FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Runtime knihovna interpretu příkazů dclsh
Student:	Martin Tománek
Vedoucí:	Ing. Jiří Kašpar
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství 2021
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Vytvořte runtime knihovnu pro projekt interpretu příkazů dclsh.
Vyjděte z nepublikovaného experimentálního skriptového jazyka TPP (Kašpar, Schmidt, 2001).
Nastudujte koncept rozšiřitelných datových typů (tříd) v TPP a jeho implementaci.
Navrhněte potřebná programová rozhraní a implementujte je v jazyce C.
Pro výslednou sadu funkcí navrhněte a implementujte regresní testy.
Programová rozhraní zdokumentujte v angličtině.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 2. prosince 2023 v Praze.

Bakalářská práce

IMPLEMENTACE RUNTIME KNIHOVNY INTERPRETU PŘÍKAZŮ DCLSH

Martin Tománek

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jiří Kašpar
16. května 2024

Obsah

Seznam obrázků

Seznam tabulek

Chtěl bych poděkovat především svému vedoucímu Ing. Jiřímu Kašparovi za skvělé vedení a poskytnutou podporu. Dále bych rád poděkoval své rodině za jejich neustálou láskyplnou podporu při studiu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 16. května 2024

Abstrakt

Cílem této práce je tvorba runtime knihovny pro projekt interpretu příkazů dclsh. Knihovna je implementovaná v jazyce C a poskytuje objektově orientovaný typový systém s podporou vícenásobného dědění pomocí rozhraní, automatickou správu paměti a standardní knihovnu poskytující kolekce, základní typy a strukturované typy. Součástí práce je také rešerše relevantních skriptovacích jazyků, včetně shellů.

Klíčová slova dclsh, skriptovací programovací jazyk, shell, automatická správa paměti, objektově orientovaný typový systém, C

Abstract

The aim of this thesis is to create a runtime library for the dclsh command interpreter project. The library is implemented in C and provides an object-oriented type system with support for multiple inheritance through interfaces, automatic memory management, and a standard library offering collections, basic types, and structured types. The thesis also includes a survey of relevant scripting languages, including shells.

Keywords DCLsh, scripting programming language, shell, automatic memory management, object-oriented type system, C

Seznam zkratek

DCL	DIGITAL Command Language
VMT	Virtual Method Table
IMT	Interface Method Table
GC	Garbage Collector
API	Application Programming Interface
GC	Garbage Collector
DLL	Dynamic Link Library

Úvod

Operační systémy jsou zásadní součástí výpočetních technologií. Jeden z hlavních způsobů komunikace programů a lidí s UNIX operačními systémy je za pomocí shellu. Práce s shellem ale nemusí být vždy přívětivá, bezpečná nebo efektivní. Psaní skriptů správně je obtížné – programátor musí chápat přesné chování spojovacích operátorů, jakou roli hraje při výpočtu prostředí a navíc stav interpretu shellu. Při psaní shell skriptů je často obtížné se zotavit z chyby a je i možné se dostat do stavu, ze kterého není jednoduché se navrátit. Skripty se skládají ze spojování jednoduchých příkazů, bohužel většina těchto příkazů neškáluje v paralelním kontextu. Shell také při mnoha operacích spouští podprogramy a subshelly, které můžou mít nemalou režii.[1]

Dclsh shell je interpretovaný programovací jazyk, který má za cíl poskytnout služby shellu a vyhnout se jeho nedostatkům. Hlavní inspirace pochází ze skriptovacího jazyka DIGITAL Command Language (DCL) a TPP. Popis výpočtu je pomocí objektově orientovaných konstruktů, zpracování chyb využívá výjimkový aparát a paralelní výpočty se provádí ve vláknech.

Celou implementaci jazyka je možné rozdělit na runtime část, poskytující rozhraní pro reprezentaci výpočtu, a interpretovou část, která pracuje se zdrojovým kódem a interpretuje ho.

Cíl této práce je implementovat runtime část jazyka dclsh a navrhnout rozhraní pro zpřístupnění implementovaných funkčností interpretu.

Začneme popisem konceptů programovacích jazyků, rešerší skriptovacích jazyků, včetně vybraných shellů a poté popíšeme jazyk dclsh a implementaci jeho runtime části.

Pro jazyk budeme muset navrhnout a implementovat automatickou správu paměti, vlastní alokátor, rozšiřitelný objektově orientovaný typový systém, umožňující definování tříd a rozhraní, které mohou být i generické, standardní knihovnu, obsahující základní datové typy a kolekce, a nakonec výjimkový aparát. Jednotlivé funkčnosti budou otestovány jednotkovými testy.

Popíšeme také souhru interpretu s runtime částí. Dané interakce budou také doprovázeny regresními testy, které v tomto kontextu poslouží jako dokumentační prvek popisované souhry.

Počítáme s tím, že na práci bude v budoucnu navázáno vytvořením interpretové části.

Cíle

Hlavním cílem práce je vytvořit runtime část jazyka dclsh tak, aby se na ní dalo navázat implementací interpretu.

1. Provést rešerši skriptovacích jazyků, včetně shellů a jazyka TPP, ze kterého jazyk dclsh vychází
2. Popsat koncepty a funkčnosti jazyka dclsh
3. Navrhnout a implementovat runtime knihovnu pro jazyk dlcsh.
4. Otestovat a zdokumentovat implementované řešení.

Kapitola 1

Koncepty programovacích jazyků

V této kapitole položíme teoretický a terminologický základ na téma programovacích jazyků v rozsahu potřebném pro zbytek této práce. Definujeme programovací jazyk, vysvětlíme kategorizaci programovacích jazyků, co je to koncept programovacího jazyku a popíšeme vybrané koncepty programovacích jazyků.

1.1 Kategorizace programovacích jazyků

Programovací jazyk je definován jako systém pravidel pro manipulaci dat pro popis výpočetu[2]. V této sekci definujeme kategorizaci programovacích jazyků, která bude použita v dalším textu.

1.1.1 Imperativní a deklarativní jazyky

Jeden z hlavních způsobů dělení programovacích jazyků je na *deklarativní jazyky* a *imperativní jazyky*. Imperativní jazyky dovolují popsat výpočet krok za krokem, zatímco deklarativní jazyky umožňují *deklarovat* co chceme vypočítat, bez specifikace *jak* přesně chceme daný výpočet provést.[3]

1.1.2 Styly programování

Jazyky je možné kategorizovat podle podporovaných stylů programování[2]. Pro potřeby tohoto textu definujeme imperativní, funkcionální a objektově orientovaný programovací programovací styl.

1.1.2.1 Imperativní programovací styl

Imperativní styl používá k popisu výpočtu sekvenci příkazů, které modifikují hodnoty pomocí vedlejších efektů, především přiřazením hodnoty do proměnné. [2]

Imperativní styl je do určité míry abstrakcí nad *von Neumannovou* architekturou počítače. Dva zásadní komponenty této architektury je paměť, která uchovává jak instrukce, tak data a dále procesor, který poskytuje operace umožňující měnit obsah dané paměti. Proměnná je potom abstrakce jazyka za paměťovou buňku počítače. [4]

1.1.2.2 Funkcionální programovací styl

Ve funkcionálním stylu se výpočet popisuje primárně voláním řady funkcí, které si navzájem vracejí návratové hodnoty. Čistě funkcionální styl nevyužívá proměnné, přiřazovací operátor, vedlejší efekty a příkazy, tedy výpočet je definován pomocí výrazů. Pro repetici se primárně používá rekurze. Dalším znakem je možnost přiřadit funkci do proměnné, předat ji jako argument jiné funkci a navrátit funkci jako výsledek jiné funkce, tyto funkce nazýváme *funkce vyššího řádu*. [2]

1.1.2.3 Objektově orientovaný programovací styl

Výpočet v objektově orientovaném programování spočívá v interakci mezi samostatnými objekty, které si individuálně obhospodařují svůj stav a interagují spolu pomocí metod. Objektově orientované programování nabízí abstrakci ve formě tříd, které za jménem svého typu a rozhraním jejich metod schovávají potenciálně komplikované implementační detaily a výpočty. Toto schovávání detailů implementace budeme nazývat jako *enkapsulaci*. Třídy avšak navíc umožňují jednoduchou rozšiřitelnost poskytnuté abstrakce pomocí *dědičnosti* a díky funkci *dynamického výběru implementace metody* (anglicky *dynamic method dispach*) mohou rozšiřující třídy upřesnit chování abstrakce, i v kontextu, kdy byla očekávaná nerozšířená verze. [3]

Podrobněji je objektově orientovaný styl popsaný v kapitole 1.2.5.

1.1.2.4 Průnik programovacích stylů

Jednotlivé programovací styly se avšak vzájemně nevylučují. Jazyky podporující více programovacích stylů jsou v převaze oproti jazykům podporující pouze jeden styl programování.[2]

Imperativní programovací jazyky typicky podporovali jen omezenou formu funkcionálního programování. Nejdůležitější limitující faktor byla chybějící podpora funkcí vyšších řádů v imperativních jazycích. [4]

Mnoho objektově orientovaných jazyků, například Java a C#, vychází z imperativního programovacího stylu. [4] Dále tyto jazyky jsou schopné podporovat i funkcionální programovací styl. Například v jazyku Java je možné definovat generické rozhraní, které popisuje typ funkce a poté vytvořit funkci jako instanci třídy, která implementuje dané rozhraní. Funkce z typu `int` do typu `boolean` je tedy možné vytvořit jako anonymní vnitřní třídu implementující rozhraní `Func1<Integer, Boolean>`. [5]

1.1.3 Kompilace a interpretace jazyků

Programovací jazyk je možné implementovat třemi způsoby: kompilací, čistou interpretací a hybridní implementací. Kompilační přístup používá *kompilátor*, což je program, který překládá programy napsané v vysokoúrovňovém jazyku do strojového kódu. Čistě interpretované jazyky se nepřekládají, místo toho jsou ve své původní formě interpretovány programem, který nazýváme softwarový interpret (hardwarový interpret je potom procesor samotný[2]). Systémy s hybridní implementací překládají program napsaný v vysokoúrovňovém jazyku do mezitvaru,

který je poté interpretován. Typicky jsou hybridní systémy pomalejší než kompilované, možné urychlení hybridních systému je použitím `Just-in-Time` kompilátoru, který je schopný přeložit daný mezitvar přímo do strojového kódu, tedy převádí hybridní systém do systému s odloženou kompilací. Použití `Just-in-Time` kompilátorů je rozšířené například v Java a .NET ekosystému. [4]

1.2 Koncepty programovacích jazyků a jejich implementace

Koncept programovacího jazyku, například předávání parametrů funkci nebo typování, definujeme typicky jako univerzální princip, podle kterého se jednotlivé jazyky liší v implementaci daného principu. [2] Typicky když někomu představujeme nový jazyk, instinktivně začneme popisem konceptů, které daný jazyk implementuje a jak jsou implementovány.

1.2.1 Jména, vázání a rozsah platnosti

Vázání (anglicky *binding*) je asociace jedné entity s jinou entitou, například proměnou lze svázat s datovým typem. Rozlišujeme dynamické a statické vázání – statické probíhá před během programu a dynamické za běhu programu.[5]

Proměnnou je možné buďto deklarovat, nebo ji referencovat. Pravidla definující rozsah platnosti proměnných popisují k jakým deklaracím je reference na proměnnou navázána, proměnné pak náleží do určitých oblastí platnosti proměnných (anglicky *scope*). Je-li rozsah platnosti proměnné určen staticky, tak byly reference navázány na deklarace před během programu. Je-li rozsah platnosti proměnné určen dynamicky, tak byly reference navázány na deklarace za běhu programu.[5]

Blok kódu je syntaktická jednotka, jejíž začátek a konec je jasně definován, např. v jazyku C definujeme blok pomocí složených závorek . Proměnné v jazycích, které využívají bloky, mají typicky lexikální rozsah platnosti, což je druh statického rozsahu platnosti. Chceme-li zjistit, ke které deklaraci je reference proměnné vázána, prohledáme bloky od nejvnitřnějšího a hledáme první odpovídající deklaraci.[2]

V jazycích používající dynamický rozsah platnosti jsou proměnné vázány v závislosti na toku výpočtu za běhu programu. Daná reference je navázána na deklaraci proměnné, která se vyskytla naposledy během provádění a ještě nebyla uvolněna návratem z podprogramu.[3] Pro implementaci dynamického určování rozsahu proměnných stačí prohledávat zásobník volání za běhu z vrchu směrem dolů[2].

1.2.2 Typové systémy

V jazycích se statickou typovou kontrolou jsou typy kontrolovány a typové chyby detekovány před během programu. V dynamicky typovaných jazycích jsou typové kontroly a typové chyby detekovány až za běhu programu. [2]

Typ definujeme jako množinu hodnot a přípustných operacích na daných hodnotách. Při typové kontrole se zjišťuje, zda-li jsou hodnoty typů a operace na daných typech přípustné. [2]

Dovoluje-li jazyk explicitně porušit typové omezení, tak říkáme, že je *slabě typovaný*, naopak jazyk který typové omezení nedovoluje obejít, nazýváme *silně*

typovaným. [2]

Jazyky dále mohou být explicitně typované, tedy vyžadují u každé proměnné deklarovaný typ, nebo implicitně typované, u kterých se typ uvádět nemusí. [2]

1.2.2.1 Typová konverze

Konverze je změna typu jedné hodnoty na typ druhý. Konverze může být buďto implicitní, nebo explicitní. Implicitní konverze se provádí automaticky v různých případech (v závislosti na jazyku, operačním systému a jiných faktorech) bez zásahu programátora. Explicitní typovou konverzi je možné provést buďto konverzní funkcí, nebo za pomocí *type casting*, přičemž dojde k reinterpretaci bitů jedné hodnoty na jiný typ.[2]

1.2.2.2 Polymorfismus

Část kódu (data, struktury, procedury) je polymorfní, pokud je schopná pracovat s vícero typy, které mezi sebou mají nějaké společné charakteristiky. [3]

Při parametrickém polymorfismu jsou předávány typy, buďto explicitně nebo implicitně, jako parametry. Při podtypovém polymorfismu je část kódu schopná pracovat s typem explicitně definovaným typem `T` a také s jakýmkoliv podtypem tohoto typu. [3]

Staticky typované objektově orientované jazyky typicky podporují podtypový polymorfismus a explicitní parametrický polymorfismus implementují pomocí generik. Dynamicky typované jazyky implementují oba mechanismy za pomocí odložení kontroly typu až do doby běhu programu. [3]

1.2.3 Kontrola toku výpočtu

Na úrovni výrazů je tok výpočtu definován operátorovou *precedencí*, která určuje prioritu mezi operátory, a operátorovou *asociativitou*, která v případě rovné priority operátorů rozhoduje zda-li se provede první levý nebo pravý operátor.[4]

Na úrovni příkazů rozlišujeme mezi podmíněné příkazy a cykly. Podmíněné příkazy nám dovolují rozdělit tok programu na dvě a více cest, například příkazy `if-then-else` a `switch-case`. Cykly nám dovolují opakovat provedení kusů kódu, například príkazy `for` a `while`. [4]

1.2.3.1 Popis chybného tok výpočtu pomocí výjimek

Výjimečné situace a chyby se dají řešit mnoha způsoby. Bez jazykové podpory pro výjimky může být kód řešící chyby značně nepřehledný – často dochází k duplikaci, které je náchylné k chybám. Výjimkový aparát jasně odděluje chybový tok výpočtu od normálního. [4]

Je-li vyhozena výjimka, je možné se ji pokusit ošetřit *nějakým handlerem* a pokračovat *někde* dále ve výpočtu nebo program ukončit. Výběr *jak* výjimky reprezentovat, *jaký* handler použít a *kde* pokračovat dále ve výpočtu v případě úspěšného ošetření a *v jakém* případě program ukončit je na jazyku samotném. [4] Specifikům vybraných programovacích jazyků na toto téma se budeme více věnovat v kapitole popisující jazyk dclsh 3 a v rešerši 2.

1.2.4 Podprogramy

Podprogram je konstrukt pro abstrakci výpočtu – umožňuje asociovat jméno s potenciálně komplikovanou operací. Místo plného popisu výpočtu je pak tedy možné pouze *zavolat* podrogram pomocí jeho jména a případných parametrů. Volající pak čeká na konec volaného podprogramu před pokračováním se svým vlastním výpočtem. Podprogramy rozdělujeme na procedůry a funkce – funkce vrací volajícímu po skončení výpočtu nějakou hodnotu, zatímco procedury žádnou hodnotu nevrací. Funkce, které nemají žádné vedlejší efekty nazýváme *čisté funkce*. [4]

1.2.4.1 Předávání parametrů

Formální parametry nazýváme parametry v hlavičce podprogramu, se kterými podprogram pracuje jako s lokálními proměnnými. Skutečné parametry jsou proměnné nebo výrazy dosazeny při volání funkce. [4]

Jsou-li parametry předány hodnotou (anglicky *pass-by-value*), tak se skutečné parametry použijí k inicializaci formálních parametrů, typicky kopírováním. Dále je možné ještě možnost předat parametry pomocí *pass-by-result*, kde se formální parametry neinicialuzují a po návratu z podprogramu se jejich hodnoty předají zpátky do volajícího, je proto nutné, aby v místě volání byly dosazeny proměnné a ne výrazy. Kombinací předávání hodnotou a *pass-by-result* je *pass-by-value-result*, která se od *pass-by-result* liší v tom, že se formální parametry inicializují hodnotami v skutečných parametrech. Při předávání parametru referencí (*pass-by-reference*) se předává typicky jen adresa dané hodnoty nebo tzv. *handle* na předávanou hodnotu. [4]

1.2.5 Koncepty objektově orientovaného stylu programování

Objektově orientovaný programovací styl byl zmíněn v kapitole popisující kategorizující jazyky 1.1.2.3, ve které jsme představili třídu a tři její definující vlastnosti: enkapsulace, dědičnost a dynamický výběr implementace metody. Podprogramy uvnitř třídy se typicky nazývají *metody* a proměnné uvnitř třídy *atributy*, dohromady tvoří *členy* třídy. [3]

Pro enkapsulaci členů se používají *modifikátory přístupu* umožňující definovat viditelnost členů, například můžeme chtít, aby byl člen viditelný pouze ve třídě, ve které byl deklarován, nebo můžeme chtít, aby byl viditelný všude, kde je třída importována. Jednotlivé jazyky definují tyto modifikátory odlišně. [3]

Pro inicializaci objektů se používají speciální podprogramy zvané *konstruktory*. Typicky je možné definovat více konstruktorů, které se od sebe liší typem a počtem parametrů. Nějaké jazyky podporují i *destruktory*, které se volají při zániku objektu. [3]

Většina objektově orientovaných jazyků dovoluje definici *třídních* (také známých jako *statických*) členů, ve smyslu že nenáleží jednotlivým instancím, ale náleží třídě samotné, tedy všechny instance sdílí jednu kopii tohoto členu.[3]

Koncept *abstraktní metody*, tedy metody, které chybí tělo, neboli definice, lze nalézt v mnoha objektově orientovaných jazycích. Třídu nazýváme *abstraktní*, jestli má alespoň jednu *abstraktní metodu*, neabstraktní třídy nazýváme *konkrétní*. Jelikož abstraktní třída není plně definovaná, není ji možné instanciovat, tedy slouží

pouze jako základ pro ostatní třídy, které její chybějící dílky musí při dědění doimplementovat, nebo být také abstraktní. [3]

Procesem dědění je možné vytvořit nový podtyp, který přejímá všechny členy od svých nadtypů. Obecná mnohonásobná dědičnost (přítomná například v jazyku C++) je avšak problematická. Sporná situace vzniká, když se dědí ze dvou nadtypů se stejně pojmenovanými členy. Jaký z nich se má zdědit? Měli by být dostupní oba dva? Jak je rozlišit? Další pochybnosti nastávají v případě, když dva děděné nadtypy mají jeden společný nadtyp. Měli bychom mít společné členy dvakrát nebo jen jednou? Kvůli této složitosti je často volen kompromis v podobě *mix-in inheritance*. Základní princip *mix-in* dědičnosti spočívá v povolování pouze jednoduché dědičnosti mezi třídami a navíc poskytování konstruktu *rozhraní*, který se chová jako (abstraktní) třída, ale jako členy může mít pouze abstraktní metody. Rozhraní mohou dědit mezi sebou dědit neomezeně, třídy mohou také dědit od neomezeného množství rozhraní, ale aby mohli být konkrétní, musí zděděné abstraktní členy doimplementovat. Moderní implementace rozhraní dovolují definovat i těla metod.[3]

1.2.6 Souběžný výpočet

Souběžným výpočtem rozumíme situaci, kdy je v jeden okamžik prováděno více výpočtů zároveň, kde každý výpočet má svůj vlastní kontext. Souběžný systém je *paralelní*, pokud se hardwarově opravdu provádí výpočet více úkolů najednou, tedy při souběžný výpočet není nutné mít více jak jeden proces, ale pro realizaci paralelního výpočtu musíme mít procesorů více. [3]

Souběžnost a paralelnost je možné využít k návrhu programů, které jdou přirozeně rozdělit na více oddělených činností, například weboví prohlížeči zároveň vykreslují stránky, komunikují se serverem a reagují na interakci od uživatele, nebo webové servery typicky ke každému požadavku přiřadí samostatné vlákno. Paralelní výpočet je využitelný ke zrychlení výpočtu, pokud je možné výpočet rozdělit na více podúkolů, které jdou poté spojit, zároveň je paralelnost a souběžnost nutná pro distribuovaný výpočet na fyzicky oddělených zařízení. [3]

1.2.7 Správa paměti

Je-li možné předurčit dobu životnosti hodnot, tak si typicky vystačíme s alokací na *zásobníku*. V případě, kdy chceme avšak použít hodnoty, u kterých neznáme jejich životnost, například dynamické datové struktury typu list a strom, funkční hodnoty a podobně, tak si s zásobníkem nevystačíme, musíme použít jinou strukturu – *haldu*. Haldu je možné obhospodařovat manuálně nebo automaticky. Manuální správa paměti může být avšak značně komplikovaná a náchylná k chybovosti. [5]

1.2.7.1 Automatická správa paměti

Mluvíme-li o automatické správě paměti (anglicky *garbage collection*), myslíme tím mechanismus automatického uvolňování paměti, uložené na haldě, která již není používána. Data nejsou používány, pokud nejsou dosažitelné z proměnných, které jsou aktivní (tedy součástí běžících funkcí nebo metod), poté mohou být bezpečně uvolněny. [5]

Jedna implementace automatické správy paměti je pomocí počítání referencí. Každému objektu se musí přidat navíc čítač, reprezentující počet referencí směřovaných na daný objekt, který je při vytvoření objektu nastavený na nulu a při vytvoření reference na objekt se zvedá a při odstranění reference na objekt se snižuje (v tomto kontextu je objekt *něco*, na co se dá navázat jméno). Dostane-li se čítač odstraněním poslední reference na nulu, tak se musí všechny případné reference držené daným objektem také snížit a objekt samotný se může uvolnit. [5]

Další častá implementace je tzv. *mark-and-sweep*. V *mark* fázi se označí všechny dostupné bloky, v *sweep* fázi se potom neoznačené bloky uvolní. Typicky obě fáze trvají značnou dobu, protože se musí projít všechny alokované objekty a dostupné dokonce dvakrát. Tedy spouštění *garbage collectoru* zapříčiní dlouhou pauzu ve výpočtu, které může být nepříjemná v interaktivních aplikacích a nebo katastrofická v aplikacích reálného času. [5]

Kapitola 2

Rešerše skriptovacích jazyků

V této kapitole popíšeme vybrané skriptovací jazyky, včetně shellů. Zaměříme se na implementaci sdílených konceptů a funkčností.

