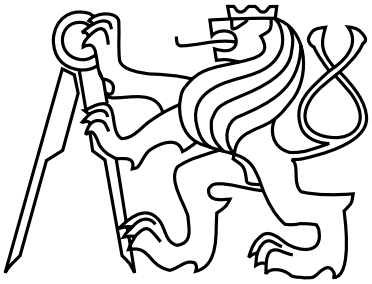
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Interpret grafových algoritmů

Bc. Michal Turek

Vedoucí práce: RNDr. Marko Genyk-Berezovskyj

Studijní program: Elektrotechnika a informatika, strukturovaný,
Navazující magisterský

Obor: Výpočetní technika

12. prosince 2009

Poděkování

Rád bych poděkoval vedoucímu práce, RNDr. Marku Genyk-Berezovskému, za jeho připomínky a cenné rady.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 12. prosince 2009 .....

Abstract

The core of this diploma thesis is design and implementation of a special programming language based on C syntax, which is suited for the graph manipulations. Integrated development environment was created, it consists of a text editor for programmers, debugger and visualization window. The progress of the interpreted graph algorithm can be displayed using 3D graphics.

Abstrakt

Jádrem diplomové práce je návrh a implementace speciálního programovacího jazyka založeného na syntaxi C, který slouží pro manipulaci s grafy. Bylo vytvořeno integrované vývojové prostředí sestávající z programátorsky zaměřeného textového editoru, debuggeru a vizualizačního okna. Průběh interpretovaného grafového algoritmu může být zobrazován pomocí 3D grafiky.

Obsah

1 Úvod **1**

2 Popis problému, specifikace cíle **3**
2.1 Požadavky na aplikaci . . . 3
2.1.1 Navrhovaný jazyk . . . 3
2.1.2 Textový editor . . . 4
2.1.3 Zobrazovací jednotka, vizualizace . . . 4
2.1.4 Manipulace s grafem, krokování skriptu . . . 4

3 Analýza a návrh řešení **5**
3.1 Volba jazyka a knihoven pro implementaci . . . 5
3.2 Návrh a specifikace vytvářeného jazyka . . . 6
3.2.1 Datové typy a proměnné . . . 6
3.2.2 Operátory . . . 7
3.2.3 Řídící struktury . . . 7
3.2.4 Funkce . . . 7
3.2.5 Preprocesor . . . 8
3.2.6 Datové kontejnery . . . 8
3.2.7 Grafy, vrcholy a hrany . . . 9
3.2.8 Ukázka skriptu . . . 9
3.3 Analýza reprezentace grafu . . . 10
3.3.1 Obecné požadavky . . . 10
3.3.2 Reprezentace grafu . . . 11
3.3.2.1 Standardní maticové reprezentace . . . 11
3.3.2.2 Reprezentace množinami vrcholů a hran . . . 13
3.3.2.3 Testovací implementace . . . 14
3.4 Proměnné ve skriptu, typy a operace . . . 15
3.4.1 Implementace operací větvením . . . 15
3.4.2 Double dispatching pattern . . . 16
3.5 Reprezentace skriptu a vykonávání . . . 17
3.5.1 Abstraktní strom syntaxe . . . 17
3.5.2 Podpora krokování a debuggingu . . . 18
3.5.3 Úniky paměti, správce paměti . . . 18
3.5.3.1 Garbage collector . . . 19
3.5.3.2 Chytré ukazatele . . . 19
3.6 Grafické uživatelské rozhraní . . . 20
3.7 Shrnutí . . . 20

4 Realizace **23**

4.1 Základní kód a interpret pro příkazovou řádku . . . 23
4.1.1 Lexikální analyzátor . . . 23
4.1.1.1 Procházení zdrojových dat . . . 23
4.1.1.2 Identifikátory . . . 23
4.1.2 Gramatika, parser . . . 24
4.1.3 Hierarchie tříd, základní třída BaseObject . . . 24
4.1.3.1 Výpis struktury objektů, metoda dump() . . . 25
4.1.3.2 Kontrola úniků paměti . . . 25
4.1.3.3 Pořadí inicializace statických objektů . . . 26
4.1.4 Hodnoty a proměnné ve skriptu, Value* hierarchie tříd . . . 27
4.1.4.1 ValueNull a ValueBool . . . 28
4.1.4.2 Přiřazování do proměnných . . . 28
4.1.4.3 Grafové třídy . . . 29
4.1.5 Abstraktní strom syntaxe, Node* hierarchie tříd . . . 30
4.1.5.1 Strukturované skoky . . . 30
4.1.5.2 Funkce . . . 31
4.1.6 Kontext skriptu . . . 31
4.1.6.1 Funkce ve skriptu, jejich volání a spuštění skriptu . . . 31
4.1.6.2 Globální proměnné . . . 32
4.1.6.3 Pozice v kódu . . . 32
4.1.6.4 Paralelizace skriptů, více kontextů . . . 33
4.1.7 Logování, textový výstup . . . 33
4.1.8 Generátory zdrojových kódů . . . 34
4.2 Grafické uživatelské rozhraní . . . 35
4.2.1 Oddělení od interpretu pro příkazovou řádku . . . 35
4.2.2 Vlákna a jejich synchronizace . . . 35
4.2.3 Spuštění a běh skriptu . . . 36
4.2.4 Ladění skriptu . . . 36
4.2.5 Hlavní okno . . . 38
4.2.6 Textový editor . . . 39
4.2.6.1 Zvýrazňování syntaxe . . . 39
4.2.6.2 Zvýraznění aktuálního řádku . . . 40
4.2.6.3 Číslování řádek . . . 40
4.2.6.4 Indikace přednastavené šířky řádku . . . 40
4.2.6.5 Automatické odsazování textu . . . 40
4.2.6.6 Inteligentní klávesa Home . . . 40
4.2.6.7 Vyhledávání a nahrazování textu . . . 41
4.2.7 Textový výstup ze skriptu . . . 41
4.2.8 Panel proměnných skriptu a panel zásobníku volání . . . 41
4.2.9 Vizualizace grafu . . . 42
4.2.9.1 Propojení skriptu a vizualizací . . . 42
4.2.9.2 Registrace objektů a rendering grafu . . . 42
4.2.9.3 Změna pohledu kamery na scénu . . . 43
4.2.9.4 Screenshot vizualizačního okna . . . 43
4.2.10 Konfigurace grafické aplikace . . . 43

5 Testování **45**
5.1 Unit testy . . . 45

5.2 Unit testy ve skriptu 46
5.3 Rychlost vykonávání skriptu, srovnávací testy 47

6 Závěr **49**

Literatura **51**

A Gramatika jazyka **53**

B Formát datových souborů s grafem **57**

C Vytvoření nové zabudované funkce **59**

D Hierarchie tříd **61**

E Seznam použitých zkratek **63**

F Obsah přiloženého CD **65**

Seznam obrázků

1.1 Typ grafů, jež se používají pro difúzní algoritmy 1

3.1 Rozlišení dvou paralelních hran pomocí dodatečného vrcholu 12
3.2 Vrcholy, které musejí být aktualizovány po smazání některého z nich 12
3.3 Vyhodnocení příkazu 19

4.1 Hierarchie Value* tříd 28
4.2 Dva možné přístupy k ukládání pozic v abstraktním stromu syntaxe 33
4.3 Okno grafické aplikace 39

5.1 Rychlost vykonávání ve srovnání s jazykem Perl 48
5.2 Režie výjimek při návratu z funkce 48

Seznam tabulek

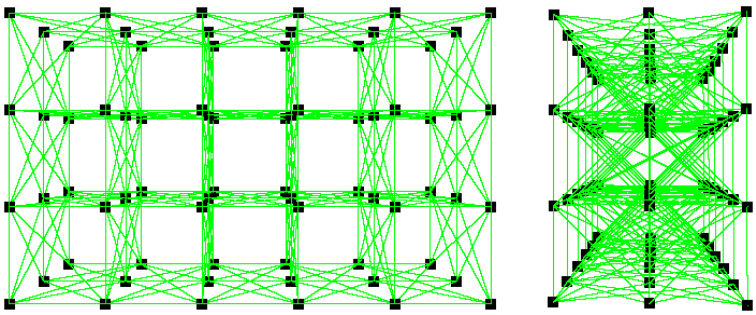
5.1 Rychlost vykonávání ve srovnání s jazykem Perl 48

Kapitola 1

Úvod

Algoritmy *difúze* [10] slouží pro zpracování obrazů. Každý, kdo se jimi zabývá, ví, že fungují, ale nikdo přesně neví, proč vlastně fungují. Výstupem tohoto projektu je integrované vývojové prostředí pro psaní, ladění a vizualizaci průběhu takovýchto algoritmů.

Za vstupní data se považují pixely obrázku a výstupem je například, ve které jeho části se nachází objekt s definovanými vlastnostmi, kde se vyskytuje pozadí a podobně. Obrázek je převeden na speciální graf s několika vrcholy pro každý pixel a pospojovaný ohodnocenými hranami. Při zobrazení ve 3D prostoru může vytvořený graf vypadat podobně jako na schématu 1.1.



Obrázek 1.1: Typ grafů, jež se používají pro difúzní algoritmy

Algoritmus pracuje s lokálními částmi grafu, iterativně aktualizuje ohodnocení hran a končí poté, co je splněna globální podmínka. Difúze je blízká *loopy belief propagation* strategii [11]. Snaží se podobným způsobem o totéž – tj. nalézt v ohodnoceném grafu podgraf s celkově maximální vahou hran a uzlů, přičemž množina přípustných podgrafů, na nichž se maximum hledá, je nějakým způsobem omezena.

Cílem diplomové práce je návrh a implementace jazyka založeného na syntaxi C, který slouží pro manipulaci s grafy. Jazyk má podporu pro základní datové typy, jako jsou celá a desetinná čísla, řetězce, pole a struktury. Proměnné mohou být lokální i globální, bez deklarací.

Výčet kontrolních struktur sestává z `if-else` podmínek, `for`, `while` a `foreach` cyklů, deklarace funkcí a jejich volání. `Include` a `define` direktivy preprocesoru jsou implementovány na úrovni lexikálního analyzátoru. Jazyk nepodporuje ukazatele ani objektově orientované programování.

Speciální datové typy zahrnují grafy, vrcholy, hrany a jejich množiny. Interpret obsahuje zabudované funkce pro vytváření a rušení těchto proměnných, vracení sousedů daného vrcholu a další operace. Každému vrcholu a hraně může být přiřazen libovolný počet dalších hodnot, k přístupu k nim slouží jméno předávané tečkovou notací `objekt.vlastnost`. Speciální řídící struktury pro zpracování komplexních proměnných jsou implementovány na bázi iterátorů, jež jsou interně volané i ve `foreach` cyklu.

Grafické uživatelské rozhraní sestává především z programátorsky zaměřeného textového editoru, který poskytuje zvýrazňování syntaxe, číslování řádků, zvýraznění aktuálního řádku, automatické odsazování a inteligentní klávesu `Home`. Všechny ostatní části aplikace jsou tvořeny panely, jež mohou být uživatelem libovolně přesunuty na jiné místo, skryty a nebo také *vytrhnuty* z hlavního do samostatného okna. To je výhodné především při používání dvou monitorů. V panelech je možné zobrazit výstup z parsování a interpretace skriptu, zásobník volání funkcí, seznam definovaných proměnných a vizualizační okno.

Vizualizace jsou přímo propojené s laděním skriptu. Vykonávání kódu může být pozastaveno breakpointem a poté krokováno s možností vstupování do funkcí (step in), opuštění dané funkce (step out) či vykonání následujícího příkazu (step over). Při každém zastavení skriptu se aktualizuje panel zásobníku volání, seznam definovaných proměnných a vizualizace. Se zobrazeným grafem se dá následně otáčet, přibližovat ho, oddalovat a provádět další 3D operace. Graf může být nahrán jak kódem skriptu, tak explicitně uživatelem.

Základní kód interpretu a rozhraní pro příkazovou řádku je napsáno v jazyce C++ s využitím parser generátoru GNU Bison (LALR gramatika) a nezávisí na žádné další knihovně kromě standardní STL. Tato část je tudíž zkompilovatelná na libovolné platformě, pro kterou existuje C++ kompilátor. Grafická část aplikace používá Qt a vizualizace OpenGL, obě knihovny jsou dostupné na všech majoritních operačních systémech.

Kapitola 2

Popis problému, specifikace cíle

2.1 Požadavky na aplikaci

Navrhněte a implementujte interpret jazyka, který bude zaměřen na práci s grafy. Jazyk bude obsahovat základní datové struktury nezbytné pro manipulaci s grafy a bude tak podporovat efektivní zápis grafových algoritmů. Interpret vybavte uživatelským rozhraním skládajícím se z editoru pro zápis algoritmů a dále grafickým oknem interaktivně zobrazujícím průběh algoritmu nad konkrétním grafem. V grafické části aplikace využijte knihovnu OpenGL.

Zásadní jsou neorientované grafy, implementace orientovaných je volitelná.

2.1.1 Navrhovaný jazyk

Jazykové konstrukce by měly být podobné jazyku C. Zahrnují základní datové typy, datové a řídící struktury

- celá a desetinná čísla,
- pole,
- podmínky, cykly, deklarace a volání funkcí,
- výstup textu na vlastní konzoli,
- direktiva `include` v jednoduché verzi, tj. nikoli hlavičky, ale celé soubory.

Zvažte

- datový typ boolean,
- znaky a řetězce, v jednoduché formě bez složitých funkcí pro manipulaci,
- ukazatele a struktury jako v jazyce C,
- objektově orientované vlastnosti,
- čtení a zápis souborů, jde především o načítání specifikací grafu z externích souborů.

Specifické elementy jazyka zahrnují

- primitivní typy:
  - graf, vrchol, hrana,
  - množina vrcholů, množina hran,
- základní funkce:
  - vytvoř/smaž graf/hranu/vrchol,
  - ohodnoť hranu/vrchol,
  - vrať všechny sousedy daného vrcholu,
  - vytvoř/zruš množinu vrcholů/hran, přidej/odeber vrchol/hranu,
  - vrať stupeň vrcholu,
- řídící struktury:
  - pro všechny sousedy daného vrcholu vykonej `příkaz`,
  - pro všechny vrcholy/hrany v dané množině vykonej `příkaz`,
  - pro všechny vrcholy/hrany grafu vykonej `příkaz`.

2.1.2 Textový editor

Textový editor by měl být programátorsky zaměřený, zvýrazňování syntaxe je vítáno. Další vymoženosti podle dohody a možností.

2.1.3 Zobrazovací jednotka, vizualizace

Vizualizační jednotka zobrazuje aktuálně zpracovávaný graf ve 3D, nemusí to být editor, ve kterém je možné měnit strukturu grafu. Uživatel má možnost graf interaktivně natáčet, přibližovat, oddalovat a posouvat v libovolném směru. Dále může zapnout a vypnout zobrazování jednotlivých částí grafu, aby bylo možné dosáhnout méně přeplněného výstupu.

Pokud bude v jazyce možnost přidávat barevné (texturové) atributy vrcholům, hranám nebo jejich množinám, promítaly by se i do zobrazení. Vizualizační okno má vlastní možnost načtení grafu, aplikace umožní volit, zda má algoritmus pracovat s grafem, který je v grafickém okně, nebo zda si jej načte sám podle zápisu v kódu.

2.1.4 Manipulace s grafem, krokování skriptu

Interpretace kódu může být krokována, při pozastavení se překreslí vizualizační okno a tím se aktualizuje i pohled na graf. Dokud není vydán příkaz k dalšímu kroku, je možné s grafem posouvat, natáčet ho a podobně. Parametry samotného grafu během průběhu programu není vhodné měnit, avšak parametry zobrazení nejspíše ano (barvy, měřítka a podobně). Debugging není třeba nijak zvlášť podporovat, postačí konzole, kam si programátor může vypsat ladící a trasovací tisky.

Kapitola 3

Analýza a návrh řešení

Základním cílem tohoto projektu je vyvinout integrované vývojové prostředí pro programovací jazyk orientovaný na manipulace s grafy. Tato komplexní úloha může být rozdělena na několik oddělených částí, které jsou řešeny v následujícím pořadí:

1. Návrh jazyka, který odpovídá všem požadavkům.
2. Návrh reprezentace grafů v aplikaci, ohled na přístup z interpretu.
3. Implementace lexikálního analyzátoru a parseru.
4. Implementace interpretu pro příkazovou řádku.
5. Návrh a implementace grafického vývojového prostředí.
6. Implementace vizualizačního okna, propojení s interpretem.

3.1 Volba jazyka a knihoven pro implementaci

Na samém začátku je možné pro tvorbu systémových aplikací uvažovat tři běžně používané jazyky: C++, Javu a C#. Třetí z nich může být automaticky vyškrtnut, protože neodpovídá základnímu požadavku na přenositelnost. Jazyk C++ je portovatelný na úrovni zdrojových kódů a Java na úrovni byte kódu. Žádné skriptovací jazyky, jako například Perl a Python, nejsou uvažovány. Ačkoli mnoho vývojářů může tvrdit přesný opak, nejsou příliš vhodné pro tak velkou aplikaci a psát interpret v interpretovaném jazyce není příliš vhodné.

Obrovskými výhodami Javy je její automatický správce paměti a spousta existujících knihoven a frameworků. Na druhou stranu, kvůli kompilaci do strojového kódu a z ní vyplývající rychlosti provádění, je naprostá většina kompilátorů a interpretů napsaná v C nebo C++.

Autor práce preferuje při vývoji systémových aplikací C++ a má s ním také mnohem více zkušeností, a proto byl nakonec zvolen právě tento jazyk. Žádná knihovna, kromě standardního STL, není při tvorbě základního kódu použita – interpret pro příkazovou řádku by měl mít co nejméně externích závislostí. Díky tomu je bez větších problémů portovatelný na všechny platformy, pro které existuje C++ kompilátor.

Grafická část aplikace je naprogramovaná s použitím knihovny Qt. Je jednou z nejlépe navržených knihoven pro tvorbu GUI aplikací, přenositelná na všechny majoritní operační

systémy a bezplatná pro nekomerční použití. Vizualizace grafů používají OpenGL, jedná se o standard pro 3D grafiku. Druhá možná volba, Direct3D, není použitelná na ne-Microsoft platformách, a proto je zbytečné o ní uvažovat.

