud objekt nemá rodičovské sloty, je hned první prvek prototypu nedefinovaná hodnota (`nil`)
- za sekcí rodičů následuje sekce metod. Každá metoda je tvořena dvojicí prvků, kde první prvek je reference na symbol představující selektor metody a druhý prvek je reference na příslušnou kompilovanou metodu
- objekt může obsahovat více metod nebo žádnou. V takovém případě je dalším prvkem po separátoru za blokem rodičovských slotů další separátor (`nil`)
- sekce rodičů je následována blokem datových slotů. Ty mají stejný formát jako rodičovské sloty, takže jsou tvořeny trojicí prvků, kdy první prvek představuje selektor pro čtení hodnoty slotu, druhý je selektor pro zápis hodnoty slotu nebo je stejný jako selektor pro čtení a třetí prvek je reference na hodnotu slotu. Ta může být i nedefinována (`nil`)
- sekce datových slotů je ukončena separátorem (nil)
- za sekcí datových slotů následuje sekce indexovaných elementů. Ta může být libovolně dlouhá a není ukončena separátorem. Končí společně s hranicí objektu



Obrázek 8.1: Formát prototypu

- prázdný prototyp je tedy tvořen nejméně třemi nedefinovanými hodnotami (`nil`)

Tato struktura prototypů byla zvolena z následujících důvodů:

- nejsou použity asociace, což by představovalo další zbytečné objekty pro každý prototyp spojené s větší paměťovou náročností a zároveň by zpomalilo proces vyhledávání slotů v objektech
- sekce slotů mají vždy pevnou velikost (tři prvky pro rodičovské a datové sloty a dva prvky pro sloty s metodami), což zjednodušuje vyhledávání. Pokud by například byl datový slot určený pouze pro čtení tvořen dvěma prvky, tedy selektorem pro čtení hodnoty slotu a referencí na hodnotu slotu, musel by být nějakým způsobem zajištěn mechanismus pro rozlišení oddělování jednotlivých slotů od sebe

- vyhledávání selektoru zaslané zprávy v objektu je velice rychlé, protože stačí vždy pouze porovnávat reference na selektory slotů s referencí na selektor zprávy
- prototyp může být z úrovně Smalltalku vytvořen jako obyčejné pole a práce s ním je velice pohodlná a konzistentní s ostatními objekty
- sekce indexovaných prvků za oblastí s definicí slotů může být velice jednoduše použita pro vytvoření prototypů hrajících roli kolekcí. Velikost indexovaného pole prototypu je rovna velikosti prototypu bez velikosti sekce s definicí slotů
- hned na začátku objektu jsou vždy definováni jeho rodiče (až po první separátor), což zrychluje proces delegace
- objekty metod jsou upřednostněny před objekty datových slotů, takže pokud je u objektu předefinován přístup ke slotu metodou, je tato metoda nalezena jako první
- duplicita jmen slotů není explicitně zakázána jeho strukturou. Naopak u datových a rodičovských slotů určených pro čtení je to běžný stav. Je vždy nalezen první slot s odpovídajícím selektorem. Nicméně vytvoření více slotů se stejnými přistupujícími selektory je zakázáno na úrovni primitiv pro manipulaci s objekty
- druhý prvek u bloků rodičovských a datových slotů určených pouze pro čtení by mohl být i nedefinován (`nil`). Jedná se pouze o konvenci, která nemá dopad na rychlost vyhledávání slotů v objektech.

8.3 Mechanismus vyhledání slotů v prototypu

Pokud je objektu zaslána zpráva, provede se její zpracování následujícím způsobem:

1. začnou se procházet selektory pro rodičovské sloty. Vždy se porovná selektor zaslané zprávy s položkou prototypu. Pokud není nalezena shoda, pokračuje se dalším selektorem. Reference na hodnoty slotů jsou samozřejmě přeskakovány. Takto se pokračuje až do nalezení separátoru (`nil`)
2. pokud je nalezena shoda selektoru s prvním prvkem bloku rodičovského slotu, jsou na zásobníku provedeny takové operace, které simulují zavolání zprávy s výsledkem, který odpovídá hodnotě daného slotu.
3. pokud je nalezena shoda selektoru s druhým prvkem bloku rodičovského slotu, což je selektor pro zápis do slotu, jsou na zásobníku provedeny takové operace, které simulují zavolání zprávy zapisující hodnotu do slotu. Výsledkem pak je reference na příjemce zprávy (nikoliv na zapisovanou hodnotu). Prohledávání slotů v objektu se poté ukončí
4. v okamžiku, kdy se narazí na separátor oddělující sekci rodičovských slotů od sekce metod objektů, změní se režim prohledávání
5. porovnává se postupně selektor zaslané zprávy se selektory slotů s metodami, přičemž reference na kompilované metody (hodnoty těchto slotů) jsou přeskakovány
6. pokud se nalezne shoda selektoru zaslané zprávy se jménem slotu, provede se volání referencované kompilované metody s patřičnými dopady na zásobník. Prohledávání slotů objektu se ukončí

7. v okamžiku, kdy se narazí na separátor oddělující sekci slotů s metodami od sekce datových slotů, změní se režim prohledávání

8. začnou se prohledávat datové sloty, přičemž se zpracovávají naprosto stejně, jako rodičovské sloty

9. pokud se narazí na selektor oddělující sekci datových slotů od indexovaných položek, algoritmus přejde na vyhledávání v rodičovských slotech objektu

8.4 Mechanismus vyhledávání slotů v rodičovských objektech

Vyhledávání slotů v rodičích objektu probíhá pomocí vyhledávací metody DFS (Depth First Search). Při tom se reference na již vyhledané objekty ukládají do speciálního pole, aby nebyly prohledávány vícekrát a tím se zamezilo zacyklení.