2.1 Přehled skriptovacích jazyků

Skriptovací jazyky jsou rozmanité a mají širokou škálu různorodých využití. Tradiční jazyky (například C, Java, C#) dávají převážně důraz především na efektivitu, udržovatelnost a statickou detekci chyb, oproti skriptovacím jazykům, které typicky zdůrazňují dynamické kontroly, snadnost vyjádření výpočtu a rychlost vývoje. Skriptovací jazyky lze najít na mnoha místech – třeba na terminálu najdeme shelly, dále exitují speciální matematické a statistické jazyk, jako například Matlab nebo Mathematica, všeobecně použitelné skriptovací jazyky, například Python nebo Perl, jazyky používány převážně na webu, například JavaScript nebo PHP. Využít se dají i k přizpůsobení nebo rozšíření funkčnosti skriptovatelných systémů, jako jsou editory, tabulkové procesory a video hry.[3]

Bylo by možné jmenovat několik společných vlastností napříč skriptovacími jazyky, například jednoduché vyjadřování a minimalizace šablonovitého kódu, nepovinné deklarace proměnných, jednoduchý přístup k funkčnostem operačního systému, dynamické typování a vysokoúrovňové datové typy, například množiny, listy a n-tice, jako součást syntaxe a sémantiky jazyka samotného.[3]

Mezi historicky první skriptovací jazyky můžeme zařadit shelly, neboli interprety příkazové řádky, například `bash` na UNIX operačních systémech nebo `command` na operačních systémech Windows, popřípadě `DCL` mimo jiná na platformu OpenVMS[3]. Mezi jejich funkčnosti patří spouštění programů, práce se systémem souborů, přeposílání vstupů ze souborů a výstupů do souborů a vytváření tzv. rour (anglicky *pipeline*), které napojují standardní výstup jednoho programu na standardní vstup programu druhého[6]. Navíc shelly typicky nabízí možnost použít proměnné, konstrukty pro kontrolu toku výpočtu a možnost definovat podprogramy[7]. Obecně shelly poskytujou jednoduchou kontrolu nad operačním systémem a schopnost poslepovat vícero programů dohromady v typicky imperativním programovacím stylu. Shelly se typicky dají použít interaktivně na terminálu, kdy přijímají vstup z klávesnice, nebo dávkově pomocí skriptů[3].

Víceúčelové skriptovací jazyky jsou tradičně interpretované jazyky podporující

imperativní programovací styl, příchodem éry webu se avšak skriptovací jazyky vyvinuly do jazyků podporující více programovacích stylů (např. jazyky JavaScript, Python, PHP podporují navíc jak objektově orientované, tak funkcionální styly). [2]

2.2 Výběr skriptovacích jazyků pro rešerši

Jazyk dclsh má za cíl nabídnout funkčnosti shellu, tedy v rešerši se podíváme na bash a DCL. Dále vychází návrh jazyku dclsh z jazyku TPP, který také zanalyzujeme. Nakonec se podíváme na populární skriptovací jazyky – Python, PHP a JavaScript [8] (popularita byla měřena dle počtu dotazů na vyhledávači Google), které s jazykem dclsh sdílí mnohé rysy, jako například jsou primárně interpretované, poskytují rozšiřitelné typy, jsou dynamicky typované, většinou mají dynamický rozsah proměnných, automaticky spravují svou paměť (převážně pomocí počítání referencí), poskytují možnost oddělit chybový výpočet od normálního výpočtu pomocí výjimek, také se jednotlivé jazyky odlišně staví k objektově orientovanému stylu programování.

2.3 Bash

Shell bash je vytvořený pro operační systémy GNU, ve kterých je to výchozí shell. Vychází z shellů Bourne shell, Korn Shell a C-shell. Bash je v souladu s POSIX specifikací standardního Unix shellu.[7] Krátká ukázka je dostupná v příloze A.1.

Výpočet v shellu probíhá následovně: přečtený vstup je rozdělený na jednotlivá slova a operátory, které se následně syntakticky zanalyzují, pak se provedou jednotlivé expanze symbolů, následně se nastaví případné přesměrování, nakonec se provede výpočet a vrátí se návratový kód. Návratové kódy jsou čísla v rozmezí 0-255 (velikost 1B), nulový návratový kód značí úspěch příkazu, nenulový návratový kód neúspěch příkazu, popřípadě hodnota kódu při nenulové hodnotě specifikuje typ chyby.[7]

Proměnné jsou dynamicky a implicitně typované a nemusí být před použitím deklarované a jejich platnost je pouze v aktuálním shellu. Podporované typy jsou řetězce, numerické hodnoty, pole, asociativní pole, reference, také se dá zakázat zapisování do proměnné (tedy read-only).[7]

Určité znaky vyvolávají expanzi jednotlivých slov, tedy nahradí se za jedno nebo více slov dalších, například použité proměnné se nahradí svou hodnotou nebo aritmetické výrazy se vyhodnotí pomocí aritmetické expanze.[7]

Příkazy jsou čteny ze standardního vstupu a výsledky výpočtu se posílají standardní výstup, bez konfigurace je standardní výstup a vstup terminál. Bash dovoluje tyto standardní vstupy a výstupy programů jednoduše přesměrovávat buď to mezi sebou pomocí rour nebo ze soborů a do souborů.[7]

Skripty mají přístup k pozičním parametrům ve formě číslovaných proměnných, začínající od nuly (například `$1, $2`, a k dalším speciálním proměnám například proměnná s číslem procesu (`$$`) nebo počtem pozičních argumentů (`$#`).[7]

Bash a Unixové shelly obecně jsou skvělé pro kompozici programů, roury dobře popisují souběžný výpočet pomocí datových proudů (anlicky *stream*), a navíc je možné je použít i v interaktivním režimu. Na druhou stranu je výpočet v shellu

doprovázen globálním stavem, který zabraňuje jakékoliv pokročilé statické analýze kódu. Vysoká interaktivita shellu je doprovázena stručnou syntaxí a shell má přístup k celému systému, což může vést na situace, kdy jeden překlep může až potenciál vymazat celé pevné disky. Také chybí podpora ochranných mechanismů pro zpracování chyb.[1]

2.4 DCL

DCL, neboli DIGITAL Command Language, je shell dostupný, mimo jiné, na operačních systémech OpenVMS. DCL interpretuje příkazy buďto rovnou z příkazové řádky nebo ze souborů v podobě skriptů.[9]

Proměnné mohou uchovávat buďto řetězcové hodnoty typu STRING, nebo celočíselné hodnoty typu INTEGER (32 bitů). Logické hodnoty jsou interpretované z typů STRING a INTEGER – TRUE je buďto lichý INTEGER, STRING začínající znakem `t/T/y/Y` a nebo STRING reprezentující lichý INTEGER, zatímco FALSE je reprezentovaný jako doplněk TRUE. Pro práci s hodnoty nabízí DCL operátory, např. `+, -`, a lexikální funkce, např. `F$length(STRING)`. Absolutní čas je reprezentován jako STRING ve formátu `[dd-mmm-yyyy][:hh:mm:ss.cc]`, například `11-DEC-2002:13`. Pro reprezentaci časového intervalu (delta time), je použit opět STRING ve formátu `"+[dddd-][hh:mm:ss.cc]"`, např. `"+3-"` jsou 3 dny.[10]

DCL rozlišuje mezi *procedury* a *podprogramy*. Procedury jsou volané uvnitř skriptu pomocí `@` a názvu příkazu, podle kterého se najde a vykoná chtěný `.COM` soubor se skriptem. Podprogramy jsou volané uvnitř jednoho skriptu pomocí příkazu `CALL`.[9]

Při volání jak procedur, tak podprogramů se vytvoří nová tabulka proměnných. Každý proces má ještě jednu globální tabulku proměnných. Proměnné se hledají v hierarchii tabulek od lokální, přes všechny nelokální až po globální. Případné parametry se přiřadí do pozičních proměnných P1-P8. Existuje ještě způsob jak zavolat podprogram pomocí `GOSUB`, ale při tomhle volání se nevytvoří nová tabulka proměnných.[9]

Pole je možné v DCL implementovat vícero způsoby – například pomocí substituce nebo expanze. Řídící struktury a cykly se implementují pomocí návěští, podmínek a skoků `GOTO`. Rourování a přesměrování je pomocí příkazu `pipe` v kombinaci s oddělovači (například `|`, `<`), které se příkazu předávají formou parametrů.[9]

K jednotlivým slovům na řádce lze doplnit extra kvalifikátory (přepínače) pomocí `/` syntaxe. Kvalifikátory je možné specifikovat u příkazů, u parametrů. Dokonce některé kvalifikátory jsou poziční, tedy mění chování příkazu podle jejich umístění, jsou-li umístěny před parametry, tak ovlivní všechny parametry, pokud jsou ale umístěny za parametr, tak ovlivní pouze ten parametr.[10]

DCL rozlišuje mezi 5 úrovněmi závažnosti při chybě – severe error, error, warning, informational a success. Výchozí chování interpretu při zjištění chyby závažnosti error nebo severe error je ukončit současnou proceduru. Syntax `ON condition THEN [$] command` umožňuje nastavit pro skript vlastní *error handler*, který zachytává chyby podle závažností error, severe error a warning. Například chceme-li ukončit proceduru i při pouhém warning, můžeme nastavit `$ ON WARNING THEN EXIT`.[10]

Ukázky probraných konceptů jsou dostupné v příloze A.2.

2.4.1 CDL

CDL, potažmo Command Definition Language, je jazyk pro popis příkazů, typicky jsou tyto popisy uloženy v souborech s příponou `.CLD`.[11]

CDL umožňuje definovat kvalifikátory a parametry pro jednotlivé příkazy pomocí `QUALIFIER` a `PARAMETER` syntaxe. Hodnoty (například argumenty kvalifikátorů nebo parametrů) lze také popsat – přesněji jejich povinnost a typ. Popřípadě očekáváme-li argument zadaný jako list čárkou oddělených hodnot, můžeme toto specifikovat `LIST` syntaxe.

Zabudované typy hodnot jsou následující (DCL také dovoluje dodefinovat dodatečné vlastní typy):

1. `$ACL`, access control list – používaný pro určování kdo má přístup k souborům.
2. `$DATETIME`, absolutní čas nebo absolutní čas +/- časový interval.
3. `$DELTATIME`, časový interval.
4. `$EXPRESSION`, výraz, který je vyhodnocen a předán je jeho výsledek. `$FILE`, platný identifikátor souboru. `$INFILE`, již existující soubor. `$DIRECTORY`, adresář. `$NUMBER`, celé číslo zadané v buďto decimální, osmičkové nebo hexadecimální soustavě. `$PARENTHESIZED_VALUE`, hodnota musí být uzávorkovaná, předává se i včetně závorek. `$QUOTED_STRING`, řetězec ohraničený uvozovkami, uvozovky se předávají. `$REST_OF_LINE`, předá se doslovný zbytek řádku jako řetězec.

[11]

CDL nabízí i způsob jak zakázat určité kombinace použití příkazu pomocí `DISALLOW` syntaxe. Například můžeme zakázat použití kvalifikátoru `Q` spolu s prvním parametrem `P1`, který má hodnotu `RED`.[11]

Další užitečná funkce je definice vícero významů pro jeden příkaz pomocí `DEFINE SYNTAX` syntaxe. Například je-li specifikován kvalifikátor `LINE`, tak se příkaz může chovat jinak než bez něj. Obdobně funguje příkaz `mv` na UNIX shellech, kde v závislosti na přepínačích a parametrech může `mv` přejmenovávat nebo přemisťovat soubory.[11]

2.5 TPP

Jazyk TPP je experimentální interpretovaný skriptovací jazyk, vytvořený Ing. Jiřím Kašparem (vedoucí této práce) a doc. Ing. Janem Schmidtem, Ph.D, kolem roku 2000. Dokumentace [12] a zdrojové kódy v jazyku C++ [13] byly poskytnuté vedoucím této práce. Návrh jazyku dclsh vychází z jazyku TPP.

Všechny informace v této sekci pochází z dokumentace jazyka [12].

2.5.1 Typový systém

Jazyk TPP je objektově orientovaný a podporuje jednoduchou dědičnost. Nejsou podporováni statičtí ani abstraktní členové, potažmo třídy. Další funkčnost je

možné vytvářet pomocí funkcí, které náleží do jednotlivých modulů, potažmo zdrojových souborů. Není avšak možné nikde explicitně deklarovat typ u proměnných.

Proměnná do které nebyla přiřazena hodnota má typ `UNDEF`, tedy neplatná hodnota, která se rovná pouze jiné neplatné hodnotě.

Typy v standardní knihovně jsou rozšiřitelné pomocí C/C++ rozhraní. Při rozšiřování třídy se vytvoří nová třída se stejnou množinou hodnot, která může vyměnit definici metod rozšířené třídy, nevyměněné metody se dědí.

Typy se na sebe mohou konvertovat, každá třída má jak příchozí a odchozí konverzní metodu, potřebujeme-li konverzi z typu `A` do typu `B` a je definovaná jak příchozí konverzní metoda na `A` z `B`, tak odchozí konverze z `B` do `A`, tak má přednost odchozí konverze. Implicitní konveze se provádí v podmíněných příkazech (konverze na třídu `BOOL`),

Operátory jsou speciální metody, tedy náleží třídám. Při binárních operacích se vybírá operátor levého operandu.

2.5.2 Reprezentace výpočtu

Specificky jazyk TPP definuje různé *jednoduché jmenné prostory* (sémanticky oblasti platnosti proměnných), například lokální, systémový, výjimkový nebo globální. Jmenné prostory obsahují jak pojmenované proměnné, tak poziční proměnné, každá proměnná je součástí právě jednoho jmenného prostoru. Při referování proměnné jsou jmenné prostory postupně prohledávány. Při deklaraci proměnné je možné uvést do jakého jmenného prostoru ji chceme vložit (implicitně je vložena do lokálního jmenného prostoru).

Jazyk TPP umožňuje také použít *permanentní* proměnné, které se uloží a zůstávají i po ukončení chodu interpretu, jsou totiž ukládány ve souborech na cestě dané proměnnými `EXTPATH` (definuje adresáře) a `EXTEXT` (definuje koncovky chtěných souborů v adresáři).

Do proměnné je také možné přiřadit řetězcovou hodnotu třídy `STR`, kterou je možné interpretovat jako spustitelnou funkci napsanou v jazyce TPP. Jinak mají funkce hodnotu třídy `function`.

2.5.3 Automatická správa paměti

Paměť je automaticky spravovaná pomocí počítání referencí.

2.5.4 Výjimkový aparát

Výjimka samotná nemá hodnotu v jazyku TPP, ale při vyvolání se vytvoří jmenný prostor výjimky, který pak zaniká po ošetření výjimky. V tomto jmenném prostoru jsou proměnné popsané v tabulce 2.1. Poziční parametry z lokálního jmenného prostoru jsou zastíněny (anglicky *shadowed*) pozičními parametry jmenného prostoru výjimky a tedy nedostupné.

2.5.5 Standardní knihovna

Standardní knihovna obsahuje neplatný typ `UNDEF`, elementární typy pro logickou hodnotu `BOOL`, řetězcové hodnoty `STR`, celočíslené hodnoty se znaménkem `INT`,

Proměnná	Třída	Význam
ExcId	STR	Identifikace výjimky, unikátní řetězec
ExcSeverity	STR	Závažnost výjimky, 1 znak
ExcMessage	STR	Text hlášení výjimky a definice pozičních parametrů
ExcChannel	STR, pouze pro zápis	Textový „kanál“ pro hlášení výjimky
poziční parametry	neurčena	Vstupní a výstupní proměnné pro hlášení a opravu výjimky, závisí na výjimce

Tabulka 2.1 Proměnné jmenného prostoru výjimky, převzato z dokumentace jazyku TPP[12]

`FLOAT`, `BIG_INT`, třídy pro práci s časem `DATETIME`, `DATE`, `DELTATIME`, `TIME` a strukturované typy list `LIST`, `LISTELEMENT`, a asociativní kontejner (implementovaný jako AVL strom) `CONTAINER`, `CONTELEMENT`.

Dále poskytuje funkce pro práci s operačním systémem, souborovým systémem, konverzní funkce, datové a časové funkce, textové funkce, třídu pro práci se *socket*, třídu pro práci s konzolí

2.6 PHP

Jazyk PHP je programovací jazyk pro všeobecné využití, je avšak také velmi dobře přizpůsobený pro použití na webu, dokonce je možné vložit jeho zdrojový kód přímo do HTML (Hypertext Markup Language).[14] Referenční implementace od *PHP Group* je v jazyce C pod a je dostupná v repozitáři *php-src* na stránce GitHub - [15].

2.6.1 Typový systém jazyku PHP

Jakýkoliv výraz v PHP je jedním z těchto zabudovaných typů, které jsou úzce spojené s runtime jazyka – `null`, `bool`, `int`, `float`, `string`, `array`, `object`, `callable`, `resource`, `never`, `void`.[14]

Typy `null`, `bool`, `int`, `float` a `string` jsou skalárními typy. Typ `null` má jedinou hodnotu a to `null`, kterou mají všechny neinicializované proměnné, `bool` má dvě možné hodnoty `true` a `false`. `int` vyjadřuje celá čísla, zatímco `float` čísla s desetinou čárkou, nakonec `string` reprezentuje řetězec, kde každý znak může nabývat jednu z 256 hodnot.[14]

Typ `array` je asociativní mapa, tedy úložiště párů klíčů a k nim asociovaných hodnot.[14]

`callable` reprezentuje hodnoty, které lze spustit, či zavolat. Například funkce, statická metoda, nestatická metoda, objekty implementující speciální funkci `__invoke` a anonymní funkce mají tento typ.[14]

Přístup k různým zdrojům, například k souborům, do databáze nebo ke grafickému plátnu (anglicky *canvas*), zajišťuje typ `resource`.[14]

Typy `never` a `void` jsou speciální typy, které je možné jenom navrátit z funkce. `never` vyjadřuje, že funkce, která ho vrací nikdy neskončí – buďto je vyvolaná výjimka nebo je program ukončen voláním funkce `exit` nebo je v funkci nekonečná smyčka. `void` říká, že funkce nevrací hodnotu, ale může skončit.[14]

Všechny typy je možné předávat jak referencí, tak hodnotou. Výjimkou jsou proměnné typu `object`, které se zdánlivě předávají pouze referencí, jelikož jejich hodnota samotná je jen reference na daný objekt.[14]

2.6.1.1 Objektově orientovaný systém jazyku PHP

Mimo zabudované základní typy poskytuje PHP navíc ještě objektově orientované rozšíření prostřednictvím typu `object`, kterým je možné vytvářet nové typy za pomocí tříd a rozhraní. Těmto novým typům se říká třídní typy a všechny jsou zároveň typem `object`, avšak neexistuje žádný společný třídní nadtyp všech třídních typů.[14]

Každá hodnota s typem `object` vznikla instanciováním nějaké třídy. Třídy mohou mít abstraktní a statické členy, mají-li alespoň jeden abstraktní člen, musí být také abstraktní. Jednotlivým členům je možné definovat jejich viditelnost pomocí modifikátorů `public`, `private` a `protected`. Bez specifikování modifikátoru je člen `private`. Atributům je možné přidat `readonly` modifikátor, který zaručí, že jejich hodnota je nastavena pouze jednou uvnitř třídy samotné. Pokud je třídě přiřazen modifikátor `readonly`, přidá se prakticky tento modifikátor před všechny atributy třídy. *mix-in inheritance* funguje obdobně jako v klasických programovacích jazycích.[14]

2.6.1.2 Implementace hodnoty

Všechny hodnoty jsou uložena v struktuře `zval`, která je složená z `_zend_value`, který reprezentuje data samotná (64 bitů), pak metadata v podobě `union` obsahující typovou informaci (32 bitů) a `union` (32 bitů) pro další libovolné data. Data samotná jsou buďto jednoduché hodnoty, například `zend_long`, ty jsou uloženy rovnou a nebo komplexnější, například `zend_string`, na ně je uložen pouze ukazatel. Pro práci s těmito struktury se interně abstrahuje použitím maker, např. `Z_TYPE`, `Z_OBJ`.[16]

2.6.1.3 Explicitní typové deklarace

Typový systém je sice dynamický, ale části jazyka je možné typovat i staticky pomocí typových deklarací. Přesněji je možné deklarovat typ u argumentů funkcí, návratových hodnot a u třídních atributů [14].

Za běhu se pak provádí v místech typových deklarací typová kontrola, zaručující správnost dosazených typů. Při dosazení nesprávného typu je vyhozena výjimka [14].

Zároveň je možné na základě deklarovaných typů provádět na zdrojovém kódu do určité míry statickou typovou analýzu a podchytit tím chybný kód před spuštěním programu [17].

Je možné typové deklarace zadávat i formou sjednocení typů. Implementačně se jen přidá další typová kontrola. Obdobně je možné typové deklarace zadat formou průniku typů, ale průnik je limitován pouze na třídní typy.[14]

2.6.1.4 Konverze mezi typy

Jazyk PHP je schopný různé typy interpretovat jako typy jiné, například určité hodnoty `string` (třeba `"0"`) je možné interpretovat jako `int` nebo `float`. V určitých kontextech se provádí tato interpretace hodnot automaticky. Původní hodnoty se avšak nemění, pouze se vytvářejí nové hodnoty s žádaným typem podle původních hodnot, které jsou za původní hodnoty dosazeny.[14]

Rozlišujeme mezi více konverzními kontexty:

- Numerický kontext při používání aritmetických operátorů.
- Řetězcový kontext při používání funkcí `echo, print` a při interpolaci řetězců a při použití spojovacího operátoru `..`
- Logický kontext při používání podmíněných příkazů, ternárního operátoru nebo logických operátorů.
- Celočíselný a řetězcový kontext při používání bitových operátorů.
- Komparativní kontext při používání komparativních operátorů.
- Funkční kontext při předávání hodnoty jako typovaný argument nebo když se hodnota vrací z funkce, která má deklarovaný návratový typ.