3.2 Návrh a specifikace vytvářeného jazyka

Jak už bylo zmíněno, syntaxe jazyka je z větší části založena na všeobecně známém C, což programátorům přináší výhodu rychlého učení. Jazyk rozlišuje malá a velká písmena, je procedurální a nepodporuje objektově orientované vlastnosti.

3.2.1 Datové typy a proměnné

Proměnné se nedeklarují, v průběhu vykonávání skriptu mohou libovolně měnit svůj datový typ. Uvnitř interpretu jsou rozlišovány typy `null`, `bool`, `int`, `float`, `string`, `struct`, `array`, `graph`, `vertex`, `edge` a `set`. Jazyk nepodporuje modifikátory typu, jako jsou například `unsigned` nebo `const`, protože nepřinášejí žádné výrazné vylepšení, programátor se musí obejít bez nich.

Viditelnost proměnných je definována na úroveň funkcí, tudíž opuštění daného bloku (například cyklu) nezpůsobuje zrušení v něm používaných proměnných. Globální proměnnou je možné ve funkci zpřístupnit po vzoru jazyka PHP klíčovým slovem `global`.

```
function example()
{
    global g_var;
    g_var = "some value";
}
```

Při úvahách o deklaracích proměnných byly zvažovány možnosti

- jako v C – povinná,
- jako v C – nepovinná,
- jako v JavaScriptu pomocí klíčového slova `var` – nepovinná,
- bez deklarací.

Na první pohled je jasné, že prostřední dvě možnosti nepřinášejí nic užitečného. Pokud jsou deklarace volitelné, nejde těžit z jejich výhod a pouze přidávají nová klíčová slova, což je patrné především u JavaScriptového způsobu.

Výhodou deklarací je snadná detekce mnoha typů chyb a případně možnost větších optimalizací. Na druhou stranu příliš svazují, což je patrné především u skriptovacích jazyků, které se snaží o co nejmenší množství kódu na co největší funkcionalitu. Právě z tohoto důvodu byla zvolena verze bez deklarací.

Poznámka na okraj. Deklarace by se mohly hodit i pro automatizovaný překlad do jiných běžně používaných jazyků, než ve kterém byl napsán vlastní algoritmus. V tomto případě by se pravděpodobně jednalo o kompilovatelné C++ nebo Javu. Tato myšlenka je inspirována Google Web Toolkitem, který překládá javovské programy do HTML, CSS a JavaScriptu, nejedná se o nic nereálného.

3.2.2 Operátory

Jazyk podporuje prakticky všechny operátory z jazyka C s výjimkou operátoru čárky a bitových operátorů. Priority jsou stejné jako v C.

- unární operátory:
  - `-`, `!`, `++`, `--`
- binární operátory:
  - aritmetické: `+`, `-`, `*`, `/`, `%`
  - přiřazovací: `=`, `+=`, `-=`, `*=`, `/=`, `%=`
  - přiřazení reference: `&=`
  - logické: `==`, `!=`, `<=`, `>=`, `<`, `>` `&&`, `||`
  - přístup k prvkům: `.`, `[]`
- ternární operátor:
  - `? :`

3.2.3 Řídící struktury

Zápis podmínek a cyklů je naprosto stejný jako v C, programátor může použít konstrukce `if`, `if-else`, `for` a `while`. Vícenásobné větvení `switch` není implementováno, protože je bez problémů nahraditelné vnořenými `if-else` podmínkami.

Jazyk dále přináší cyklus `foreach`, který umí iterovat přes všechny prvky složeného datového typu, jímž může být například pole, struktura a množina, ale také vlastnosti vrcholů a hran. Cyklus `foreach` interně používá iterátory.

```
foreach(item; iterableObject)
{
    println(item);
}
```

Předčasné ukončení cyklu je možné provést klasicky pomocí strukturovaných skoků `break` a `continue`.

3.2.4 Funkce

Funkce se ve skriptu deklarují podobně jako v jazyce PHP. Klíčové slovo `function` následuje jméno funkce, v kulatých závorkách čárkami oddělený seznam parametrů a ve složených závorkách tělo funkce. Při parsování nezáleží na pořadí funkcí v souboru, interpretace začíná až po zpracování celého zdrojového kódu, kdy už jsou všechny dostupné. Samotný skript se spouští funkcí `main(argv)` s polem argumentů v parametru.

U volání funkcí jsou k dispozici dvě ekvivalentní syntaxe. V první verzi je možné všechny parametry předat klasicky do kulatých závorek, u druhého způsobu je první parametr zapsán tečkovou notací před jméno funkce a celý zápis potom vypadá jako volání metody nad daným objektem. Forma je inspirována jazykem Python a jeho `self` parametrem u metod třídy.

```
function someFunc(parameter1, parameter2, parameter3)
{
    // ...
}

someFunc(object, parameter2, parameter3);
object.someFunc(parameter2, parameter3);

someFunc(objects[2].position.x, parameter2, parameter3);
objects[2].position.x.someFunc(parameter2, parameter3);
```

Veškeré parametry se předávají hodnotou. Při návrhu bylo zvažováno také předávání odkazem, tato technika se používá především k vracení několika hodnot z funkce najednou a omezení velikosti kopírovaných dat. Autor ale nakonec došel k závěru, že není vyloženě nutná, hodnoty se mohou uložit do struktury nebo pole a vrátit ve formě kontejneru. Jednoduchost jazyka a nedeklarativnost proměnných k tomu přímo vybízí.

3.2.5 Preprocesor

Pro zahrnutí zdrojového souboru do překladu, například dodatečné knihovny funkcí, slouží příkaz `include` s cestou a jménem souboru v parametru. Nové makro, respektive symbolickou konstantu, je možné definovat příkazem `define`. Kdykoli se ve zdrojovém souboru objeví specifikovaný identifikátor, je automaticky nahrazen hodnotou makra. Obě konstrukce se zpracovávají na úrovni lexikálního analyzátoru, parseru se předává až výsledný proud tokenů.

```
include("filename");
define("name", "value");
```

3.2.6 Datové kontejnery

Základními datovými kontejnery na úrovni jazyka zůstávají pole, struktury a množiny, ty se však dají jednoduše rozšířit. Zásobník a fronta se z pole vytvoří definováním operací `pushFront()`, `pushBack()`, `popFront()`, `popBack()`, `front()` a `back()`. Struktura se může bez jakýchkoli změn používat jako mapa (asociativní pole) a s její pomocí se dá jednoduše implementovat i spojový seznam.

Na rozdíl od jiných skriptovacích jazyků není definována speciální syntaxe pro deklarace a vytváření netriviálních typů proměnných. Pro zachování jednoduchosti stačí pouze zavolat zabudované funkce `array()`, `struct()` a další, které vrátí nově vytvořený objekt daného datového typu, a následně jej začít používat.

```
arr = array(5);
arr[0] = "item";
arr[1] = array(10);
arr[3] = 3.14;

st = struct();
st.name = "item";
st.valid = true;
```

V prvcích polí i struktur mohou být uloženy libovolné datové typy – tímto se dají například vytvořit dvou a vícerozměrná pole, spojové seznamy, stromy a další běžné datové struktury, hloubka zanoření není nijak omezena. V programátorem neinicializované položce se vždy nachází hodnota `null`, datový typ prvků je možné kdykoli změnit pouhým přiřazením.

3.2.7 Grafy, vrcholy a hrany

První operací při práci s grafovými vlastnostmi jazyka je vždy volání zabudované funkce `graph()`, která vytvoří objekt grafu. Od této chvíle se může přistoupit k definici vrcholů a hran a následné práci s nimi. Vrcholy i hrany se zároveň chovají jako struktury, tudíž je u nich možné používat libovolný počet libovolně pojmenovaných vlastností.

```
g = graph();
v1 = g.generateVertex();
v2 = g.generateVertex();
e1 = g.generateEdge(v1, v2);

v1.color = "red";
v2.visited = false;
e1.value = 3.14;

foreach(vertex; v1.getNeighbors())
    doSomething(vertex);
```

Pro vytváření množin vrcholů a hran slouží zabudovaná funkce `set()`, jež se chová podobně jako pole a struktury popsané v kapitole 3.2.6 na straně 8. Tento kontejner není určen výhradně pro grafové objekty, ale může ukládat libovolné datové typy, včetně různých kombinací.

3.2.8 Ukázka skriptu

Příkladem skriptu zapsaného ve vytvářeném jazyce budiž rekurzivní prohledávání grafu do hloubky. Parametrem uživatelsky definované funkce `dfs()` je libovolný vrchol grafu. Pokud už byl daný vrchol navštíven, větev procházeného stromu se ukončí, v opačném případě se označí vrchol za navštívený a funkce se rekurzivně zavolá nad všemi jeho sousedy.

```
define("NEW", "0");
define("CLOSED", "1");

// Recursive depth first search
function dfs(v)
{
    if(v.state == CLOSED)
        return;

    v.state = CLOSED;
```

```
    foreach(neighbor; v.getNeighbors())
        dfs(neighbor);
}
```

Výše definovaný algoritmus funguje nad všemi orientovanými i neorientovanými spojitými grafy. Od začátku vývoje se předpokládá načítání grafů z externích souborů, nicméně jejich vytváření je možné i v kódu programu.

3.3 Analýza reprezentace grafu

3.3.1 Obecné požadavky

Jedním ze základních kamenů vytvářené aplikace je reprezentace grafů, vrcholů, hran a jejich množin v kódu programu. Přímo ze zadání práce vyplývá několik skutečností. Následuje stručný přehled požadavků, který bude následně rozebírán a diskutován. Jedná se o

- typ podporovaných grafů,
- zapnutí a vypnutí orientace, inverze orientace,
- dynamičnost grafů, složitost vytváření nových vrcholů a hran,
- vlastnosti vrcholů a hran,
- zpětná reference na objekt celého grafu,
- identifikovatelnost sebe sama v grafu,
- přístup k sousedním vrcholů,
- přístup k počátečnímu a koncovému vrcholu hrany,
- reprezentace množin vrcholů a hran, množinové operace,
- iterace přes všechny vrcholy/hrany grafu/množiny,
- paměťový management, paměťové nároky.

První a nejdůležitější z otázek se ptá, jakou třídu grafů je třeba podporovat. Jelikož je program psán úplně od začátku a neexistují žádná omezení plynoucí z již existujícího kódu a napojování se na něj, je rozumnou volbou uvažovat maximální možnou množinu všech grafů. Ta zahrnuje orientované i neorientované grafy, spojité i nespojité, několik paralelních hran mezi dvěma vrcholy a samozřejmě také smyčky nad vrcholem – tj. hrana začíná i končí ve stejném vrcholu. Dále v textu uvidíme, že toto rozhodnutí silně omezí možné způsoby implementace, popř. je výrazně zkomplikuje.

Ačkoli se předpokládají spíše statické grafy, které se na začátku vytvoří a dále se už nemění, přidávání a mazání vrcholů a hran by mělo být *rozumně* složité. Kopírovat matici sousednosti o milionu prvků kvůli jednomu přidanému vrcholu rozhodně není správnou cestou.

Zadání práce požaduje, aby vrcholy i hrany mohly být nějakým, nespecifikovaným, způsobem ohodnoceny. Opět je vhodné uvažovat maximalistickou variantu. Předpokládáme, že každá

z hran může být ohodnocena libovolným počtem vlastností libovolného typu, to samé platí pro vrcholy. Z tohoto rozhodnutí plyne, že by vrchol i hrana měly mít svou vlastní třídu (ve významu objektového programování), aby se s nimi dalo jednodušeji manipulovat.

Jelikož jsou vrcholy a hrany v interpretu zpřístupnitelné i jako samostatné proměnné, je třeba, aby si nějakým způsobem pamatovaly, ke kterému grafu patří. Tento požadavek se řeší uložením ukazatele na objekt grafu do atributu třídy. Vrcholy i hrany by dále měly být v grafu schopny identifikovat sebe sama. Pokud by byl graf implementován například maticí sousednosti, je tento požadavek naprosto zásadní, protože bez něj by například nebylo možné najít sousedy daného vrcholu.

Další z důležitých otázek je reprezentace množin vrcholů a hran, přidávání a odebírání prvků, množinové a jiné operace. Pokud by byly množiny reprezentované identicky jako celý graf, zůstane spousta kódu ekvivalentní nebo alespoň podobná. Předpokládá-li se časté přidávání a odebírání prvků, měly by být tyto operace jednoduché a rychlé.

Očekávaná velikost grafů jsou miliony vrcholů a hran – obrovské paměťové nároky. Každé z primitiv může být odkazováno proměnnou v interpretu, či jiného místa, a proto je zajisté potřeba nějaký druh paměťového managementu, který zamezí chybám při práci s dynamickou pamětí.

3.3.2 Reprezentace grafu

Existuje několik možností, jak reprezentovat graf v programu. Při výuce teorie grafů bývají nejčastěji zmiňovány maticové reprezentace a spojové seznamy, ale existují i další, méně obvyklé, způsoby. Označme množinu vrcholů, resp. hran, symbolem V, resp. H. Pro uložení grafu se běžně používají

- matice incidence $|V| \times |H|$,
- matice sousednosti $|V| \times |V|$,
- spojové seznamy – seznam následníků apod.,
- jiné reprezentace.

Cílem této části je najít nejvhodnější reprezentaci grafu, kterou by bylo možné použít ve specifické oblasti interpretu.

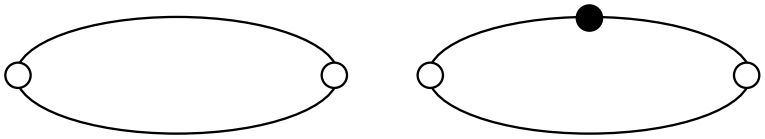
3.3.2.1 Standardní maticové reprezentace

Matice sousednosti je v grafové teorii definována jako dvourozměrné pole $|U| \times |U|$. V jazyce C++ s použitím knihovny STL by mohly být atributy třídy grafu definovány například následovně.

```
class Graph
{
    vector<Vertex*> m_vertices;
    vector< vector<Edge*> > m_edges;
};
```

Parametry datových kontejnerů nemohou být primitivní datové typy, protože každý vrchol i každá hrana musejí umět ukládat dodatečné vlastnosti typu klíč–hodnota definované v požadavcích na systém.

Na první pohled se zdá, že by mohlo být výhodné použít tuto reprezentaci, nicméně existuje několik výrazných problémů. Největším z nich jsou paralelní hrany, protože počáteční a koncový vrchol neurčují hranu jednoznačně. Řešením by mohlo být přidání dodatečného vrcholu podle schématu 3.1, kterým se dvojice paralelních hran rozbije na tři jednoznačně identifikovatelné hrany. Druhou, horší, možností zůstává trojrozměrné pole a trojrozměrné indexy. Ani jedno z řešení rozhodně není hezké.

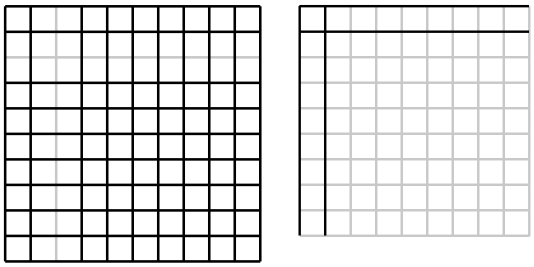


Obrázek 3.1: Rozlišení dvou paralelních hran pomocí dodatečného vrcholu

```
vector< vector< vector<Edge*> > > m_edges;
```

Další nevýhodou je příliš složité přidávání a mazání vrcholů a z toho vyplývající časté realokace velkých bloků paměti. Toto řešení by tedy bylo vhodné spíše pro statické grafy, které se na začátku vytvoří a pak zůstanou nezměněny až do konce algoritmu.

Problémem je i složité procházení vrcholů a hran a hledání sebe sama v grafu. Každý vrchol a hrana musí mít u sebe uložen index do matice reprezentující graf, který je navíc nutné aktualizovat při libovolné změně matice sousednosti. Kupříkladu schéma 3.2 ukazuje problémy s mazáním hrany. Tímto by se výrazně zvýšily nejen paměťové, ale i výpočetní nároky.



Obrázek 3.2: Vrcholy, které musejí být aktualizovány po smazání některého z nich

Další problémy pravděpodobně by pravděpodobně vznikly při vytváření množin vrcholů a hran, které by měly naprosto odlišnou implementaci vzhledem k vlastní třídě grafu. Paměťový management by byl extrémně složitý a snadno náchylný k chybám.

Výše uvedená diskuze odhalila množství nepřekonatelných problémů, které by vznikly při použití matice sousednosti. Dá se také předpokládat, že se úplně stejné problémy objeví i u matice incidence a všude tam, kde je vlastní struktura grafu, návaznost vrcholů a hran, definovaná centrálně ve třídě grafu.

3.3.2.2 Reprezentace množinami vrcholů a hran

Tato reprezentace grafů se učí spíše zřídka, nicméně v tomto konkrétním případě (interpretu), vychází jako velice elegantní. Samotný nápad vznikl při odhalení slepé uličky u maticových reprezentací, které strukturu grafu definují centrálně v jediné datové struktuře. Bylo potřeba najít nějaké distribuované řešení.

Datové položky třídy grafu jsou tvořeny pouze dvěma množinami nebo seznamy vrcholů a hran. Dá se říci, že ani tyto dva kontejnery ukazatelů by nemusely být ve třídě grafu definovány, nicméně je třeba nějakým způsobem uvolňovat dynamickou paměť, uložení pointerů je proto nutností.

```
class Graph
{
    set<Vertex*> m_vertices;
    set<Edge*> m_edges;
};
```

Vlastní struktura grafu je definovaná ne v jeho třídě, ale přímo u vrcholů a hran. Vrchol obsahuje množinu hran, které s ním incidují, a hrany ukládají ukazatele na svůj počáteční a koncový vrchol. Referenci na graf je nutné uchovávat kvůli interpretu.

```
class Vertex
{
    Graph* m_graph;
    set<Edge*> m_edges;
};

class Edge
{
    Graph*  m_graph;
    Vertex* m_begin
    Vertex* m_end;
};
```

Hlavní výhodou této reprezentace je možnost práce s naprosto obecným grafem, není třeba se starat o paralelní hrany ani smyčky nad vrcholy. Už bylo zmíněno, že druhý typ reprezentace, matice sousednosti, neumí reprezentovat paralelní hrany.

