Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Programovací jazyk pro optické výpočty

Plzeň 2014 Martin Hrach

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 22. června 2014

Martin Hrach

Abstract

This work aims to create user interface for program Rayleigh. Main task is to make this program accessible to nonprogrammers. Now this program is only nonexecutable library. User interface will be availible via scripting interface.

Abstrakt

Cílem práce je vytvořit uživatelské rozhraní k programu Rayleigh a zpřístupnit tento program neprogramátorům. Program nyní existuje pouze jako knihovna, která není přímo spustitelná. Uživatelské rozhraní bude formou skriptového rozhraní.

Poděkování

Rád bych poděkoval ing. Petru Lobazovi, vedoucímu mé diplomové práce, za jeho pomoc při realizaci této práce.

Obsah

1 Úvod

Práce je zaměřena na zpřístupnění optické knihovny neprogramátorům. Optická knihovna Rayleigh, vytvořena ing. Petrem Lobazem, neměla žádné uživatelské rozhraní a byla poskytnuta pouze jako dynamická knihovna sloužící k simulaci vlnové optiky. Knihovna mohla být připojena k programu napsaném v nízkoúrovňovém jazyce, jako je C++. Neprogramátoři tedy neměli možnost tuto knihovnu využít.

Tato práce ovšem rozšíří program Rayleigh právě k využití neprogramátorům. Výsledkem bude provázání knihovny se skriptovacím rozhraním, ve kterém neprogramátoři napíší formou skriptu, co požadují, a skript předají programu, kde bude vykonán.

Jazyk, ve kterém budou skripty psány, by měl být jednoduchý, ale zároveň by mělo být umožněno ovládat každý prvek programu. Také by jazyk měl být schopen popsat složitější scénu méně kroky (například cykly pro zadání vstupních dat). Jelikož počítání optických simulací je časově náročné, jazyk může sloužit i k napsání skriptů, které budou postupně provádět několik různých výpočtů a bude možné celý skript spustit později (například nechat počítat několik optických simulací přes noc).

Syntakticky je jazyk podobný jazyku C, ale zjednodušený (například se uživatel nemusí starat o paměťové záležitosti jako uvolňování či pointery). Jazyk umí pracovat s základními datovými typy (čísla nebo řetězce] reprezentovány proměnnými a také složitější struktury pro optické simulaci. Připravené proměnné (základní nebo optická) lze poté využít ve funkcích zajišťující požadované činnosti. Vlastní optické simulace nad popsanými daty budou prováděny knihovnou Rayleigh podle vstupního skriptu.

2 Výchozí stav

Cílem práce je vytvoření uživatelského rozhraní, které bude provázáno s knihovnou Rayleigh. Tato knihovna slouží k simulaci vlnové optiky, zejména se zaměřením na počítačem generovanou holografii.

Ještě před tím, než podrobně prozkoumáme knihovnu a budeme se věnovat možnostem uživatelského rozhraní, je nutné si něco povědět o problematice, kterou knihovna řeší, tedy o fourierovské optice.

2.1 Optika

Optika je částí fyziky, která se zabývá světlem. Světlo lze chápat jako elektromagnetické vlnění, které se šíří od svého zdroje.

V některých případech lze k popisu světla využít paprskové optiky (poměrně jednoduchá představa známá ze základní školy). Ale existují i jevy, které nelze popsat paprskovou optikou. Proto je nutné dívat se na světlo jako na vlnu.

2.1.1 Vlnová optika

Vlnová optika[5, s. 15] pohlíží na světlo jako na elektromagnetické pole, které má podobné chování jako mechanické vlnění. V tomto pohledu je tedy světlo elektromagnetické vlnění.