[14]

Je také možné konverzi použít explicitně pomocí například `(int)`, `(string)`, `(object)`.[14]

2.6.2 Automatická správa paměti jazyku PHP

Komplexní hodnoty sdílí stejnou hlavičku `zend_refcounted_h`, ve které je referenční čítač a další typové informace. Čítač se při referenci a snižuje se při zničení reference, dosáhne-li hodnoty 0, hodnota se uvolní. Jsou-li hodnoty neměnné, typicky když jsou ve sdílené paměti, tak se jejich reference nepočítají, vysokoúrovňové rozhraní pro práci s hodnotami toto řeší automaticky. [16]

Mimo samostatné počítání referencí jsou také řešeny cyklické reference pomocí algoritmu popsaném v tomto článku [18], tedy navíc běží ještě *mark-and-sweep* cyklický garbage collector, který pokud se hodnotě sníží reference, ale ne na nulu, tak ji dá na zásobník, protože je potencionálně cyklická. Když je je v základním nastavení na zásobníku potencionálních cyklických referencí 10 000 hodnot, spouští se *mark-and-sweep*. Cyklický *gargage collector* je možné zapnout a vypnout podle potřeby a dokonce je možné i vynutit *garbage collection* ihned v PHP kódu.[14] Pokud víme, že nepracujeme s cyklickými daty, tak v nízkkoúrovňovém API jsou varianty funkcí, které snižují referenční čítače, ale neřeší možnost cirkulárních závislostí. [16]

2.6.3 Výjimkový aparát

Výjimky jsou narozdíl od jazyku dclsh odlišeny od sebe pomocí typu a ne pomocí řetězcového atributu a závažnosti výjimky. Dále PHP dovoluje nastavit pouze jednu globální obslužnou rutinu pro všechny výjimky.[14]

2.7 Python

Python je nejpopulárnější jazyk dle [8], budeme se zaměřovat na verzi 3 a dále. Novější verze jsou vytvořené v jazyce C pod *Python Software Foundation* a jsou dostupné v GitHub repozitáři *CPython* [19]. Python je kompilovaný do *bytecode* před jeho interpreptací.[20]

2.7.1 Typový systém

Všechna data v Pythonu jsou *objekty*, které mají identitu, typ a hodnotu. Identita objektu je de facto adresa v paměti, tedy neměnitelná. Typ objektu také nelze změnit a definuje jaké operace hodnoty danného typu podporujují, typ samotný je také objekt. Hodnoty typů mohou být *měnitelné* (například tabulky, pole) nebo *neměnné* (například čísla, řetězce). Typy je možné deklarovat, ale fungují jako tzv. *type hints* a za runtime nejsou tyto deklarace kontrolovány. Nabízí tedy pouze potenciální statickou typovou kontrolu a popřípadě mohou sloužit jako dokumentační prvek.[21]

Třídy jsou také objekty a slouží k instanciování nových objektů, je dokonce možné je přiřadit do proměnné. Třídní proměnné jsou dostupné pod tabulkou `__dict__`. Python implementuje obecné vícenásobné dědění a používá speciální *C3 Method Resolution Order*, který je popsaný v tomto článku [22]. Instance tříd jsou tedy vytvořené voláním objektu třídy a členové jsou dostupní pod tabulkou `__dict__`.[21]

Třídní členy jsou pro všechny viditelné. Python komunita sama časem začala používat konvenci prefixování `_` před členy, které by měli být považovány za neveřejné. Ještě později byl přidán mechanismus *name mangling*, který zaměnil prefix členů, které mají prefix s `__` za `_classname__`. Například z členu `__function` uvnitř třídy `classic` by se stal `_classic__function`. [21]

Funkce jsou také reprezentovány jako objekt, který má položky `__globals__`, což je tabulka reprezentující globální jmenný prostor modulu, ve kterém je funkce definovaná a `__closure__`, která obsahuje vázání na volné proměnné dané funkce. Instanční metody jsou jen kontejnery, které svazují `__self__` s funkcí třídy, která akceptuje jako první parametr právě *self*.[21]

Obdobně k třídám je moduly, neboli jednotlivé soubory zdrojového kódu, také možné importovat jako objekty a každý modul má tabulku `__dict__`, díky které je možné přistupovat k definicím uvnitř modulu.[21]

Python navíc podporuje generické funkce a třídy, tato typová informace je typicky použita pro statickou typovou kontrolu, jinak se generické entity chovají stejně jako negenerické protějšky.[21]

Hodnoty výjimek jsou objekty, které dědí ze třídy `BaseException` a jednotlivé výjimky se rozlišují podle jejich typu.[21]

2.7.1.1 Implementace v jazyku C

Všechny typy dědí z typu `object`, který je zabudovaný základní typ a obsahuje základní funkcionality. Všechny datové typy, v `CPython` mají stejnou hlavičku `PyObject` obsahující referenční čítač a ukazatel na strukturu reprezentující typ `PyTypeObject`. Kolekce navíc mají od `PyObject` v hlavičce `PyVarObject` položky

reprezentující základní typ a počet prvků v kolekci. Ve zkratce CPython tímto způsobem provádí ruční dědění v jazyku C, jelikož díky totožné hlavičce je možné interpretovat podtyp jako jeho nadtyp.[20]

Každý typ (potažmo třída) definovaný uživatelem (klíčové slovo `class`) má asociativní tabulku, obsahující všechny členy (instanční metody a proměnné, statické metody a proměnné) přístupnou pod `__dict__()` v Pythonu a pomocí funkce `PyObject_GenericGetDict()` v C.[20]

Standardní instalace pythonu obsahuje značné množství modulů. Nějaké jsou napsané v Python jazyku samotné, ostatní zase v jazyku C, například funkce `print` je v Pythonu dostupná z modulu `__builtins__` a zdrojový kód je v souboru `builtinmodule.c`.[20]

2.7.1.2 Typové konverze

Typy se na sebe implicitně konvertují v speciálních případech, například při vyhodnocování výrazu `int + float` proběhne konverze na `float`. Tuto konverzi avšak provádí kompilátor. Až na speciální případy je tedy konverze prováděna explicitně pomocí normálních funkcí. Standardní knihovna poskytuje konverzní funkce pro většinu základních typů. [21]

2.7.2 Reprezentace výpočtu

Jazyk Python pro reprezentaci výpočtu používá struktury *Interpreter state*, *Thread state* a *Stack frame* a *Frame object*.[20]

`Interpreter state` obsahuje list vláken, informace o *garbage collectoru* a další potřebné informace.[20]

Dále Python reprezentuje jednotlivá vlákna strukturou `PyThreadState`, která obsahuje unikátní identifikátor vlákna, odkaz na vlastnící *Interpreter state*, zrovna probíhající *Call frame* a hloubku volání, zásobník zrovna šetřených výjimek.[20]

Python obdobně jako jazyk dclsh používá pro reprezentaci volání funkce strukturu *Stack frame* obsahující parametry, návratovou hodnotu a obdobný stav výpočtu.[20]

Struktura *Frame object* obaluje zkompilovaný *Code Object* (což je zkompilovaný zdrojový soubor v bytecode) a uchovává navíc runtime stav v podobě globálních proměnných, lokálních proměnných a zásobníku hodnot.[20]

2.7.3 Automatická správa paměti

Python v základu standardně počítá reference všech objektů, dále poskytuje volitelný `garbage collector`, který je schopný detekovat a uvolnit referenční cykly.[21]

Jak již bylo zmíněno, každý `PyObject` (potažmo objekt) má čítač referencí, který je ovládaný funkcemi `_Py_DECREF()` a `Py_INCREF()`, které pracují s položkou `ob->ob_refcnt`.[20]

Místo klasického *mark-and-sweep garbage collector* používá Python vlastní algoritmus, který hledá cyklické reference v kontejnerových typech (tedy typy objektů obsahující reference na další objekty), u kterých se cyklické reference mohou objevit. Všechny typy, které jsou spravované *garbage collectorem* mají v C stejnou hlavičku ještě před normálními objektovými daty jménem `PyGC_Head`, která obsahuje ukazatel dopředu a dozadu, tedy spravované objekty se při vzniku přidají

do dvousměrného seznamu a při zániku odeberou. Každý spravovaný typ musí definovat funkci `tp_traverse`, která zavolá `visit` funkci s daným argumentem na každý obsažený objekt. V případě hledání cyklických referencí zmenší `visit` u každého objektu speciální referenční čítač `ob->gc_ref` (odlišný od normálního `ob->ob_refcnt`) o jeden, pokud spadne na nulu, objekt se připraví na uvolnění.[20]

Python navíc používá generační *garbage collection*, která se zakládá na pozorování, že 80 % objektů jsou krátce po vzniku zničeny. Python má tři generace, každá má nastavenou hodnotu, kdy se spustí *garbage collection* na tu generaci a všechny mladší generace. Pokud objekt není uvolněn v *garbage collection* cyklu, tak postupuje do starší generace.[20]

Pomocí modulu v jazyku Python `gc` je možné pracovat s volitelným *garbage collectorem*, přesněji ho vypnout, změnit jeho frekvenci nebo nastavit debugovací možnosti.[21]

2.8 JavaScript

Obdobně jako PHP je vhodný pro použitíá na webových serverech, je JavaScirpt vhodný pro využití na webu, přesněji ve webových prohlížečích poskytuje interaktivní funkce přímou změnou HTML dokumentu na straně uživatele. Jako jeden z mála skriptovacích je JavaScript formálně definován mezinárodní standardnizační organizací *Ecma International* a navíc má vícero implementací. Například jazyky Python a PHP mají jednu rozšířenou implementaci, která funguje zároveň jako definice jazyka samotného.[3]

2.8.1 Typový systém

Jazyk JavaScript má 8 základních typů – `undefined`, `null`, `boolean`, `number`, `bigint`, `string`, `symbol` a `object`.[23]

Typ `undefined` má jedinou hodnotu a to `undefined`, která je přiřazena proměnnám, které nebyly inicializovány. Podobně typ `null` má také jednu hodnotu `null`, avšak význam je odlišný, tedy vyjadřuje neexistenci hodnoty.[23]

Logické hodnoty reprezentuje JavaScript typem `boolean` a řetězce typem `String`. Čísla jsou reprezentovány typem `number`, který je implementován jako 64 bitové číslo s plovoucí desetinou čárkou. Při chybných výpočtech jako například `0 / 0` vzniká numerická hodnota `NaN`, všechny aritmetické operace s `NaN` mají za výsledek opět `NaN` a zároveň platí, že `NaN` se nerovná ničemu, ani sobě. Pro reprezentaci technicky arbitrárně velikých celých čísel je typ `BigInt`, který není možné používat s normálními `number`.[23]

Konverze mezi typy jsou prováděny implicitně a JavaScript obecně je notorický za nečekané chování při těchto konverzích. [24].

V JavaScriptu není možné deklarovat u proměnných a atributů objektů jejich typ. Populární alternativa je tedy jazyk TypeScript, který přidává do jazyku JavaScript syntaxi pro deklaraci typů a taky se do něj kompiluje.[25]

2.8.1.1 Objektově orientovaný systém

JavaScript, obdobně jako PHP, má omezené množství základních datových typů, z nichž jeden typ je `object`, pomocí kterého je možné programovat v objektově

orientovaném stylu, avšak implementace a detaily objektového systému se značně liší.

Objekt v JavaScriptu se chová jako asociativní kontejner, tedy asociuje klíče s hodnotami. Objektové proměnné mají referenční sémantiku oproti ostatním typům. Kdykoliv je možné hodnotě přidat nový klíč a asociovanou hodnotu nebo podle klíče asociovaný pár odebrat. Pole, neboli `arrray`, je potom objekt s numerickými klíči se speciální syntaxí.[23]

Každý objekt má také jeden svůj designovaný *prototyp*, což je nějaký jiný objekt. Při hledání hodnoty dle poskytnutého klíče v objektu se první prohledává objekt samotný a při neúspěchu se pokračuje prohledáváním objektů v řetězci prototypů. Řetězec prototypů je avšak prohledáván pouze při čtení z objektu, ne při zápisu. Na konci každého řetězce je speciální objekt `Object.prototype`, jehož prototyp je `null` a obsahuje základní metody například `toString()`.[23]

Tímto mechanismem je implementováno dědění, které je ale velice dynamické, jelikož je kdykoliv možné změnit prototyp objektu na jiný nebo změnit jeho obsah.[23]

Funkce jsou v JavaScriptu objekty se speciální syntaxí a proto se s nimi dá pracovat jako s funkcemi vyššího řádu. Funkce mohou používat hodnotu `this`, která je dosazena až za běhu programu podle objektu, který metodu volá, pokud je funkce volaná bez objektu, tak má `this` hodnotu `undefined`.[26]

Třídy a třídní hierarchie jde tedy pomocí prototypování tvořit "ručně„ alternativně JavaScript nabízí syntax deklarace třídy s jednoduchou dědičností, konstruktory, statickými a schovanými členy. Tento zjednodušený syntax je jen syntaktická nádstavba nad prototypováním.[27]

2.8.2 Automatická správa paměti

Implementace JavaScriptového *Garbace collector* jsou různé a využívají různé optimalizace napříč všemi implementacemi JavaScriptu. Všechny se avšak zakládají a *mark-and-sweep* algoritmu.

Kapitola 3

Runtime a interpret jazyku dclsh

V této kapitole popíšeme jazyk dclsh a jeho rozdělení na runtime a interpretovou část. Popíšeme podrobněji koncepty jazyku dclsh, které je nutné implementovat v runtime části. Nakonec porovnáme jazyky, které byly v rešerši s jazykem dclsh.

Dclsh je interpretovaný skriptovací programovací jazyk, podporující imperativní a objektově orientovaný styl programování.

Přesný syntax a konstrukce jazyka samotného nejsou v této práci pevně definovány, ale popis je doprovázen názornými ukázkami, jak by mohl jazyk vypadat. Popisovaná specifikace vznikla formou konzultací s vedoucím práce [28].

Mezi kompetence runtime části jazyku dclsh patří typový systém, datová reprezentace výpočtu a automatická správa použité paměti.

Návrh jazyku dclsh vychází z jazyku TPP, dále nabízí funkčnosti shellu v podobě jednoduchého přesměrovávání a rourování. Funkčnosti shellu jsou poskytnuty pomocí datových proudů (anglicky *streams*) v kombinaci s jednotmým rozhraním pro volání jak metod, funkcí a jiných dclsh shkriptů, tak externích programů. Navíc poskytuje objektově orientovaný typový systém, možnost deklarovat typ u proměnných, schopnost definovat chybný tok výpočtu pomocí výjimek a možnost deklarovat neglobální proměnné.

3.1 Typový systém

Runtime část jazyku dclsh má na starost reprezentaci objektově orientovaného typového systému.

Jazyk podporuje definici tříd a rozhraní. Všichni členové tříd jsou viditelní všude a nemůžou být statičtí ani abstraktní. Zabudované třídy avšak své chování a data schovávají. Třídy samotné avšak abstraktní být mohou, tedy nějaké třídy není možné instanciovat. Typ dědičnosti je *mix-in* bez možnosti definovat těla metod v rozhraních, tedy rozhraní obsahují pouze deklarace metod, které dědící třídy musí implementovat. Jak třídy, tak rozhraní mohou obsahovat v definici typové parametry.

Dclsh obdobně jako skriptovací jazyky zmiňované v rešerši používá dynamické typování a povoluje typy explicitně deklarovat. Plánujeme do budoucna využít deklarovaných typů pro optimalizaci při kompilaci modulu – známe-li typ proměnné,

```
// typedef - pojmenování typu

typedef Vector<Int> intvector; // vektor intu
typedef Array<Int> intarray; //pole intu
typedef Int[10] intarray; // pole intu
typedef Vector<Int,5> vecint5; // vektor 5 intu
typedef Array<Int,5> arrint5; // pole 5 intu
typedef List<Int> numlist; // seznam intu

// Některé typy mají výchozí typové parametry
Vector<valuetype=Object>
Array<valuetype=Object>
List<valuetype=Object>
Tree<keytype=Int, valuetype=Object>

Object ob1; // Jakýkoliv typ
Object<Int,String> ob2; // Pouze Int nebo String
```

Výpis kódu 1 Ukázka použití *templates*.

můžeme předpočítat index do virtuální tabulky metod a vyhnout se dynamickému vyhledávání (anglicky *lookup*) podle jména metody. Obdobnou optimalizaci bychom měli být schopni provést i pro atributy třídy.

Pro sémantickou konzistenci jsou explicitně netypované hodnoty implicitně typované jako typ `Object`, který je nadtyp všech typů. Proměnné typované jako `Object` hledají své členy vždy podle řetězcového jména členu (*lookup*).

Proměnné typu `Object` je možné omezit na určité sjednocení typů pomocí syntaxe `Object<Null, Int>`, která sjednocuje typ `Null` a `Int`. Podobný mechanismus jsme viděli u typových deklarací v jazyku PHP.

3.1.1 Šablonovité třídy

Třídy a rozhraní mohou mít typové parametry ve své definici. Při instanciaci generické třídy musí být předány mimo formálních argumentů i typové argumenty. Na rozdíl od jazyků, používající *type erasure*, jako je například Java[29] se vytvářejí nové konkrétní typy a třídy při novém unikátním použití typových parametrů. Typovými argumenty se myslí argumenty, které jsou předávány typu. Mohou být předávány jak typy, tak i jiné hodnoty, viz. názorná ukázka 1

3.1.2 Konverze mezi typy

Mezi jakoukoliv nestejnou dvojicí tříd `A` a `B` je možné definovat dvě speciální konverzní funkce, první z typu `A` do typu `B` a druhou z typu `B` do typu `A`.

Je-li někde vyžadován typ `A`, ale je k dispozici pouze hodnota typu `B` a je navíc definovaná konverzní funkce z typu `B` do `A`, tak dclsh provede implicitně konverzi, jinak vyvolá výjimka.

Při konverzi není měněná konvertovaná hodnota, pouze se vytváří nová, která z konvertované hodnoty vychází.

3.1.3 Definování nových typů

Nové třídy a rozhraní je možné přidat dynamicky interpretováním modulu, který obsahuje novou definici třídy nebo rozhraní. Interpretovaný typ musí být registrován interpretem za běhu u runtime části podle poskytnutého rozhraní. Runtime si nový typ zpracuje, uloží a bude jeho funkčnosti poté poskytovat zpátky interpretu. Alternativně je možné definovat nový typ v jazyce C. Tímto způsobem může mít uživatel více kontroly nad implementací daného typu.

3.1.4 Standardní knihovna

Standardní knihovna jazyku dclsh poskytuje základní datové typy, strukturované typy, kolekce a datové proudy.

Hodnota typu `Undef` je přiřazena do neinicializovaných proměnných. Typ `Null` vyjadřuje, že v proměnné není žádná hodnota. Výjimku reprezentuje typ `Exception`.

Mezi základní datové typy patří logický typ `Bool`, celočíselné typy `Int` a `UInt` (64 bitů), typ čísel s plovoucí desetinnou čárkou `Float`. Řetězce jsou reprezentovány pomocí znakového typu `Char` a řetězcovým typem `String`. Dále, podobně jako v jazyku DCL a TPP, absolutní čas a datum reprezentují typy `Time`, `Date` a `DateTime`, a relativní čas typ `DeltaTime`.

Strukturované typy jsou reprezentované pomocí `Struct` a `CStruct`, které asociují jména položek s jejich hodnotou a typem. Také jsou sami o sobě abstraktní a musí se jim dodefinovat typové argumenty, podle kterých se definuje jaké typy jsou schopné držet. Typ `CStruct` je kompatibilní se strukturami v jazyku C. Další strukturovaný typ je výčtový typ `Enum`, který jako `Struct` je abstraktní a musí se mu dodefinovat typové argumenty. Přesněji je nutné doplnit unikátní řetězce, které se stanou prvky výčtu.

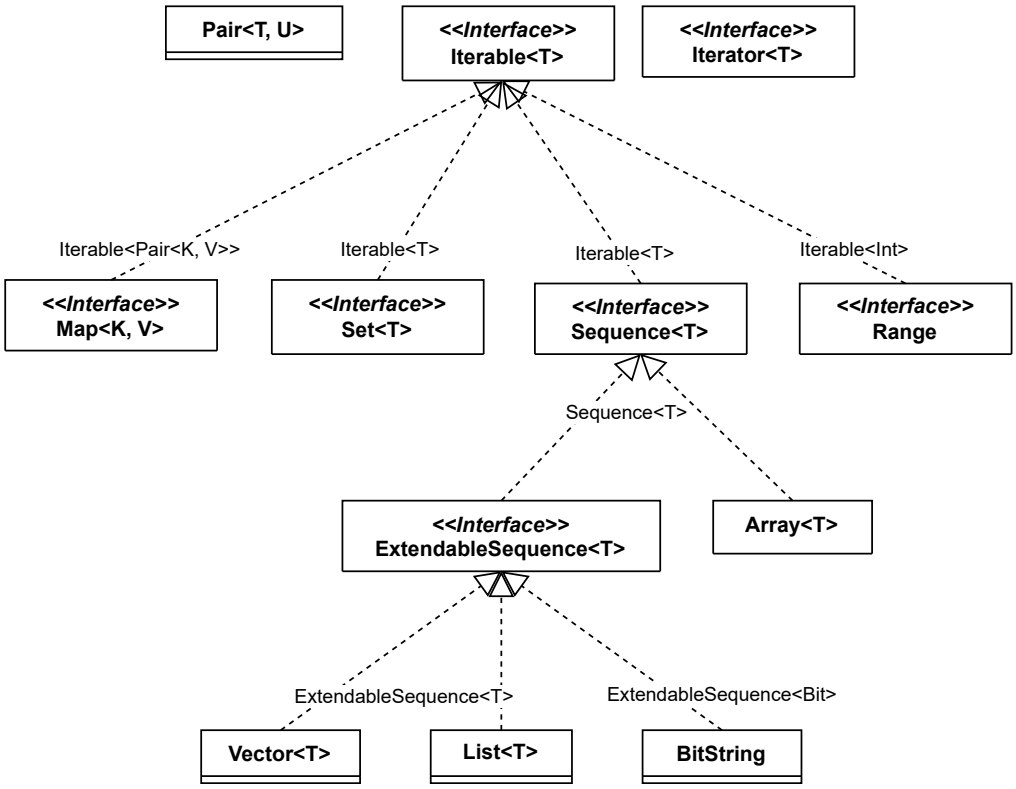
Všechny kontejnery implementují rozhraní `Iterable<T>`, díky čemuž jsme schopni přes všechny kontejnery iterovat. Hierarchii kontejnerových rozhraní je zobrazena na obrázku 3.1. Podrobnější informace o jednotlivých rozhraní lze najít v sekci zabývající se kontejnery v návrhové kapitole 4.2.3.