Druhá obrovská výhoda samozřejmě spočívá v ekvivalentní implementaci grafu a množin jeho vrcholů a hran. Není nutné programovat identické operace dvakrát a poté je navíc složitě napojovat na sebe. Výhodou je i přirozená iterovatelnost množin, díky níž nebylo složité implementovat procházení grafu pomocí cyklu `foreach`.

Jedinou možnou nevýhodou je mírná duplicita informací při definici struktury grafu, která je podobná obousměrně propojenému spojovému seznamu. Průchod je možný od vrcholů k hranám a také naopak od hran k vrcholům. V rozhraní tříd je nutné ošetřit, aby kvůli této duplicitě nemohly vzniknout žádné problémy.

3.3.2.3 Testovací implementace

V následujícím výpisu kódu je ukázána až překvapivá elegance řešení na bázi množin vrcholů a hran ve spojení se standardní knihovnou STL a její šablonou `set`. Jsou ukázány čtyři metody třídy grafu, které se starají o vytváření a mazání vrcholů a hran. Zdají se natolik jednoduché, že veškeré komentáře jsou asi zbytečné.

```
Vertex* Graph::generateVertex(void)
{
    Vertex* vertex = new Vertex(this);
    m_vertices.insert(vertex);
    return vertex;
}

Edge* Graph::generateEdge(Vertex* begin, Vertex* end)
{
    Edge* edge = new Edge(this, begin, end);
    m_edges.insert(edge);

    begin->addEdge(edge);
    end->addEdge(edge);

    return edge;
}

void Graph::deleteVertex(Vertex* vertex)
{
    set<Edge*> edges = vertex->getEdges();
    for(set<Edge*>::iterator it = edges.begin(); it != edges.end(); ++it)
        deleteEdge(*it);

    m_vertices.erase(vertex);
    delete vertex;
}

void Graph::deleteEdge(Edge* edge)
{
    edge->getBeginVertex()->deleteEdge(edge);
    edge->getEndVertex()->deleteEdge(edge);
    m_edges.erase(edge);
    delete edge;
}
```

Výhradně tyto čtyři funkce pracují při vytváření a mazání vrcholů a hran s pamětí, a tudíž je možnost jejích úniků v této části aplikace vyloučená.

Implementace podmnožin vrcholů a hran je téměř identická jako implementace vlastního grafu. Jediný rozdíl se nachází v těchto čtyřech metodách – nepracují s pamětí, ale pouze se starají o ukládání a vracení již existujících ukazatelů na objekty.

Přiřazování mezi proměnnými v interpretu není třeba nějak speciálně ošetřovat. Pokud je proměnná `vertex` vrcholem grafu, pak jeho přiřazení do jiné proměnné nevytváří v původním grafu ani v žádném jiném nový vrchol, nemůže jít o hlubokou kopii. Protože žádný vrchol nemůže bez grafu existovat, je jasné, že proměnná `alias` z příkladu níže musí odkazovat na stejný vrchol jako proměnná `vertex`. Stejnou úvahu lze provést i pro hrany.

```
vertex = g.generateVertex();
alias = vertex;
```

Výše uvedený kód může vypadat jako nežádoucí a jeho použití okrajové, ale naprosto stejné operace se provádějí při předávání parametrů funkcím, a proto je dobré mít ověřeno, že při jejich volání nenastanou žádné problémy. Jak už bylo řečeno, o veškerou dynamicky alokovanou paměť grafu se stará třída `Graf` a žádná jiná.

3.4 Proměnné ve skriptu, typy a operace

3.4.1 Implementace operací větvením

Existují v zásadě dvě možnosti, jak implementovat sčítání, odčítání a ostatní operace nad hodnotami a proměnnými ve skriptu. Nejjednodušší z nich předpokládá použití `unionu` a pomocné proměnné, která ukládá datový typ uložené hodnoty. Příkladem budiž ukázka z autorova dřívějšího interpretu [9, třída `CNodeValue`].

```
class CNodeValue
{
private:
    TYPE m_type;
    union
    {
        bool m_b;
        int m_i;
        float m_f;
    };
};
```

Toto řešení je funkční, nicméně při více podporovaných datových typech a operacích začíná být extrémně nepřehledné a náchylné k chybám. Každá z funkcí implementujících například některou aritmetickou operaci musí obsahovat větvení podle typu, navíc, v případě binární operace, vnořené.

```
const CNodeValue CNodeValue::operator+(const CNodeValue& object) const
{
    // ...

    switch(m_type)
    {
    case LEX_BOOL:
```

```
        switch(object.m_type)
        {
            case LEX_BOOL:  return CNodeValue(m_b + object.m_b);
            case LEX_INT:   return CNodeValue(m_b + object.m_i);
            case LEX_FLOAT: return CNodeValue(m_b + object.m_f);
            default: assert(false); return *this;
        }

        // ...
    }
}
```

Výše uvedený kód ošetřuje sčítání boolean hodnoty s boolean hodnotou, celým číslem a desetinným číslem. Jedná se pouze o část funkce, v případě tří různých datových typů by musela funkce obsahovat ještě další dvě ekvivalentní sekce, což přináší devět podmínek.

Navrhovaný jazyk předpokládá jedenáct různých datových typů (viz sekce 3.2.1 na straně 6), což pouze pro sčítání dává celkem 121 větvení! Psát a poté udržovat podobný kód pro všechny operace je naprosto nereálné. Navíc přidání nového datového typu způsobí nutnost složitých úprav všech existujících funkcí.

3.4.2 Double dispatching pattern

Odpovědí na všechny výše uvedené problémy je návrhový vzor *Multiple dispatching pattern* [2, str. 679], který poskytuje možnost volání typových operací nad kombinací beztypových objektů. Základem je abstraktní třída `Value`, která deklaruje všechny potřebné operace – pro každý datový typ a operaci jedna virtuální metoda, plus jedna čistě virtuální metoda pro neznámý datový typ.

```
class Value
{
    // +
    virtual CountPtr<Value> add(const Value&     right) const = 0;
    virtual CountPtr<Value> add(const ValueBool&  left) const;
    virtual CountPtr<Value> add(const ValueInt&   left) const;
    virtual CountPtr<Value> add(const ValueFloat& left) const;
    // ...
};
```

Při sčítání celého čísla s ostatními datovými typy je nejdříve nutné definovat sčítání celého čísla s neznámým datovým typem na pravé straně. V takovém případě funkce neví, jak tuto operaci provést, a proto požádá tento neznámý typ o sečtení se sebou samým. Zároveň mu oznámí, že se jedná o sčítání s celým číslem, protože typ ukazatele `this` je zde `ValueInt*`.

```
CountPtr<Value> ValueInt::add(const Value& right) const
{
    return right.add(*this);
}
```

Dejme tomu, že se jedná také o celé číslo, zavolá se tedy následující metoda a provede sečtení celých čísel, jehož výsledkem je taktéž celé číslo. V tuto chvíli jsou již všechny potřebné datové typy známy a operace se může vykonat.

```
CountPtr<Value> ValueInt::add(const ValueInt& left) const
{
    return CountPtr<Value>(new ValueInt(left.getVal() + m_val));
}
```

Jednou z výhod Double dispatching patternu je také to, že není nutné definovat všechny metody pro všechny kombinace datových typů. Pokud operace s danými parametry nedává smysl, například sčítání čísla s grafem, ponechá se vykonání operace na rodičovské třídě `Value`, jež může kupříkladu vypsat chybu a vrátit `null` hodnotu.

```
CountPtr<Value> Value::add(const ValueGraph& /* left */) const
{
    cerr << "Operation is not supported" << endl;
    return VALUENULL;
}
```

V případě přidání nového datového typu se žádná z již existujících funkcí nemění, pouze se do základní třídy dopíše nová virtuální operace a do potomků, kde to dává smysl, její implementace.

3.5 Reprezentace skriptu a vykonávání

3.5.1 Abstraktní strom syntaxe

Zdrojový soubor skriptu se překládá pomocí lexikálního analyzátoru a parseru do vnitřní reprezentace. Jejím základním prvkem je abstraktní třída `Node`, ze které jsou odvozeny všechny ostatní třídy, například třída pro větvení, třída pro cykly a další.

Posláním parseru je sestavit *abstraktní strom syntaxe*, který se následně používá pro vykonávání programu. Povinností jednotlivých stavebních kamenů je definovat metodu `execute()`, jež se volá rekurzivně na všech úrovních stromu.

```
class Node
{
public:
    virtual CountPtr<Value> execute(void) = 0;
};
```

Kontext skriptu (viz kapitola 4.1.6 na straně 31), ve kterém jsou uložené například hodnoty proměnných je vhodné definovat v externí třídě, pravděpodobně singletonu. Druhou možností by bylo předávat kontext metodám v parametru, ale to bylo v současnosti označeno za zbytečné plýtvání výkonem, nepředpokládají se vícevláknové skripty.

Alternativní možností vykonávání skriptu by mohl být překlad do instrukcí virtuálního procesoru a jejich následná interpretace. To se však aktuálně jeví jako zbytečně náročné řešení, které nepřináší téměř nic navíc.

3.5.2 Podpora krokování a debuggingu

Výhodou lineárního pole instrukcí a virtuálního procesoru je snadná implementace debuggingu. V kterémkoli okamžiku je možné pozastavit vykonávání, protože kód skriptu je včetně pozice uložen výhradně v datové struktuře. Po případném opuštění interpretující funkce zůstane stav virtuálního procesoru zachován a pozastavený skript může být kdykoli opětovně spuštěn.

Naproti tomu, při použití abstraktního stromu syntaxe je kód reprezentován především datovými typy uzlů a jejich strukturou. Vykonávání spočívá v rekurzivním volání, kterézto není možné přerušit a na stejném místě později obnovit. Vzniká otázka, jak pozastavit tato rekurzivní volání.

Odpovědí je *vícevláknové programování* a jeho synchronizační prostředky. V případě řádkové utility se debugging nepředpokládá, u grafického rozhraní je další vlákno pro skript naopak nezbytné. Bez něj by aplikace po spuštění déletrvajícího skriptu nemohla reagovat na žádné požadavky a uživateli by se jevila jako zamrzlá.

Princip pozastavení skriptu demonstruje následující výpis kódu používající synchronizační prostředky knihovny Qt. Pokud skript narazí například na breakpoint, zavolá funkci `pause()`, jež ve `wait()` čeká na signál k opětovnému spuštění. Tento příkaz vydává hlavní vlákno aplikace – kupříkladu po kliknutí uživatele na tlačítko *Pokračovat*.

```
QMutex dbgMutex;
QWaitCondition waitCondition;

// Script thread (e.g. breakpoint)
void pause(void)
{
    dbgMutex.lock();
    waitCondition.wait(&dbgMutex);
    dbgMutex.unlock();
}

// Main/GUI thread (e.g. click on next command button)
void resume(void)
{
    waitCondition.wakeAll();
}
```

Ač se to nemusí zdát, je výše uvedené řešení plně funkční. Bylo implementováno v jednoduché testovací aplikaci a pracuje bez jediného problému.

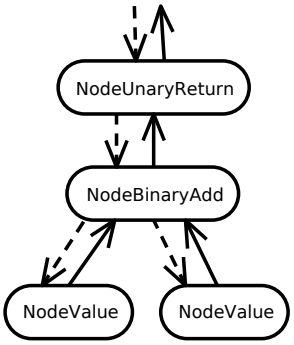
3.5.3 Úniky paměti, správce paměti

V kapitole 3.5.1 na straně 17 byla představena virtuální funkce `execute()`, jejíž rekurzivní volání umožňují interpretaci skriptu. Jednou ze zcela zásadních věcí je její návratová hodnota, která slouží především pro získání výsledku aritmetických výrazů.

Mějme například následující jednoduchý skript, jehož úkolem je sečíst dvě čísla.

```
function main(argv)
{
    return 1 + 2;
}
```

Součet se v syntaktickém stromu uloží celkem do tří uzlů. Listy vracejí svoji hodnotu ve formě celých čísel, binární operátor následně tyto dvě hodnoty sečte a vrátí výsledek operátoru `return`. Vykonávání pokračuje v podobném duchu až do vynoření se z kořene stromu. Celá situace je znázorněna ve schématu 3.3.



Obrázek 3.3: Vyhodnocení příkazu "return 1 + 2;"

Je jasné, že třída pro sčítání nezná při kompilaci datový typ návratové hodnoty, může jím být například `null`, `int`, `float` aj., a tudíž návratovou hodnotou funkce musí být ukazatel na abstraktní objekt typu `Value` představený v sekci 3.4.2 na straně 16.

Z toho dále vyplývá, že se musí objekty vytvářet dynamicky, a s tím bohužel souvisí velice častý problém programátorů v jazycích C a C++: *Kdy je možné tyto objekty smazat a která část kódu se o to postará.*

3.5.3.1 Garbage collector

Jednou z možností je používat *Garbage collector* obdobný javovskému. Velkou výhodou by bylo vyřešení naprosto všech, současných i budoucích, problémů s uvolňováním dynamické paměti. Nevýhodou zůstává složitá implementace a dodatečná režie při vykonávání. V případě použití nějakého již existujícího garbage collectoru vzniká závislost na externí knihovně a s tím souvisí i nižší či složitější přenositelnost programu.

V tomto případě se zdá, že nevýhody převažují nad výhodami. Navíc se přece jedná o návratovou hodnotu *výhradně* jedné funkce. Možná by stačilo zapouzdřit ukazatel na návratovou hodnotu do pomocné třídy, která by se postarala o smazání ve chvíle, kdy už objekt není potřeba.

3.5.3.2 Chytré ukazatele

Nejjednodušší implementací chytrých ukazatelů je šablona `auto_ptr` z knihovny STL. Stačí nebo je potřeba nalézt silnější nástroj? `Auto_ptr` se po požádání stává výhradním vlastníkem

ukazatele a při své destrukci maže i předanou dynamickou paměť. *Výhradní vlastník* znamená, že v jednom okamžiku existuje vždy právě jedna `auto_ptr` reference na danou paměť.

Splnit tento požadavek bohužel nelze. Chceme-li například vrátit hodnotu uloženou v proměnné interpretovaného skriptu, první referencí zůstane původní proměnná a druhá vznikne ve formě vracené hodnoty. Musíme tedy hledat silnější nástroj.

Pokud je jediným problémem `auto_ptr` pouze možný větší počet odkazů, může se použít chytrý ukazatel na bázi čítání referencí [8]. Ten zapouzdřuje ukazatel na dynamický objekt spolu s čítačem, který se při vytvoření objektu nastaví na jedna a při každém zavolání kopírovacího konstruktoru nebo operátoru přiřazení inkrementuje. V destruktoru se naopak počet referencí dekrementuje a při dosažení nulové hodnoty se objekt automaticky smaže.

Toto řešení by mohlo fungovat, je však nutné si dát pozor na cyklické reference, kdy první objekt odkazuje na druhý a zároveň druhý odkazuje na první, paměť by se nikdy neuvolnila. Zmíněná situace bohužel může nastat, protože právě tímto způsobem je uložena struktura grafu (viz kapitola 3.3.2.2 na straně 13). Vrcholy obsahují seznam incidujících hran a hrany odkaz na počáteční a koncový vrchol.

Z toho plyne, že pokud se na tomto konkrétním místě chytrý ukazatel s čítáním referencí nepoužije, nedojde ke vzniku cyklických referencí, a tudíž dostáváme řešení problému návratové hodnoty `execute()`.

3.6 Grafické uživatelské rozhraní

Grafické uživatelské rozhraní je naprogramováno s použitím knihovny Qt. Sestává z hlavního okna aplikace `QMainWindow`, jehož centrálním widgetem je textový editor pro programátory na bázi `QPlainTextEdit`. Jedná se o MDI aplikaci, jednotlivé editory je možné přepínat, po vzoru webových prohlížečů, pomocí panelů či listů. Menu, nástrojová lišta a stavový řádek jsou pro daný typ aplikace samozřejmostí.

Knihovna Qt poskytuje možnost vkládat do okrajů hlavního okna plovoucí panely, objekty třídy `QDockWidget`, jež mohou být přesunuty na jiné místo v aplikaci a také *vytrhnuty* do samostatného okna. Tato funkcionalita se určitě hodí u vizualizací, v případě jednoho monitoru je OpenGL widget panelem v hlavním okně aplikace, zároveň však existuje možnost zobrazit ho i v samostatném okně například na druhém monitoru.

Panelů může být definováno libovolné množství, pro začátek se kromě vizualizací předpokládá i zobrazování výstupu z vykonávaného skriptu, seznam definovaných proměnných spolu s jejich hodnotami a zásobník volání funkcí.

3.7 Shrnutí

V této kapitole byly definovány základní myšlenky a rozhodnutí pro tvorbu interpretu speciálního programovacího jazyka pro manipulaci s grafy. Aplikace se naprogramuje v jazyce C++ s využitím standardní knihovny STL, grafická část v Qt a OpenGL.

Analýza odhalila nemožnost reprezentovat graf maticí sousednosti ani jinými technikami, ve kterých je struktura grafu uložena centrálně. Navržené řešení spočívá v obecných množinách vrcholů a hran, kde si vrcholy udržují seznam incidujících hran a hrany odkazy na svůj

počáteční a koncový vrchol. Díky této technice může být podporován libovolný typ grafů bez jakéhokoli omezení a hlavně s rozumnými složitostmi vykonávání všech operací.

Hodnoty a proměnné se ve skriptu implementují pomocí hierarchie tříd, jednotlivé operace pracují na bázi návrhového vzoru Double dispatching.