Vyhledávání slotů v rodiči probíhá rekurzivně podle algoritmu pro vyhledávání slotů v prototypu. Na rozdíl od Selfu se nijak nekontroluje vznik případných nejednoznačností. Vždy se použije první nalezený slot. Tento přístup má následující důsledky:

1. objekty se i při vyhledávání slotů chovají jako uzavřené entity

2. na rozdíl od Selfu je podstatně jednodušší přetěžovat definice slotů a tím lépe pracovat s jmennými prostory

3. vyhledávání slotů je rychlejší, protože se po nalezení slotu nemusí pokračovat ve vyhledávání případných kolizí

Nicméně, zda se algoritmus DFS skutečně osvědčí, se ukáže až při praktickém nasazení. Nezle vyloučit, že tato vlastnost v pozdějších návrzích úprav virtuálního stroje změní.

Pokud objekt žádné rodiče nemá nebo v nich nebyl hledaný slot nalezen, provede se běžné smalltalkovské volání zaslané zprávy. To znamená, že zprávu obdrží ke zpracování třída, jejíž instance ve skutečnosti prototypy jsou (`MarvinPrototype`). Zprávy, kterým takto třída rozumí, lze z pohledu porototypů vnímat jako obdobu primitivních volání.

Pokud rodičovský slot referencuje objekt, který není prototyp a jedná se tedy o obyčejný smalltakovský objekt, provede se vyhledání patřičné metody ve třídě tohoto objektu a následně v jeho supertřídách. Pokud metoda odpovídající selektorem zaslané zprávě není v této třídě nalezena, pokračuje se dalším rodičem. Pakliže taková třída nalezena je, což znamená, že objekt referencovaný jako rodič zprávě rozumí, je mu tato zpráva zaslána a výsledek její invokace je následně vrácen.

Za pozornost stojí, že se v tomto případě nejedná o obvyklou delegaci na rodiče, ale o přímé zaslání zprávy. Delegace by pro interoperabilitu jazyka Marvin a Squeaku byla jistě podstatně zajímavější řešení, ale bohužel není možná.

Důvod spočívá v tom, že běžné smalltalkovské objekty používají přímý přístup ke svým instančním proměnným. V jejich bytekódu jsou instanční proměnné odkazovány pomocí indexů a proto u nich nelze najít skutečná jména. Prototypy oproti tomu žádné instanční proměnné nemají. To znamená, že pokud bychom se pokusili provést delegaci na standardní smalltalkovský objekt a zavolali bychom metodu, která k instančním proměnným přistupuje, vedlo by to k chybě. Případné řešení tohoto problému by si vyžádalo podstatně větší zásahy do virtuálního stroje, než které Marvin v současnosti provádí.

8.5 Mechanismus volání v kontextu předka

Zpracování volání v kontextu předka, která se v Marvinovi provádí pomocí speciálního pseudoslotu `resend`, odpovídá na úrovni volání smalltalkovskému `super`. I pro tuto operaci virtuální stroj používá jiný režim zpracování pro instance prototypové třídy (`MarvinPrototype`).

Proces obsluhy resendu se velmi podobá běžnému volání zpráv pro prototypy. Hlavní rozdíl spočívá v tom, že se sloty nezačínají vyhledávat v příjemci zprávy, ale v rodičích vlastníka volané metody. Referenci na tohoto vlastníka je nejdříve potřeba získat. To se provádí tak, že se z objektu aktivního kontextu referencovaného virtuálním strojem získá reference na právě prováděnou kompilovanou metodu a z této metody je přečtena hodnota posledního literálu. Ta odkazuje objekt, který je majitelem kompilované metody, tedy našeho hledaného vlastníka. Referenci na něj musí do posledního literálu zapsat přímo kompilátor, který provádí překlad metody.

Přímý resend není na úrovni virtuálního stroje doposud podporován.

8.6 Zhodnocení

Provedené zásahy do virtuálního stroje jsou realizovány velmi citlivě. Při práci s běžnými objekty vyžadují pouze provedení jednoho porovnání dvou celočíselných hodnot navíc, což je vzhledem k celkové režii volání zpráv naprosto zanedbatelné.

Samotné hledání slotů v prototypech je v některých případech dokonce rychlejší než běžné volání zpráv ve Smalltalku. U něj je ale tento proces značně optimalizován pomocí speciálních vyrovnávacích paměti. Ve výsledku jsou tak programy využívající prototypy zatím pomalejší. Nicméně vhodnými optimalizacemi by mělo být možné dosáhnout u beztřídních programů minimálně stejné výkonnosti.

Jistý výkonnostní handicap oproti Smalltalku představuje absence instančních proměnných, k nimž je přístup z bytekódu velice rychlý. Jedna z možných variant rozšíření schopností Squeaku o beztřídní přístup počítala se simulací datových slotů pomocí instančních proměnných, ale tato varianta by ve výsledku přinesla mnoho problémů a vyžádala by si větší paměťovou náročnost prototypů.

Zvolený způsob reprezentace beztřídních objektů lze z výkonnostního i paměťového hlediska považovat téměř za optimální. Na druhou stranu lze i zde dosáhnout dalších výrazných úspor. Především je možné zavést po vzoru implementace Selfu mapy, což znamená, že by podobné objekty sdílely deskriptor popisující jejich strukturu a objekty samy by pak obsahovaly pouze data a referenci na tento deskriptor. Tím by se dosáhlo nižší paměťové náročnosti. Výpočetní náročnost by se ovšem pravděpodobně nesnížila.

Kapitola 9

Implementace

Implementace má dvě víceméně samostatné části. Ta první se dotýká pouze změn v image Squeak Smalltalku. Jedná se o vytvoření překladače jazyka Marvin přímo do bytekódu Squeaku.

Druhá část implementace je tvořena modifikací virtuálního stroje, tzn. jeho rozšířením o podporu prototypů a delegace. Tyto modifikace bylo poté také nutné zohlednit na úrovni image Squeaku.