Asociativní kontejner představuje rozhraní `Map<K, V>`, které asociuje typ klíče `K` s typem hodnot `V`. Množinový kontejner je reprezentován typem `Set<T>`.

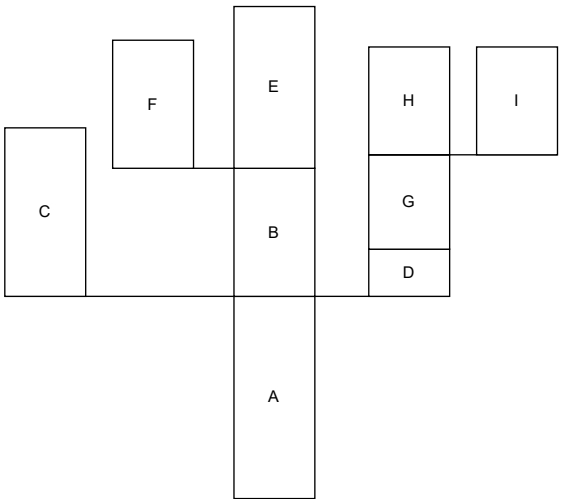
Pro paměťově úspornou reprezentace řady celých čísel slouží typ `Range`, který také využívá rozhraní `Iterable`.

Rozhraní `Sequence<T>` udává společné metody indexovaných sekvenčních kontejnerů, například přístup k prvku na indexu pomocí metody `at(Int index)`, zatímco rozhraní `ExtendableSequence<T>` přidává navíc metody, pomocí kterých je možné kontejner zvětšit nebo zmenšit. Kontejner `Array` představuje pole fixní délky, zatímco `Vector` je natahovací pole a `List` je seznam. Typ `BitString` kopmaktně uchovává jednotlivé bity.

Plánované jsou dvě implementace asociativního kontejneru `Map` – `HashMap` a `TreeMap`. `HashMap` používá pro hashovací tabulku, zatímco `TreeMap` používá vyvážený binární vyhledávací strom. Velmi podobně jsou rozděleny množinové kontejnery s rozhraním a implementacemi `HashSet` a `TreeSet`.



Obrázek 3.1 Hierarchie kontejnerových tříd. Vlastní obrázek.



Obrázek 3.2 `Cactus stack` obsahující `Callframe` A, B, C, D, E, F, G, H, I. Z `Callframe` A jsou volány C, B a D, z `Callframe` B jsou volány E a F, z `Callframe` D pouze G a z `Callframe` G jsou volány H a I. Volání C, D, F a I jsou realizované samostatnými vlákny (asynchronní) Vlastní obrázek.

3.2 Rozsah platnosti proměnných

Jazyk podporuje jak definici funkcí, tak metod.

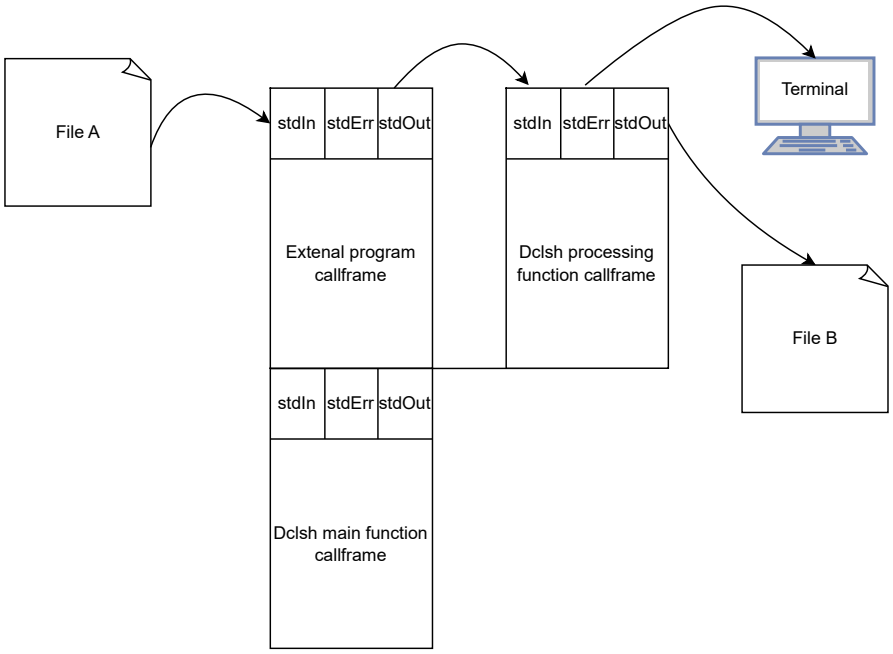
Uvnitř metod je, pro zajištění enkapsulace, rozsah platnosti proměnných omezen na sjednocení globální oblasti a oblasti platnosti, kterou vymezuje `Callframe`, reprezentující volání dané metody. Vždy je uvnitř metody dostupná a implicitně dosazená speciální proměnná `self`, která reprezentuje objekt, na kterém byla metoda zavolána.

Funkce mají dynamický rozsah proměnných, tedy nenajde-li se proměnná na vrcholu zásobníku volání funkcí, prohledává se zásobník volání dále, ve snaze najít chtěnou proměnnou. Proměnné uvnitř funkcí musí mít dále kvalifikátor viditelnosti proměnné – buďto `local`, značící viditelnost uvnitř funkce, `global`, značící globální viditelnost, nebo `common`, značící viditelnost uvnitř ostatních funkcích ve stejném modulu. Zdrojový kód se zapisuje do souborů, každý soubor definuje *modul*. V modulu je možné definovat funkce, proměnné, třídy a rozhraní.

3.3 Přesměrování, rourování a souběžný výpočet

Volání funkcí, metod, dalších dclsh skriptů a externích programů má jednotné rozhraní v podobě `Callframe`, které jsou uspořádány do tzv. *cactus stack*, vyzobrazeném na obrázku 3.2. V případě volání funkcí a metod mají `Callframe` vlastní zásobník, což umožňuje souběžný výpočet.

Funkce jednoduchého přesměrování a rourování poskytuje `Callframe`, prostřednictvím zabudovaných datových proudů (*streams*). Každý `Callframe` má tři datové proudy v podobě standardního vstupu, standardního výstupu a chybového výstupu, které se dají mezi sebou napojovat. Díky tomuto návrhu je možné v dclsh tvořit programy podobným způsobem jako v shellech s větší bezpečností a přehledností, viz. ukázka potencionálního využití 3.3.



Obrázek 3.3 Příklad použití zabudovaných datových proudů. Hlavní funkce main spustila externí program, který čte ze souboru A a výsledky předává datovým proudem procesní funkci, která proud postupně zpracovává a výsledky posílá datovým proudem do souboru B a případné chyby posílá na terminál. Vlastní obrázek.

3.4 Předávání parametrů

Parametry můžou být předávány buď to hodnotou, referencí nebo konstantní referencí. Konstantní reference se chovají jako reference, které ale nedovolují volat na proměnných nekonstantní metody nebo měnit hodnoty jejich atributů.

Dále rozlišujeme mezi inherentně referenčními a hodnotovými typy, které se při předávání chovají odlišně, viz. tabulka 3.1. Hodnotové typy jsou vybrané typy, například `Int`, `Float`, `Array`, které jsou součástí základní knihovny, všechny ostatní a uživatelsky definované typy jsou referenční.

	Hodnotou	Referencí	Konstantní referencí
Hodnotový typ	Kopie hodnoty	Vznik reference	Vytvoření konstantní reference
Referenční typ	Kopie reference, kopie hodnoty při zápisu	Kopie reference	Vznik konstantní reference

Tabulka 3.1 Tabulka předávání parametrů podle typu

3.5 Výjimkový aparát

Výjimkový aparát odděluje chybový tok výpočtu od normálního. V jazyku dclsh je možné výjimky vyvolávat, zachytávat a případně je ošetřit, nebo je nechat šířit dále.

```
// Ve skriptu:

// handler zachycuje podle závažnosti error
on error then handler1;

// handler zachycuje podle závažnosti warning
on warning then handler2;

// handler zachycuje podle řetězcového jména "eof"
on "eof" then { handler3; }

// Ve funkci:
try {
    ...
    val = throw(exc1);
    ...
}
    // Pokračování za val = ... s náhradní hodnotou newval
    catch ("exc1") { handler1; continue(newval); };

    // Pokračuje za try blokem
    catch ("bad") { handler2; break; };

    // Reklasifikace výjimky, eskalace výš
    catch (error) { handler2; throw("syserror"); };

    // Provede jak při vyhození výjimky, tak bez ní
    finally {...};
```

Výpis kódu 2 Ukázka použití výjimek

Výjimka samotná je hodnota, která má typ `Exception` a je ji možné vyvolat pomocí výrazu `throw`. Třída `Exception` má za atributy řetězec jména výjimky, řetězec popisující výjimku a číselnou závažnost výjimky. Závažnost je na stejné škále jako v jazyku DCL, tedy `INFO`, `WARNING`, `ERROR`, `FATAL`.

Ve skriptu je možné deklarovat obslužnou rutinu pro jednotlivé výjimky.

Specifika výjimkové sémantiky a obecně kontrola toku výpočtu je v zásadě v režii interpretu, runtime část pouze poskytuje výjimkový typ a potřebnou reprezentaci výpočtu, viz. Výpis kódu 2. 2.

3.6 Operátory

Dclsh umožňuje zápis operací pomocí operátorů. Operátory jsou pouze jinak zapsané volání metod. Operátorů bude omezené množství, nebude možné je dále rozšiřovat, výčet a asociativita a precedence lze nalézt v tabulce B.1. Třída operátoru se vybírá podle typu levého operandu.

3.7 Automatická správa paměti

Každá hodnota má asociovaný referenční čítač, který počítá počet referencí. Spadne-li čítač na nulu, nedostupná hodnota se ihned uvolní z paměti.

3.8 Porovnání s rešeršovanými jazyky

Narozdíl od omezeného výběru datových typů probíraných shellů, poskytuje dclsh objektově orientovaný systém umožňující definovat nové interpretované typy pomocí dědění. Jazyky dclsh a PHP jsou si podobné v tom, že oba podporují definici tříd a rozhraní, avšak PHP objektově orientovaný styl programování adoptovalo až postupem času (velká část základní knihovny je napsaná v imperativním stylu), tedy nejsou zmiňované konstrukty pro práci v jazkyu vůbec nutné. Jazyky JavaScript a Python mají značně dynamičtější typové systémy, zakládající se na implementování hodnoty jako hašovací tabulky a potenciálně dovolují měnit třídní hierarchie za běhu. Avšak oba jazyky nabízí `class` syntax, který potenciální komplexitu věrohodně schovává. Python dokonce nabízí obecné vícenásobné dědění.

Jazyk dclsh také unikátně dovoluje explicitně deklarovat typ kdekoliv a plánuje se pomocí těchto typových deklarací určité optimalizace. Další unikátností je definování typu `Object` jako nadtyp všech typů, kterým jsou sémanticky implicitně typovány nedeklarované proměnné. Navíc je to jediný typ, jehož členy se vyhledávají podle jména. Jazyk PHP také umožňuje do značné míry typy explicitně deklarovat a také na místech deklarací probíhá typová kontrola, avšak například typování funkčních hodnot je omezeno na pouhé `callable`, což je nic neříkající o reprezentované funkci. Pro explicitní typové deklarace v JavaScriptu se používá jiný jazyk TypeScript. Python explicitní typové deklarace povoluje, ale za běhu programu je nebere v potaz.

Všechny jazyky díky jejich dynamickému typování umožňují vytvářet polymorfní typy. Dclsh unikátně podporuje tvorbu explicitních generických typů ve formě šablonovitých tříd. Python explicitní generické typy podporuje, ale nejsou za běhu programu zohledňovány. JavaScript a PHP explicitní generické typy nepodporují.

Výjimkový aparát dclsh je inspirovaný jazyky DCL a TPP. Bash výjimky nepodporuje. JavaScript, PHP a Python výjimky implementují a jejich implementace je značně podobná.

Jedinečně poskytuje jazyk dclsh možnost specifikovat speciální konverzní funkce mezi typy, které jsou následně implicitně používány. Implicitní typové konverze sdílí shelly, TPP, JavaScript a dclsh. PHP provádí různé konverze implicitně podle kontextu pomocí systému `Type Jugggling`. Python až na výjimky vyžaduje implicitní konverzi mezi typy.

Také schopnost použít konstantní reference ve funkcích není přítomná v žádném dalším zkoumaném jazyku.

Kapitola 4

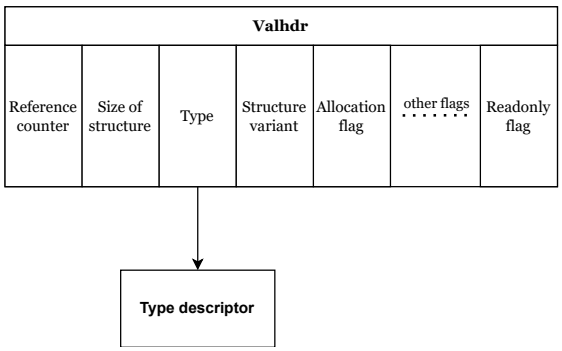
Návrh a architektura runtime knihovny dclsh

Popíšeme architekturu runtime části jazyka dclsh a rozhraní, které vystavuje interpretu.

Runtime knihovna je psaná v jazyku C podobně jako podobné jazyky, hlavním důvodem je potencionální rychlost. Také ostatní populární skriptovací jazyky jsou implementovány v jazyku C. Části runtime knihovny jsou navrženy ve stylu připomínající objektově orientovaný styl, obdobně jako jsme viděli v implementaci jazyku Python, i přes nepřímou nepodporu jazyka samotného. Obecně není-li objektový styl potřeba, je lepší se mu v jazyce C vyhnout, protože je pomalejší než iterativní řešení a zdlouhavý na implementaci.

4.1 Reprezentace dat a výpočtu

Všechny v této sekci probírané datové struktury mají společnou hlavičku `VALHDR`, která uchovává potřebné společné informace.



Obrázek 4.1 Diagram společné části všech datových struktur. Vlastní obrázek.

- `Reference counter` je referenční čítač.
- `Size of structure` udává velikost struktury.

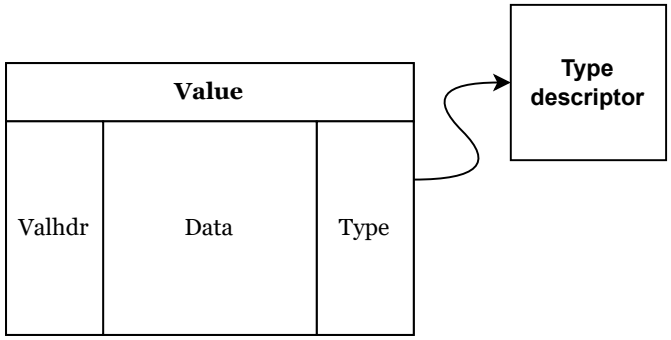
- `Type` je ukazatel na popisovač třídy.

- `Structure variant` uchovává informaci (pomocí čísla) určující, která struktura zrovna používaná.

4.1.1 Hodnota

V jádru výpočetního modelu jazyku dclsh je hodnota (potažmo objekt), která je vždy instancí jedné třídy. Hodnota musí držet potřebné data, podle toho jakou třídu implementuje, odkaz na danou třídu/typ a také referenční čítač, nutný pro automatickou správu paměti.

Paměť dostupná uvnitř hodnoty je neměnná, prakticky je nastavená podle paměťově nejnáročnější zabudované třídy (union).

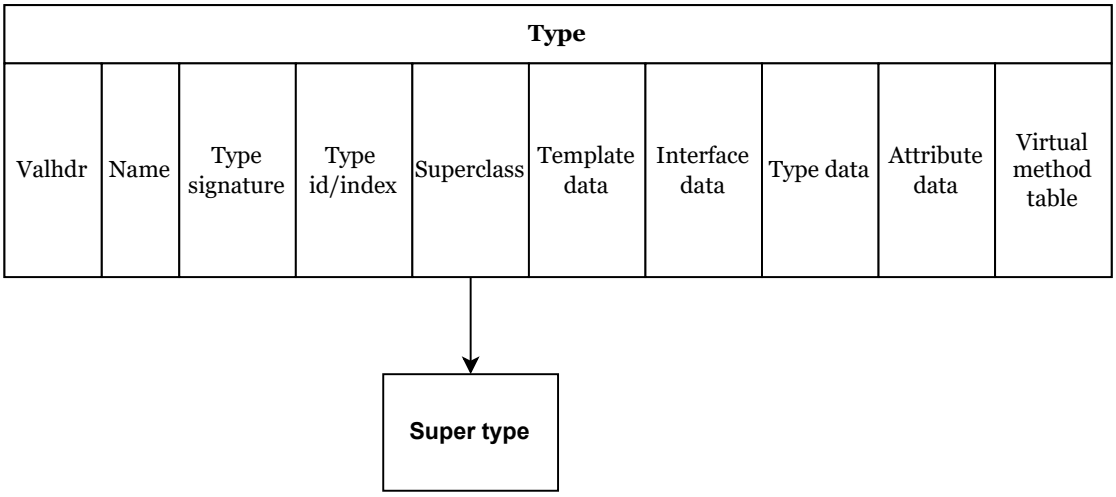


Obrázek 4.2 Diagram hodnoty. Vlastní obrázek.

4.1.2 Třída

Třídy definují operace nad hodnotami, které jsou instancí daných tříd. Třídy se také mezi sebou se dědí a implementují rozhraní. Třída zároveň v očích runtime přímo reprezentuje typ hodnoty, jelikož rozhraní není možné instancovat. Jednotlivé třídy musí mít také povědomí o tom, zda-li jsou šablony, nebo byli šablonou vytvořeny.

Interpretovane typy se odvozují z existujících typů a je třeba si vybrat takový, který vyhovuje, žádná další paměť se uvnitř hodnot nealokuje. Například pro Galoisovo těleso se hodí `Int`, pro obecnou třídu `Struct`.



Obrázek 4.3 Diagram typového/třídního deskriptoru. Vlastní obrázek.

- `Name` obsahuje řetězcové jméno typu.
- `Type signature` obsahuje reprezentaci signatury typu, například `Vector<Int>`.
- `Type id/index` je číslo sloužící jako unikátní identifikátor typu a také je to index do tabulky, uchovávající všechny typy.
- `Superclass` odkaz na třídu, ze které daná třída dědí.
- `Template data` obsahuje informace související s mechanismem šablonování, tedy je-li typ šablona a pokud ano, tak jaké typy byly vytvořeny pomocí dané šablony.
- `Inteface data` uchovává jaké rozhraní třída implementuje a případné další data zabývající se rozhraními.
- `Type data` vymezený prostor pro data třídy. Například zabudovaný typ `Enum` si tu ukládá jaké položky vyjmenovává a jejich počet.
- `Attribute data` informace o atributech, které třída obsahuje.
- `Virtual method table` je virtuální tabulka metod třídy. Metody samotné budou mít jednotné rozhraní, které podporuje jak interpretované metody, tak zabudované C funkce.

4.1.2.1 Konverzní funkce

Typy jsou uloženy v globálním poli typů, ke kterému se přistupuje pomocí indexu, který také definuje identitu daného typu. Konverzní funkce jsou uloženy v globální dvojdimenzionální tabulce, ke které se přistupuje pomocí dvou indexů, první index představuje konvertovaný typ a druhý index cílový typ konverze. Nedefinované políčka v tabulce obsahují funkci, která zkonzumuje argument navrátí výjimku. Na obrázku 4.4 je zobrazené schéma globální typové tabulky a globální konverzní tabulky.

Typová tabulka				
0	1	2	3	4
Bool	Int	UInt	Char	String

Konverzní tabulka						
Y \ X		0	1	2	3	4
		Bool	Int	UInt	Char	String
0	Bool	x	I->B	U->B	C->B	S->B
1	Int	B->I	x	U->I	C->I	S->I
2	UInt	B->U	I->U	x	C->U	S->U
3	Char	B->C	I->C	U->C	x	S->C
4	String	B->S	I->S	U->S	C->S	x

Obrázek 4.4 Globální typová a konverzní tabulka. Vlastní obrázek.

4.1.2.2 Optimalizace volání třídních metod

Pracuje-li interpret s netypovanou proměnnou/parametrem/hodnotou (potažmo `Object`, musí interpret při použití metody/atributu daný člen hledat v třídním deskriptoru podle řetězcového jména. Avšak zná-li interpret při načítání modulu třídu hodnoty, může daný modul určit na jakém indexu ve virtuální tabulce metod je chtěná metoda, potažmo jak najít atribut v hodnotě samotné.

4.1.2.3 Vytvoření nového typu pomocí šablony

Při použití šablony vzniká nový deskriptor typu, který může být také šablona, nebo již konkrétní typ. Původní šablona si drží pole s typy, které byly podle ní vytvořeny, aby bylo možné dohledat, zda-li je nutné vytvářet nový typ, nebo zda-li již existuje.

Při vytvoření nového typu pomocí šablony se změní typová signatura, samozřejmně také `Template data`, a případné rozhraní, atributy a metody mohou změnit signaturu.

4.1.2.4 Dědění

Při dědění může rozšiřující typ přidat nové metody a atributy nebo nahradit již existující metody novými definicemi. Dále může také nový typ implementovat nové rozhraní, nebo dodat nové typové parametry, či dosadit do typových parametrů rozšiřující třídy.

Algoritmus dědění vypadá následovně:

1. Dosazujeme-li konkrétní typové parametry do rozšiřující třídy, sémanticky vlastně dědíme z typu, který vznikne dosazením daných typových argumentů. Tedy pokud již neexistuje tento typ, vytvoříme ho a dědíme z něj.

2. Přímočaře musíme změnit jméno, signaturu a nastavit korektní id/index a nadtyp.
3. Definujeme-li nové typové parametry, musíme je registrovat v `Template data` a případně v dotknutých metodách, atributech a rozhraních.
4. Zaregistrujeme nové atributy.
5. Ve virtuální tabulce metod přepíšeme nahrazované metody a přidáme nové na konec.
6. Zkontrolujeme zda-li je možné nové rozhraní implementovat. Pokud ano, zaktualizujeme `Interface data` a případně virtuální tabulku metod, provádíme-li optimalizace, viz. sekce zabývající se rozhraními.