Nejprve je vhodné zmínit se o šíření vlnění. Šíření vlnění lze popsat pomocí vlnoploch. Jednu vlnoplochu si lze představit jako plochu složenou z takových míst, kam světlo dorazí od zdroje za stanovený čas. V homogenním (stejnorodém) prostředí se světlo šíří ve všech směrech stejně rychle a bodový zdroj světla tak vytváří kulové vlnoplochy. U jednoduchých vlnoploch se lze vrátit k paprskové optice a paprsky lze považovat za kolmice k vlnoplochám (toto zjednodušení lze ovšem uplatnit jen v jednoduchých situacích, protože ne vždy lze určit kolmice k vlnoplochám). Navíc v případě, že se v prostoru nacházejí vlny z více zdrojů, jednotlivé vlny mezi sebou reagují (interferují) a již nelze definovat paprsky vůbec, a proto se využívá pouze vlnové představy.

Nyní je nutné nějak popsat samotné vlnění. Přistoupíme-li k představě (sice nesprávné, ale názorné), že prostor, kde se vlnění šíří, je složen z jednotlivých částic vzájemně propojených silami, je nutné sledovat výchylku každé částice v prostoru. K popisu výchylky lze využít funkce $\vec{u}(\vec{r},t) = \vec{u}(x,y,z,t)$. Tato funkce $\vec{u}$ udává výchylku v daném místě prostoru a v daném čase. V obecném případě je výchylkou vektor, tedy jednotlivé částice mohou být odchýleny od své polohy v jakémkoli směru.

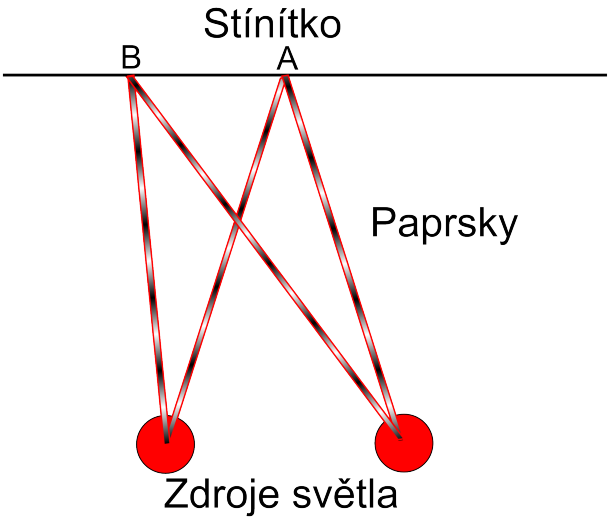
Pokud se vzdáme informace o směru výchylky (v případě světla tedy polarizace světla) a bude nás zajímat pouze její velikost, lze vektorovou veličinu redukovat na skalár a funkci tak zjednodušit na $u(\vec{r},t)$. Tímto jsme dosáhli skalární aproximace vlnění. Tato aproximace ztrácí některé informace o vlnění, ale pro většinu aplikací studujících šíření světla volným prostorem je postačující. Právě tato aproximace je též užita v knihovně Rayleigh.

U světla je nutné definovat nějaké optické veličiny. Pro vlnovou optiku jsou nejdůležitější veličiny: amplituda A, fáze φ a vlnová délka λ. Amplituda udává rozkmit vlnění a kvadrát amplitudy udává intenzitu (na kterou reagují například oči či světlocitlivý materiál, jako je fotografický film). Vlnová délka určuje vzdálenost, po které se vlnění dostane do stejné fáze. Fáze udává posun v periodě vlnění. Fáze se po uražení vzdálenosti rovné vlnové délce začne opakovat. U viditelného světla navíc vlnová délka určuje jeho barvu.

U většiny aplikací studujících šíření světla se většinou nezajímáme o časově proměnné veličiny. Zajímá nás pouze časově neproměnná hodnota. I v knihovně Rayleigh se pracuje s časově neproměnnými veličinami.

Světlo má kromě výše zmíněných vlastností i další. Jednou z dalších vlastností světla je koherence (soudržnost). Běžné bílé světlo není koherentní (koherence se ztrácí po určité vzdálenosti). Ztráta koherence se projeví tím, že u světla nelze předpokládat fázi odpovídající teoretickému vzorci. Různé typy světel mají různé délky koherence. Běžné světlo je koherentní v řádech mikrometrů. Délka koherence laserového světla závisí na jeho zdroji a pohybuje se od mikrometrů až po stovky metrů. Výkonné lasery (například řezací) mají běžně délku koherence v řádech mikrometrů, naopak stovky metrů jsou typické pro aplikace v atomové fyzice. Lasery pro holografii mají obvykle délku koherence několik centimetrů až několik metrů.