Tato kapitola si klade za cíl popsat konkrétní vytvořenou implementaci a upozornit na problémy, které bylo při její tvorbě nutné překonat.

K implementaci bylo použito výhradně volně šiřitelných otevřených prostředků. V první řadě se jedná o samotný Squeak doplněný o program SmaCC. Dále byl použit kompilátor GCC, který je používán k překladu vygenerovaných zdrojových kódů virtuálního stroje.

9.1 Překladač

K tomu, aby mohl být jazyk Marvin do Squeaku začleněn, bylo nutné pro něj vytvořit překladač. Bohužel nebylo prakticky možné vyjít ze standardního překladače Smalltalku, protože ten je naprogramován metodou rekurzivního sestupu a je pro překlad Smalltalku výrazně optimalizován. Bylo nutné vytvořit překladač zcela nový.

9.1.1 SmaCC

Pro vytvoření překladače jsem použil program SmaCC [6] společnosti The Refactory, Inc. Jedná se o volně šiřitelný generátor překladačů napsaný v jazyce Smalltalk. Je to v současnosti nejpoužívanější program tohoto druhu pro Smalltalk. Oproti jiným generátorům překladačů má celou řadu předností. Umožňuje například vytvářet překladače pro nejednoznačné gramatiky a gramatiky s překrývajícími se tokeny. Navíc je také rychlejší.

SmaCC pro generování překladačů využívá volitelně dvě metody překladu zdola nahoru. Konkrétně se jedná o metody LALR(1) a LR(1). U překladače jazyka rozsahu Marvina lze výstupy obou těchto metod považovat za srovnatelné.

SmaCC se pro implementaci ukázal jako velice výhodné řešení. Jednak umožňuje velmi snadno provádět i poměrně razantní zásahy do již hotové gramatiky, jednak dokáže v plné síle využívat všech hlavních předností Smalltalku. V první řadě se jedná o skutečnost, že při modifikaci akcí spojených s jednotlivými přepisovacími pravidly není nutné nechávat celý překladač znovu přegenerovávat, což výrazně zrychluje vývojový cyklus, kdy lze kód překladače modifikovat i během samotného překladu.

V budoucnu by samozřejmě bylo vhodné vytvořit překladač jazyka Marvin ručně metodou rekurzivního sestupu. Takový překladač by byl rychlejší a měl by podstatně lepší možnosti při

zpracovávání chyb. Na druhou stranu je k tomuto kroku možno přistoupit až v okamžiku, kdy bude možné gramatiku jazyka Marvin prohlásit za ustálenou, což v této chvíli rozhodně nelze.

9.1.2 Lexikální analyzátor

Tvorba lexikálního analyzátoru byla poměrně snadná, protože pro většinu tokenů stačilo pouze zapsat jejich specifikaci. Pouze v případě jednořádkových komentářů bylo nezbytné dopsat část analyzátoru ručně.

9.1.3 Syntaktický analyzátor

Metoda překladu zdola nahoru přirozeně přímo ovlivnila i konstrukci překladače. Objekty nesoucí informace o kódu a jeho kontextu je nutné vytvářet často přímo u pravidel v nejnižších patrech gramatiky a při jejich postupném přenosu výš a výš je postupně slučovat nebo transformovat. SmaCC standardně přenáší na vyšší úrovně kolekce tokenů. Ty je nutné nahrazovat přímo příslušnými deskriptory generovaných objektů, jako jsou sloty, kontexty s kódem, objekty apod.

9.1.4 Obecný překlad

V první fázi překladu je nejdříve vytvořena obecná struktura generovaných entit. Jsou to:

Sloty Jsou identifikovány svým jménem a typem. U slotů majících metody rovněž obsahují i množinu argumentů. Každému slotu přísluší hodnota. Pro její získání z výrazu uvedeného na pravé straně definice slotu je u datových a rodičovských slotů vytvořena zcela nová speciální instance překladače, která tento výraz přeloží a nechá vykonat v kontextu objektu `lobby`.

Konstanty Obsahují smalltalkovské objekty pro jednoduché literály, jako jsou čísla a řetězce.

Metody Obsahují množinu slotů, které v nich byly definovány. Dále obsahují příslušný přeložený kód, který je obsahem metody.

Bloky Jedná se o speciální druh metod. Na této obecné úrovni překladu se od nich prakticky neliší.

Objekty Tvoří je množina slotů. Nemají přiřazen žádný kód.

Nejdůležitějším produktem překladu v této fázi jsou tzv. kontexty s kódem. Ty obsahují literály, což jsou výše uvedené entity, dále selektory, což jsou jména volaných zpráv, a samozřejmě vygenerované instrukce. Těch současná implementace rozlišuje šest:

send: Zašle objektu na vrcholu zásobníku zprávu identifikovanou určitým selektorem.

selfSend: Zašle příjemci zprávu identifikovanou určitým selektorem.

resend: Zašle příjemci zprávu identifikovanou určitým selektorem v kontextu předka.

pushSelf Na vrchol zásobníku umístí příjemce zprávy.

pushLiteral: Na vrchol zásobníku umístí referenci na určený literál.

pop Odebere jednu položku z vrcholu zásobníku a získanou hodnotu zahodí.

returnTop Návrat z metody s vrcholem zásobníku jako výsledkem.

returnImplicit Implicitní návrat z kontextu.

9.1.5 Překlad do nativního bytekódu

Poté, co je dokončen obecný překlad, přistoupí se k samotnému generování nativního bytekódu. U něj se sestaví pole selektorů a literálů a určí jejich definitivní pořadí, aby mohly být v bytekódu odkazovány pomocí indexů.

Nejproblematičtější jsou v této fázi bloky. S těmi se totiž nepracuje jako s obyčejnými literály, ale jsou generovány pomocí speciální sekvence instrukcí bytekódu obsahující i skoky. Navíc literály a selektory, které využívají, nepřísluší jim, ale metodě, které ve které jsou použity.