4.1.3 Rozhraní

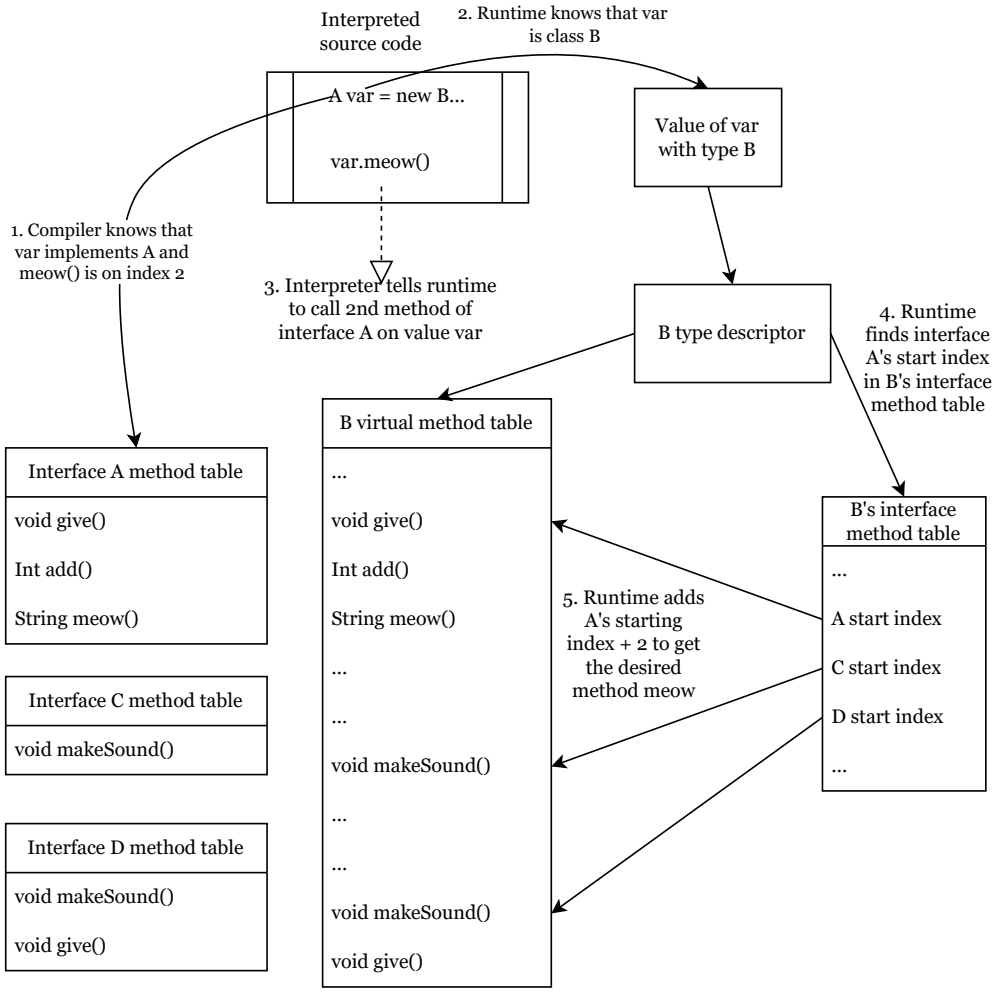
Rozhraní si musí držet signatury metod a případně z jakých dalších rozhraní metody dědí. Pomocí těchto informací se dají provádět kontroly tříd, jestli opravdu implementují daná rozhraní.

4.1.3.1 Optimalizace volání metod rozhraní

Známe-li jaké rozhraní hodnota implementuje, můžeme optimalizovat volání metod, podobně jako plánujeme u tříd, která nám ušetří vyhledávání jména metody pomocí řetězcového jména. Rozhraní udává určitý počet metod, které musí třídy implementovat. Jednotlivé metody rozhraní můžeme tedy vložit do tabulky metod rozhraní a tím je oindexovat. Při kompilaci modulu na místě použití metody od proměnné typovanou jako rozhraní, jsme schopni předpočítat jaký index má použitá metoda v tabulce metod daného rozhraní. Jednotlivé třídy si pro každé rozhraní, které implementují, mohou navíc držet index udávající, kde ve své virtuální tabulce metod začínají metody daného rozhraní. Tedy pro ilustraci na obrázku 4.5 když voláme metodu `String meow()` na proměnné, která je typovaná jako rozhraní `A`, tak se můžeme podívat do jejího opravdového typu `B` a najít si, kde začínají implementace metod od rozhraní `A`. Spojíme-li pak tyto dva indexy a podíváme se do virtuální tabulky metod typu `B`, dostaneme žádanou implementaci metody `String meow()`.

V tomto schématu je ale nutné uchovávat pro každé implementované rozhraní třídy odkaz do virtuální tabulky metod té dané třídy, kde ty implementace metod rozhraní lze nalézt. Tyto odkazy je možné ukládat v poli, což by zapříčinilo lineární prohledávání, nebo v potenciálně rychlejším, ale paměťově náročnějším asociativním kontejneru asociující typ rozhraní s žádaným indexem.

Navíc i virtuální tabulka metod musí být větší než obvykle, nejnaivnější implementace je pro každé implementované rozhraní přidat na konec tabulky kopie ukazatelů na metody, odpovídající metodám rozhraní, jak je to udělané na schématu 4.5. Na schématu jde ale vidět, že rozhraní `C` se "vejde„ do rozhraní `D`, tedy `C start index` by mohl být rovný `D start index` a metoda `void makeSound()`, na kterou ukazoval `C start index` je ve virtuální tabulce metod zbytečně. Dají se provést různé další komplexnější optimalizace velikosti společné virtuální tabulky metod třídy a rozhraní, kterými se zabírá tato práce [30].



Obrázek 4.5 Schéma pro optimalizaci volání metod na typované proměnné s typem rozhraní. Vlastní obrázek.

4.1.4 Podprogramy

Podprogramy mohou být buď funkce nebo metody a navíc ty můžeme mít interní nebo interpretované.

Subroutine					
Valhdr	Name	Singature	Required stack depth	Variables, parameters	Kind descriptor

Obrázek 4.6 Diagram reprezentace podprogramu. Vlastní obrázek

- `Valhdr` je společná část všech datových struktur.
- `Name` je řetězcové jméno podprogramu.

- `Signature` reprezentuje signaturu podprogramu. Například `"int max(int a, int b)"`.
- `Required stack depth` udává jak velký potřebuje podprogram zásobník.
- `Variables, parameters` popisují proměnné a parametry, tedy zda jsou lokální nebo globální, hodnotové nebo referenční, jaké mají řetězcové jméno, typy a výchozí hodnoty.
- `Kind descriptor` uchovává nutné informace pro reprezentaci daného typu podprogramu.
  - U interpretovaných metod/funkcí musíme vědět, na jaké řádce a v jakém modulu, potažmo souboru je metoda/funkce definována, spolu s odkazem na přeložený "kód„ metody/funkce.
  - U interních funkcí/metod si stačí držet ukazatel na danou C funkci.
  - Speciálně u metod musíme mít uložený odkaz na typ, ve kterém je metoda definována.
  - U externích programů zde bude uložený popis příkazového rozhraní.

4.1.5 Reprezentace volání

V dclsh reprezentuje volání funkce, metody, jiného skriptu nebo externího programu pomocí jednotného rozhraní v podobě `Callframe`. `Callframe` vytváří *cactus stack* tak, že každé vnořené volání si drží odkaz na volajícího rodiče.

Volání dalšího skriptu je pomocí `main` metody uvnitř daného skriptu. Tedy reprezentace uvnitř `Callframe` je stejná jako při volání interpretované funkce.

- Pro volání externího programu je pouze nutné předat parametry volání formou parametrů na příkazové řádce, dále napojit datové proudy z daného programu a držet si odkaz na proces, který program spustil.
- Při volání interpretované metody nebo funkce je nutné poskytnout interpretu výpočetní zásobník, programový čítač a případné proměnné nebo parametry.
- Při volání zabudované metody nebo funkce se předávají proměnné pomocí zásobníku volajícího `Callframe`.

4.2 Standardní knihovna

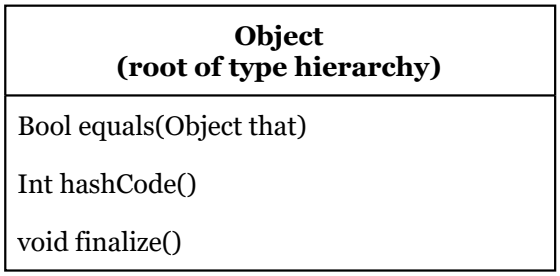
4.2.1 Základní typy

4.2.1.1 Object

Object je kořenová abstraktní třída celé typové hierarchie jazyku dclsh, tedy je to nadtyp všech ostatních typů. Object poskytuje základní základní operace společné pro všechny hodnoty:

- `Int hashCode()`, výpočet hashe (důležité pro implementaci kontejnerů)
- `Bool equals(Object that)`, vrací true, pokud jsou si objekty rovny.

- `void finalize()`, metoda, která se zavolá při uvolnění objektu – destruktor.



Obrázek 4.7 Diagram třídy Object. Vlastní obrázek.

4.2.1.2 Undef

Undef reprezentuje chybějící hodnotu, všechny operátory třídy Undef vyvolají výjimku. Také nejsou definované žádné výchozí konverzní metody.

4.2.1.3 Null

Null představuje chybějící hodnotu, všechny operátory definované pro typ Null vyústí opět v hodnotu Null.

Z jiných typů na Null konverze definovány nejsou, ale na Null se základní typy většinou konvertovat dokážou.

- Z `Null` na `Bool` – `false`.
- Z `Null` na `Int` – `0`.
- Z `Null` na `Uint` – `0`.
- Z `Null` na `Float` – `0`.
- Z `Null` na `Char` – `0`.
- Z `Null` na `String` – prázdný řetězec.

4.2.1.4 Logický typ Bool

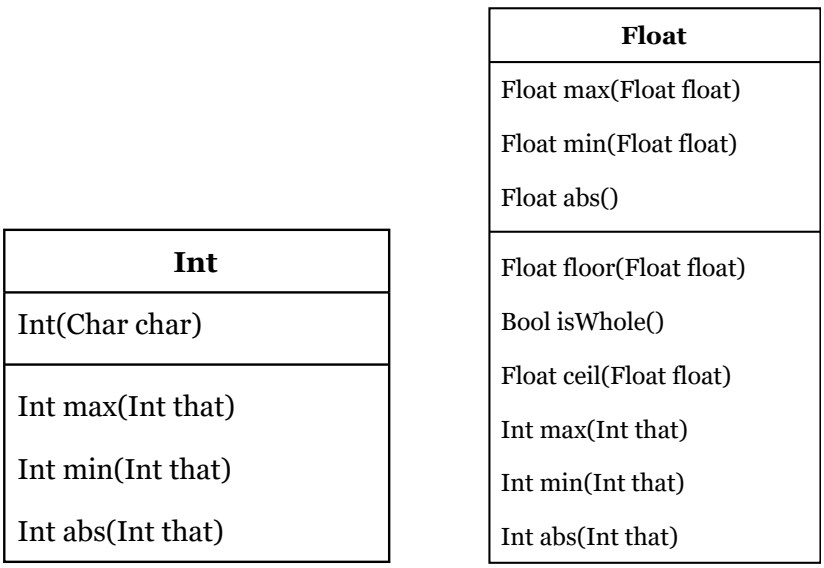
Bool má pouze 2 hodnoty – true nebo false. S logickými hodnotami se pracuje pouze s pomocí operátorů, popsané tabulkou 4.2.

4.2.1.5 Číselné typy

Typ Int představuje celočíselné číslo se znaménkem a typ Uint bez znaménka, oba celočíselné typy mají přesnost 64 bitů. Operace na celých číslech jsou definovány pomocí operátorů, typ Int je popsaný v tabulce 4.3. Pro Uint by tabulka vypadal podobně, jen bez operace unární mínus.

Výraz	Třída výsledku	Význam
`Bool && Bool`	Bool	Logický AND
`Bool || Bool`	Bool	Logický OR
`Bool ^^Bool`	Bool	Logický XOR
Proměnná `&&= Bool`	Bool	Logický AND a přiřazení
Proměnná `||= Bool`	Bool	Logický OR a přiřazení
Proměnná `^^= Bool`	Bool	Logický XOR a přiřazení
`Bool == Bool`	Bool	Porování na rovnost
`Bool != Bool`	Bool	Porování na nerovnost

Tabulka 4.2 Operátory třídy Bool.



(a) Metody třídy Int. **(b)** Metody třídy Float.

Obrázek 4.8 Metody tříd Float a Int.

Int a Uint poskytují respektivě metody:

- Konstruktor s parametrem typu `Char` vytvoří Int s Unicode hodnotou poskytnutého znaku.
- `max` vrátí větší ze dvou čísel.
- `min` vrátí menší ze dvou čísel.
- `abs` vrátí -1, je-li číslo záporné, 1 jestli kladné a 0, pokud je číslo právě 0.

Čísla s pohyblivou desetinou čárkou Float jsou manipulovány pomocí stejných operátorů jako celočíslené čísla s výjimkou bitových, inkrementačních, dekrementačních operátorů, které spolu s operacemi modulo nejsou definovány. Navíc od typu Int jsou dostupné pomocné funkce:

- `floor`, navrátí dolní celou část čísla.

- `ceil`, navrátí horní celá část čísla.
- `isWhole`, vrátí true je-li číslo celé.

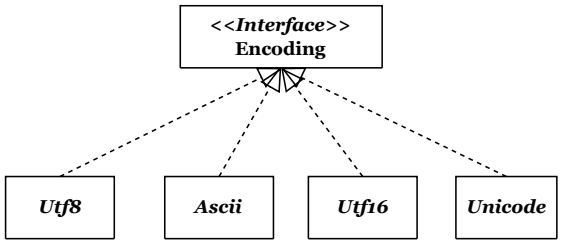
Výraz	Třída výrazu	Význam
`Int + Int`	Int	Celočíselné sčítání
`Int - Int`	Int	Celočíselné odčítání
`Int * Int`	Int	Celočíselné násobení
`Int / Int`	Int	Celočíselné dělení
`Int % Int`	Int	Modulo
Proměnná `+= Int`	Int	Celočíselné sčítání a přiřazení
Proměnná `-= Int`	Int	Celočíselné odčítání a přiřazení
Proměnná `*= Int`	Int	Celočíselné násobení a přiřazení
Proměnná `/= Int`	Int	Celočíselné dělení a přiřazení
Proměnná `%= Int`	Int	Celočíselné modulo a přiřazení
`+Int`	Int	Unární plus
`-Int`	Int	Unární mínus
`++Int`	Int	Prefixový inkrement
`--Int`	Int	Prefixový dekrement
`Int++`	Int	Postfixový inkrement
`Int--`	Int	Postfixový dekrement
`Int & Int`	Int	AND po bitech
`Int | Int`	Int	OR po bitech
`Int ^Int`	Int	XOR po bitech
Proměnná `&= Int`	Int	AND po bitech a přiřazení
Proměnná `|= Int`	Int	OR po bitech a přiřazení
Proměnná `^= Int`	Int	XOR po bitech a přiřazení
`Int << Int`	Int	Bitový posun doleva
`Int >> Int`	Int	Bitový posun doprava
Proměnná `<<= Int`	Int	Bitový posun doleva a přiřazení
Proměnná `>>= Int`	Int	Bitový posun doprava a přiřazení
`Int <` `Int`	Bool	Menší než
`Int` $\leq$ `Int`	Bool	Menší rovno než
`Int >` `Int`	Bool	Větší než
`Int` $\geq$ `Int`	Bool	Větší rovno než
`Int ==` `Int`	Bool	Porování na rovnost
`Int !=` `Int`	Bool	Porování na nerovnost
`Int :: Uint`	String	Formátování čísla vkládáním mezer vlevo.

Tabulka 4.3 Operátory třídy Int.

4.2.1.6 Znak

Znak je reprezentován typem Char, který nabízí jednoduché funkce pro ulehčení práce se znaky ve vícero kódováních. Interně je Char reprezentován jako Unicode znak pomocí 32 bitů, ale externě se může chovat jako kdyby měl kódování jiné. Podporovaná kódování jsou ASCII, UTF-8, UTF-16 nebo Unicode. Jednotlivé kódování jsou také reprezentovány třídami, viz. obrázek 4.9

Třída Char nebude mít definovaný literál, jediný způsob jak Char vytvořit je použít jeden z jeho konstruktorů. Mimo jiné poskytuje Char užitečné metody v podobě predikátů a konverzí, viditeln dostupné v obrázku 4.10. Také ještě operátory, uvedene v tabulce 4.4 podle kterých je možné jednoduše vytvářet nové znaky.



Obrázek 4.9 Digram kódování znaků. Vlastní obrázek.

Char
Char(Int code) Char(String string)
Encoding getEncoding()
Bool isLower() Bool isUpper() Bool isLetter() Bool isDigit() Bool isIdent() Bool isWhitespace() Bool isDiacritical() Bool isCombining()
Char toLower() Char toUpper() Char toNodia() Char toGeneral()
Char toUnicode() Char toAscii() Char toUtf8() String toCombining()

Obrázek 4.10 Metody třídy Char. Vlastní obrázek.

- `Char(Int code)`, vytvoří Char s kodem znaku v Unicode.

- `Char(String str`, vytvoří Char podle prvního znaku přiloženého String.
- `isDiacritical` vrací true, pokud je znak s diakritikou.
- `isCombining` vrací true, pokud je znak kombinující diakritika z Unicode.
- `toNodia` odstraní diakritiku.
- `toGeneral` odstraní diakritiku a převede na velké písmo.
- `toCombining` převede Unicode na retezec obsahující písmeno + kombinující diakritiku.

Výraz	Třída výsledku	Význam
`Char + Int`	Char	Zvýšení externí hodnoty znaku o číslo, nepřetéká (zastaví se na maximum)
`Char - Int`	Char	Zmenšení externí hodnoty znaku o číslo, nepodtéká (zastaví se na minimum)
`Char - Char`	Int	Vzdálenost mezi znaky
`++Char`	Char	Prefixový znakový inkrement, hodnota se změní jako při přičtení jedničky
`--Char`	Char	Prefixový znakový dekrement, hodnota se změní jako při přičtení jedničky
`Char++`	Char	Postfixový znakový inkrement, hodnota se změní jako při přičtení jedničky
`Char--`	Char	Postfixový znakový dekrement, hodnota se změní jako při přičtení jedničky

Tabulka 4.4 Operátory třídy Char.

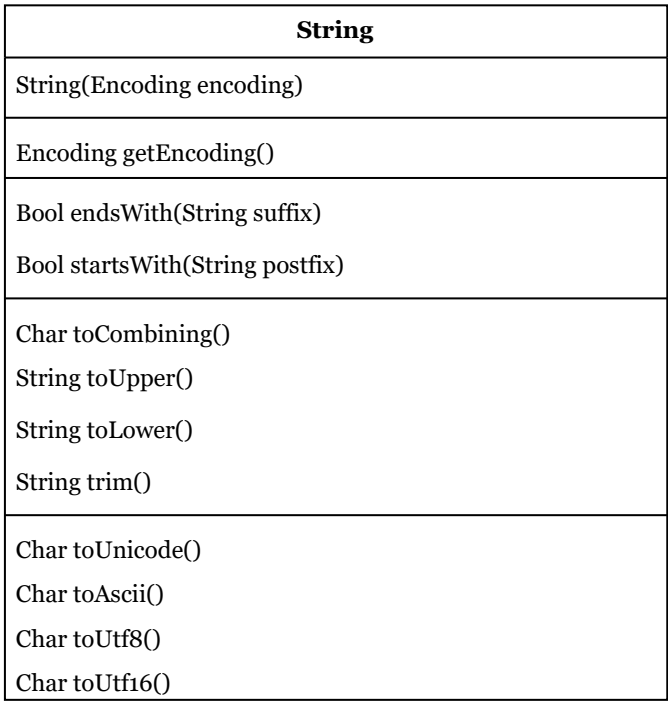
4.2.1.7 Řetězec

Řetězec představuje třída String, která jako znak je zakódovaná buďto v ASCII, UTF-8, UTF-16 nebo Unicode. String s výchozí kódování je možné vytvořit řetězcovým literálem.

Výraz	Třída výsledku	Význam
`String + String`	String	Zřetězení dvou řetězců
`String - String`	String	Odstranění jednoho výskytu řetězce v řetězci
`String  Uint`	String	Replikace řetězce
`String / String`	String	Odstranění všech výskytů řetězce v řetězci
`String % String`	Int	Odstranění všech výskytů řetězce v řetězci
`String == String`	Bool	Porovnání na rovnost
`String == String`	Bool	Porovnání na nerovnost
`String < String`	Bool	Menší než

`String` ≤ `String`	Bool	Menší rovno než
`String` > `String`	Bool	Větší než
`String` ≥ `String`	Bool	Větší rovno než
`String[Int]`	String	Vrací znak na pozici poskytnutého čísla
`String[Int, Int]`	String	Podřetězec pomocí indexu a délky, záporný levý index se počítá od konce, záporná délka znamená vybírá podřetězec nalevo od indexu

Tabulka 4.5 Operátory třídy String.



Obrázek 4.11 Metody třídy String.

4.2.1.8 Konverze mezi základními typy

Mezi základními typy jsou definované výchozí konverze následovně:

- Z `Bool` na `Int`, `Uint` – false jako 0, true jako 1.
- Z `Bool` na `Char` – znak 0, pokud false, jinak 1.
- Z `Bool` na `String` – jednoznakový řetězec s 0, pokud false, jinak s 1.
- Z `Bool` na `Float` – 0 pokud false, 1 pokud true
- Z `Uint` na `Int` a zpátky – standardní C reinterpretace.
- Z `Char` na `Int`, `Uint` – interní kód znaku

- Z `Float` na `Int`, `Uint` – zaokrouhlení na nejbližší celé číslo.
- Z `Int`, `Uint`, `Float` na `Bool` – false, pokud je číslo 0, jinak true.
- Z `Int`, `Uint` na `Char` – znak ve výchozím kódování s hodnotou konvertovaného čísla
- Z `Int`, `Uint` na `String` – konverze čísla do formy řetězce ve výchozím kódování.
- Z `Int`, `Uint` na `Float` – přidání desetinné části.
- Z `Char` na `Bool` – true pro následující hodnoty `Y, y, 1, T, t`, jinak false.
- Z `String` na `Bool` – true, začíná-li řetězec na jeden z těchto znaků `Y, y, 1, T, t`, jinak false.
- Z `String` na `Int` – zparsování čísla z řetězce.