Skript je v interpretu reprezentován abstraktním stromem syntaxe. Jeho vykonávání spočívá v rekurzivním volání metody `execute()` nad každým uzlem tohoto stromu. Návratovou hodnotou je chytrý ukazatel na bázi čítání referencí zapouzdřující obecnou hodnotu. Skript může být, v případě debuggingu, pozastaven technikami vícevláknového programování.

Grafické uživatelské rozhraní zahrnuje hlavní okno s textovým editorem pro programátory a několik speciálních plovoucích panelů, které mohou být uživatelem vytrhnuty do samostatného okna.

Kapitola 4

Realizace

4.1 Základní kód a interpret pro příkazovou řádku

4.1.1 Lexikální analyzátor

Jako základ pro lexikální analyzátor byl použit zdrojový kód z předchozího autorova projektu [9, třída `CLexan`]. Jeho jádrem je rozsáhlý stavový automat, který zpracovává příchozí znaky a v závislosti na nich vrací lexikální tokeny.

Analyzátor bere v úvahu konstanty `__FILE__`, `__LINE__` a `__FUNCTION__`, které se při svém použití automaticky nahrazují za pozice ve zdrojových kódech. Dalšími vnitřními příkazy jsou jednoduchá bezparametrická makra `define("name", "value")` a vkládání externích souborů pomocí konstrukce `include("filename")`.

4.1.1.1 Procházení zdrojových dat

Makra i soubory se v principu zpracovávají naprosto stejně. Lexikální analyzátor obsahuje zásobník objektů typu `LexanIterator`, který slouží pro zanoření se do další úrovně ve zdrojových kódech skriptu. Například při inkluzi souboru se na vrcholu zásobníku vytvoří nová položka a při dokončení zpracování se odstraní. Překlad končí ve chvíli, kdy je zásobník vyprázdněn.

Abstraktní třída `LexanIterator` definuje rozhraní pro nejnižší vrstvu – vlastní čtení znaků ze vstupního proudu. Jediný požadavek na potomky je specifikace metody `get()`, která má za úkol vrátit následující znak v pořadí a `unget()` sloužící pro návrat o znak zpátky. Třída dále ukládá pozici ve zdrojových kódech používanou především v chybových zprávách.

Aktuálně jsou definováni pouze dva potomci. `LexanIteratorFile` má za úkol iterovat přes znaky v souboru, používá ho konstrukce `include`. Řetězce, nebo-li `define` makra, se zpracovávají třídou `LexanIteratorString`. V budoucnu je možné, bez větších zásahů do aplikace, přidat další zdroje kódu, například soubory nahrávané z internetu.

4.1.1.2 Identifikátory

Řetězce ve skriptu, jména proměnných a funkcí, se během lexikální analýzy nahrazují za celočíselné identifikátory, stejnou myšlenku používá i interpret JavaScriptu ve webovém prohlížeči Links [7, str. 25].

Ve chvíli, kdy lexikální analyzátor narazí ve zdrojovém proudu na shluk znaků, který by mohl být identifikátorem, zkontroluje nejdříve, zda se nejedná o klíčové slovo. V případě negativního výsledku projde seznam skriptem definovaných maker a pokusí se je rozbalit. Neuspěje-li ani zde, použije makro `STR2ID()`, které požádá tabulku symbolů, třídu `StringTable`, o překlad řetězce na celočíselný identifikátor. Opačný směr umožňuje makro `ID2STR()`.

Algoritmus překladu je velice jednoduchý. Tabulka symbolů je tvořena obyčejným polem, jehož jednotlivé položky ukládají řetězcovou reprezentaci, hodnota indexu představuje reprezentaci číselnou. Při žádosti o překlad z řetězce na identifikátor zkusí třída najít a vrátit pozici již existujícího prvku. Pokud neuspěje, přidá řetězec na konec pole a vrátí jeho pozici.

Asymptotická složitost operace vychází v nejhorším případě lineární, protože se musí při prvním výskytu řetězce projít kompletně všechny dříve definované prvky. Její snížení, například seřazením položek podle abecedy, není možné, pořadí v poli se nesmí změnit, protože by se zneplatnily již existující překlady. Opačný směr představuje operaci s konstantní složitostí, identifikátor je současně indexem do pole.

```
identifier StringTable::getID(const string& str)
{
    vector<string>::iterator pos = find(m_data.begin(), m_data.end(), str);

    if(pos == m_data.end())
        m_data.push_back(str);

    return pos - m_data.begin();
}

string& StringTable::getString(identifier id)
{
    return m_data[id];
}
```

Hlavní výhodou této techniky je vyšší rychlost interpretace, porovnání dvou čísel trvá mnohem kratší dobu než porovnání dvou řetězců. Úspora vzniká při každém přístupu k hodnotě proměnné a při volání funkcí. Druhá z výhod spočívá ve snížených paměťových nárocích, všechny řetězce jsou uloženy bez duplicit na jednom místě a ve skriptu je reprezentují pouze celočíselné hodnoty. Jak už bylo zmíněno, jediné zpomalení nastává při vytváření tabulky symbolů, je přímo úměrné počtu různých identifikátorů ve zdrojovém textu a jejich délce.

4.1.2 Gramatika, parser

Gramatiku vytvořeného jazyka je možné najít v příloze A na straně 53, alternativně ve zdrojovém souboru `parser.y`. Jedná se o standardní vstupní kód pro generátor gramatik GNU Bison, který má za úkol vystavět abstraktní strom syntaxe. Při vytváření se vycházelo z předpisu pro jazyk Ansi C [1].

4.1.3 Hierarchie tříd, základní třída BaseObject

V objektově orientovaném návrhu aplikačního rozhraní je vždy dobré zavést kořen hierarchie tříd, který poskytuje, či pouze definuje, základní obecné operace nad všemi používanými

objekty. V jazyce Java se taková třída jmenuje `Object`, knihovna Qt zavádí `QObject`, `CObject` je k nalezení v knihovně MFC. Vyvíjená aplikace použije jméno `BaseObject`.

4.1.3.1 Výpis struktury objektů, metoda dump()

`BaseObject` v první řadě definuje čistě virtuální metodu `dump()`, díky níž je možné vypsat obsah daného objektu. Využití je předpokládáno především při stavbě a ověřování abstraktního stromu syntaxe, metoda umožňuje jeho přehledný výpis.

Pokud každá odvozená třída definuje svoji metodu `dump()` – a ona ji definovat musí, protože je čistě virtuální – může do předaného proudu vypsat informace o sobě a o objektech, které v sobě uchovává. Mějme například jednoduchý demonstrační skript.

```
function main(argv)
{
    value = 1 + 2 * 3;
}
```

Po přeložení kódu do vnitřní reprezentace a zavolání `dump()` vypadá vypsaná struktura objektů následovně.

```
<Function name="main" id="63">
    <Parameter name="argv" id="64" />
    <NodeBinaryAss>
        <ValueIdentifier name="value" id="65">
            <NoValue />
        </ValueIdentifier>
        <NodeBinaryAdd>
            <ValueInt value="1" />
            <NodeBinaryMult>
                <ValueInt value="2" />
                <ValueInt value="3" />
            </NodeBinaryMult>
        </NodeBinaryAdd>
    </NodeBinaryAss>
</Function>
```

XML formát výstupu samozřejmě není nutný, byl použit pouze kvůli přehlednosti výpisu strukturovaných dat.

4.1.3.2 Kontrola úniků paměti

Druhou funkcionalitou poskytovanou základní třídou `BaseObject` jsou testy na úniky dynamické paměti. Jedná se pouze o rychlé ověření, zda změny provedené při vývoji nezpůsobily nějaký nečekaný problém, a proto záměrně není ani implementace nijak složitá.

Třída definuje statické úložiště adres alokovaných objektů. Při své konstrukci do něj vloží ukazatel na právě vytvořený objekt, v destruktoru ho smaže, toť vše. Jelikož se jedná o kořen

hierarchie tříd, tento konstruktor a destruktor se z definice volá u naprosto všech vytvářených objektů.

Těsně před ukončením funkce `main()` se zjistí aktuální počet alokovaných dat. Pokud není nulový, je jasné, že se někde v aplikaci objevil únik paměti a tato skutečnost je signalizována vývojáři. Díky RTTI, které jazyk C++ podporuje operátorem `typeid`, je možné získat i datový typ neuvolněných objektů.

Kód pro testy může způsobovat výrazné zpomalení běhu aplikace, a proto byl obalen do podmíněné sekce preprocesoru `CHECK_MEMORY_LEAKS`, jež je zapnutá pouze při *debug* kompilaci. Ve výsledné verzi samozřejmě nejsou tyto testy potřeba a ve spustitelném souboru se ani fyzicky nenacházejí.

Pozorný čtenář si pravděpodobně uvědomil, že nejsou kontrolovány pouze dynamické, ale i statické alokace. Možná by stálo za úvahu vložit kód do přetížených operátorů `new` a `delete`, nicméně statické objekty jsou uvolněny vždy, a proto nemají žádný vliv na správnost zjištěné informace.

Dále se nekontrolují alokace paměti, která nepatří do `BaseObject` hierarchie, ty se však v kódu vyskytují spíše zřídka. Pro zjištění jejich úniků slouží, taktéž používaný, nástroj Valgrind (http://www.valgrind.org/).

4.1.3.3 Pořadí inicializace statických objektů

Učebnice jazyka C++ se pouze velice zřídka věnují tématu *pořadí inicializace statických objektů* (angl. static object initialization order), jednou z mála výjimek je [3]. Jedná se o situaci, kdy linker při spojování jednotlivých modulů může způsobit zásadní nefunkčnost naprosto správného kódu!

Termín *statický objekt* je zde používán ve významu atributu třídy deklarovaného s modifikátorem `static`, jenž způsobuje sdílení mezi odpovídajícími objekty. Jako praktickou ukázku nastavší chyby můžeme použít statickou proměnnou `m_allocated_objects` třídy `BaseObject` z kapitoly 4.1.3.2 na straně 25.

Statický objekt je vytvářen a inicializován ještě před vlastním spuštěním funkce `main()`, jazyk C++ pro instanciaci a inicializaci používá speciální konstrukci zapsanou vně všech funkcí a tříd.

```
class BaseObject
{
    static set<BaseObject*> m_allocated_objects;
};
set<BaseObject*> BaseObject::m_allocated_objects;
```

Definujme ještě jednu třídu, singleton poděděný od `BaseObject`, jenž v sobě obsahuje taktéž statický objekt.

```
class Singleton : public BaseObject
{
    static Singleton m_instance;
};
Singleton Singleton::m_instance;
```

Inicializuje vygenerovaný kód `m_allocated_objects` dříve než proměnnou `m_instance`, či naopak? Jedná se o zásadní otázku, protože `Singleton` dědí od `BaseObject`, a tudíž pracuje s proměnnou `m_allocated_objects`, která však v tuto chvíli ještě nemusí existovat. Vše tedy závisí na linkeru, nevhodné pořadí spojení modulů může způsobit neoprávněný přístup do paměti a následný pád aplikace ještě před vlastním spuštěním funkce `main()`.

Pokud nejde použití statických objektů v aplikaci zamezit nebo je jiné řešení silně nepraktické, existuje proti chybě relativně snadná obrana. Stačí založit nový soubor se zdrojovými kódy a přesunout do něj všechny problematické inicializace.

Jazyk C++ garantuje pořadí inicializací v rámci jednotlivých modulů. Nahoru je tedy třeba zapsat ty, jež se mají provést nejdříve, a dolů ty, které na nich závisí. Obsah souboru by tedy vypadal následovně.

```
set<BaseObject*> BaseObject::m_allocated_objects;
Singleton Singleton::m_instance;
```

Vyvíjená aplikace nakonec používá zcela jiné řešení. Statické nejsou samotné objekty (singletony), ale pouze ukazatele na ně, inicializace na `NULL` vždy proběhne bezchybně. O vytvoření a smazání vlastních objektů se stará kód na začátku a konci funkce `main()`, který může explicitně ovlivnit pořadí.

Další možností by mohlo být nastavení priorit modulů při linkování, například u Makefile/gcc se jedná pouze o pořadí zápisu na příkazové řádce. Problémy spojené s tímto řešením nejspíše vyvstanou až někdy v budoucnu.

Už bylo řečeno, že chyba není fyzicky přítomná v kódu, a tudíž se na ni při libovolné změně může snadno zapomenout. Typicky se jedná o přidání nového souboru, či o aktualizaci nebo změnu buildovacího systému. Používaná vývojová prostředí, současná i budoucí, tato nastavení také nemusí vůbec podporovat, a proto je lepší vzít v úvahu spíše řešení spočívající ve změně zdrojových kódů.

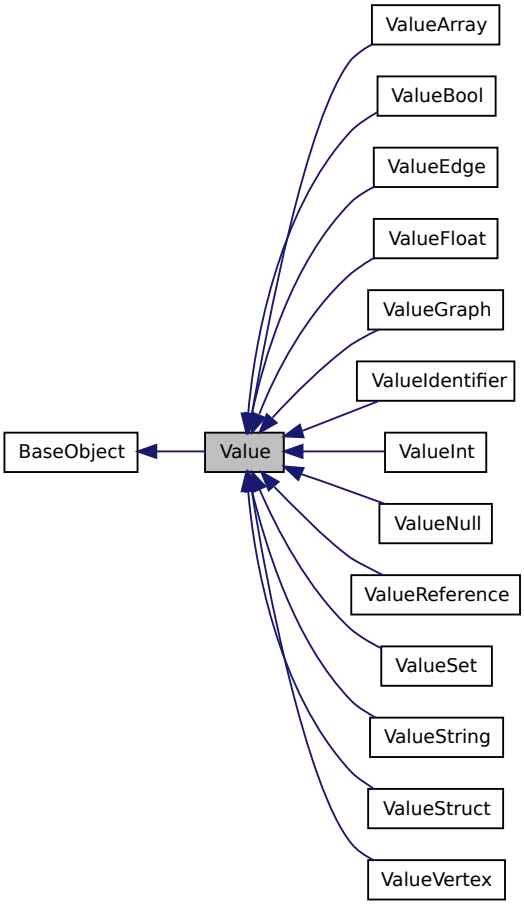
Je pravda, že výše uvedená ukázka je k chybě výrazně náchylnější než běžná použití statických objektů. Na druhou stranu, autor se již dříve, při portaci legacy aplikací na nový operační systém, setkal s obdobným problémem. V obou případech se navíc jedná o dva různé kompilátory – Microsoft Visual Studio 2003/2005 oproti GNU gcc.

4.1.4 Hodnoty a proměnné ve skriptu, Value* hierarchie tříd

Jak už bylo zmíněno v kapitole 3.4.2 analýzy na straně 16, binární operace nad hodnotami ve skriptu jsou vykonávány s využitím návrhového vzoru Double dispatching. Kořenem hierarchie je abstraktní třída `Value`, potomci se jmenují `ValueNull`, `ValueBool`, `ValueInt`, `ValueFloat` atd., class diagram této skupiny tříd je uveden na obrázku 4.1.

Iterování přes prvky složených datových typů je podporováno čtyřmi, taktéž virtuálními, metodami `iterator()`, `hasNext()`, `next()` a `resetIterator()`. Jednoduché datové typy nejsou iterovatelné, a proto jakékoli pokusy o procházení hned na začátku odmítnou.

Potomci, především grafové třídy, definují i další metody, není reálné, aby všechny operace, včetně silně specifických, byly virtuální. Ke konverzi z obecného `Value` ukazatele na konkrétní datový typ slouží skupina virtuálních funkcí `toValueBool()`, `toValueInt()` atd., které vracejí `NULL`, pokud se nejedná o daný datový typ. Na tomto místě by šlo samozřejmě použít i RTTI přetypování `dynamic_cast<>`, obě techniky jsou ekvivalentní. Použití leží především v zabudovaných funkcích interpretu, které v parametrech předpokládají konkrétní datové typy a volají jejich specifické operace.



Obrázek 4.1: Hierarchie Value* tříd

4.1.4.1 ValueNull a ValueBool

Obě tyto třídy jsou definovány jako singleton a v kódu mohou být používány pomocí zkráceného zápisu `VALUENULL`, `VALUEBOOL_TRUE` a `VALUEBOOL_FALSE`. Jedná se o makra, která volají statické metody `getInstance()`.

Implementace pomocí singletonu byla zvolena z důvodu, aby nebylo nutné neustále vytvářet nové a nové ekvivalentní objekty.

4.1.4.2 Přiřazování do proměnných

K dotazu, zda je možné do nějakého `Value` objektu přiřadit jinou hodnotu, slouží metoda `isLValue()`. Vlastní přiřazení je následně možné provést funkcí `assign()` nebo `assignRef()` v případě přiřazování reference.

Proměnné jsou ve skriptu reprezentovány třídou `ValueIdentifier`, která v sobě uchovává identifikátor hodnoty odkazující do kontextu skriptu. Všechny operace se vykonávají stejně jako u ostatních tříd, rozdíl spočívá pouze v tom, že se neprovádějí přímo s daným objektem, ale s odkazovaným.

Jelikož je navržený jazyk beztypový, proměnné musejí umět měnit nejen svou hodnotu, ale i datový typ, jedná se tedy o složitější operaci, než je například pouhá záměna jednoho

čísla za jiné. Interpret tuto situaci řeší obalením vlastní hodnoty další třídou pojmenovanou `ValueReference`, přiřazení poté probíhá podle následujícího schématu:

1. Operátor přiřazení vyhodnotí výraz na své levé straně:
   - Proměnná požádá kontext o nalezení hodnoty uložené pod daným jménem.
   - Pole poskytne prvek na daném indexu.
   - Struktura najde prvek specifikovaného jména.
   - Kombinace předchozích možností (např. `objects[2].position.x`).
2. Výsledkem je vždy nějaká hodnota obalená ve třídě `ValueReference`.
3. Operátor přiřazení vyhodnotí výraz na své pravé straně.
4. `ValueReference` je požádáno o záměnu obalené hodnoty za novou.

V případě, že proměnná daného jména, popř. položka struktury, ještě neexistuje, nově vytvořená `ValueReference` obaluje `ValueNull` hodnotu. Pole se při přístupu za své hranice automaticky nezvětšuje, je vypsána chyba.