Pro převod obecných kontextů kódu do bytekódu se využívají tzv. generátory. Při tom platí, že generátor pro bytekód bloku je speciální případ generátoru metody. Od něj se liší především tím, že má definovaný rodičovský kontext, pomocí něhož získává referenci na mateřskou metodu. Rodičem bloku může být buď přímo metoda nebo jemu nadřazený blok. Pro metodu platí, že je sama svým rodičem, díky čemuž lze pro generování metod i bloků použít stejné instrukce.

Liší se například i implicitní návrat z kontextu metody a implicitní návrat z kontextu předka. U metody je vrácen příjemce zprávy, kdežto u bloku je to nedefinovaná hodnota v případě prázdného bloku nebo výsledek posledního výrazu.

V této fázi překladu se také provádí simulace přístupu ke slotům metod a bloků prostřednictvím lokálních dočasných proměnných. Toho se dociluje pomocí speciálního zpracování instrukce `selfSend:`, kdy se porovnávají selektory zasílaných zpráv se jmény lokálních slotů.

9.1.6 Evaluátor

Pokud se zaměříme na evaluátor outlineru ve vývojovém prostředí Self, zjistíme, že ten provádí zapsaný kód v kontextu nějakého objektu. Standardně jím bývá `lobby`. Pokud by tento evaluátor uměl vyhodnocovat jen jednotlivý výraz a my bychom chtěli použít lokální sloty (jako obdobu dočasných proměnných), museli bychom, museli bychom vytvořit objekt obsahující tyto sloty a metodu definující akci, které nad těmito sloty chceme provést. Následně bychom museli tomuto objektu připravenou metodu zavolat.

Naopak pokud bychom v evaluátoru zpracovávali blok kódu, neměli bychom k dispozici žádné lokální sloty pro uchovávání dočasných výsledků.

Proto evaluátor v Selfu a testovací okno ve SmaCC v případě jazyka Marvin postupují tak, že napsaný kód chápou jako blok, která je součástí daného objektu. Tím je v případě testovacího okna SmaCC standardní objekt `lobby`. Díky tomu je výsledkem celého vyhodnocení výsledek posledního výrazu.

9.1.7 Integrace se Smalltalkem

Marvin má možnost využívat Smalltalkovské objekty při delegaci, jak bylo uvedeno výše. Dále může pracovat přímo se smalltalkovskými objekty, které si vytvoří jako literály. Existuje však ještě další mechanismus, který dává beztřídním programům možnost, jak pracovat s běžnými smalltalkovskými objekty.

Překladač jazyka Marvin má k dispozici již standardní prototyp, který je po vzoru Selfu pojmenován `lobby`. V jeho kontextu dochází k vyhodnocování výrazů přiřazovaných do datových objektů apod. V jeho kontextu je rovněž prováděn výsledný program.

Hlavní funkcí tohoto objektu je zpřístupňovat důležité globální sloty, jako je `nil`, `true`, `false` apod. Důležitou roli ale hraje také při procesu hledání slotů.

Jestliže je objektu zaslána nějaká zpráva, pokusí se v něm nebo jeho rodičích virtuální stroj nalézt slot stejného jména, jako je selektor zprávy. Pokud tato zpráva nalezena není, pokusí se tuto zprávu zpracovat třída prototypu (`MarvinPrototype`). Když ji ani ta nenajde, vyvolá se výjimka,

kterou třída prototypu zpracuje tak, že příjemci zašle zprávu `slotNotFound:`. Ten se ji opět pokusí najít.

Pokud virtuální stroj slot `slotNotFound:` nenajde, je přeposlána třídě prototypu, která ji zpracuje vygenerováním chyby, takže k zacyklení nedojde. Pokud ale objekt deleguje svoje zprávy na objekt `lobby`, ten slot se jménem `slotNotFound:` obsahuje a provede příslušnou metodu. Tato metoda má naprosto specifické postavení. Nejedná se totiž o kompilovanou metodu, která by byla vytvořena překladačem jazyka Marvin, ale jedná se kompilovanou metodu, která je součástí Smalltalku. Jejím cílem je prozkoumat globální systémový slovník `Smalltalk` a pokusit se v něm nalézt objekt, jehož klíč se shoduje se selektorem nenalezené zprávy. Pokud se takový objekt najde, je vrácen jako výsledek volání této zprávy.

Tímto způsobem lze přímo přistupovat z beztřídních programů ke globálním objektům Smalltalku a je možné díky tomu zcela běžným způsobem vytvářet instance tříd apod. Nicméně protože tento proces je relativně pomalý, nabízí `lobby` ještě další slot se selektorem `St:`, který jako parametr očekává symbol se jménem globálního objektu. Jeho metoda provádí vyhledávání v systémovém slovníku rovnou bez další přídavné režie.

9.1.8 Stav implementace

Překladač jazyka Marvin je v současné implementaci funkční a lze s ním již pracovat. Bohužel ale zatím neimplementuje zcela dokonale všechny rysy jazyka. Jedná se především o lokální sloty metod a bloků, u nichž zatím ignoruje přiřazené hodnoty. Ty je potřeba v době překladu vyhodnotit, umístit do literálů a při invokaci metod jimi naplnit příslušné lokální dočasné proměnné simulující sloty.

Mezi další ne zcela dokončené úkoly patří mimo jiné i překlad do bytekódů, kdy se v současnosti nevyužívají úplně všechny vícebytové rozšířené instrukce bytekódu, takže přípustné rozsahy operandů jsou menší, než by musely být.

Současná omezení kompilátoru jazyka Marvin jsou důsledkem toho, že primárním cílem při tvorbě implementace bylo prověřit realizovatelnost všech navržených postupů a některé méně důležité a bezproblémově realizovatelné detaily překladu byly prozatím opomenuty.

9.2 Virtuální stroj

Hlavní myšlenky úprav, které byly provedeny ve virtuálním stroji, jsme již probrali v kapitole o napojení jazyka Marvin na virtuální stroj. Zde již jen prodiskutujeme několik technických detailů.