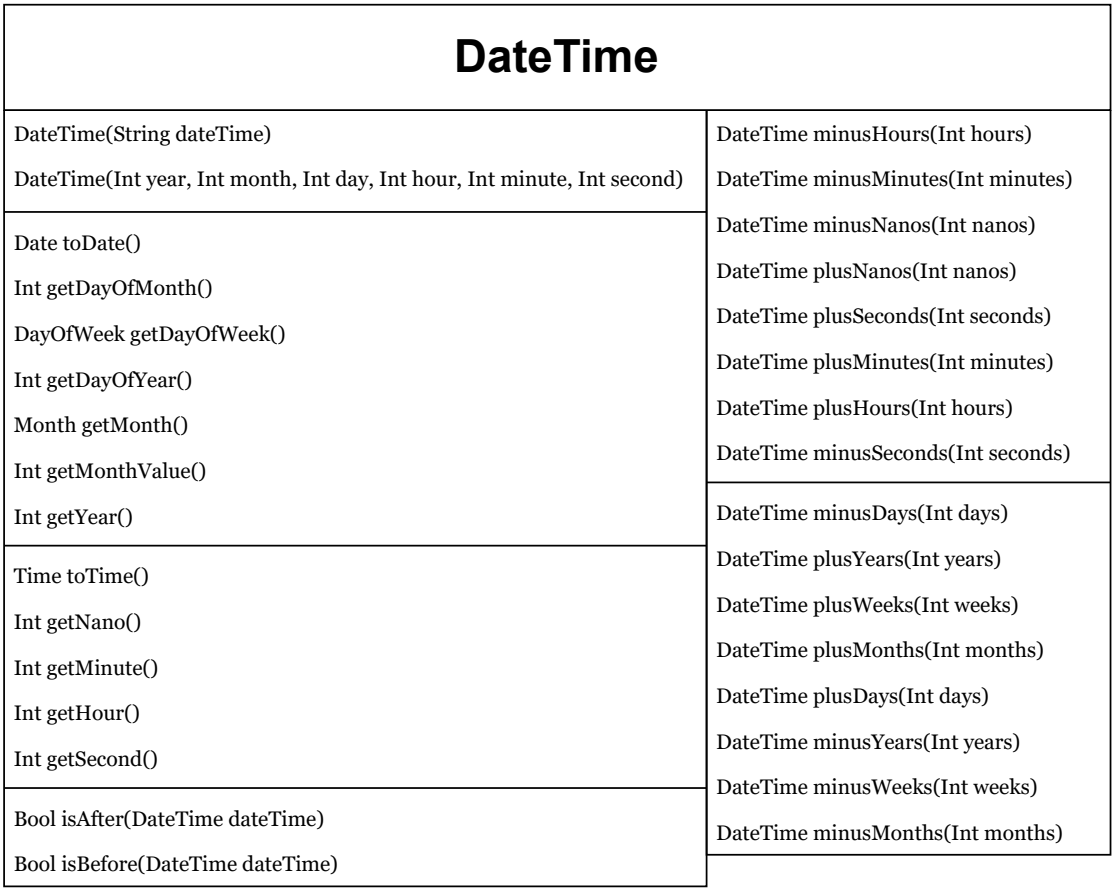
4.2.2 Časové typy

Časové typy jsou Date, Time a DateTime reprezentují absolutní čas a třída `DeltaTime` relativní čas, trvání. Návrh je inspirovány jazykem DCL a TPP. Pro inicializaci časových typů jsou musí být použité normální funkce, alternativně konverze poskytují jednotlivé třídy konstruktory. Výchozí konverze mezi časovými typy jsou pouze dvě, při konverzi z DateTime do Date se ztratí Time položka a při konverzi z DateTime do Time se ztratí Date položka.

- Funkce `dateNow`, `timeNow`, `DateTimeNow` navrátí hodnoty podle současného času.
- Každá časová třída má také konstruktor, který z řetězce ve formátu stejném jako v jazykce DCL vytvoří příslušnou třídu, například `DateTime("11-DEC-2002:13")`.

4.2.2.1 DateTime

Třída DateTime reprezentuje specifický čas a specifické datum. Poskytnuté metody jsou samovysvětlující.



Obrázek 4.12 Metody třídy DateTime. Vlastní obrázek.

Výraz	Třída výsledku	Význam
DateTime + DeltaTime	DateTime	Přičtení relativního času
DateTime + Int	DateTime	Přičtení počtu dnů
DateTime – DateTime	DeltaTime	Relativní čas mezi dvěma DateTime
DateTime – DeltaTime	DateTime	Odečtení relativního času
DateTime – Int	DateTime	Odečtení počtu dnů
DateTime :: String	String	Formátování řetězce
DateTime == DateTime	Bool	Porovnání na rovnost
DateTime != DateTime	Bool	Porovnání na nerovnost
DateTime < DateTime	Bool	Menší než
DateTime ≤ DateTime	Bool	Menší rovno než
DateTime > DateTime	Bool	Větší než
DateTime ≥ DateTime	Bool	Větší rovno než

Tabulka 4.6 Operátory třídy DateTime.

4.2.2.2 Date

Date reprezentuje datum pomocí roku, měsíce a dnu. Poskytnuté metody jsou samovysvětlující.

Výraz	Třída výsledku	Význam
Date + Time	DateTime	spojení data a času
Date + DeltaTime	Date	přičtení relativního času (v celých dnech)
Date + Int	Date	přičtení počtu dnů
Date – Date	DeltaTime	rozdíl dvou dat
Date – DeltaTime	Date	odečtení relativního času (v celých dnech)
Date – Int	Date	odečtení počtu dnů
Date :: String	String	Formátování řetězce
Date == Date	Bool	Porovnání na rovnost
Date != Date	Bool	Porovnání na nerovnost
Date < Date	Bool	Menší než
Date ≤ Date	Bool	Menší rovno než
Date > Date	Bool	Větší než
Date ≥ Date	Bool	Větší rovno než

Tabulka 4.7 Operátory třídy Date.

4.2.2.3 Time

Třída Time reprezentuje absolutní čas v rámci jednoho dne. Poskytnuté metody jsou samovysvětlující.

Výraz	Třída výsledku	Význam
Time + DeltaTime	Time	Přičtení času
Time + Int	Time	Přičtení počtu sekund
Time – DeltaTime	Time	Odečtení času
Time – Int	Time	Odečtení počtu sekund
Time :: String	String	Formátování řetězce
Time == Time	Bool	Porovnání na rovnost
Time != Time	Bool	Porovnání nerovnost
Time < Time	Bool	Menší než
Time ≤ Time	Bool	Menší rovno než
Time > Time	Bool	Větší než
Time ≥ Time	Bool	Větší rovno než

Tabulka 4.8 Operátory třídy Time.

Time
Time(String time) Time(Int hour, Int minute, Int second)
DateTime atDate(Date date)
Int getHour() Int getMinute() Int getSecond()
Bool isAfter(Time that) Bool isBefore(Time that)
Time minusHours(Int hours) Time minusMinutes(Int minutes) Time minusNanos(Int nanos) Time minusSeconds(Int seconds) Time plusHours(Int hours) Time plusMinutes(Int minutes) Time plusSeconds(Int seconds)
Int toSecondOfDay()

(a) Metody třídy Time. Vlastní obrázek.

Date
Date(String date) Date(Int year, Int month, Int dayOfMonth)
DateTime asStartOfDay() DateTime atTime(Time time) Int getDayOfMonth() Int getDayOfYear() Int getMonthValue() Int getYear()
Int lengthOfMonth() Int lengthOfYear() Bool isLeapYear()
Date minusDays(Int days) Date minusMonths(Int months) Date minusWeeks(Int weeks) Date minusYears(Int years) Date plusDays(Int days) Date plusMonths(Int months) Date plusWeeks(Int weeks) Date plusYears(Int years)

(b) Metody třídy Date. Vlastní obrázek.

Obrázek 4.13 Metody tříd Float a Int.

4.2.2.4 DeltaTime

DeltaTime
DeltaTime(DateTime from, DateTime to) DeltaTime(Int seconds)
Int getDays() Int getHour() Int getMinutes() Int getSeconds()

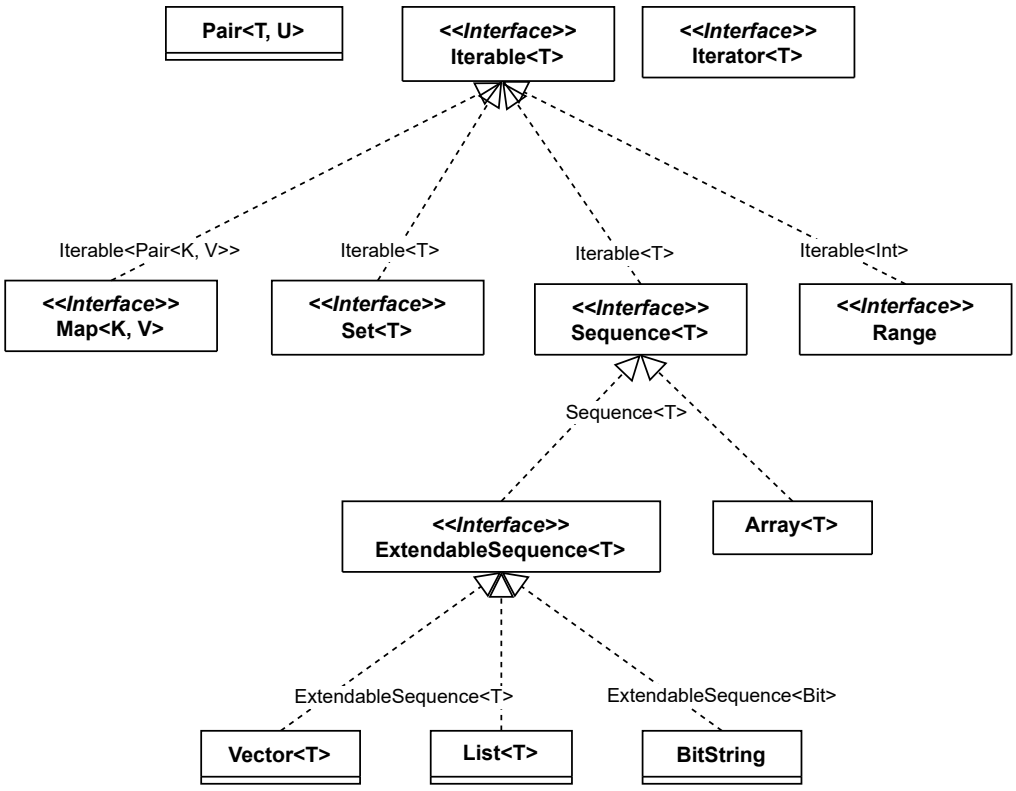
Obrázek 4.14 Metody třídy DeltaTime. Vlastní obrázek.

DeltaTime představuje relativní čas, potažmo trvání. Poskytnuté metody jsou samovysvětlující.

Výraz	Třída výsledku	Význam
DeltaTime + DeltaTime	DeltaTime	Přičtení času
DeltaTime + Int	DeltaTime	Přičtení počtu sekund
DeltaTime – DeltaTime	DeltaTime	Odečtení času
DeltaTime – Int	DeltaTime	Odečtení počtu sekund
DeltaTime :: String	String	Formátování řetězce
DeltaTime == DeltaTime	Bool	Porovnání na rovnost
DeltaTime != DeltaTime	Bool	Porovnání nerovnost
DeltaTime < DeltaTime	Bool	Menší než
DeltaTime ≤ DeltaTime	Bool	Menší rovno než
DeltaTime > DeltaTime	Bool	Větší než
DeltaTime ≥ DeltaTime	Bool	Větší rovno než

Tabulka 4.9 Operátory třídy DeltaTime.

4.2.3 Kolekce



Obrázek 4.15 Hierarchie kontejnerových tříd. Vlastní obrázek.

Kolekce nám umožňují uchovávat a manipulovat se skupinou prvků najednou. Návrh kolekcí v jazyku dclsh se inspiroval kolekcemi z jazyků Scala [31] a Kotlin

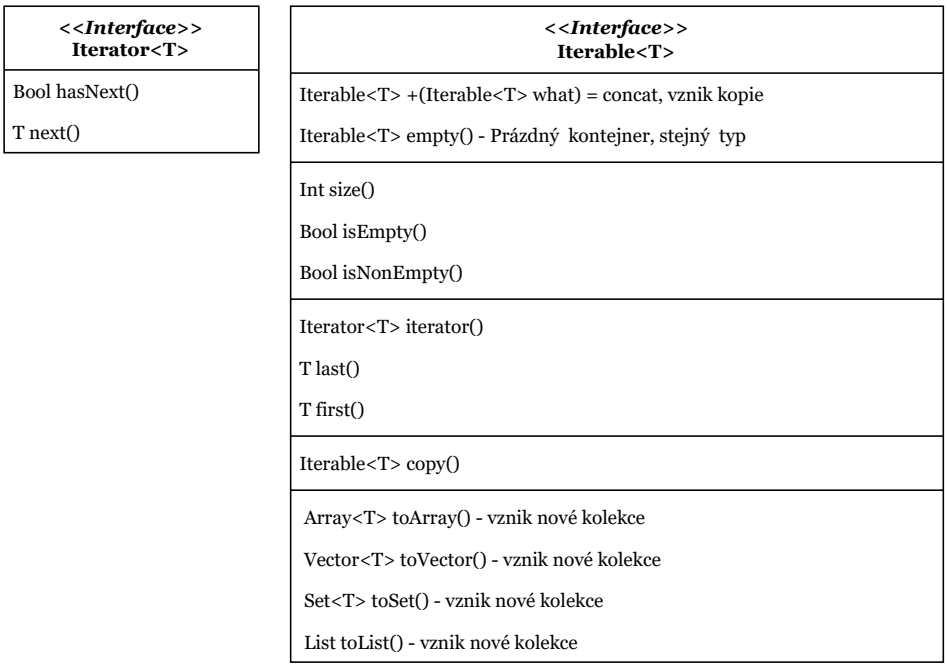
[32]. Pro stručnost předpokládáme konceptuální znalost probíraných kolekcí, jelikož se sémanticky neliší od jiných jazyků a také podrobný popis a rozbor kolekcí by mohla být závěrečná práce sama o sobě.

4.2.3.1 Iterable a Iterator

Na vrcholu hierarchie tříd kolekcí stojí třída `Iterable`, poskytující základní metody, které implementují všechny ostatní kolekce. Konverzní metody jsou pouze obyčejné metody, nejsou to speciální konverzní funkce, avšak jako při všech ostatních konverzích se vytváří nová hodnota.

Přes všechny kolekce je možné iterovat pomocí iterátoru, reprezentovaným rozhraním `Iterator`. Všechny kolekce musí implementovat svůj vlastní `Iterator`. Poskytnuté funkce iterátoru v rozhraní jsou:

- `Bool hasNext()` vrátí true, pokud současný element, na který ukazuje iterátor je definovaný.
- `T next()` vrátí současný element a posune iterátor dále, pokud je dostupný následující element.

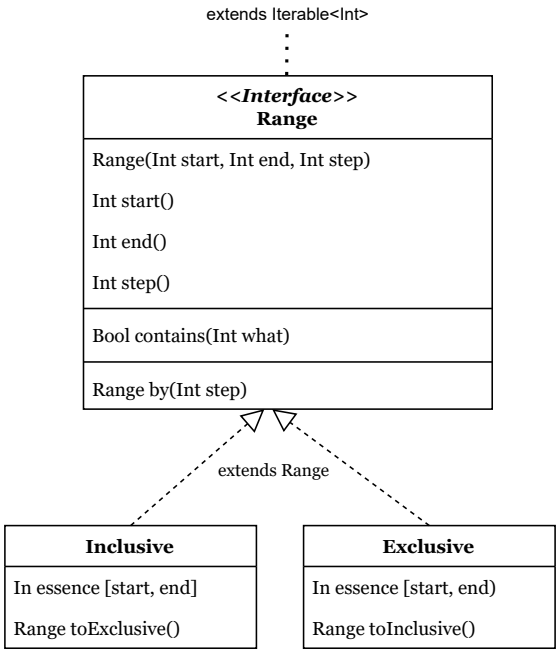


Obrázek 4.16 Diagram rozhraní `Iterator` a `Iterable`. Vlastní obrázek.

4.2.3.2 Range

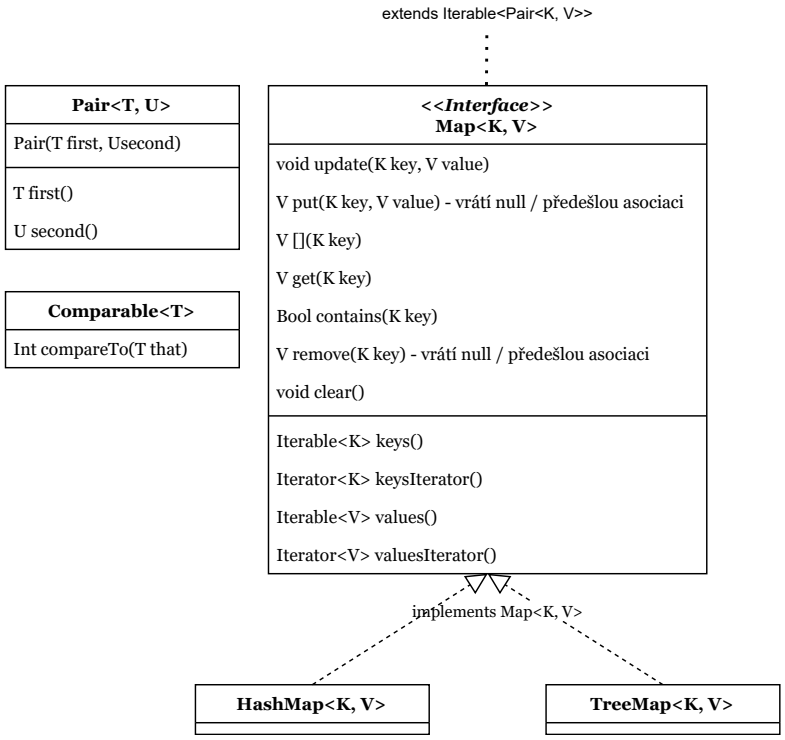
Kolekce, implementující rozhraní `Range`, efektivně uchovávají posloupnost celých čísel. Vnitřní reprezentace potřebuje pouze číslo pro začátek, pro konec a také číslo, udávající vzájemnou rovnoměrnou vzdálenost mezi jednotlivé prvky.

Řadu čísel včetně koncového prvku reprezentuje třída `Inclusive`, řadu čísel mimo koncový prvek reprezentuje třída `Exclusive`.



Obrázek 4.17 Diagram rozhraní `Range` a implementujících tříd `Inclusive` a `Exclusive`. Vlastní obrázek.

4.2.3.3 Asociativní a množinové kontejnery Map a Set

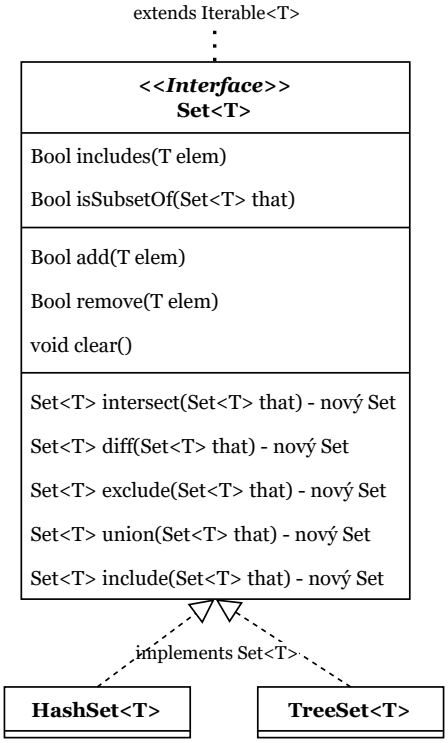


Obrázek 4.18 Diagram rozhraní `Map` a implementací `TreeMap` a `HashMap`. Vlastní obrázek.

Rozhraní `Map` představuje kolekci párů klíčů a hodnot (pár je reprezentován třídou `Pair`), kde nemůžou být dva páry se stejným klíčem. Typicky pokud je změněna hodnota klíče, když je již v kolekci, tak se může porušit invariant implementace kolekce, poté jsou operace nad touto kolekcí nedefinovány.

Jedna implementace rozhraní `Map` je `TreeMap`, která pro implementaci používá samovyvažující binární vyhledávací strom. Protože je nutné udržovat v tomto stromě uspořádání jednotlivých prvků, musí mít typ klíče buď to implementovanou metodu `compareTo` z rozhraní `Comparable`, nebo mít definovaný operátor `<`.

Další implementace rozhraní `Map` je `HashMap`, která pro implementaci používá hashovací tabulku. Pro efektivní implementaci musí mít klíčový typ dobře definovanou metodu `hashCode`, kterou má každá třída, protože je zděděná z typu `Object`.



Obrázek 4.19 Diagram rozhraní `Set` a implementací `TreeSet` a `HashSet`. Vlastní obrázek.

Reprezentaci množinových kolekcí zajišťuje rozhraní `Set`. `Set` prakticky funguje jako `Map`, kde klíč je zároveň i hodnota, tedy nejsou povoleny duplikátní prvky v `Set` a musí být brán zřetel na měnitelnost prvků, které jsou již v kolekci.

Jako `Map` má i `Set` dvě implementace, jednu pomocí samovyvažovacího binárního vyhledávacího stromu `TreeSet` a druhou pomocí hashovací tabulky `HashSet`. Požadavky na klíčový typ v implementaci `TreeMap` a `HashMap` jsou respektivně kladeny na typ prvku v implementacích `TreeSet` a `TreeMap`.