4.1.4.3 Grafové třídy

Všechny grafové třídy jsou, stejně jako ostatní hodnoty, potomky abstraktní třídy `Value`, a tudíž musí i ony implementovat Double dispatching metody vyhodnocování operátorů. Třída `ValueGraph` reprezentuje graf na principu množin vrcholů a hran, který byl představen v kapitole 3.3.2.2 na straně 13.

Pro ukládání vrcholů a hran se nepoužívá původně preferovaná standardní šablona `set`, ale vlastní třída `ValueSet`, která zobecňuje práci s těmito množinami a současně zpřístupňuje interpretu i nový datový typ. Toto řešení vychází především z problémů s *dangling* ukazateli v případě, kdy jsou veškerá data grafu uvolněna, ale z interpretu stále existují odkazy na vrcholy a hrany – například reference v proměnné nebo v některé z množin.

K řešení se opět využívá principu čítání ukazatelů. Aby nedocházelo ke dříve zmíněným kruhovým závislostem mezi vrcholy a hranami (kapitola 3.5.3.2 na straně 19), musí hrany ukládat pouze obyčejné ukazatele na počáteční a koncový vrchol a vždy testovat jejich platnost.

Graf může být vytvořen ručně opakovaným voláním funkcí `generateVertex()` a `generateEdge()`, či automaticky z datového souboru funkcí `loadFromFile()`. Formát souborů je specifikován v příloze B na straně 57.

Vykonávané operace rozlišují, zda je graf orientovaný či ne, orientace hran může být také kdykoli invertována. Matice sousednosti není přímo přístupná, protože se s ní vnitřně nepočítá, ale může být na žádost `getAdjacencyMatrix()` vygenerována a vrácena.

Třídy `ValueVertex` i `ValueEdge` v sobě obsahují `ValueStruct` atribut a mapují na něj všechny nedostupné operace, aby jejich rozhraní bylo ekvivalentní – jedná se o variaci na návrhový vzor *Proxy*. Skládání dostalo přednost před dědičností, protože vrchol ani hrana *nejsou* ve své podstatě strukturou. Návrh funkcionality je pouze takový, že poskytují stejné služby, ale to není dostatečným důvodem pro dědičnost.

Poslání množin vrcholů a hran reprezentovaných třídou `ValueSet` je především udržovat záznamy o registrovaných objektech. Kromě toho se navíc mohou iterovat a je s nimi možné vykonávat operace průniku, sjednocení či rozdílu. `ValueSet` neslouží výhradně pro vrcholy a hrany, ale pro libovolný typ `Value` objektů.

4.1.5 Abstraktní strom syntaxe, Node* hierarchie tříd

Abstraktní třída `Node` byla již představena v analýze, v kapitole 3.5.1 na straně 17. Z ní vycházejí všechny ostatní třídy, jejichž objekty ve výsledku vytvářejí strukturu abstraktního stromu syntaxe. Hierarchie tříd je k nalezení v příloze D na straně 61.

Unární operátory jsou potomky třídy `NodeUnary`, která předpokládá jednoho následníka ve stromě, `NodeBinary` logicky dva. Ternální operátor je interně mapován na `if-else` větvení poskytované třídou `NodeCondition`. Dalšími podtřídami jsou například `NodeLoop` a `NodeForeach` cykly, `NodeBlock` zapouzdřuje skupinu příkazů, `NodeFunctionCall` slouží pro volání funkcí. Prázdný příkaz `NodeEmptyCommand` se využívá spíše interně.

Třída `NodeValue` definuje spojnici mezi `Node` a `Value` hierarchiemi, `NodeGlobal` poskytuje propagaci globálních proměnných do lokálního kontextu funkce. `NodePosition` slouží pro aktualizaci pozice odkazující do zdrojových kódů skriptu.

4.1.5.1 Strukturované skoky

Strukturované skoky `break`, `continue` a `return` používají mechanismus výjimek, což je jedna z mála výraznějších daní za jednoduchost vykonávání skriptu pomocí rekurzivního volání metody `execute()`. Příkaz skoku, byť strukturovaný, musí být nějakým způsobem schopen opustit zanořené volání a vystoupat do požadovaného místa. Výjimky jsou zde opravdu tou nejelegantnější cestou.

Místo, kde se má skok zastavit, je definováno strukturou objektů ve stromu. Například výše uvedený příkaz `break` by se měl zastavit za koncem nejzanořenějšího cyklu. V případě `continue` se samozřejmě jedná o návrat na začátek před další iteraci. Přesně podle této definice funguje uzel `NodeLoop`, výjimka typu `NodeJumpBreak` je zachytávána za vlastním tělem cyklu, `NodeJumpContinue` uvnitř.

```
CountPtr<Value> NodeLoop::execute(void)
{
    m_init->execute();

    try
    {
        while(m_condition->execute()->toBool())
        {
            try
            {
                m_body->execute();
            }
            catch(NodeJumpContinue* ex) { }

            m_inc->execute();
        }
    }
    catch(NodeJumpBreak* ex) { }
}
```

Protože volání cyklů probíhá od vnějších k vnitřním, je případná výjimka vždy zachycena v nejvnitřnějším možném cyklu. Výkonnostní poměry u těchto typů konstrukcí příliš neutrpí, `break` a `continue` se v reálných programech vyskytují spíše zřídka.

U návratových hodnot funkcí je situace opačná, `return` se vyskytuje téměř v každé. Ano, zde jsou výjimky bolestné, nicméně, z druhé strany pohledu, ne tak moc jako implementace bez jejich služeb.

Příkaz `return` by mohl nastavit hodnotu sdílené proměnné v kontextu na výstupní hodnotu z funkce a zároveň v další proměnné signalizovat, že probíhá návrat z funkce. Metoda `execute()` by u naprosto všech tříd musela testovat tuto signalizační proměnnou, a v době mezi `return` až do ukončení těla funkce by se nezanořovala do dalších příkazů. Obrovské množství těchto větvení by jistě přesáhlo režii jedné nastavší výjimky.

4.1.5.2 Funkce

Funkce, potomci třídy `NodeFunction`, se dělí na dvě skupiny – na zapsané ve skriptu a na zabudované do interpretu. Každou z nich identifikuje jméno a parametry, při volání se ovšem používá pouze jméno. První typ při inicializaci očekává tělo s příkazy a pozici ve zdrojových kódech, kde je definována. U zabudovaných funkcí, potomků `NodeFunctionBuiltin`, tyto informace nemají význam, tělo je specifikováno přímo C++ kódem metody `execute()`.

Funkci umí zavolat třída `NodeFunctionCall`. Ta se ji nejdříve pokusí najít pomocí uloženého jména a zkontroluje počty deklarovaných a předaných parametrů. Pokud vše souhlasí, vyhodnotí parametry, vloží na zásobník volání novou položku, podle specifikovaných jmen parametrů definuje požadované proměnné a spustí tělo funkce.

4.1.6 Kontext skriptu

V textu byl již několikrát zmíněn, do této chvíle bez dostatečného vysvětlení, termín *kontext skriptu*. V kódu se jedná o jednu z funkcionálně nejbohatších tříd aplikace pojmenovanou `Context`. Tato třída mimo jiné ukládá zabudované i uživatelské funkce, globální proměnné a zásobník volání funkcí spolu s lokálními proměnnými. Dále si vede záznam o aktuální pozici ve zdrojových kódech, který slouží především při vypisování chybových zpráv, a také umí pomocí databáze řetězců převádět jména identifikátorů skriptu z řetězcové do číselné formy a naopak (viz kapitola 4.1.1.2 na straně 23).

4.1.6.1 Funkce ve skriptu, jejich volání a spuštění skriptu

Všechny definované a tudíž i volatelné funkce se ukládají do asociativního pole indexovaného číselným identifikátorem, datová položka se skládá z ukazatele na abstraktní nadtřídu `NodeFunction`. Kontext nepotřebuje znát, o který z obou typů funkcí se jedná, samotné objekty to vědí a díky polymorfismu se při spuštění volají odpovídající metody. Hledání pouze podle jména znemožňuje přetěžování funkcí, k jeho zprovoznění by bylo potřeba ukládat dodatečné informace, například počet argumentů, a používat je i při vyhledávání.

Zásobník volání je taktéž uložen v kontextu skriptu. Jedná se o pole objektů typu `CallStackItem`, jež v sobě ukládají jméno právě spuštěné funkce, pozici v kódu, odkud byla zavolána, a hodnoty lokálních proměnných na dané úrovni.

Vlastní spuštění funkce je úkolem třídy `NodeFunctionCall`. Objekty tohoto datového typu si v sobě uchovávají identifikátor volané funkce, kód nutný pro zjištění hodnot parametrů a pozici v kódu, odkud je funkce volána. Celý životní cyklus funkce vypadá následovně:

1. V kontextu je podle jména nalezen objekt `NodeFunction*`.
2. Ověří se počet deklarovaných a předaných parametrů.
3. Vyhodnotí se kód nutný pro získání hodnot parametrů.
4. Na vrchol zásobníku volání je přidána nově vytvořená položka.
5. Nastaví se hodnoty argumentů (lokální proměnné) na hodnoty předané v parametrech.
6. Zavolá se `execute()` metoda objektu funkce.
7. Vykonají se všechny příkazy obsažené v těle funkce.
8. Po skončení se z vrcholu zásobníku volání odstraní dříve přidaná položka.
9. Vrátí se návratová hodnota.

Již bylo řečeno, že skript se spouští voláním funkce `main(argv)`. Tuto operaci opět poskytuje kontext, konkrétně se jedná o metodu `executeScriptMain()`, jež vytvoří objekt typu `NodeFunctionCall`, do konstruktoru mu předá identifikátor *main* a `ValueArray` parametr. Následně je odstartováno rekurzivní volání `execute()` metod nad uzly abstraktního stromu syntaxe.

4.1.6.2 Globální proměnné

Seznam lokálních i globálních proměnných je stejně jako kontejner definovaných funkcí reprezentován asociativním polem. Kontext neposkytuje žádné metody pro získání nebo nastavení hodnoty globální proměnné. Naproti tomu umožňuje funkcí `propagateGlobalVariable()` namapovat její hodnotu do adresního prostoru právě zavolané lokální funkce, což ve skriptu přesně odpovídá chování operátoru `global` představeného v kapitole 3.2.1 na straně 6.

Celý mechanismus funguje na principu sdílení `ValueReference` objektu mezi globálními a lokálními proměnnými. Změní-li se hodnota lokální proměnné, která odkazuje na stejný objekt jako globální proměnná, je automaticky aktualizována i ta, protože sídlí na stejném místě v paměti.

4.1.6.3 Pozice v kódu

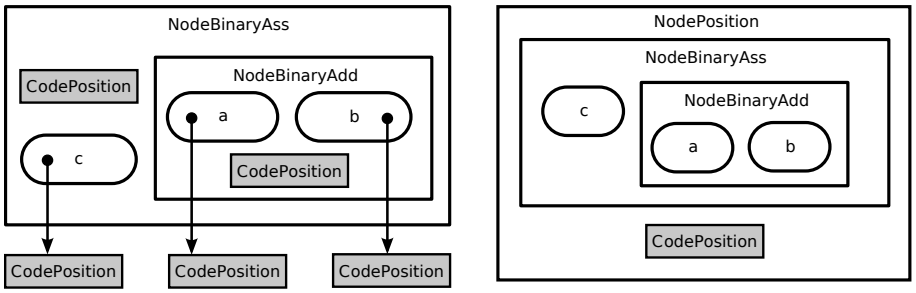
Je jasné, že není potřeba, aby byla pozice ve zdrojových kódech reprezentovaná třídou `CodePosition` přítomna v naprosto každém z uzlů stromu syntaxe. Dokud by se nezměnilo jméno vstupního souboru ani číslo řádky, docházelo by pouze k paměťově náročným duplikacím, které by při vykonávání nepřinášely žádnou novou informaci.

Bylo by žádoucí, aby každá řádka vstupního souboru měla svou uzlovou reprezentaci ve stromu, avšak syntaktický analyzátor v dané implementaci nepracuje po řádcích, ale po tokenech. Pokud se berou v úvahu *příkazy* namísto řádků, je možné se k tomuto požadavku přiblížit. Téměř každý příkaz (výraz zakončený středníkem) bývá uveden na samostatné

řádce, v některých méně častých případech i na dvou či třech. To ale nevadí, pozice v chybových hlášeních mají pouze o něco menší přesnost, stále však naprosto dostatečnou.

Místa ve zdrojových kódech jsou tedy uchovávány pouze ve speciálních uzlech stromu, příkazy se obalují objekty typu `NodePosition`, jejichž prací je aktualizovat pozici v kontextu a zavolat metodu svého následníka ve stromu.

Oba způsoby ukládání pozic v jednoduchém demonstračním příkazu `c = a + b;` jsou zobrazeny ve schématu 4.2. Třída `NodeBinaryAdd` odpovídá sčítání a `NodeBinaryAss` přiřazení. Na první pohled je jasné, že ukládání pozic do každého uzlu stromu by bylo zbytečné a navíc by docházelo ke značnému plýtvání pamětí.



Obrázek 4.2: Dva možné přístupy k ukládání pozic v abstraktním stromu syntaxe

Třída `NodePosition` v sobě zapouzdřuje objekt typu `CodePosition`, jenž je jednoduchým kontejnerem uchovávajícím jméno souboru a číslo řádky. Jméno souboru opět není ukládáno v řetězcové formě, ale jako číselné ID, což přispívá k dalšímu šetření paměti.

Kontext skriptu obsahuje datovou `CodePosition` položku, jež je během průchodu abstraktním stromem syntaxe s pomocí `NodePosition` uzlů a volání `setPosition()` průběžně aktualizována a může být použita například při výpisu chybové zprávy.

4.1.6.4 Paralelizace skriptů, více kontextů

Kontext skriptu je ostatními třídami používán téměř jako singleton, přistupují k němu pomocí makra `CONTEXT`, které se rozbaluje na volání `getContext()` třídy `ObjectCreator`. V průběhu vykonávání skriptu existuje vždy právě jeden kontext, a tudíž není paralelní běh na úrovni skriptu aktuálně podporován.

Pokud by vznikla potřeba vícevláknového programování, existuje v zásadě pouze jediná možnost, jak doprogramovat jeho podporu. Spočívá v dodatečném argumentu `execute()` metod, který si uzly stromu předávají při rekurzivním vykonávání. Nevýhody jsou zřejmé, naprostá většina uzlů kontext ke své činnosti nepotřebuje, a tudíž by jeho neustálé kopírování, byť i ve formě reference, zpomalovalo běh skriptu.

4.1.7 Logování, textový výstup

Výpis textového výstupu ze skriptu, varovných, chybových a ladících zpráv kompletně zajišťuje třída `Logger`. Aplikace definuje několik maker, například `ERR()`, `WARN()` nebo `INFO()`, které slouží jako aplikační rozhraní pro výpis zpráv uživateli.

Přímý přístup k vstupně/výstupním proudům jazyka C++ není používán, aby bylo jednodušší přesměrovat veškerý výstup skriptu do některého z oken aplikace při běhu v grafickém rozhraní. Tomuto tématu se více věnuje kapitola 4.2.1 na straně 35.

4.1.8 Generátory zdrojových kódů

Interpret obsahuje několik skupin tříd, jejichž kód je velice podobný, typickým příkladem jsou třídy aritmetických a logických operátorů. Vezměme si třeba operátory pro sčítání a odčítání, jejich výkonná část se liší jediným příkazem. Sčítání volá metodu `add()` a odčítání `sub()`, jinak je celý kód identický.

```
return m_left->execute()->add(*(m_right->execute()));
return m_left->execute()->sub(*(m_right->execute()));
```

Kromě těchto dvou je potřeba dalších dvacet binárních operátorů, k tomu několik unárních operátorů a nesmí se zapomenout ani na zabudované funkce. Vytvářet všechny třídy ručně je zdlouhavé a neefektivní, v případě změn je třeba manuálně upravovat stejným způsobem spoustu souborů.

Nepříliš vhodné řešení této situace je využití C++ maker, jimiž lze generovat libovolný kód, tedy i třídy. Jedna z primitivnějších a ne moc hezkých implementací by mohla vypadat například následovně.

```
#define UNARYOPERATOR(NAME, CODE)             \
class NAME : public NodeUnary                 \
{                                             \
public:                                       \
    NAME(Node* next) : NodeUnary(next) { }    \
                                              \
    virtual CountPtr<Value> execute(void)     \
    {                                         \
        CODE                                  \
    }                                         \
};

UNARYOPERATOR(NodeUnarySub, return m_next->execute()->subUn();)
UNARYOPERATOR(NodeUnaryNot, return m_next->execute()->logNOT();)
```

Jedná se o plně funkční kód v profesionální praxi běžně používaný, ale obecně hodně špatný a naprosto nevhodný pro libovolnou situaci. Kvůli těmto konstrukcím především není možné ladění programu. Při psaní také nefunguje automatické doplňování kódu, protože s podobným zápisem tvůrci žádného IDE zkrátka nepočítali. Počáteční úsporu času a zrychlení vývoje velice rychle, při prvních problémech a neúspěšných pokusech o ladění, vystřídají myšlenky na kompletní přepsání ručním rozbalením maker.

Řešením situace je sáhnout po externím skriptu, jehož jediným posláním je podle připravené šablony generovat C++ soubory s definicí jednotlivých tříd. Pokud vznikne požadavek na úpravu, změní se šablona a skript všechny soubory během krátké chvíle automaticky přegeneruje.

Ve vyvinuté aplikaci toto navržené řešení používají perlovské skripty `gen_operators.pl` a `gen_builtin.pl`, které generují C++ kód unárních operátorů, binárních operátorů a také všech zabudovaných funkcí. Návod na vytvoření nové zabudované funkce je uveden v příloze C na straně 59.

4.2 Grafické uživatelské rozhraní

4.2.1 Oddělení od interpretu pro příkazovou řádku

Grafické uživatelské rozhraní je pouze alternativou k interpretu pro příkazovou řádku, a proto by měly být oba zdrojové kódy co nejvíce odděleny. Ideálem zůstává samostatná knihovna zapouzdřující veškerý základní kód a dvě nezávislá uživatelská rozhraní – konzolová a okenní aplikace. Vytvořené programy se dají zkompilovat nezávisle na sobě, sdílí většinu zdrojových kódů, nicméně nepoužívají žádnou knihovnu.