9.2.1 Primitivy

Smalltalk standardně obsahuje několik objektů, které pro něj mají výjimečnou roli. Jsou to tzv. speciální objekty. Jedná se například o některé singletony (`true`, `false`, `nil`), významné třídy (`Display`, `Process` apod.) nebo některé důležité selektory metod.

K těmto speciálním objektům bylo nutné přičlenit i třídu, jejíž instance jsou prototypy a mají tedy pozměněný režim zasílání zpráv. Tato třída nemá pevně stanovené jméno. O tom, jakou třídu má virtuální stroj pro prototypy použít, se dozví pomocí nové speciální primitivy. Množina primitiv byla rozšířena ještě o jednu, která naopak prototypovou třídu vrátí.

9.2.2 Kontrola cyklů

Ke kontrole cyklů při vyhledávání slotů je použito statické pole, kam se postupně ukládají prozkoumané objekty. Před zkoumáním slotů v dalším objektu se toto pole nejdříve sekvenčně projde a zjistí se, jestli se reference na zkoumaný objekt v něm již nachází. Pokud ano, objekt již znovu zkoumán není. Pokud ne, je na konec tohoto pole vložena jeho reference a virtuální stroj se v objektu pokusí nalézt daný slot.

Velikost pole prozkoumaných objektů je v současné době zvolena na 512 objektů. Při překonání tohoto limitu je hlášena chyba. Dynamicky alokovaný prostor není vytvářen hlavně kvůli rychlosti a jednoduchosti, protože tato operace není v generátoru virtuálního stroje zcela bez problémů a je zde potřeba již pracovat i s garbage collectorem.

9.2.3 Generování interpretu

Programování virtuálního stroje ve Smalltalku má celou řadu úskalí. V žádném případě nelze říct, že pokud se programátorovi podaří změny virtuálního stroje úspěšně naprogramovat a prověřit v simulátoru uvnitř Smalltalku, má vyhráno a následná konverze do C a překlad do nativního strojového kódu je již jen formalita. Naopak. Především v hlavní smyčce interpretu, do které byly potřeba dělat největší zásahy, je použita celá řada optimalizací zásadním způsobem ovlivňující přítomnost dostupnost proměnných.

V důsledku toho jsem byl nucen oddělit proces vyhledání slotů a proces provedení příslušné metody či přístupu ke slotu v nalezeném objektu. Proto se ve skutečnosti objekt, ve kterém je hledaný slot nalezen, prohledává dvakrát. Toto chování lze sice odstranit, ale pouze za cenu přidávání dalších globálních proměnných interpretu. Nicméně i přesto se s tímto optimalizačním krokem v příštích verzích počítá.

Kapitola 10

Závěr

V rámci řešení diplomového projektu byl představen programovací jazyk Self a jím použitý koncept objektově orientovaného programování, který k implementaci sdíleného chování nevyužívá třídy. Je zde ukázáno, že odstranění tříd z objektového jazyka vede k jeho podstatnému zjednodušení a podaří se tak vyřešit řada neduhů, kterými čistě objektově orientované jazyky často trpí. Navíc beztřídní jazyky přináší několik velice zajímavých nových rysů, jako je dynamická dědičnost a snazší oddělení uživatelských prostorů.

Porovnal jsem současné přístupy, které jsou používány pro doplnění vývojového prostředí Squeak Smalltalk o podporu beztřídního programování a zhodnotil jejich výhody a nevýhody. Na základě těchto poznatků jsem navrhl zcela nový pohled na řešení tohoto problému, který revolučním způsobem modifikuje stávající virtuální stroj tak, aby byl schopen podporovat beztřídní objekty a delegaci.

Zároveň jsem navrhl nový programovací jazyk, který si klade za cíl skloubit nejlepší vlastnosti jazyků Self a Smalltalk-80 tak, aby společně s provedenými modifikacemi virtuálního stroje umožnil programátorům ve Squeaku používat beztřídní programování ve formě, která je velice blízká jejich stávajícím návykům a zvyklostem, a mohou si ji tedy rychle a snadno osvojit. Při tom je však také kladen velký důraz na to, aby beztřídní paradigma implementoval ve stejné šíři, v jaké bylo použito v programovacím jazyce Self. Pro tento nový jazyk jsem ve Smalltalku vytvořil funkční kompilátor, který provádí překlad přímo do nativního squeakovského bytekódu, takže výsledné programy jsou se Squeakem velice dobře integrovány a jsou výkonnostně srovnatelné s programy napsanými ve Smalltalku. Přestože realizované modifikace stávajícího virtuálního stroje Squeaku jeho možnosti značně vylepšují, jsou provedeny velice citlivě, takže na upraveném virtuálním stroji lze bez jakýchkoliv omezení spouštět i původní obrazy objektové paměti.

Implementace některých málo důležitých rysů navrženého programovacího jazyka nebyla zatím bezezbytku dokončena, což ale není důsledek jejich obtížné realizovatelnosti, ale pouze velkého rozsahu této práce. Jedná se například o přiřazování hodnot lokálním datovým slotům nebo o pokrytí plné množiny přípustných bytekódů při překladu.

Tento projekt nabízí obrovskou škálu možných navazujících prací. Kromě dalších vylepšování překladače a optimalizací výkonné části se v první řadě jedná o vytvoření sady vývojových nástrojů (outliner, dekompilátor, ladící nástroje), které by práci s beztřídními systémy usnadnily a poskytly vývojářům stejné pohodlí, na jaké jsou zvyklí z implementací Smalltalku nebo Selfu. Otevřená rovněž zůstává otázka integrace realizovaných rozšíření se standardním Squeakem tak, aby se navržený programovací jazyk (v případné modifikované podobě) stal vedle Smalltalku jedním ze dvou mateřských jazyků této progresivní platformy.