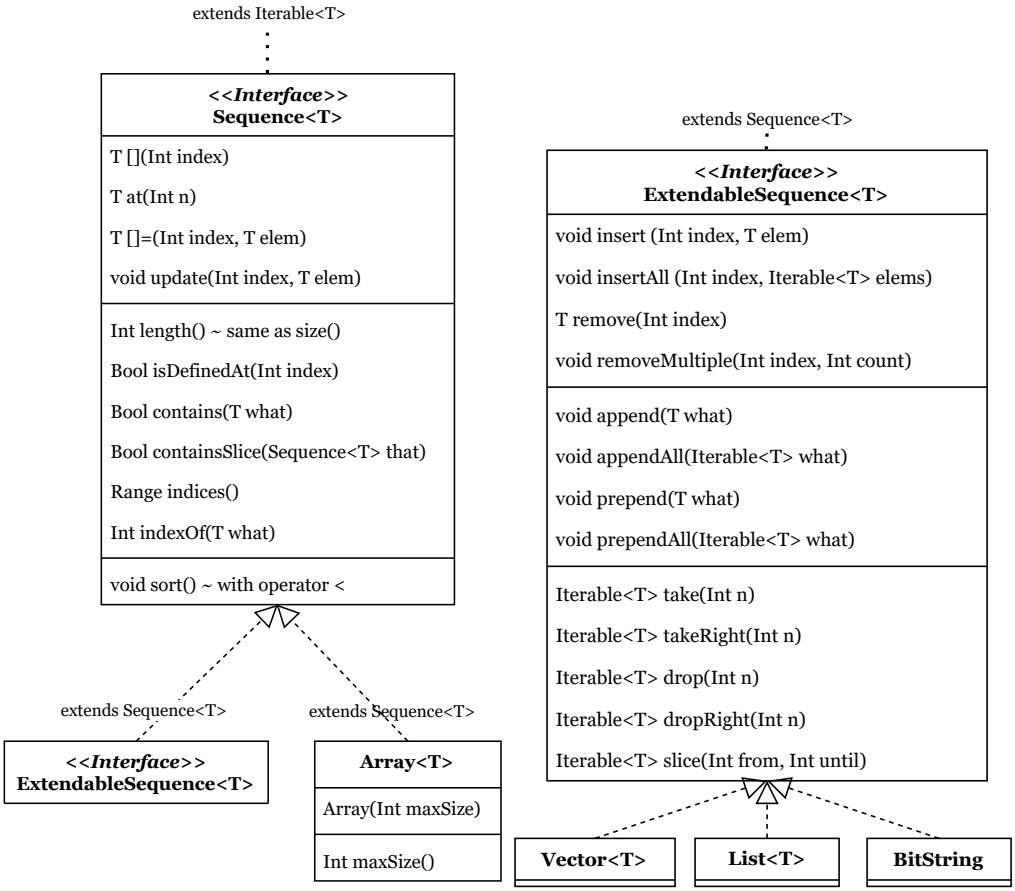
4.2.3.4 Sequence

Rozhraní `Sequence` implementují třídy reprezentující indexovatelné kolekce prvků. `Sequence` nabízí metody pro přístup a změnu uložených prvků. Třídy implemen-

tující rozhraní `ExtendableSequence` jsou `Sequence`, které navíc dokážou změnit svou velikost. Přesněji metody v rozhraní `ExtendableSequence` všechny mění velikost kontejneru, nějaké metody (`take`, `drop`, `slice`) vrací nový kontejner s vyndanými prvky.

Skoro přímou implementací `Sequence` je pevné velikosti představuje třída `Array`, která má navíc hodnotovou a ne referenční sémantiku. Navíc je plánované, že typ tohoto pole bude možné zapsat syntaxí `int[]` a při inicializaci bude možné uvést velikost pole `int[10]`.

`ExtendableSequence` implementují třídy `Vector`, `List` a `BitString`. `List` je implementován jako spojový seznam, `Vector` jako natahovací pole a `BitString` úsporně ukládá pole jednotlivých bitů a nabízí rozhraní pro jednoduchou bitovou manipulaci.



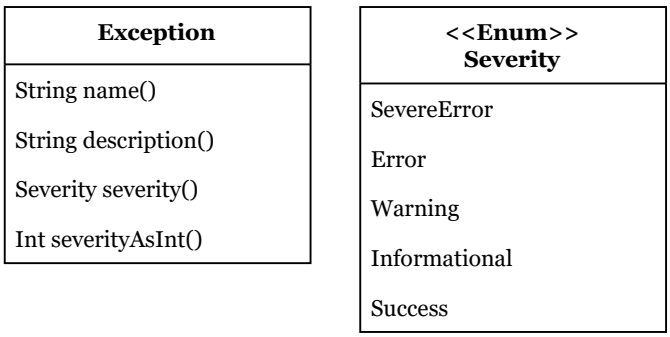
(a) Diagram rozhraní `Sequence` a třída `Array`. Vlastní obrázek. **(b)** Diagram rozhraní `ExtendableSequence` a implementací `List`, `Vector` a `BitString`. Vlastní obrázek.

Obrázek 4.20 Diagram sekvenčních kolekcí. Vlastní obrázek.

4.2.4 Výjimky

Výjimky jsou reprezentovány třídou `Exception`. Návrh je silně inspirovaný jazykem DCL:

- `name`, jméno výjimky.



Obrázek 4.21 Model třídy `Exception` a enumera závažnosti `Severity`. Vlastní obrázek.

- `description`, popis výjimky.
- `severity`, závažnost výjimky.

Jelikož se jednotlivé výjimky rozlišují navzájem od sebe pomocí závažnosti a jména, měla by se v budoucnosti zavést specifikace nebo konvence pro pojmenovávání výjimek.

4.2.5 Datové proudy

Obecně datové proudy jsou fronty, která umožňuje jak přidávání, tak odebírání prvků. Klasické datové proudy jsou implementovány cyklickou frontou (například roura).

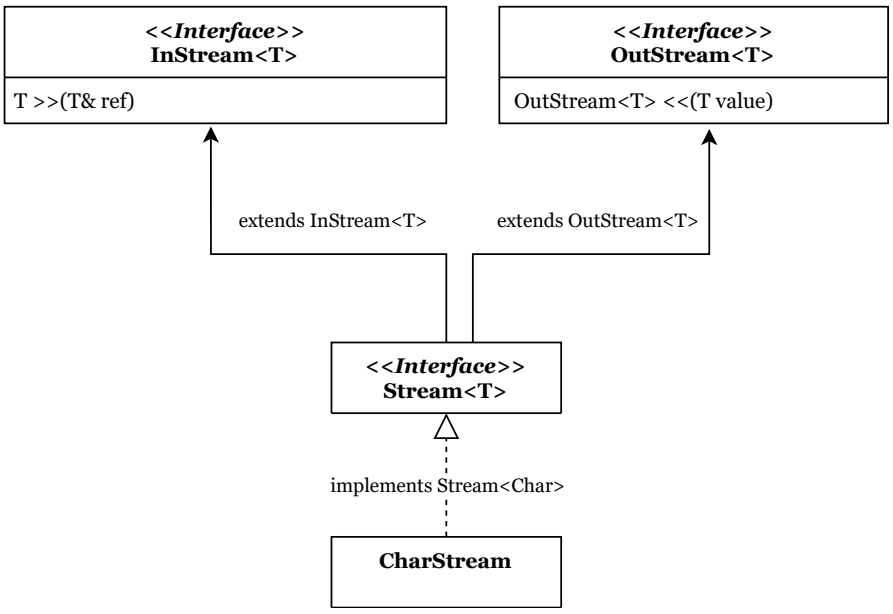
Například datový proud `Stream<Char>` může reprezentovat tok znaků z klasické roury a `Stream<String>`, může popisovat tok celých řádků.

`OutStream` je datový proud, který má již definovaného konzumenta (například soubor, do kterého zapisuje), můžeme do něj pouze prvky přidávat. `InStream` je naopak proud s již definovaným producentem (například terminál), proto je z něj možné prvky pouze vybírat.

S datovými proudy se interaguje pomocí operátorů `<<` a `>>`. Přesněji `<<` hodnoty do proudu vkládá, pokud se typy neshodují, tak se pokusí hodnotu první zkonvertovat na chtěný typ proudu a poté ji tam vloží. Operátor `>>` vkládá do přiložené reference novou hodnotu vybranou z datového proudu a vloženou hodnotu také vrací.

Pro načítání typů z klasických datových proudů `Stream<String>` a `Stream<Char>` musí každá třída `T` implementovat metodu metodu `T get(T& val, Object<Stream-<Char>, <Stream<String>>`. Tato metoda z datového proudu vyčte jednu hodnotu `T`. Volat se bude tato metoda jen v případě, kdy se bude lišit typ reference od typu proudu (`String` nebo `Char`).

Pokud je při čtení fronta prázdná, vrátí se výjimka `stmempty`, při vkládání do plné fronty se vrátí výjimka `stmfull`. Standardní ošetření těchto výjimek pozastaví vlákno a bude čekat na data/uvolnění bufferu/data z terminálu/socketu a podobně.



Obrázek 4.22 Diagram datových proudů `Stream`. Vlastní obrázek.

4.3 Správa paměti

Téma alokace a automatické dealokace paměti je rozsáhle. Pro jazyk dclsh jsme vybrali implementaci alokátoru a *garbage collectoru*, která je jednoduchá na implementaci. Později je možné toto řešení nahradit řešením, které je více přizpůsobené pro použití jazyku dclsh.

4.3.1 Alokace paměti

Programovací jazyk dclsh kontinuálně vytváří a uvolňuje deskriptory hodnot, které mají relativně malou velikost. Pro opakované malé alokace nemusí být `malloc` ideálním kandidátem, protože se s ním může pojit určitý balast, vycházející z jeho podstaty být univerzální alokátor [33]-.

Naše řešení najednou naalokuje více paměti v jednom bloku, dojde-li paměť v bloku, naalokuje se blok další. Jednotlivé bloky přidáváme do řetězu bloků.

Do každého bloku se vleze určitý počet hodnot pevné délky, které spravuje. Při alokaci najde blok volnou paměť, ke které si poznačí, že již volná není a předá na ni ukazatel. Analogicky při uvolňování paměti dostane blok ukazatel na paměť, která se má uvolnit a poznačí si, že je daná paměť volná, tedy ji může příště použít při alokaci.

Blok můžeme rozdělit na tzv. *bitfield*, ve kterém si poznačujeme, jaké hodnoty jsou volné, a na paměť pro hodnoty samotné. Jeden bit v *bitfield* odpovídá přesně jedné hodnotě a je-li bit s hodnotou 1, tak je korespondující hodnota alokovaná, v opačném případě je volná.

bitfield je na začátku bloku a po něm následuje paměť určená pro hodnoty. Můžeme si představi *bitfield* jako pole bitů a spravovanou paměť jako pole hodnot. Mapování bitů z *bitfield* na hodnoty v spravované paměti je poté pomocí jejich

vzájemných indexů. Tedy bit na indexu 0 v *bitfield* spravuje hodnotu na indexu 0 v spravované paměti.

Popisovaný alokátor je úsporný na paměť (pro každou hodnotu musíme navíc držet pouze 1 bit informace), ale při hledání volné hodnoty se musí procházet pole bitů, ve snaze najít bit s hodnotou 1. Avšak vyžadované manipulátory bitů pro implementaci tohoto alokátoru je možné využít dále při implementaci třídy `BitString` ve standardní knihovně.

4.3.2 Automatické uvolňování paměti

Vybraná implementace referenčního čítače je jedna z nejjednodušších na implementaci, přesto je nutné přesně specifikovat kdy se má reference snížit a kdy zvýšit. Například když jsou argumenty předávány pomocí zásobníku, je nutné zvýšit referenční čitač daných hodnot při přidáváná na zásobník a poté je musí podprogram zkonzumovat.

Použitý algoritmus je možné rozšířit algoritmem detekcí cyklických referencím. Jako možná inspirace by mohly sloužit rešeršované jazyky PHP a Python, které používají *garbage collector* s detekcí cyklických referencí. Alternativně by mohl být použít úplně jiný algoritmus pro automatickou správu paměti, například algoritmus na bázi `mark-and-sweep` je použit v různých implementacích jazyku *JavaScript*.

4.4 Programové rozhraní

Pomocí následujících programových rozhraní komunikuje dclsh runtime s vnějším světem. První rozhraní je mezi runtime částí a interpretem dclsh, pomocí kterého poskytuje runtime interpretu veškeré prostředky pro výpočet a případné ladění. Druhé rozhraní umožňuje uživatelům implementovat své vlastní zabudované typy v jazyce C.

4.4.1 Rozhraní mezi runtime a interpretem

Pro interpret je veškerý kontext výpočtu definovaný pomocí `Callframe` právě interpretované funkce/metody. Interpret může v daném `Callframe` řidit tok výpočtu změnou programového čítače, volat funkce/metody a přiřazovat do proměnných a případně zpracovávat výjimky. Pro komunikaci s runtime používá interpret výpočetní zásobník v `Callframe`, na který ukládá operandy nebo argumenty funkcí nebo mezivýpočty výrazů.

Při načítání modulu může interpret narazit na nové definice tříd a rozhraní. Interpret v tomto případě musí dané definice převést do interní formy a poté je postupně registrovat u runtime. Jednotlivé registrace budou probíhat postupně, tedy například budeme mít jedno volání pro registraci nové metody nebo vytvoření nového typu pomocí šablony.

4.4.2 Rozhraní pro definici nových typů v jazyku C

Uživatelé mohou definovat nové zabudované třídy pomocí mechanismu dědění – musí nový typ odvodit od již existujícího typu (například ale třeba stačí `Object`)

a mohou mu vyměnit definice metod, přidat nové metody, přidat/doplnit typová argumenty, definovat implementované rozhraní nebo dokonce přidat nové atributy.

Avšak pro rozšiřování zabudovaných tříd, typu `Int`, `Vector` je nutné znát interní implementaci, protože se nechovají jako interpretované třídy, jelikož používají `Data` segment v deskriptoru hodnoty přímo a neukládají si v něm atributy.

Nové definice třídy, které splňují vystavené C rozhraním pro definici nových typů je nutné zkompilovat do podoby DLL (Dynamic Link Library)

Kapitola 5

Implementace runtime knihovny dclsh

V tuto chvíli jsou dle návrhu implementované následující datové struktury: `Value` (hodnota), `Type` (třída), `Method` (popis metody/funkce) a `Callframe` (popis volání metody, funkce, skriptu externího programu). Jak bylo řečeno v návrhu, obsahují všechny zmiňované struktury jednu hlavičku `VALHDR`.

5.1 Správa paměťi

Navrhovaný alokátor, zakládající se na `bitfield` je implementovaný a otestovaný, avšak ještě teď není knihovnou používaný, jelikož by v tuto chvíli pouze stěžoval ladící proces. Ale moduly implementující rozhraní pro práci s bitovými hodnoty `bit_accessor` a `bitstring` budou dále užitečné při budoucí implementaci třídy `BitString`.

Implementace automatické správy automatické správy paměti je pomocí funkcí `Value *ref(Value *value)` a `void unref(Value **p)`, pomocí kterých se zvedá a snižuje počet referencí na strukturu.

Pro práci s hodnoty je definovaný následující kontrakt:

- Použití `Value *ref(Value *value)` znamená vznik nového výskytu hodnoty `value`, která je z této funkce vracená.
- Použití `void unref(Value **value)` znehodnotí odkaz na výskyt hodnoty `value` (nastaví se na `NULL`) a byl-li to poslední odkaz na danou hodnotu, uvolní se.
- Všechny metody, konstruktory, funkce vrací odkaz na `Value`, u které je již čítač referencí zvýšen, tedy není nutné volat funkci `ref`.
- Hodnoty, které jsou pouze pro čtení jsou uvnitř `ref` a `unref` ignorovány.

```
struct Value {
  /* value descriptor */
  VALHDR; /* std. header = 16 bytes */
  union {
    struct {
      char *exc_name;       /* exception name */
      char *exc_text;       /* explanation */
      uint8_t exc_severity; /* seriousness of the exception */
      char exc[31];         /* buffer */
    };
    struct {
      int64_t i_val;  /* INT */
      int64_t i_val2; /* rezerva pro odvozene typy */
    };
    struct {
      uint64_t ui_val;  /* UINT */
      uint64_t ui_val2; /* rezerva pro odvozene typy */
    };
    struct {
      double f_val;  /* FLOAT */
      double f_val2; /* rezerva pro odvozene typy */
    };
    uint64_t ch_val; /* CHAR (UTF32) */
    uint64_t b_val;  /* BOOL */
    struct {
      /* STRING, BITSTRING */
      uint32_t str_size;
      /* allocated size = maximal length in bytes including
                                  zero terminator */
      uint32_t str_length;   /* actual length (without the terminator) */
      uint8_t *str_val;      /* pointer to the string */
      union {                // STRING vs. strref
        uint8_t str_buf[32]; /* internal character buffer */
      };
    };
    struct {
      /* DATE, TIME, TIMESTAMP, DELTATIME */
      uint64_t timestamp;
    };
    struct {
      /* REF - reference to a variable */
      Value **variable; /* address of variable pointer */
      Type *var_type;   /* type of a variable */
    };
  };
};
```

Výpis kódu 3 Deskriptor hodnoty.

5.2 Implementace tříd

Pro implementaci třídy je nutné přidat do dvou `union` uvnitř `Value` (položka `Data` z návrhu) a do `union` uvnitř `Type` (položka `Type data` z návrhu) požadované datové položky, nutné pro datovou reprezentaci hodnot daného typu a potřebné typové informace.

Pak je vhodné definovat statický deskriptor typu a ten předvyplnit statickým obsahem, tedy vyplnit hlavičku `VALHDR`, jméno typu, signaturu typu a id typu. Operátory je také možné doplnit do deskriptoru typu staticky, protože pro ně má každá třída předurčené místo, metody na druhou stranu musí být alokovány dynamicky.

Každá zabudovaná třída také musí implementovat inicializační funkci (například `initializeInt`), pomocí které se inicializují dynamické prvky typového deskriptoru `Type`, tedy například se vyplní globální konverzní tabulky a případně se doplní metody. Mimo inicializační funkci je nutné definovat i funkci, pomocí které se deskriptor třídy uvolní (například `finalizeInt`).

Zabudované funkce/metody mají univerzální rozhraní `Value* (*Function)(uint32_t argc, Value **argv)`, tedy při volání je nutné předat počet argumentů a odkaz na zásobník, kde je možné příslušné argumenty najít. Návratový typ je `Value`, což může být výsledek nebo potenciálně výjimka. Metody mají tradičně jako první argument hodnotu `self`, tedy objekt, na kterém je metoda zavolaná.

V tuto chvíli jsou takto implementovány typy `Object`, `Undef`, `Null`, `Ref` (typ reference na hodnotu), `Bool`, `Exception` a `Char`.

Pro třídy `Set` a `Map` byla také vytvořena knihovna poskytující AVL strom, který si drží v každém uzlu ukazatel na rodiče, čímž dovoluje jednoduchou iteraci přes dané datové struktury. Navíc je daná implementace iterativní a ne rekurzivní, tedy neplýtvá zbytečně místem na zásobníku. Další využití pro tento kontejner bude například při implementaci metody `lookup` (vyhledávání členu objektu pomocí řetězcového jméne), alternativně kdekoliv, kde bude potřeba rychlý asociativní kontejner.

```
struct Type {
  /* value type descriptor */
  VALHDR; /* std. header */
  const char *name;
  char *signature;  /* "object<int,string>", ...*/
  uint8_t type_def; /* base type, i.e. record variant */
  uint16_t type_id; /* type index */
  Type *superclass; /* superclass descriptor or null */
  union {
    /* variants  */
    struct {
      /* type_def = STR */
      uint16_t str_attrib;
      uint16_t str_charset;
    };
    struct {
      /* type_def = VECTOR, ARRAY */
      Type *v_type; /* type of item */
    };
    struct {
      /* type_def = OBJECT */
      Type **objecttypes;
    };
  };
  uint32_t meth_allocated;
  uint32_t meth_used;
  Method **methods; /* array of method descriptors */
  Function *meth_functable; /* methods function pointers */
  struct {
    /* internal methods */
    void (*op_free)(Value *self); /* Value destructor */
    void (*op_finalize)(); /* Type destructor */
    /* helper routines */
    void (*op_dump)(Value *self, uint32_t level);
    int (*op_cmp)(Value *v1, Value *v2);
    /* reference interface */
    Value *(*op_get)(Value *v); /* get Value from reference */
    Type *(*op_gettype)(Value *v); /* get type of reference val */
    /* stream interface */
    Value *get(int32_t argc, Value **argv)
  };
  union { /* operators virtual function table */
    Function operators[F_MAX];
    struct {
      Function op_forget;
      /* ... */
      Function op_function;
    };
  };
};
```

Výpis kódu 4 Deskriptor typu.

Kapitola 6
Testování

V této kapitole se nachází popis testovací strategie a analýza napsaných testů.

Každá implementovaná třída v jazyku dclsh implementuje i speciální testovací funkci (například `Bool` má `testBool`) se signaturou `uint32_t (*) (uint32_t level)`, ve které jsou napsány testy jednotlivých metod dané třídy. Daná testovací funkce vrací počet zaznamenaných chyb.

Testy jsou spouštěny testovací terminálovou aplikací, která akceptuje čtyři přepínače:

- `-d`/`--debuglevel` určuje jaké typy ladícího hlášení chceme vypisovat. V tuto chvíli jsou čtyři různé ladící hlášení, aplikace očekává celé číslo mezi 0-15, které interpretuje jako bitové pole. Například číslo 1 (0001) určuje, že chceme pouze první typ ladícího hlášení, zatímco zvoli-li bychom 15 (1111), tak budou aktivované všechny typy ladící hlášení.
- `-l`/`--level` číselná testovací úroveň je předávána jednotlivým testovacím funkcím jako parametr. Umožňuje nastavovat vlastní chování jednotlivých testovacích funkcí.
- `-c`/`--class` přijímá čárkou oddělený list názvů tříd, které by se měli testovat.
- `-t`/`--trace` přijímá čárkou oddělený list názvů tříd, které by měli při testování tisknout ladící informace na výstup.

Můžeme ladit buďto třídy pomocí makra `TRACE`, nebo obecné mechanismy runtime pomocí makra `DEBUG`. U variant těchto maker s číslem na konci, například `TRACE8`, proběhne ladící výpis, pokud byl v příkazové řádce nastaven typ ladění odpovídající danému číslu.

Obecné mechanismy runtime se ladí pomocí maker `DEBUG`:

- `DEBUG1` ladí funkce `ref` a `unref`.
- `DEBUG2` při výpisu hodnot navíc vypisuje i jejich hlavičku.
- `DEBUG4` při porovnávání hodnot v testech navíc vypíše testované hodnoty.
- `DEBUG8` upozorní na podezřelé hodnoty referenčních čítačů při provádění testů.

Pro ladění metod používáme makra `TRACE`. Sice jsou definované i varianty `TRACE1, TRACE2, TRACE4, TRACE8`, ale v tuto dobu nejsou nutné. `TRACE` makra tisknou ladící výstup pouze u testovaných tříd, které byly vyjmenovány v přepínači `--trace`.

Testovací modul nabízí jednotlivým testovaným modulům univerzální testovací funkce (například `test2` pro testování s dvěma argumenty), které přijímají testované argumenty, ukazatel na testovanou funkci, název testu a očekávaný výsledek. Tyto univerzální testovací funkce napodobují chování interpretu pomocí instrukcí například `push`, `pop` a například `call2`.