Některé třídy, typicky logování, není možné sdílet, a proto aplikace zavádí singleton `ObjectCreator`, jenž se stará o vytváření a poskytování těchto objektů.

Pokud skript potřebuje vypsat chybové hlášení, požádá creator o logovací třídu a s její pomocí vypíše chybovou zprávu. V případě, že aplikace běží v textovém režimu, vrátí creator logovací třídu pro textový režim, která vypíše text na konzoli. U grafického režimu je vrácen jiný datový typ vkládající text do logovacího okna. Oba objekty mají stejné rozhraní, protože jsou potomky společné třídy.

Ano, řešení používá návrhový vzor *továrna*. Zůstaneme-li u příkladu s logováním, aplikace definuje celkem tři třídy pro výstup zpráv, abstraktní nadtřídu `Logger` a dva konkrétní potomky `CliLogger` a `GuiLogger`. Dále využívá, taktéž tři, továrny – rozhraní `ObjectFactory` a jeho potomky `CliFactory` a `GuiFactory`. Rodičovská rozhraní jsou kompilována v obou verzích programu, `Cli*` třídy pouze v konzolové a `Gui*` třídy pouze v grafické části.

Nyní se vrátíme zpět k singletonu `ObjectCreator`, jenž poskytuje vlastní objekty. Při spuštění programu je mu předán některý z potomků `ObjectFactory`, s jehož pomocí může velice elegantně vytvořit správný typ objektů, aniž by věděl, ve které verzi programu se nachází. Pracuje pouze s obecnými rozhraními rodičovských tříd a díky polymorfismu volá správné metody. Tovární třída je vytvořena na začátku funkce `main()`, ta se samozřejmě musí v obou verzích programu taktéž lišit.

4.2.2 Vlákna a jejich synchronizace

Rozdíl mezi oběma verzemi aplikace je i ve způsobu, jakým je skript spouštěn. Konzolový program se nemusí starat o interakci s uživatelem, a tudíž mu i pro běh skriptu postačuje pouze jedno vlákno.

Okenní rozhraní, na rozdíl od něj, neslouží pouze pro jednorázové vykonání skriptu, ale umožňuje i psaní kódu v editoru, vizualizace a další činnosti, a proto musí být vykonávání skriptu vloženo do dalšího, pracovního, vlákna. Bez něj by déletrvající skript způsobil zamrznutí GUI a nemožnost debuggingu ani vizualizací.

Spolu s druhým a dalšími vlákny v aplikaci automaticky přichází i požadavek na jejich synchronizaci. Podobně jako u logování v kapitole 4.2.1 na straně 35 je vhodné napsat tento kód pro obě aplikace zvlášť.

Opět vytvoříme tři třídy, obecné rozhraní `Mutex` a konkrétní `CliMutex` a `GuiMutex`. Rodičovská třída požaduje po svých potomcích definovat pouze metody `lock()` a `unlock()`. Už bylo řečeno, že konzolová aplikace nepotřebuje synchronizaci vláken, a proto může `CliMutex` definovat tyto dvě funkce jako prázdné. `GuiMutex` využije služeb knihovny Qt a její třídu `QMutex`, ten je nutné definovat jako rekurzivní (`QMutex::Recursive`), aby mohl být se zanořením do funkcí zamknut v daném vláknu opakovaně.

Aby se omezil počet možných chyb se synchronizací, byla dále, po vzoru `QMutexLocker` z knihovny Qt, vytvořena třída `MutexLocker`. Ta v konstruktoru zamyká předaný mutex, v destruktoru je automaticky odemknut. Nyní stačí na začátku synchronizované funkce vytvořit statický objekt tohoto datového typu a je zajištěno, že se `lock()` a `unlock()` zavolá automaticky. Makro `ACCESS_MUTEX_LOCKER` dále zkracuje celý zápis a zároveň umožňuje synchronizaci kompletně odstranit – vhodné pro konzolovou aplikaci.

```
#ifdef DISABLE_THREAD_SYNCHRONIZATION
#define ACCESS_MUTEX_LOCKER
#else
#define ACCESS_MUTEX_LOCKER \
    MutexLocker mutexLocker(ObjectCreator::getInstance().getAccessMutex())
#endif
```

4.2.3 Spuštění a běh skriptu

Z uživatelského hlediska se skript spouští kliknutím na tlačítko *Run*, které je namapováno na slot `runScript()` hlavního okna. Při jeho aktivaci se automaticky uloží všechny otevřené soubory, z dokumentu v aktivním editoru se vezme disková cesta se jménem skriptu, inicializuje se `ScriptThread` objekt (dědí ze třídy `QThread`) a spustí se nové pracovní vlákno.

V metodě `run()` vlákna se nejdříve resetuje kontext skriptu do výchozího stavu a nastaví se požadované běhové parametry. Dále je naparsován vstupní soubor, vytvořen abstraktní strom syntaxe a spuštěna hlavní funkce skriptu.

Samotný skript komunikuje s GUI vláknem spíše zřídka. Výjimkou je zasílání chybových a informačních zpráv k zobrazení v panelu s textovým výstupem a signál o pozastavení běhu breakpointem. Ten má za následek aktualizaci uživatelského rozhraní včetně panelu aktuálně definovaných proměnných a překreslení vizualizací.

4.2.4 Ladění skriptu

V požadavcích na vytvářenou aplikaci je definováno, že musí umožňovat krokování skriptu, které ve své složitější formě odpovídá debuggingu. Kapitola 3.5.2 na straně 18 analyzovala možnosti pozastavení skriptu a pro grafickou aplikaci přišla s nápadem použít synchronizační prostředky vláken. `QWaitCondition` parametr je obsažen ve třídě `GuiContext` a vlastní pozastavení je možné realizovat zabudovanou funkcí `pauseExecution()`.

```
void GuiContext::pauseExecution(void)
{
    emit executionPaused();
    m_dbgMutex.lock();
    m_waitCondition.wait(&m_dbgMutex);
    m_dbgMutex.unlock();
}
```

Implementovaný debugging podporuje celkem čtyři níže uvedené způsoby obnovení běhu skriptu známé z běžných vývojových prostředí:

- Continue – zastavit na následujícím breakpointu.
- Step into – zastavit na následujícím příkazu (i uvnitř volané funkce).
- Step over – zastavit na následujícím příkazu, ignorovat příkazy uvnitř volaných funkcí.
- Step out – zastavit po opuštění aktuálně prováděné funkce.

Zdrojový kód definuje tyto čtyři způsoby jako výčet `SteppingType`, jenž nabývá hodnot `ST_NONE`, `ST_INTO`, `ST_OVER` a `ST_OUT`. Aktuálně používané krokování je uloženo v kontextu v proměnné `m_steppingType`.

Nejjednodušším typem pokračování běhu je samozřejmě *continue*. V jeho případě se pouze nastaví krokování na `ST_NONE` a probudí se vlákno skriptu. Tato funkce je volána z GUI vlákna například po kliknutí uživatelem na odpovídající tlačítko v nástrojové liště.

```
void GuiContext::debugRun(void)
{
    ACCESS_MUTEX_LOCKER;
    m_steppingType = ST_NONE;
    m_waitCondition.wakeAll();
}
```

Další tři typy krokování však vyžadují o něco více pozornosti. Jak bylo vidět před chvílí, spustit vlákno není nic složitého, problém spíše spočívá v jeho opětovném pozastavení. Nejvhodnější místo pro tuto činnost je metoda kontextu `setPosition()`, jež má za úkol aktualizovat pozici ve skriptu – podrobnosti viz kapitola 4.1.6.3 na straně 32. Funkce se volá periodicky po každém provedeném příkazu a to je to, co nás zajímá.

Nová implementace na začátku zavolá metodu předka a poté se větví podle nastaveného typu krokování. Jak už bylo řečeno, `ST_NONE` nedělá nic a `ST_INTO` se pozastaví na nejbližším následujícím příkazu. `ST_OVER` a `ST_OUT` navíc berou v úvahu úroveň zanoření na zásobníku volání funkcí, parametr `m_callStackSize` byl na svou hodnotu nastaven spolu s `m_steppingType` ve chvíli, kdy uživatel kliknul na odpovídající tlačítko pro pokračování.

```
void GuiContext::setPosition(const CodePosition* pos)
{
    Context::setPosition(pos);

    switch(m_steppingType)
    {
    case ST_INTO:
        pauseExecution();
        break;

    case ST_OVER:
        if(getStackSize() <= m_callStackSize)
            pauseExecution();
        break;
```

```
    case ST_OUT:
        if(getStackSize() < m_callStackSize)
            pauseExecution();
        break;

    case ST_NONE:
    default:
        break;
    }
}
```

Ačkoli se to nemusí zdát, pouze toto maličké množství kódu realizuje kompletní debugging ve vytvořené aplikaci. Jediný rozdíl, oproti běžným IDE, spočívá v definici breakpointu, jedná se zde o zabudovanou funkci zapsanou přímo do zdrojových kódů skriptu.

Jasnou výhodou z implementačního hlediska je snadnost řešení. Kontext si nemusí vést databázi breakpointů a v každém kroku pro všechny z nich porovnávat, zda jejich pozice spadá mezi číslo řádku předchozího a aktuálního příkazu. Při zápisu do zdrojového kódu pozastaví breakpoint skript automaticky.

Druhou obrovskou výhodou je možnost parametrizace zabudované funkce, aktuální implementace zastavuje skript pouze, když je předán argument odpovídající logickému true. Jak ukazují následující příklady, díky jediné dodatečné podmínce ve zdrojových kódech, dostává programátor k dispozici opravdu komplexní nástroj.

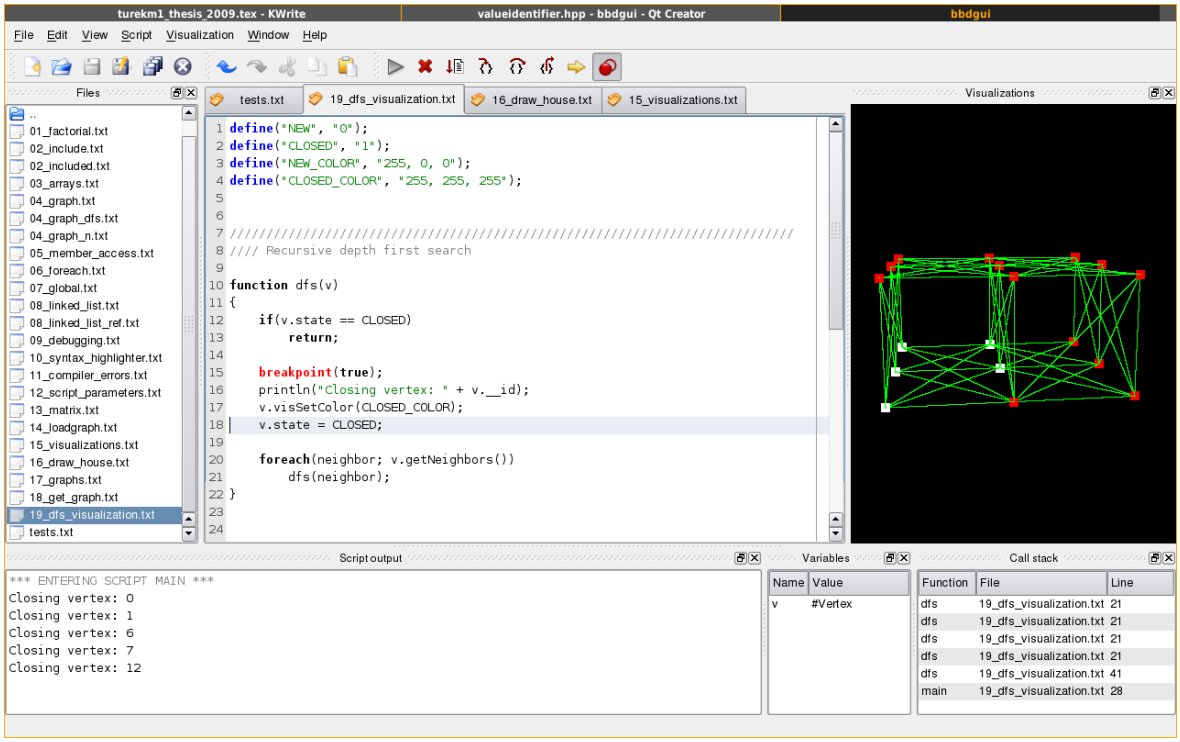
```
breakpoint(true);// Active
breakpoint(false);// Nonactive

breakpoint(cond1 || cond2);// Depends on the values

for(i = 0; i < 1000; ++i)
    breakpoint(i % 100 == 99);// Every hundredth iteration
```

Nevýhodou tohoto řešení je určitě statičnost, není možné definovat nový breakpoint po spuštění skriptu, protože zdrojový kód je již zpracován a změny se nevezmou v úvahu. Další možnou nevýhodou je nestandardnost řešení oproti tomu, na co jsou programátoři zvyklí.

4.2.5 Hlavní okno

Hlavní okno aplikace dědí ze třídy `QMainWindow` a používá standardní rozvržení sestávající z menu, nástrojových lišt a stavového řádku. Jedná se o MDI aplikaci, centrální widget tvoří textové editory pro úpravu kódu, které se dají přepínat pomocí záložek.

Posledním výrazným prvkem jsou dokovatelná okna, která se dají libovolně přesouvat po celé ploše hlavního okna, skrývat a znovu zobrazovat a také kompletně vytrhávat ven do samostatných oken. Aplikace si pamatuje rozložení ovládacích prvků mezi jednotlivými spuštěními.



Obrázek 4.3: Okno grafické aplikace

4.2.6 Textový editor

Textový editor rozšiřuje standardní `QPlainTextEdit` a kvůli logickému oddělení částečně nesouvisejícího kódu používá dvě samostatné třídy: `TextEditorProgrammers`, z níž dále dědí `TextEditor`.

Programátorský textový editor poskytuje zvýrazňování syntaxe, číslování řádek, zvýraznění aktivní řádky, automatické odsazování zdrojového kódu a inteligentní klávesu *Home*. Od něj odvozená třída dále přidává vytváření, otevírání a ukládání souborů, hledání a nahrazování textu a další činnosti.

4.2.6.1 Zvýrazňování syntaxe

Pravidla a vlastní funkcionalitu zvýrazňování syntaxe definuje třída `TextEditorHighlighter`, jež vychází ze standardního `QSyntaxHighlighter` a je zaregistrována k zobrazovanému dokumentu. V jejím konstruktoru je definováno, jak se mají obarvovat klíčová slova, řetězce, čísla, komentáře a další elementy jazyka, tyto formáty jsou dále přiřazeny k regulárním výrazům, pomocí kterých se hledají jejich textové reprezentace v kódu skriptu.

Vlastní obarvování probíhá v metodě `highlightBlock()`, jež prochází řetězec s textem zvýrazňované řádky, v něm hledá text odpovídající specifikovanému regulárním výrazu a při nalezení aplikuje daný formát. Celé řešení vychází z oficiálního tutoriálu dodávanému ke knihovně Qt [5].

4.2.6.2 Zvýraznění aktuálního řádku

Změní-li se pozice textového kurzoru v dokumentu, editor emituje signál `cursorPositionChanged`. Tento signál je namapován na obslužný slot `highlightCurrentLine()`, jenž vytvoří objekt typu `QTextEdit::ExtraSelection`. Nastaví mu barvu pozadí a řekne, aby se roztáhl přes celou řádku, ve které se nachází kurzor. V podstatě se jedná o ekvivalentní zvýraznění, jaké probíhá při označení textu ke kopírování, editor jich podporuje libovolné množství.

4.2.6.3 Číslování řádek

Převážná část kódu pro přidání čísel řádek do levého okraje editoru se nachází ve třídě `TextEditorProgrammers`. Nejdříve je nutné vytvořit prázdné místo, do kterého se později vykreslí. K tomu slouží funkce `updateLineNumberAreaWidth()` volaná například při přidání nového řádku do dokumentu nebo při změně velikosti okna. Požadovaná hodnota se získá vynásobením šířky jednoho znaku s počtem znaků (číslic) v textové reprezentaci počtu řádků.

Zobrazená čísla řádků se musí měnit zároveň s posunem v dokumentu. Požadavek na rolování oznamuje editor signálem `updateRequest`, stačí ho namapovat na slot `updateLineNumberArea()`, ve kterém se pomocí standardní funkce `scroll()` posune viewport o požadovaný počet pixelů.

Vlastní vykreslení čísel řádků se nachází ve funkci `lineNumberAreaPaintEvent()`, jež si najde číslo prvního viditelného řádku a poté v cyklu kreslí číslice. Téměř celý kód byl převzat z oficiálního tutoriálu knihovny Qt [4].

4.2.6.4 Indikace přednastavené šířky řádku

Programátoři často zarovnávají své zdrojové kódy na 80 znaků, některé textové editory umožňují zobrazit vertikální linku, jež specifikuje tento pravý okraj dokumentu. Zdrojový kód pro její vykreslení je umístěn ve funkci `paintEvent()` textového editoru.

4.2.6.5 Automatické odsazování textu

Automatické odsazování textu je implementováno v metodě `autoIndent()` editoru, která se volá po každém stisku klávesy *Enter*. Funkce zkopíruje všechny bílé znaky (mezery, tabulátory) na začátku předchozího řádku do právě vytvořeného. Pokud navíc jako první ne-bílý znak nalezne složenou závorku, přidá ještě jeden tabulátor popř. odpovídající počet mezer.

4.2.6.6 Inteligentní klávesa Home

Nové chování klávesy *Home* má na starost metoda `homeKey()`. Ve standardním nastavení přesouvá editor při jejím stisku textový kurzor na začátek aktuálního řádku a v případě současného stisku klávesy *Shift* vybírá mezilehlý text do bloku.