Literatura

[1] Křivánek, Pavel. Squeak: návrat do budoucnosti. http://www.root.cz/serialy/squeak-navrat-do-budoucnosti/, 2004.

[2] WWW stránky. AspectS (oficiální stránky). http://www.prakinf.tu-ilmenau.de/ hirsch/Projects/Squeak/AspectS/ (květen 2005).

[3] WWW stránky. Croquet Project. http://www.opencroquet.org/ (květen 2005).

[4] WWW stránky. Prototypes (oficiální stránky). http://russell-allen.com/squeak/prototypes/ (květen 2005).

[5] WWW stránky. Self (oficiální stránky). http://research.sun.com/self/ (květen 2005).

[6] WWW stránky. SmaCC by Refactory, Inc. http://www.refactory.com/Software/SmaCC/ (květen 2005).

[7] WWW stránky. Squeak (oficiální stránky). http://www.squeak.org (květen 2005).

[8] WWW stránky. Sun Microsystems, Inc. http://www.sun.com/ (květen 2005).

[9] WWW stránky. The Home of the Slate Programming Language. http://slate.tunes.org/ (květen 2005).

[10] Sun Microsystems, Inc. *The SELF 4.1 Programmer's Reference Manual*. 1995–2000. Dokument dostupný na http://www.cs.auc.dk/ guttorm/Undervisning/Self-4.1-Pgmers-Ref.pdf (květen 2005).

[11] Švec, Martin. *Programovací jazyk a aplikační prostředí SELF*. 2004. Dokument dostupný na http://www.fit.vutbr.cz/study/courses/TJD/public/0304TJD-Svec.pdf (květen 2005).

[12] Wolczko, M.; Smith, R. B. Prototype-Based Application Construction Using SELF 4.0. http://research.sun.com/self/release_4.0/Self-4.0/Tutorial/index.html/ (květen 2005).

Přílohy

Příklad

Následující příklad napsaný v jazyce Marvin názorně demonstruje funkčnost volání v kontextu předka. Výsledkem jeho vyhodnocení je číslo 3.

```
| p1 |
p1: (|
    parent* = [(|
        parent* = [(|
            a = (^a).
            b = [1] |)].
        a = (^resend a) |)].
    a = [(|
        a = [3].
        b = [4] |)].
    method = (^resend a a) |).
p1 method
```

Příklad bytekódu metod, které jsou vyhodnoceny již během překladu a které generují hodnoty datových slotů:

```
17 <20> pushConstant: 3
18 <7C> returnTop
```

Příklad bytekódu metod, které jsou vyhodnoceny již během překladu a které do datového slotu umístí literál pro objekt:

```
21 <22> pushConstant: a MarvinPrototype
22 <7C> returnTop
```

Metoda s jednoduchým zasláním zprávy příjemci:

```
17 <70> self
18 <83 00> send: a
20 <7C> returnTop
```

Metoda s jednoduchým zasláním zprávy příjemci v kontextu předka:

```
17 <70> self
18 <85 00> superSend: a
20 <7C> returnTop
```

Metoda se zasláním zprávy v kontextu předka, která výsledku zašle další zprávu:

```
17 <70> self
18 <85 00> superSend: a
20 <83 00> send: a
22 <7C> returnTop
```

Výsledná metoda celého příkladu. Je vyhodnocena jako blok s jednou simulovanou dočasnou proměnnou, kterou nejdříve naplní objektem v literálu, poté jí zašle zprávu a vrátí výsledek. Všimněte si zbytečné manipulace se zásobníkem. Při simulaci lokálních proměnných se překladač musí snažit udržovat stejný stav zásobníku, jako kdyby šlo o skutečné volání slotů. Důsledkem toho jsou právě zbytečné operace se zásobníkem.

```
45 <70> self
46 <28> pushConstant: a MarvinPrototype
47 <68> popIntoTemp: 0
48 <87> pop
49 <10> pushTemp: 0
50 <87> pop
51 <70> self
52 <87> pop
53 <10> pushTemp: 0
54 <83 02> send: method
56 <7D> blockReturn
```

Lexikální analyzátor

Definice lexikálního analyzátoru jazyka Marvin zapsaná v syntaxi generátoru SmaCC

```
<identifier> :
    [a-zA-Z_] [a-zA-Z0-9_]*
    ;

<parentSlotName> :
    <identifier> \*
    ;

<keyword> :
    <identifier> \:
    ;

<multikeyword>:
    <identifier> \: (<identifier> \: )+
    ;

<argumentName> :
    \: <identifier>
    ;

<operatorChar> :
    \! | \@ | \$ | \% | \& | \* | \- | \+
|   \= | \~ | \/ | \? | \< | \> | \, | \; | \\
    ;

<binaryKeyword> :
    <operatorChar>+
    ;

<character> :
    \$ .
    ;

<base> :
    [0-9]+ r
    ;

<exponent> :
    e [\+ | \-]? [0-9]+
    ;

<integer> :
    <base>? [0-9A-Z]+ <exponent>?
    ;
```

```
<float> :
    <base>? [0-9A-Z]+ \. [0-9A-Z]+ <exponent>?
    ;

<negative> :
    \-
    ;

<number> :
    <negative>? ( <integer> | <float> )
    ;

<resend> :
    resend
    ;

<self> :
    self
    ;

<string>:
    \' [^\']* \' (\' [^\']* \')*
    ;

<symbol> :
    \# <string>
|   \# <identifier>
|   \# <binaryKeyword>
|   \# <keyword>
|   \# <multikeyword>
    ;

<whitespace> :
    \s+
    ;
```