Jakékoliv další testovací užitkové funkce si můžou moduly implementovat samy, třeba stručné konstruktory hodnot a tak podobně.

```
#define I0 &int_zero
...
#define S01 &str_minusone
#define F &bool_false
#define T &bool_true
#define SF &str_false
...
#define CT &char_true

uint32_t testBool(uint32_t level) {
  test1(F, Bool_not, T, "! False");
  test1(T, Bool_not, F, "! True");
  test2(F, F, Bool_or, F, "False || False");
  test2(F, T, Bool_or, T, "False || True");
  ...
  test2(F, T, Bool_xor, T, "False ^^ True");
  test2(T, F, Bool_xor, T, "True ^^ False");
  test2(T, T, Bool_xor, F, "True ^^ True");
  ...
  /* Updates */
  test2r(F, F, Bool_upd_or, F, "False ||= False");
  test2r(F, T, Bool_upd_or, T, "False ||= True");
  test2r(T, F, Bool_upd_or, T, "True ||= False");
  test2r(T, T, Bool_upd_or, T, "True ||= True");
  ....
  /* Conversions */
  test1(I0, Bool_not, T, "! 0");
  ...
  test2(T, ST, Bool_or, T, "True || \"True\"");

  return errors;
}
```

Výpis kódu 5 Testovací funkce třídy `Bool`.

```
#define I0 &int_zero
#define I1 &int_one
#define I01 &int_minusone
#define I(N) newInt(N)
...
#define F &bool_false
#define SC(N) constString(N)

uint32_t testInt(uint32_t level) {
  test2r(I1, I1, Int_upd_add, I(2), "a += b");
  test2r(I(1), I1, Int_upd_sub, I0, "a -= b");
  ...
  test2r(I(2), I(1), Int_upd_sh_right, I(1), "a >>= b");
  ...
  test2(I(1), I(1), Int_eq, T, "a == b");
  test2(I(1), I(2), Int_eq, F, "a == b");
  ...
  test1(I(1), Int_bit_not, I(~1), "~a");

  test1(I0, Int_not, T, "! 0");
  test1(I1, Int_not, F, "! 1");
  ...
  test2(I1, I1, Int_or, T, "1 || 1");
  ...
  /* Conversions */
  test2(I(1), S1, Int_add, I(2), "1 + \"1\"");
  ...
  test2(SC("123"), I(100), Int_sub, I(23), "\"123\" - 100");
  /* Methods */
  test1(I(2), Int_abs, I(2), "abs(2)");
  ...
  test2(I(2), I(1), Int_max, I(2), "max(2, 1)");

  return errors;
}
```

Výpis kódu 6 Testovací funkce tříd `Int`

```
$ ./dclshtest -l 1 -c bool,int

Test level = 1
Test  5.01: bool ! False          OK
Test  5.02: bool ! True            OK
Test  5.03: bool False || False    OK
Test  5.04: bool False || True    OK
Test  5.05: bool True || False    OK
...
Test  5.66: bool False || "True"  OK
Test  5.67: bool True || "False"  OK
Test  5.68: bool True || "True"    OK
Test  5:    bool                  OK
Test  6.01: int a += b            OK
Test  6.02: int a -= b            OK
...
Test  6.55: int 1 + '1'          OK
Test  6.56: int '1' - '0'         OK
Test  6.57: int "123" - 100      OK
Test  6.58: int abs(2)            OK
Test  6.59: int abs(-2)          OK
Test  6.60: int min(1, 2)        OK
Test  6.61: int min(2, 1)        OK
Test  6.62: int max(1, 2)        OK
Test  6.63: int max(2, 1)        OK
Test  6:    int                  OK
Summary:  2 test(s), 131 subtest(s), 0 error(s)
```

Výpis kódu 7 Výpis testů třídy Bool a Int.

Kapitola 7

Závěr

Cílem práce bylo vytvořit provést rešerši skriptovacích jazyků, popsat jazyk dclsh a navrhnout, implementovat, otestovat a zdokumentovat runtime knihovnu jazyka dclsh.

Byl položen teoretický základ na téma konceptů programovacích jazyků. Dále proběhla rešerše skriptovacích interpretovaných jazyků, včetně shellů. Poté byl popsán jazyk dclsh a jeho rozdělení na interpretovou a runtime část.

V další části proběhl návrh runtime části jazyku dclsh, včetně rozhraní pro definici nových typů v jazyku C a rozhraní, pomocí kterého bude interpret s runtime částí pracovat. Byl navrhnut objektově orientovaný typový systém, včetně konverzního aparátu, podpory rozhraní a standardní knihovny. Dále byl navržen způsob jak reprezentovat výjimky, volání funkcí, metod, dalších skriptů a externích programů. Také byl popsán návrh mechanismu automatické správy paměti pro jazyk dclsh.

Možné budoucí zlepšení současného návrhu by mohlo být doplnění automatického systému správy paměti o mechanismus detekce cyklů. Použity by mohly být algoritmy zmiňované v rešerši skriptovacích jazyků. Alternativně by nemuselo být na škodu zvážit jinou odnož algoritmů, které nejsou založené na počítání referencí. Toto zlepšení má cenu zvažovat až po dokončení interpretu, aby bylo možné porovnávat jednotlivé řešení automatické správy paměti na opravdových úlohách a zátěžích.

Také byla mírně opomenutá otázka podpory funkcionálního programovacího stylu v jazyku dlcsh. V současné době je možné ručně vytvořit funkce vyššího řádu pomocí objektově orientovaného aparátu. Například pomocí rozhraní `Function1<Return, Param>` s funkcí `Return invoke(Param param)`, jenž slouží jako objektová reprezentace funkce, kterou je možné předat jako parametr nebo navrátit z jiné funkce. Sémanticky se tento konstrukt chová jako funkce vyššího řádu. Avšak aby bylo funkcionální programování v jazyku dclsh příjemné, měl by interpret poskytnout stručný syntax pro typování funkcí a vytváření anonymních funkcí.

Obdobně objektově orientované programování v jazyku dclsh by mohlo být v budoucnosti rozšířené o modifikátory přístupu a možnost definovat statické členy nebo abstraktní členy. Jako inspirace pro implementaci by mohl sloužit jazyk PHP, který tyto funkčnosti implementuje. Tyto rozšíření avšak mají i svoje nevýhody, jelikož komplikují jazyk samotný a s největší pravděpodobností vyžadují navíc

kontroly za běhu.

Z důvodu časové náročnosti a velikosti projektu byla naprogramovaná relativně malá podmnožina celkového návrhu. Navíc velká část řešení nebyla vytvořená námi, ale vedoucím této práce. Rozdělení autorství jednotlivých zdrojových souborů je dostupné v textovém souboru `CONTRIBUTIONS` v přiloženém řešení. Ve zkratce vznikl naší zásluhou alokátor, AVL strom a typ `Bool`.

Práce na jazyku dclsh pro nás tímto nekončí. Myslíme si, že jazyk dclsh má zajímavé prvky, které ho dělají unikátním, například sémantika typu `Object` a podpora souběžného výpočtu pomocí datových proudů, a proto má smysl na něm dále pracovat.

Příloha A

Ukázky kódu

A.1 Bash

```
#!/bin/bash
# Shebang definuje, že chceme použít bash
# <- Komentář

echo $1 # vypíše první poziční parametr skriptu

# příkaz echo zavolaný s dvěma řetězcovými argumenty
echo "Hello" "World"

# přesměrování výstupu příkazu do souboru
echo "Hello World!" > file

wc -w file # 2, použití přepínače
wc -w < file # 2, soubor přesměrován na vstup příkazu

# proměnná použitelná v aritmetice, pokud obsahuje numerickou hodnotu
my_num=5
echo $(( 2 + my_num )) # 7

# Expanze speciálních znaků
echo * # Vypíše obsah současného adresáře
echo \* # *, potlačení expanze speciálních znaků (taky quoting)
```

Výpis kódu 8 Ukázka základního bash skriptu.

```
# použití roury přesměrovává výstup prvního příkazu
# do vstupu následujícího příkazu
echo "Hello world!! | wc -w # 2

# Hello World, vykonání 2 příkazů v sekvenci
echo "Hello"; echo "World"
# vykoná následující příkaz, jestli předešlý příkaz uspěl
false && echo "Hello" # nevypíše nic
# vykoná následující příkaz, jestli předešlý příkaz neuspěl
false || echo "Hello" # Hello

# vytvoří nový proces na pozadí, terminál je použitelný a nečeká
sort /usr/dict/words > file &
```

Výpis kódu 9 Ukázka rourování, přesměrování a sekvencování.

```
# test příkaz pro aritmetické, logické, řetězcové testy
test 0 -eq 0 # návratový kód 0
test 0 -eq 1 # nenulový návratový kód
[ 0 -eq 0 ] # Odlišný zápis test

if <prikaz-if>; then # vykoná se první, typicky příkaz test
    <prikazy> # vykonají se, pokud <vyraz-if> vrátil za kód nulu
else <prikazy> # jinak se vykonají tyhle výrazy
fi

# příklad definice funkce
hello_world () {
   echo 'hello, world'
   return 10
}
# zavolání funkce
hello_world # návratový kód 10
```

Výpis kódu 10 UKázka řídících struktur v bashi.

A.2 DCL

```
$ ! <- komentář
$ ! Většina příkazů nerozlišuje mezi malými a velkými písmeny
$ ! Příkazy přiřazení
$ ! = <- lokální přiřazení hodnoty
$ ! == <- globální přiřazení hodnoty
$ ! := <- lokální přiřazení řetězcové hodnoty
$ ! :== <- globální přiřazení řetězcové hodnoty
$ ! <- ve skriptu jsou příkazy odděleny $
$ a="1" ! přiřazení STRING do proměnné
$ b="2"
$ c= a+b ! "12"
$ d=1 ! přiřazení INTEGER do proměnné
$ e=d+b ! 3, přetížení operátorů
```

Výpis kódu 11 Ukázka práce s proměnnými v DCL. Inspirace z [9].

```
$ ! @file param1 param2 ...
$ ! Volání procedury s parametrem "delete"
$ ! Parametr v proceduře viditelný pod pozičním parametrem p1
$ @clean-up delete
$ ! Obdobně volání podprogramu nastaví "delete" jako p1 a skočí na label
$ ! CALL routine param1 param2
$ call some-label delete ! návratový kód 1 v proměnné STATUS
$ ! gosub label
$ gosub another-label ! návratový kód 0 v proměnné STATUS
$
$ some-label: SUBROUTINE ! začátek definice, nová tabulka proměnných
$
$ EXIT 1 ! návratová hodnota
$ ENDSUBROUTINE ! konec definice
$
$ other-label:
$ RETURN 1
```

Výpis kódu 12 Ukázka volání procedur a podprogramů. Inspirace z [9]

```
$ ! nahrad' od znaku offset délku size řetězcem "text"
$ ! A[offset,size] :="text"
$ ! nahrad' od bitu offset délku size hodnotou výrazu
$ ! A[offset,size] = expresssion
$
$ A="" ! pole 8 znakových slov
$ A[0*8,8]:=ASDFGH
$ A[3*8,8]:=QWERTY
$ b=F$EXTRACT(3*8,8,A) ! "QWERTY  "
$
$ COUNT=3
$ BARK := P'COUNT' ! "P3", proběhla substituce
$ ! pomocí substituce je možné implementovat pole
$ INDEX=0
$ array_'INDEX'=value ! array_0=value
$ array_'INDEX':=string ! array_0:=string
$ x=array_'INDEX' ! x=array_0
```

Výpis kódu 13 Ukázka implementace pole v DCL. Inspirace z [9]

```
$ if 1 .lt. 2
$ then
$   show time
$ else
$   show memory
$
$ ! Příklad smyčky, FOR, WHILE a tak podobně nejsou dostupné
$ count = 0
$loop: count = count + 1
$ if count .gt. 10 then goto endloop
$ show users
$ goto loop
$endloop:
```

Výpis kódu 14 Ukázka řízení kontroly toku programu v DCL. Inspirace z [9].

```
$ ! pipe command separator command ...
$ ! separátory se chovají jako v bash ||, &&, <, >, |
$ pipe cc/decc sourcefile/object=objectfile && link objectfile
$ pipe cmd1 | sort | cmd2
```

Výpis kódu 15 Ukázka příkazu pipe. Inspirace z [9].

```
$ ! soubory byly zařazeny do fronty dávkového zpracování LN03_PRINT
$ PRINT/QUEUE=LN03_PRINT SATURN.TXT,EARTH.TXT
$ ! 2 kopie danných souborů
$ PRINT/COPIES=2 SPRING.SUM,FALL.SUM
$ ! 2 kopie SPRING.SUM, 1 kopie FALL.SUM
$ PRINT SPRING.SUM/COPIES=2,FALL.SUM
```

Výpis kódu 16 Ukázka příkazu pipe. Inspirace z [9].

A.2.1 CDL

```
QUALIFIER Q
PARAMETER P1, VALUE(TYPE=COLORS)
DISALLOW Q AND P1.RED
```

Výpis kódu 17 Příklad DISALLOW, inspirace z[11].

Příloha B

Operátory jazyku dclsh

Usage	Operation	Precedence	Associativity
`a,b`	Comma	15	Left-to-right
`a = b`	Simple assignment	14	Right-to-left
`a += b`	Assignment by sum	14	Right-to-left
`a -= b`	Assignment by difference	14	Right-to-left
`a *= b`	Assignment by product	14	Right-to-left
`a /= b`	Assignment by quotient	14	Right-to-left
`a %= b`	Assignment by remainder	14	Right-to-left
`a <<= b`	Assignment by bitwise left shift	14	Right-to-left
`a >>= b`	Assignment by bitwise right shift	14	Right-to-left
`a &= b`	Assignment by bitwise AND	14	Right-to-left
`a ^= b`	Assignment by bitwise XOR	14	Right-to-left
`a |= b`	Assignment by bitwise OR	14	Right-to-left
`a &&= b`	Assignment by AND	14	Right-to-left
`a ^^= b`	Assignment by XOR	14	Right-to-left
`a ||= b`	Assignment by OR	14	Right-to-left
`a ? b : c`	Ternary conditional	13	Right-to-left
`a || b`	Logical OR	9	Left-to-right
`a ^^b`	Logical XOR	9	Left-to-right
`a && b`	Logical AND	8	Left-to-right
`a == b`	Relational =	7	Left-to-right
`a != b`	Relational $\neq$	7	Left-to-right
`a < b`	Relational operator $<$	6	Left-to-right
`a <= b`	Relational operator $\leq$	6	Left-to-right
`a > b`	Relational operator $>$	6	Left-to-right
`a >= b`	Relational operator $\geq$	6	Left-to-right
`a::b`	Format/convert	5	Left-to-right
`a << b`	Bitwise left shift	5	Left-to-right
`a >> b`	Bitwise right shift	5	Left-to-right
`a | b`	Bitwise OR (inclusive or)	4	Left-to-right
`a ^b`	Bitwise XOR (exclusive or)	4	Left-to-right

`a + b`	Addition	4	Left-to-right
`a - b`	Subtraction	4	Left-to-right
`a & b`	Bitwise AND	3	Left-to-right
`a * b`	Multiplication	3	Left-to-right
`a / b`	Division	3	Left-to-right
`a % b`	Remainder	3	Left-to-right
`++a`	Prefix increment	2	Right-to-left
`--a`	Prefix decrement	2	Right-to-left
`+a`	Unary plus	2	Right-to-left
`-a`	Unary minus	2	Right-to-left
`!a`	Logical NOT	2	Right-to-left
`~a`	Logical bitwise NOT	2	Right-to-left
`&a`	Reference	2	Right-to-left
`a++`	Suffix/postfix increment	1	Left-to-right
`a--`	Suffix/postfix decrement	1	Left-to-right
`(type) a`	Cast	0	Left-to-right
`a[b]`	Array subscripting	0	Left-to-right
`a[b:c]`	Array range subscripting	0	Left-to-right
`a.b`	Structure and union member access	0	Left-to-right
`a.b(...)`	Method call	0	Left-to-right
`a(...)`	Function call	0	Left-to-right
`(type){list}`	Compound literal	0	Left-to-right

Tabulka B.1 Operátory jazyku dclsh. Tabulka inspirována tabulkou operátorů v C.[34]

Bibliografie

1. GREENBERG, Michael; KALLAS, Konstantinos; VASILAKIS, Nikos. Unix shell programming: the next 50 years. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. Ann Arbor, Michigan: Association for Computing Machinery, 2021, s. 104–111. HotOS '21. ISBN 9781450384384. Dostupné z DOI: 10.1145/3458336.3465294.
2. PERUGINI, S. *Programming Languages: Concepts and Implementation*. Jones & Bartlett Learning, 2021. ISBN 9781284264982. Dostupné také z: https://books.google.cz/books?id=5MlcEAAAQBAJ.
3. SCOTT, M. *Programming Language Pragmatics*. Elsevier Science, 2015. ISBN 9780124104778. Dostupné také z: https://books.google.cz/books?id=jMcBAAAQBAJ.
4. SEBESTA, R.W. *Concepts of Programming Languages*. Pearson Education, 2012. ISBN 9780133072518. Dostupné také z: https://books.google.cz/books?id=vRQvAAAAQBAJ.
5. SESTOFT, P. *Programming Language Concepts*. Springer International Publishing, 2017. Undergraduate Topics in Computer Science. ISBN 9783319607894. Dostupné také z: https://books.google.cz/books?id=b7szDwAAQBAJ.
6. KOCHAN, S.G.; WOOD, P. *Unix Shell Programming*. Pearson Education, 2003. Kaleidoscope. ISBN 9780134304328. Dostupné také z: https://books.google.cz/books?id=IvDJCgAAQBAJ.
7. FREE SOFTWARE FOUNDATION, Inc. *Bash Reference Manual* [online]. 2022. [cit. 2024-05-05]. Dostupné z: https://www.gnu.org/software/bash/manual/html_node/index.html.
8. CARBONNELLE, Pierre. *PYPL PopularitY of Programming Language* [online]. [cit. 2024-04-28]. Dostupné z: https://pypl.github.io/PYPL.html.
9. ANAGNOSTOPOULOS, Paul C.; HOFFMAN, Steve. *Writing real programs in DCL*. USA: Digital Press, 1998. ISBN 1555581919.
10. VMS SOFTWARE, Inc. *VSI OpenVMS User's Manual*. VMS Software, Inc., 2024.
11. EVSTIGNEEV, Nikita. *Comman Line Description Generator*. 2021. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií.

12. ING. JIŘÍ KAŠPAR, DOC. ING. JAN SCHMIDT, PH.D. *Dokumentace jazyku TPP*. 2000.
13. ING. JIŘÍ KAŠPAR, DOC. ING. JAN SCHMIDT, PH.D. *Zdrojový kód jazyku TPP*. 2000.
14. GROUP, The PHP. *PHP: Language Reference - Manual — php.net* [online]. [cit. 2024-03-29]. Dostupné z: `https://www.php.net/manual/en/langref.php`.
15. THE PHP GROUP. *The PHP Interpreter repository* [online]. [cit. 2024-04-28]. Dostupné z: `https://github.com/php/php-src?tab=readme-ov-file`.
16. PAULI, Julien; POPOV, Nikita; FERRARA, Anthony. *PHP Internals Book* [online]. [cit. 2024-04-28]. Dostupné z: `https://www.phpinternalsbook.com`.
17. S.R.O., JetBrains. *PHP type checking — PhpStorm — jetbrains.com* [online]. [cit. 2024-03-29]. Dostupné z: `https://www.jetbrains.com/help/phpstorm/php-type-checking.html`.
18. BACON, David F.; RAJAN, V. T. Concurrent Cycle Collection in Reference Counted Systems. In: Berlin, Heidelberg: Springer-Verlag, 2001, s. 207–235. ECOOP '01. ISBN 3540422064.
19. PYTHON SOFTWARE FOUNDATION. *CPython Github repository* [online]. [cit. 2024-04-28]. Dostupné z: `https://github.com/python/cpython?tab=readme-ov-file`.
20. SHAW, Anthony. *CPython Internals: Your Guide to the Python 3 Interpreter*. Real Python (realpython.com), 2021. ISBN 9781775093343. Dostupné také z: `https://books.google.cz/books?id=fKZwzgEACAAJ`.
21. FOUNDATION, Python Software. *Python 3 Language Reference* [online]. [cit. 2024-04-29]. Dostupné z: `https://docs.python.org/3/reference/index.html`.
22. SIMIONATO, Michele. *C3 Method Resolution Order epxlanation* [online]. [cit. 2024-04-30]. Dostupné z: `https://docs.python.org/3/howto/mro.html`.
23. TOAL, Ray. *JavaScript Types* [online]. 2021. [cit. 2024-05-06]. Dostupné z: `https://cs.lmu.edu/~ray/notes/javascripttypes/`.
24. NICHOLAS THOMPSON Paul Whittemore, Prince Agrawal. *Javascript WTF* [online]. [cit. 2024-05-06]. Dostupné z: `https://javascriptwtf.com/`.
25. MICROSOFT. *TypeScript: JavaScript with syntax for types* [online]. [cit. 2024-05-06]. Dostupné z: `https://www.typescriptlang.org/`.
26. KANTOR, Ilya. *JavaScript Methods and this Keyword* [online]. [cit. 2024-05-06]. Dostupné z: `https://www.programiz.com/javascript/methods`.
27. CORPORATION, Mozilla. *Classes - JavaScript — MDN* [online]. [cit. 2024-05-06]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes`.
28. JIŘÍ KAŠPAR. *Konzultace k této bakalářské práci* [Osobní konverzace]. 2024.
29. GOSLING, James; JOY, Bill; STEELE, Guy L.; BRACHA, Gilad; BUCKLEY, Alex. *The Java Language Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. ISBN 013390069X.

30. TREPAKOV, Ivan; PAVLOV, Pavel. In: *2020 Ivannikov Memorial Workshop (IVMEM)*. 2020, s. 62–68. Dostupné z DOI: 10.1109/IVMEM51402.2020.00018.
31. TABER BAIN Dmytro Kazanzhy, Seth Tisue a další. *Introduction — Collections — Scala documentation* [online]. [cit. 2024-05-14]. Dostupné z: https://docs.scala-lang.org/overviews/collections-2.13/introduction.html.
32. FOUNDATION, Kotlin. *Collections overview — Kotlin documentation* [online]. [cit. 2024-05-14]. Dostupné z: https://kotlinlang.org/docs/collections-overview.html.
33. DJ DELORIE Andreas Schwab, Carlos Donell. *MallocInternals – glibc wiki* [online]. [cit. 2024-05-14]. Dostupné z: https://sourceware.org/glibc/wiki/MallocInternals.
34. CPPREFERENCE, Tým. *C Operator Precedence* [online]. [cit. 2024-04-05]. Dostupné z: https://en.cppreference.com/w/c/language/operator_precedence.

Obsah příloh