Nová implementace umožňuje, aby se kurzor nepřesouval přímo na začátek řádku, ale až za poslední ne-bílý znak na začátku řádku, tedy za odsazení tabulátory. V případě, že se tam již nacházel, je přesunut na začátek a naopak. Vytvořený kód samozřejmě neporušuje chování klávesových zkratek *Ctrl+Home* a *Ctrl+Shift+Home*.

4.2.6.7 Vyhledávání a nahrazování textu

Editory z knihovny Qt neposkytují dialogy na vyhledávání ani nahrazování textu. V aplikaci je definují nově vytvořené třídy `DialogFind` a `DialogReplace`, které spolupracují s kódem editoru.

4.2.7 Textový výstup ze skriptu

V kapitole 4.2.1 na straně 35 byla představena třída `GuiLogger`, jejímž prostřednictvím skript vypisuje text do okna s výstupem. Aby mohl tento objekt pro mezivláknovou komunikaci využívat signály a sloty, musí být potomkem třídy `QObject` a dialog mezi vlákny musí probíhat podle schématu využívajícího mezilehlou frontu (angl. queued connection) [6].

`GuiLogger` poskytuje funkce pro předávání informačních, varovných a chybových zpráv a také pro standardní výstup ze skriptu. V jejich tělech emituje signály s textovými parametry, které jsou na druhém konci napojeny na obslužné sloty výstupního okna.

Panel je tvořen třídou `DockScriptOutput`, jež dědí od `QDockWidget` a zároveň zapouzdřuje `QTextBrowser`. Obslužné sloty převezmou text, podle jeho typu definují barvu a všechno předají metodě `append()`. Pozice ve zdrojových kódech se obalí HTML odkazem a vloží se spolu s obarveným textem na konec logu.

Jediným důvodem pro použití třídy `QTextBrowser` namísto `QPlainTextEdit` je interní podpora odkazů. Jméno souboru a číslo řádku, na kterém se vyskytla chyba, je v logu vždy klikací, daný soubor se otevírá v editoru a kurzor je přesunut na požadovaný řádek.

Toto samozřejmě není standardní chování `QTextBrowser`, třída musí být pomocí funkce `setOpenLinks()` požádána, aby odkazy automaticky neotevírala, ale pouze generovala signály o kliknutí. Předané URL má stejný formát, jaký byl zadán do `href` parametru odkazu, tedy jméno souboru a číslo řádku oddělené dvojtečkou. Obě části se od sebe oddělí a předají se jako parametry nově emitovaného signálu. Ten je poté již jednoduše napojen na `openAndScroll()` slot hlavního okna.

4.2.8 Panel proměnných skriptu a panel zásobníku volání

Panel s výpisem hodnot lokálních proměnných obsahuje dva sloupce, jméno proměnné a její hodnotu. Aktualizace informací probíhá po každém pozastavení skriptu, všechny potřebné informace dodává kontext a jeho zásobník volání. Proměnné se procházejí v cyklu a v textové formě se vkládají do seznamu položek okna.

Hodnoty prvků, jež jsou uložené ve strukturovaných proměnných (pole, množina, graf atd.), se nezobrazují, místo nich je předáván pouze datový typ. Vypisovat všechny položky stoprvkového pole nebo grafu s milionem vrcholů by bylo jednak nepřehledné a jednak neefektivně vytěžovalo procesor i paměť. Místo toho je ve skriptu možné vyžít služeb přiřazení reference, kdy se prvek strukturovaného datového typu namapuje na jméno obyčejné proměnné a s její pomocí se zobrazí požadovaná hodnota.

```
alias &= structure.array[57];
structure.array[57] = "a new value";
// Alias variable contains the new value too
```

Výpis zásobníku volání funguje téměř ekvivalentním způsobem jako zobrazování hodnot proměnných. Obsahuje tři sloupce, jméno funkce, jméno souboru a řádek, ze kterého byla funkce zavolána.

Po přidržení kurzoru myši nad jménem souboru se zobrazí *tool tip* ukazující kompletní cestu k souboru. Třída dále zpracovává signál o dvojkliku myši, na jeho reakci otevírá soubor v editoru a roluje na specifikovaný řádek.

4.2.9 Vizualizace grafu

4.2.9.1 Propojení skriptu a vizualizací

Skript komunikuje s vizualizačním oknem výhradně pomocí zabudovaných funkcí, jejich jména začínají předponou *vis-*. Podobně jako logování z kapitoly 4.2.1 na straně 35 i tato funkcionalita musí být zobecněna, aby se dala používat s grafickou aplikací a zároveň negenerovala chyby o neexistujících funkcích při běhu z konzole.

Rodičovská třída `VisualizationConnector` definuje potřebné rozhraní, současně implementuje těla všech svých metod jako prázdná. Nejedná se tedy o čistě abstraktní třídu jako v minulých případech. Zabudované funkce pro vizualizace komunikují s GUI výhradně s jejím prostřednictvím, resp. jejího potomka. Pokus o přístup k oknu z konzolové aplikace proto nezpůsobí žádnou chybu, pouze se zavolá prázdná metoda komunikačního objektu, a to ničemu nevadí.

Odvozený `GuiVisualizationConnector` je o něco zajímavější. Všechny zabudované vizualizační funkce mají obdobu v některé z metod konektoru, ten dále v jejich těle generuje Qt signály, které jsou napojeny na obslužné sloty okna. Opačný směr komunikace probíhá pomocí obyčejného volání metod – vizualizační okno nepotřebuje žádný zobecňovací mechanismus, ví, že konektor vždy existuje.

4.2.9.2 Registrace objektů a rendering grafu

Chce-li programátor skriptu zaregistrovat graf pro zobrazení ve vizualizačním okně, zavolá zabudovanou funkci `visRegister()`. Ta se přes metodu a signál konektoru spojí se slotem widgetu, jenž si uloží předaný objekt pro další použití.

Aplikace v této části rozlišuje množiny a celé grafy. Současně s objektem se předává i řetězec se jménem pro menu, s jehož pomocí může uživatel vypínat a opětovně zapínat zobrazování.

Vlastní rendering používá grafickou knihovnu OpenGL. Vrcholy se vykreslují jako `GL_POINTS` body ve 3D prostoru, jež jsou pospojovány `GL_LINES` čárami reprezentujícími hrany. Souřadnicové koordináty se přebírají z vlastností vrcholů pojmenovaných $\_\_x$, $\_\_y$ a $\_\_z$, jsou tedy implicitně přístupné i kódu skriptu, jenž je může obyčejným přiřazením kdykoli změnit. U hran je situace obdobná, při renderingu se ovšem používá pozice počátečního a koncového vrcholu.

Nutno podotknout, že není úkolem vizualizací ani skriptu souřadnicové koordináty nějakým způsobem vytvářet. Nestarají se, jak se jejich hodnoty dostaly do vrcholů, jednoduše předpokládají, že tam jsou, a používají je. O jejich vytvoření se musí postarat tvůrce poskytovaného grafu – ať už specifikací v datovém souboru (viz příloha B na straně 57) či přímo v kódu skriptu.

Další vlastnosti, které mohou ovlivnit rendering se jmenují $\_\_r$, $\_\_g$, $\_\_b$ a $\_\_w$. První tři z nich definují barvu ve formátu RGB, při jejím nalezení se pro tento konkrétní vrchol (hranu) nepoužije obecná barva množiny, se kterou byl registrován, ale tato konkrétní.

Čtvrtý z parametrů umožňuje dodatečné vychýlení hrany ve směru souřadnicové osy z. Je-li definován a zároveň je ovlivňování pozic povoleno (zabudovaná funkce `visUseWeightWhenPaintingEdges()`), bere se při výpočtu pozice hrany v úvahu i její ohodnocení.

Představená funkcionalita souvisí s požadavkem vedoucího práce. Jde o to, že při vizualizaci speciálních grafů a při studiu difúzních algoritmů je výhodné graf zobrazovat tak, že se vrcholy zakreslí do roviny a hrany se zobrazí jako úsečky vedené paralelně *nad* touto rovinou, přičemž výška hrany nad rovinou odpovídá její váze.

4.2.9.3 Změna pohledu kamery na scénu

Rendering používá celkem pět parametrů pro definici kamery. Jsou jimi x, y a z pozice spolu s x a y rotacemi. Kamera se při vykreslování nastavuje následovně.

```
glLoadIdentity();
glTranslatef(xpos, ypos, zpos - 10.0f);
glRotatef(xrot, 1.0f, 0.0f, 0.0f);
glRotatef(yrot, 0.0f, 1.0f, 0.0f);
```

Uživatel má možnost stiskem levého tlačítka myši ve vizualizačním okně a táhnutím ovlivnit x a y pozici, k posunutí na ose z slouží kolečko. Táhnutí se stisknutým pravým tlačítkem ovlivňuje rotace, *Ctrl* a *Shift* modifikátory omezují natáčení pouze na horizontální resp. vertikální osu.

Zmíněné parametry se sdružují do třídy `VisualizationView`. Aplikace si udržuje kontejner těchto objektů, ukládání a pozdější aktivaci uloženého pohledu zpřístupňuje pomocí položek menu.

4.2.9.4 Screenshot vizualizačního okna

Díky knihovně Qt je kód funkce velice jednoduchý, zabírá pouze dva řádky. Funkce `grabWindow()` třídy `QPixmap` sejme obsah vizualizačního okna a `save()` uloží data obrázku na disk. Alternativou by mohlo být OpenGL volání `glReadPixels()`, obě metody produkují stejné výsledky. Screenshot je rovněž zpřístupněn skriptu ve formě zabudované funkce.

4.2.10 Konfigurace grafické aplikace

Grafická aplikace, na rozdíl od konzolové, nepřebírá nastavení z předaných argumentů, ale jednotlivé volby získává z konfiguračního souboru. Databáze parametrů se uchovává v singletonu `Settings`. Hodnoty jsou načteny jednorázově při konstrukci objektu, pozdější přístup poskytují odpovídající `set-` a `get-` metody. Zápis do souboru se provádí při každé změně, ne až při ukončení aplikace. Kód interně využívá služeb třídy `QSettings`.

Kapitola 5

Testování

5.1 Unit testy

Aplikace nepoužívá žádný standardní framework pro automatické testování, místo toho přichází s vlastním kódem zapouzdřeným ve třídě `Tests`. Ke spuštění jednotkových testů slouží metoda `run()`, která volá jednotlivé testovací rutiny.

```
void Tests::run(void)
{
    uint failed = 0;

    failed += !testDoubleDispatching();
    // ...

    cout << _("Number of failed tests: ") << failed << endl;
}
```

Z předchozího kódu je vidět, že jediným požadavkem na testovací funkci je její návratová hodnota. Funkce pomocí závěrečného `testResult()` vypisuje jméno testu (své jméno) a výsledek *OK* či *FAILED*.

```
bool Tests::testDoubleDispatching(void)
{
    bool result = true;

    CountPtr<Value> a(new ValueInt(5));
    CountPtr<Value> b(new ValueFloat(3.4f));
    CountPtr<Value> c(new ValueFloat(2.0f));

    verify(a->add(*b)->div(*c)->toString() == "4.2");// (5 + 3.4) / 2
    verify(a->mult(*c)->sub(*b)->toString() == "6.6");// (5 * 2) - 3.4

    return testResult(__FUNCTION__, result);
}
```

Pozorný čtenář si jistě všiml slůvka `verify()`, jímž jsou obaleny testovací podmínky. Nejedná se o funkci, ale o makro, které vyhodnotí kód v parametru, aktualizuje stavovou proměnnou `result` a v případě chyby vypíše jméno souboru s číslem řádku, kde k ní došlo. Dále je zobrazena i textová reprezentace testovacího kódu. V podstatě se jedná o alternativu ke standardnímu makru `assert()`, která však, v případě neúspěchu, nemá za následek ukončení programu.

```
#define verify(expr)                                                    \
{                                                                       \
    bool cur_result = (expr);                                           \
    result = result && cur_result;                                      \
    if(!cur_result)                                                     \
        cerr << __FILE__ << ":" << __LINE__ << "   " << #expr << endl; \
}
```

Změníme-li pokusně hodnotu proměnné *a* (jednoduchá emulace chyby), vypadá výstup z provedených testů následovně.

```
[ OK ]     testMemoryLeaks
tests.cpp:204   a->add(*b)->div(*c)->toString() == "4.2"
tests.cpp:205   a->mult(*c)->sub(*b)->toString() == "6.6"
[ FAILED ] testDoubleDispatching
[ OK ]     testValueStruct
[ OK ]     testValueString
...
Number of failed tests: 1
Number of memory leaks: 0
```

Ke spuštění všech testovacích sekvencí slouží u konzolové aplikace dodatečný argument příkazové řádky `--unit-tests`. Následující příkaz současně testuje i úniky dynamické paměti, a to nejen nástrojem *valgrind*, ale i vlastním kódem - viz kapitola 4.1.3.2 na straně 25.

```
valgrind ./bbd --unit-tests ../samples/tests.txt
```

Je nutné podotknout, že vytvořené unit testy poskytly neocenitelné služby při vývoji aplikace a především při testování změn ve fungování jednotlivých částí programu. Dokázaly velice rychle a efektivně detekovat místa v kódu, která nepracovala tak, jak by měla – mnohdy i v na první pohled nesouvisejících částech aplikace.

5.2 Unit testy ve skriptu

Obdobným způsobem, jakým jsou implementovány jednotkové testy z předchozí kapitoly, funguje i testovací skript *samples/tests.txt*. Jedná se v podstatě o unit test zapsaný v kódu skriptu a vykonávaný interpretem. Jeho obrovskou výhodou je, že se netestují pouze umělé případy navržené programátorem, ale přímo reálné interakce mezi všemi objekty interpretu.

Skript definuje testovací makro `tverify()` (jméno `verify()` se již používá pro zabudovanou funkci), které se opět rozbaluje na aktualizaci stavové proměnné. Jazyk nepodporuje výpis

předaného kódu, a proto si musí programátor vystačit pouze se jménem souboru a číslem řádku, na kterém nastala chyba. Tato informace je však plně dostatečná, zvlášť ve spojení s grafickým prostředím, kde se po kliknutí otevře místo chyby v editoru.

```
define("tverify", "result = result && verify");
```

Jednoduchá testovací funkce by mohla vypadat například následovně.

```
function testFactorial()
{
    result = true;

    tverify(factorial(1) == 1);
    tverify(factorial(2) == 2);
    tverify(factorial(3) == 6);

    return testResult(__FUNCTION__, result);
}
```

Pouze pro zajímavost, takto vypadá kód, který vznikne v lexikálním analyzátoru po rozbalení makra `tverify()` a dodání jména funkce.

```
function testFactorial()
{
    result = true;

    result = result && verify(factorial(1) == 1);
    result = result && verify(factorial(2) == 2);
    result = result && verify(factorial(3) == 6);

    return testResult("testFactorial", result);
}
```

5.3 Rychlost vykonávání skriptu, srovnávací testy

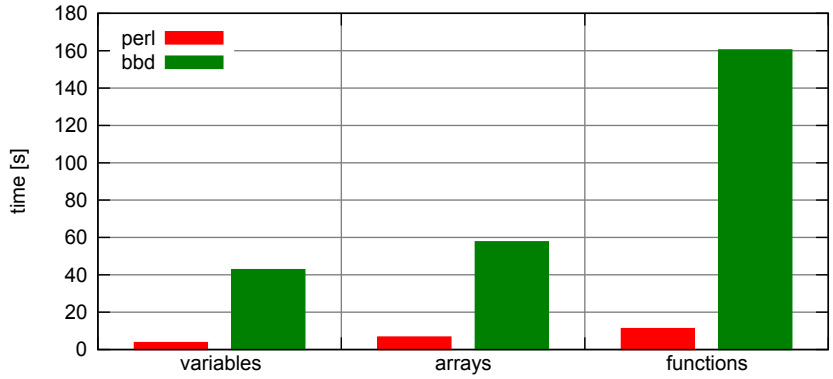
Po vytvoření nového kompilátoru či interpretu nějakého programovacího jazyka je jistě důležité znát, jak rychle dokáže vykonávat napsané programy. Byla navržena sada tří jednoduchých testů, které odpovídají na tuto otázku.

Všechny z nich obsahují cyklus o deseti milionech kroků. V prvním se testuje provádění jednoduchých aritmetických operací a přístup k proměnným (`tmp = i * 2;`), druhý rozšiřuje funkcionalitu o indexování pole (`a[i % SIZE] = i * 2;`) a třetí navíc přichází s voláním funkcí (`tmp = func(i);`).

Tyto jednoduché testy byly dále přepsány do jazyků C++ a Perl, aby se mohlo s něčím porovnávat. Zkompilované C++ programy jsou podle očekávání bezkonkurenčně nejrychlejší, doba běhu je neměřitelně malá. Při srovnání naměřených hodnot z tabulky 5.1 a následně vynesených do grafu 5.1 vyplývá, že vytvořený interpret je přibližně osmkrát až patnáctkrát

typ úlohy	perl [s]	bbd [s]	poměr [-]
variables	3.78	42.91	11.3
arrays	6.91	57.11	8.3
functions	10.48	160.38	15.2

Tabulka 5.1: Rychlost vykonávání ve srovnání s jazykem Perl



Obrázek 5.1: Rychlost vykonávání ve srovnání s jazykem Perl

pomalejší než taktéž interpretovaný jazyk Perl. Výsledky byly v podstatě očekávatelné, interpret se v dané verzi nesnaží o naprosto žádné optimalizace, a tudíž se není čemu divit.

Čtenáře možná překvapí výrazně pomalejší běh skriptu při volání funkcí než při jiných operacích. Kapitola 4.1.5.1 na straně 30 se věnuje implementaci klíčového slova `return` pomocí výjimek a jak ukazuje graf 5.2, právě ony ho způsobují.

Obrázek 5.2: Režie výjimek při návratu z funkce

Cílem těchto testů bylo především získání orientačních hodnot. Interpret byl zkompilován s optimalizačními volbami `-O2 -s -DNDEBUG` a vypnutými testy úniku paměti (viz kapitola 4.1.3.2 na straně 25). Neoptimalizovaná verze je cca. dvakrát pomalejší. Měření byla spuštěna na nezatíženém počítači několikrát, vždy s obdobnými výsledky.