Syntaktický analyzátor

Definice syntaktického analyzátoru jazyka Marvin zapsaná v syntaxi generátoru SmaCC

```
Start :
    Code
    ;

SlotName:
    <identifier>
    ;

ParentSlotName:
    <parentSlotName>
    ;

ParentSlot :
    ParentSlotName
|   ParentSlotName ‘‘=’’ ‘‘[’’ Expression ‘‘]’’
|   ParentSlotName ‘‘<-’’ ‘‘[’’ Expression ‘‘]’’
    ;

DataSlot :
    SlotName
|   SlotName ‘‘=’’ ‘‘[’’ Expression ‘‘]’’
|   SlotName ‘‘<-’’ ‘‘[’’ Expression ‘‘]’’
    ;

UnarySlot :
    SlotName ‘‘=’’ MethodObject
    ;

BinarySlot :
    <binaryKeyword> <identifier> ‘‘=’’ MethodObject
    ;

KeywordSlotDeclaration :
    <keyword> <identifier> KeywordCapPart*
    ;

KeywordSlot:
    KeywordSlotDeclaration ‘‘=’’ MethodObject
    ;

ArgumentSlot :
    <argumentName>
    ;

Slot :
```

```
    ParentSlot
|   DataSlot
|   UnarySlot
|   BinarySlot
|   KeywordSlot
|   ArgumentSlot
    ;

MethodSlot :
    DataSlot
|   ArgumentSlot
    ;

SlotList :
    # epsilon
|   Slot
|   SlotList ``.'' Slot
    ;

MethodSlotList :
    # epsilon
|   MethodSlot
|   MethodSlotList ``.'' MethodSlot
    ;

Slots :
    ``|'' SlotList ``|''
    ;

MethodSlots :
    ``|'' MethodSlotList ``|''
    ;

Block :
    ``['' ``]''
    ``['' MethodSlots ``]''
    ``['' Code ``]''
    ``['' MethodSlots Code ``]''
    ;

RegularObject :
    ``('' ``)''
|   ``('' Slots ``)''
    ;

MethodObject :
    ``('' ``)''
|   ``('' MethodSlots ``)''
```

```
|   ``('' Code ``)''
|   ``('' MethodSlots Code ``)''
    ;

Object:
    RegularObject
|   Block
    ;

Code :
    ExprSequence ``.'' ``^'' Expression ``.''?
|   ``^'' Expression ``.''?
|    ExprSequence ``.''?
    ;

ExprSequence :
    Expression
|   ExprSequence ``.'' Expression
    ;

Expression :
    KeywordMessage
    ;

KeywordCapPart :
    <keyword> BinaryMessage
    ;

KeywordMessage :
    BinaryMessage <keyword> BinaryMessage KeywordCapPart*
|   <resend> <keyword> BinaryMessage  KeywordCapPart*
|   <keyword> BinaryMessage KeywordCapPart*
|   BinaryMessage
    ;

BinaryMessage :
    BinaryMessage <binaryKeyword> UnaryMessage
|   <resend> <binaryKeyword> UnaryMessage
|   <binaryKeyword> UnaryMessage
|   UnaryMessage
    ;


UnaryMessage :
    UnaryMessage <identifier>
|   <resend> <identifier>
|   <identifier>
|   <self>
```

```
|   Primary
    ;

Primary :
    Literal
|   ‘‘(’’ Expression ‘‘)’’
    ;

BaseLiteral:
    <number>
|   <string>
|   <character>
|   <symbol>
|    ‘‘#(’’ Array ‘‘)’’
    ;

Array:
    # epsilon
|   Array ArrayLiteral
    ;

ArraySymbol :
    <identifier>
|   <binaryKeyword>
|   <keyword>
|   <multikeyword>
|   <self>
|   <resend>
    ;

ArrayLiteral:
    <number>
|   <string>
|   <character>
|   <symbol>
|   ArraySymbol
|   RegularObject
|   ‘‘#(’’ Array ‘‘)’’
    ;

Literal :
    BaseLiteral
|   Object
    ;
```

Bytekód Squeaku

Tabulka jednotlivých instrukcí bytekódu Squeak Smalltalku. Vytvořeno na základě zdrojových kódů a komentářů.

0 15 pushReceiverVariableBytecode
- vloží instanční proměnnou na vrchol zásobníku
- 16x, bitAnd: 16rF

16 31 pushTemporaryVariableBytecode
- vloží lokální proměnnou na vrchol zásobníku
- 16x, bitAnd: 16rF

32 63 pushLiteralConstantBytecode
- vloží literál na vrchol zásobníku
- 32x, bitAnd: 16r1F

64 95 pushLiteralVariableBytecode
- vloží na vrchol zásobníku hodnotu asociace uležené v daném literálu
- 32x, bitAnd: 16r1F

96 103 storeAndPopReceiverVariableBytecode
- uloží instanční proměnnou odebranou z vrcholu zásobníku
- 8x, bitAnd: 16r7

104 111 storeAndPopTemporaryVariableBytecode
- uloží lokální proměnnou odebranou z vrcholu zásobníku
- 8x, bitAnd: 16r7

112 pushReceiverBytecode
- uloží na vrchol zásobníku referenci na příjemce

113 pushConstantTrueBytecode
- uloží na vrchol zásobníku objekt true

114 pushConstantFalseBytecode
- uloží na vrchol zásobníku objekt false

115 pushConstantNilBytecode
- uloží na vrchol zásobníku objekt nil

116 pushConstantMinusOneBytecode
- uloží na vrchol zásobníku objekt -1

117 pushConstantZeroBytecode
- uloží na vrchol zásobníku objekt 0

118 pushConstantOneBytecode
- uloží na vrchol zásobníku objekt 1

119 pushConstantTwoBytecode
- uloží na vrchol zásobníku objekt 2

120 returnReceiver
- návrat z volání s referencí na příjemce jako výsledkem

121 returnTrue
- návrat z volání s true jako výsledkem

122 returnFalse
- návrat z volání s false jako výsledkem

123 returnNil
- návrat z volání s níl jako výsledkem

124 returnTopFromMethod
- návrat z volání s vrcholem zásobníku jako výsledkem v kontextu příjemce