Kapitola 6

Závěr

V této práci byl navržen programovací jazyk pro psaní grafových algoritmů a vytvořen jeho interpret včetně komplexního vývojového prostředí. Jako implementační platforma byl pro základní kód zvolen jazyk C++, grafická verze aplikace je napsána s pomocí knihovny Qt a vizualizace grafů dále využívají služeb OpenGL. Použité knihovny jsou multiplatformní, a tudíž je zaručena velice snadná přenositelnost mezi nejrůznějšími operačními systémy.

Všechny požadavky vyplývající ze zadání byly splněny, v mnoha případech i v nevyžadované rozšířené formě. Příkladem budiž volitelná implementace orientovaných grafů, požadavek na krokování skriptu či schopnosti textového editoru pro psaní kódu. Aplikace ve výsledku umí pracovat s naprosto všemi typy grafů, jednoduché krokování skriptu se vyvinulo v obecný debugger a editor poskytuje nejen zvýrazňování syntaxe, ale i automatické odsazování kódu, číslování řádků a další vymoženosti běžných programátorských editorů.

Jak ukázalo závěrečné testování, interpret má bohužel výrazné nedostatky v rychlosti vykonávání kódu. Při porovnání s ekvivalentními programy zapsanými v jazyce Perl vychází přibližně osmi až patnáctinásobně pomalejší běh. Je pravda, že se vyvinutý interpret v současné verzi nesnaží o naprosto žádné optimalizace, zde se zcela jistě nachází prostor pro další zlepšování.

Budoucnost aplikace a schopnosti nově vytvořeného jazyka leží především na rozšíření o nové zabudované funkce, které zatím poskytují pouze velice málo možností k praktickému nasazení například pro skriptování na úrovni operačního systému. Grafická část spolu s výkonnými vizualizacemi grafů bude zcela jistě využívána při zkoumání difuzních algoritmů, o nichž byla řeč na samém úvodu práce. Je jasné, že během této činnosti určitě vznikne spousta nových požadavků na nejrůznější rozšíření a zlepšení.

Literatura

[1] Degener, J.; Lee, J.: ANSI C Yacc grammar. 2008, online, rev. 01.11.2008, cit. 24.10.2009.
http://www.quut.com/c/ANSI-C-grammar-y.html

[2] Eckel, B.: *Thinking In C++; Volume 2: Practical Programming*. Upper Saddle River, NJ 07458: Pearson Prentice Hall, druhé vydání, 2004, ISBN 0-13-035313-2, online, cit. 07.05.2009.
http://mindview.net/Books/TICPP/ThinkingInCPP2e.html

[3] Henricson, M.; Nyquist, E.: *Industrial Strength C++*, kapitola 9. Stockholm: Prentice-Hall PTR, první vydání, 1997, ISBN 0-13-120965-5, s. 95–102, online, cit. 07.06.2009.
http://www.sannabremo.se/nyquist/industrial/

[4] Nokia Corporation: Code Editor Example. 2009, online, cit. 28.10.2009.
http://doc.trolltech.com/4.5/widgets-codeeditor.html

[5] Nokia Corporation: Syntax Hightlighter Example. 2009, online, cit. 28.10.2009.
http://doc.trolltech.com/4.5/richtext-syntaxhighlighter.html

[6] Nokia Corporation: Thread Support in Qt. 2009, online, cit. 28.10.2009.
http://doc.trolltech.com/4.5/threads.html

[7] Patočka, M.; Pergel, M.; Kulhavý, P.; aj.: Links – Webový prohlížeč, Vývojová dokumentace. Technická zpráva, Universita Karlova v Praze, Matematicko-fysikální fakulta, 2002, online, cit. 07.05.2009.
http://links.twibright.com/doc/vyvojova.pdf

[8] Sharon, Y.: Smart Pointers - What, Why, Which? 1999, online, cit. 07.06.2009.
http://ootips.org/yonat/4dev/smart-pointers.html

[9] Turek, M.: Borsch Interpreter. 2007, online, cit. 07.05.2009.
http://woq.nipax.cz/borsch/

[10] Werner, T.: A Linear Programming Approach to Max-sum Problem: A Review. IEEE Trans. on Pattern Recognition and Machine Intelligence (PAMI). 2007, online, cit. 05.12.2009.
http://cmp.felk.cvut.cz/~werner/

[11] Wikipedia, the free encyclopedia: Belief propagation. 2009, online, cit. 05.12.2009.
http://en.wikipedia.org/wiki/Belief_propagation

Dodatek A

Gramatika jazyka

Při vytváření níže uvedené gramatiky se vycházelo z předpisu pro jazyk Ansi C [1]. Kromě těchto pravidel poskytuje jazyk také konstrukce `define` a `include`, ty se však zpracovávají na úrovni lexikálního analyzátoru, viz kapitola 3.2.5 na straně 8.

```
primary_expression
    : LEX_NULL
    | LEX_TRUE
    | LEX_FALSE
    | LEX_INT
    | LEX_IDENTIFIER
    | LEX_FLOAT
    | LEX_STRING
    | '(' expression ')'
    ;

postfix_expression
    : primary_expression
    | postfix_expression '[' expression ']'
    | LEX_IDENTIFIER '(' ')'
    | LEX_IDENTIFIER '(' argument_expression_list ')'
    | postfix_expression '.' LEX_IDENTIFIER
    | postfix_expression '.' LEX_IDENTIFIER '(' ')'
    | postfix_expression '.' LEX_IDENTIFIER '(' argument_expression_list ')'
    | postfix_expression LEX_INC_OP
    | postfix_expression LEX_DEC_OP
    ;

argument_expression_list
    : assignment_expression
    | argument_expression_list ',' assignment_expression
    ;

unary_expression
    : postfix_expression
```

```
    | LEX_INC_OP unary_expression
    | LEX_DEC_OP unary_expression
    | '+' unary_expression
    | '-' unary_expression
    | '!' unary_expression
    ;

multiplicative_expression
    : unary_expression
    | multiplicative_expression '*' unary_expression
    | multiplicative_expression '/' unary_expression
    | multiplicative_expression '%' unary_expression
    ;

additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;

relational_expression
    : additive_expression
    | relational_expression '<' additive_expression
    | relational_expression '>' additive_expression
    | relational_expression LEX_LE_OP additive_expression
    | relational_expression LEX_GE_OP additive_expression
    ;

equality_expression
    : relational_expression
    | equality_expression LEX_EQ_OP relational_expression
    | equality_expression LEX_NE_OP relational_expression
    ;

logical_and_expression
    : equality_expression
    | logical_and_expression LEX_AND_OP equality_expression
    ;

logical_or_expression
    : logical_and_expression
    | logical_or_expression LEX_OR_OP logical_and_expression
    ;

conditional_expression
    : logical_or_expression
    | logical_or_expression '?' expression ':' conditional_expression
    ;
```

```
assignment_expression
    : conditional_expression
    | unary_expression '=' assignment_expression
    | unary_expression LEX_MUL_ASSIGN assignment_expression
    | unary_expression LEX_DIV_ASSIGN assignment_expression
    | unary_expression LEX_MOD_ASSIGN assignment_expression
    | unary_expression LEX_ADD_ASSIGN assignment_expression
    | unary_expression LEX_SUB_ASSIGN assignment_expression
    | unary_expression LEX_REF_ASSIGN assignment_expression
    ;

expression
    : assignment_expression
    ;

statement
    : compound_statement
    | expression_statement
    /* "parser.y: conflicts: 1 shift/reduce" is ok */
    | LEX_IF '(' expression ')' statement
    | LEX_IF '(' expression ')' statement LEX_ELSE statement
    | LEX_WHILE '(' expression ')' statement
    | LEX_FOR '(' expression_statement expression_statement ')' statement
    | LEX_FOR '(' expression_statement expression_statement
        expression ')' statement
    | LEX_FOREACH '(' LEX_IDENTIFIER ';' expression ')' statement
    | LEX_BREAK ';'
    | LEX_CONTINUE ';'
    | LEX_RETURN ';'
    | LEX_RETURN expression ';'
    | LEX_GLOBAL LEX_IDENTIFIER ';'
    ;

expression_statement
    : ';'
    | expression ';'
    ;

compound_statement
    : '{' '}'
    | '{' block_item_list '}'
    ;

block_item_list
    : statement
    | block_item_list statement
    ;
```

```
function_definition
    : function_and_name '(' parameter_list ')' compound_statement
    | function_and_name '(' ')' compound_statement
    ;

function_and_name
    : LEX_FUNCTION LEX_IDENTIFIER
    ;

parameter_list
    : LEX_IDENTIFIER
    | parameter_list ',' LEX_IDENTIFIER
    ;

start
    : /* empty */
    | start function_definition
    ;
```

Dodatek B

Formát datových souborů s grafem

Soubory ukládající grafy používají strukturu uvedenou níže. Jedná se o textový formát, oddělovačem hodnot může být libovolný počet bílých znaků (mezera, tabulátor, nový řádek).

```
is_directed

number_of_vertices   number_of_edges

number_of_properties_for_vertices   names_of_properties_for_vertices
number_of_properties_for_edges      names_of_properties_for_edges

vertex_id   values_of_properties
vertex_id   values_of_properties
vertex_id   values_of_properties
...

vertex_id   vertex_id   values_of_properties
vertex_id   vertex_id   values_of_properties
vertex_id   vertex_id   values_of_properties
...
```

Jako příklad následuje kompletní výpis definice jednoduchého grafu, který se ve vizualizačním okně zobrazí jako domeček ze známého úkolu *nakreslit jedním tahem*.

```
1

5 8

3    __x __y __z
1    __w

0    -1  1 0
1     1  1 0
2     1 -1 0
3    -1 -1 0
```

```
4     0  2 0

3 0    0.0
0 1    0.0
1 3    0.0
3 2    0.0
2 0    0.0
0 4    0.0
4 1    0.0
1 2    0.0
```

Dodatek C

Vytvoření nové zabudované funkce

Při vytváření nové zabudované funkce, je dobré postupovat podle následujících kroků:

1. V souboru `../samples/tests.txt` vytvořit unit test (viz kapitola 5.1 na straně 45).
2. V souboru `gen_builtin.pl`:
   (a) Napsat `$funcdecl` deklaraci/dokumentaci funkce.
   (b) Definovat kód pro vložení hlavičkových souborů.
   (c) Definovat tělo zabudované funkce.
   (d) Zavolat funkci `genBFClass()`.
3. Přegenerovat zdrojové kódy pomocí `make builtin` nebo `make rebuild`.
4. Přidat nové `.hpp` a `.cpp` soubory do projektů.
5. Zkompilovat aplikaci.
6. Otestovat kód pomocí `valgrind ./bbd --unit-tests ../samples/tests.txt`.

Jako příklad následuje kompletní výpis kódu zabudované funkce `getAdjacencyMatrix()`, která programátorovi skriptu zpřístupňuje matici sousednosti grafu.

```
$funcdecl = 'getAdjacencyMatrix(graph) : array';

$include = <<END_OF_CODE;
#include "valuegraph.hpp"
END_OF_CODE

$code = <<END_OF_CODE;
    ValueGraph* g = NULL;

    if((g = par[0]->toValueGraph()) != NULL)
        return g->getAdjacencyMatrix();
    else
    {
```

```
        WARN_P(_("Bad parameters type: $funcdecl"));
        return VALUENULL;
    }
END_OF_CODE

genBFClass('getAdjacencyMatrix', 'NodeBuiltinGetAdjacencyMatrix', 1,
    $code, $include);
```

Při dodržení výše uvedeného postupu stačí pouze vytvořit tento zdrojový kód, přegenerovat zabudované funkce, přidat novou třídu do projektových souborů a začít nově vytvořenou funkci používat.

Dodatek D

Hierarchie tříd

```
CountPtr< TYPE >
ObjectCreator
BaseObject
.   CallStackItem
.   CodePosition
.   Context
.   .   GuiContext (+ QObject)
.   ExitValue
.   Lexan
.   LexanIterator
.   .   LexanIteratorFile
.   .   LexanIteratorString
.   Logger
.   .   CliLogger
.   .   GuiLogger (+ QObject)
.   Node
.   .   NodeBinary
.   .   .   NodeBinaryAdd
.   .   .   NodeBinaryAnd
.   .   .   NodeBinaryAss
.   .   .   NodeBinaryAssAdd
.   .   .   NodeBinaryAssDiv
.   .   .   NodeBinaryAssMod
.   .   .   NodeBinaryAssMult
.   .   .   NodeBinaryAssRef
.   .   .   NodeBinaryAssSub
.   .   .   NodeBinaryDiv
.   .   .   NodeBinaryEq
.   .   .   NodeBinaryGe
.   .   .   NodeBinaryGt
.   .   .   NodeBinaryIndex
.   .   .   NodeBinaryLe
.   .   .   NodeBinaryLt
.   .   .   NodeBinaryMember
.   .   .   NodeBinaryMod
.   .   .   NodeBinaryMult
.   .   .   NodeBinaryNe
.   .   .   NodeBinaryOr
.   .   .   NodeBinarySub
.   .   NodeBlock
.   .   NodeCondition
.   .   NodeEmptyCommand
.   .   NodeForeach
.   .   NodeFunction
.   .   .   NodeFunctionBuiltin
.   .   .   .   NodeBuiltinArray
.   .   .   .   NodeBuiltinAssert
.   .   .   .   etc.
.   .   .   NodeFunctionScript
.   .   NodeFunctionCall
.   .   NodeGlobal
.   .   NodeJumpBreak
.   .   NodeJumpContinue
.   .   NodeLoop
.   .   NodePosition
.   .   NodeUnary
.   .   .   NodeUnaryDecPost
.   .   .   NodeUnaryDecPre
.   .   .   NodeUnaryIncPost
.   .   .   NodeUnaryIncPre
.   .   .   NodeUnaryNot
.   .   .   NodeUnaryReturn
.   .   .   NodeUnarySub
.   .   NodeValue
.   ObjectFactory
.   .   CliFactory
.   .   GuiFactory
.   StringTable
.   Tests
.   Value
.   .   ValueArray
.   .   ValueBool
.   .   ValueEdge
.   .   ValueFloat
```

```
.   .   ValueGraph
.   .   ValueIdentifier
.   .   ValueInt
.   .   ValueNull
.   .   ValueReference
.   .   ValueSet
.   .   ValueString
.   .   ValueStruct
.   .   ValueVertex
.   VisualizationConnector
.   .   GuiVisualizationConnector (+ QObject)
MutexLocker
Mutex
.   CliMutex
.   GuiMutex
Settings
QObject
.   VisualizationItemData
.   VisualizationView
.   Qt classes
.   .   QMainWindow
.   .   .   MainWindow
.   .   QDockWidget
.   .   .   DockCallStack
.   .   .   DockFiles
.   .   .   DockScriptOutput
.   .   .   DockVariables
.   .   .   DockVisualization
.   .   QDialog
.   .   .   DialogFind
.   .   .   DialogIncludeDirs
.   .   .   DialogReplace
.   .   .   DialogReplaceConfirmation
.   .   .   DialogScriptParameters
.   .   .   DialogSettings
.   .   .   DialogSettingsEditor
.   .   .   DialogSettingsVisualization
.   .   QPlainTextEdit
.   .   .   TextEditorProgrammers
.   .   .   .   TextEditor
.   .   QWidget
.   .   .   TextEditorLines
.   .   QSyntaxHighlighter
.   .   .   TextEditorHighlighter
.   .   QGLWidget
.   .   .   Visualization
.   .   QThread
.   .   .   ScriptThread
```

Dodatek E

Seznam použitých zkratek

AST	Abstract Syntax Tree
CLI	Command Line Interface
GNU	GNU's Not Unix!
GUI	Graphics User Interface
HTML	Hyper Text Markup Language
I/O	Input/Output
LALR	Look Ahead Left to Right (Grammar)
MDI	Multiple Document Interface
OOP	Object Oriented Programming
OpenGL	Open Graphics Library
PDF	Portable Document Format
PNG	Portable Network Graphics
RGB	Red Green Blue
RTTI	Run-Time Type Information
STL	Standard Template Library
SVG	Scalable Vector Graphics
SVN	Subversion
UML	Unified Modeling Language
URL	Uniform Resource Locator
XML	eXtensible Markup Language

⋮

Dodatek F

Obsah přiloženého CD

```
cd
|-- bbd
|-- bbdgui
|-- samples
|-- graphs
|-- man
|   |-- html
|   |-- php
|-- doc
|-- benchmarks
|-- text
|-- presentation
|-- svn
|-- svnstat
```

Adresáře `bbd` a `bbdgui` obsahují kompletní zdrojové kódy konzolové a grafické aplikace. Projektové soubory zahrnují vývojová prostředí Qt Creator, Code::Blocks a Makefile. Adresáře `samples` a `graphs` ukládají ukázkové skripty ve vytvořeném programovacím jazyce a několik datových souborů s testovacími grafy. Kompilace a spuštění jsou plně popsány v uživatelském manuálu, který se nachází v adresáři `man`. Vývojová dokumentace, včetně základních UML diagramů, generovaná ze zdrojových kódů nástrojem Doxygen je uložena v adresáři `doc`.

V `benchmarks` jsou k nalezení všechny prostředky sloužící k měření výkonu interpretu z kapitoly 5.3 na straně 47. Jedná se o zdrojové kódy testovacích sestav pro vyvinutý interpret, Perl a C++, dále spouštěcí skripty, formátování logů a konfigurační soubory pro Gnuplot s výstupem ve formátu PNG a SVG, včetně následné konverze SVG na PDF. Vše je plně automatizováno, stačí nainstalovat potřebné nástroje a spustit příkaz `make run`.

Ve složce `text` je uložen kompletní text této diplomové práce ve formátu PDF a zdrojové soubory pro typografický systém LaTeX. Prezentace diplomové práce se nachází v `presentation`, jedná se o formát PDF a zdrojové kódy napsané v LaTeX Beamer.

Adresář `svn` obsahuje dump SVN repositáře sloužícího pro vývoj a `svnstat` statistiky o tomto repositáři vygenerované nástrojem StatSVN.