125 returnTopFromBlock
- návrat z volání s vrcholem zásobníku jako výsledkem v kontextu volajícího
126 unknownBytecode
- chyba - neznámý bytecode
127 unknownBytecode
- chyba - neznámý bytecode
128 extendedPushBytecode
- rozšířené vkládání na vrchol zásobníku
  1) načte další bytekód (descriptor)
  2) index := descriptor bitAnd: 16r3F (63)
  3) typ := (descriptor >> 6) bitAnd: 16r3
        0 - pushReceiverVariable: (vloží instanční proměnnou)
        1 - pushTemporaryVariable: (vloží lokální proměnnou)
        2 - pushLiteralConstant: (vloží literál)
        3 - pushLiteralVariable: (vloží hodnotu asociace literálu)
129 extendedStoreBytecode
- rozšířené ukládání
  1) načte další bytekód (descriptor)
  2) index := descriptor bitAnd: 16r3F (63)
  3) typ := (descriptor >> 6) bitAnd: 16r3
        0 - uloží instanční proměnnou
        1 - uloží lokální proměnnou
        2 - nelze (uložení literálu)
        3 - uložení hodnoty do asociace literálu
129 extendedStoreBytecode
- rozšířené ukládání
  1) načte další bytekód (descriptor)
  2) index := descriptor bitAnd: 16r3F (63)
  3) typ := (descriptor >> 6) bitAnd: 16r3
        0 - uloží instanční proměnnou
        1 - uloží lokální proměnnou
        2 - nelze (uložení literálu)
        3 - uložení hodnoty do asociace literálu
130 extendedStoreAndPopBytecode)
- rozšířené ukládání s vyzvednutím hodnoty z vrcholu zásobníku
- obdoba extendedStoreBytecode
131 singleExtendedSendBytecode
- jednoduché zaslání zprávy
- může použít pouze prvních 32 literálů jako selektor a nanejvýš 7 argumentů
  1) načte další bytekód (descriptor)
  2) index selektoru := descriptor bitAnd: 16r1F
  3) počet argumentů := descriptor >> 5
132 doubleExtendedDoAnythingBytecode
- rozšířená exekuce
  1) načte další dva bytekódy (byte2 byte3)
  2) typ operace (opType) := byte2 >> 5
        0 - zaslání zprávy
          - index selektoru := byte3

- počet argumentů := byte2 bitAnd: 31
1 - zaslání zprávy v kontextu super
- index selektoru := byte3
- počet argumentů := byte2 bitAnd: 31
2 - vložení instanční proměnné na vrchol zásobníku
- index := byte3
3 - vložení literálu na vrchol zásobníku
- index := byte3
4 - vložení hodnoty asociace literálu na vrchol zásobníku
- index := byte3
5 - uložení vrcholu zásobníku do instanční proměnné
- index := byte3
6 - uložení vrcholu zásobníku s vyzvednutím do instanční proměnné
- index := byte3
7 - uložení vrcholu zásobníku do hodnoty asociace literálu
- index := byte3

133 singleExtendedSuperBytecode
- jednoduché zaslání zprávy v kontextu super
- může použít pouze prvních 32 literálů jako selektor a nanejvýš 7 argumentů
1) načte další bytekód (descriptor)
2) index selektoru := descriptor bitAnd: 16r1F
3) počet argumentů := descriptor >> 5

134 secondExtendedSendBytecode
- jednoduché zaslání zprávy
- může použít prvních 64 literálů jako selektor a nanejvýš 4 argumenty
1) načte další bytekód (descriptor)
2) index selektoru := descriptor bitAnd: 16r3F
3) počet argumentů := descriptor >> 6

135 popStackBytecode
- odebere vrchol zásobníku

136 duplicateTopBytecode
- na vrchol zásobníku ještě jednou přidá hodnotu aktuálního vrcholu

137 pushActiveContextBytecode
- vloží aktivní kontext na vrchol zásobníku

138 143 experimentalBytecode
- v praxi dosud nevyužívané optimalizační bytekódy

144 151 shortUnconditionalJump
- krátký nepodmíněný skok (max. 8 bytekódů)

152 159 shortConditionalJump
- krátký podmíněný skok (ifFalse, max. 8 bytekódů)

160 167 longUnconditionalJump
- dlouhý nepodmíněný skok (až 4*256+další bytekód)

168 171 longJumpIfTrue
- dlouhý podmíněný skok (ifTrue)

172 175 longJumpIfFalse)
- dlouhý podmíněný skok (ifFalse)

176 bytecodePrimAdd

177 bytecodePrimSubtract)

178 bytecodePrimLessThan
179 bytecodePrimGreaterThan
180 bytecodePrimLessOrEqual
181 bytecodePrimGreaterOrEqual
182 bytecodePrimEqual
183 bytecodePrimNotEqual
184 bytecodePrimMultiply
185 bytecodePrimDivide
186 bytecodePrimMod
187 bytecodePrimMakePoint
188 bytecodePrimBitShift
189 bytecodePrimDiv
190 bytecodePrimBitAnd
191 bytecodePrimBitOr)
192 bytecodePrimAt
193 bytecodePrimAtPut
194 bytecodePrimSize
195 bytecodePrimNext
196 bytecodePrimNextPut
197 bytecodePrimAtEnd
198 bytecodePrimEquivalent
199 bytecodePrimClass
200 bytecodePrimBlockCopy
201 bytecodePrimValue
202 bytecodePrimValueWithArg
203 bytecodePrimDo
204 bytecodePrimNew
205 bytecodePrimNewWithArg
206 bytecodePrimPointX
207 bytecodePrimPointY
208 255 sendLiteralSelectorBytecode
- jednoduché zaslání zprávy
- může použít pouze prvních 16 literálů jako selektor a nanejvýš 2 argumenty
1) index selektoru := bitAnd: 16rF
2) počet argumentů := (>> 4) bitAnd: 3) - 1