V simulacích se na rozdíl od reálného světla počítá s dokonalou koherencí. Dost často se samozřejmě potřebuje simulovat i světlo s omezenou koherencí, ale knihovna Rayleigh to v současnosti příliš nepodporuje.



Obrázek 2.1: Jednoduchý příklad interference.

Mezi další z vlastností světla patří například polarizace (směr, ve kterém částice kmitají), ale v důsledku zavedení skalární aproximace se informace o polarizaci ztrácí.

Nyní je nutné zaměřit se na některé jevy ve vlnové optice. Jedním takovým jevem je interference. Jak již bylo zmíněno, u více světelných vln dochází k jevu interference. Nutné podmínky vzniku interference jsou:

- Vlnění mají stejnou vlnovou délku.
- Vlnění jsou koherentní

V praxi lze interference docílit tak, že se využije světlo pouze z jednoho zdroje (většinou laseru, jelikož má větší délku koherence) a k rozdělení se využívá polopropustných zrcátek.

V následujících úvahách budeme uvažovat, že světlo splňuje podmínky interference (koherentnost, vlnová délka). K interferenci dochází kvůli změnám fáze světla. Výsledek interference závisí na rozdílu fází. Princip bude ukázán na příkladu. Pro zjednodušení budeme uvažovat že existují pouze dva zdroje světla (oba stejné), podobně jako je naznačeno na obrázku 2.1.

V uvažovaném příkladě jsou dva zdroje světla (naznačeny červeně) a z

těchto zdrojů vychází světlo (na obrázku jsou naznačeny pouze dva významné páry paprsků, ale ve skutečnosti se světlo šíří všemi směry). V paprscích je naznačena změna fáze (přechod mezi světlou a tmavou barvou). Na obrázku si lze všimnout, že na různá místa (na obrázku 2.1 označená A a B) stínítka dopadá světlo s různou fází (to závisí na vzdálenosti cílového místa od zdroje). V případě pravého místa dopadu (na obrázku 2.1 označené A) jsou fáze proti sobě posunuty o π radiánů. V tomto místě dochází k destruktivnímu sečtení, a tedy výsledkem bude tmavé místo (s intenzitou světla menší než mají zdroje). V druhém místě (na obrázku 2.1 označené B) naopak dopadne světlo se stejnou fází a dojde ke konstruktivnímu sečtení a místo bude světlé (v tomto místě je intenzita vyšší než u jednotlivých zdrojů).

U tohoto jednoduchého příkladu jsme zkoumali výsledek interference pouze na stínítku a cestu světla jsme naznačili paprskem. Ve skutečnosti ale k interferenci dochází v celém prostoru. V jednotlivých místech prostoru bude situace obdobná, jako v místech na stínítku - je důležitý fázový rozdíl obou potkávajících se vln. Ve výsledku tedy závisí na rozdílu vzdálenosti cílového místa k oběma zdrojům. V případě více než dvou zdrojů je nutné poskládat veškeré příspěvky.

2.2 Typické úlohy

Hlavním prvkem knihovny je simulace šíření světla volným prostorem. Konkrétně, známe-li vlastnosti světla na rovině ρ_1, dokáže knihovna vypočítat vlastnosti světla na rovině ρ_2. Vlastnosti světla v rovině ρ_1 lze definovat různým způsobem, může například simulovat chování rovinné vlny. Na rovinu ρ_2 může kromě světla z ρ_1 dopadat i mnoho dalších světel, které spolu pak interferují. Knihovna se tedy dá použít pro studium interference světelných svazků. Na principu interference pracuje i holografie, knihovnu tedy můžeme využít pro simulaci vzniku hologramu.

Na rovině ρ_2 můžeme vlastnosti světla nějak změnit, například můžeme těsně za ni „přilepit“ tenkou čočku. Pak lze studovat, jak se světlo šíří dále. Knihovnu tak lze použít pro simulaci průchodu světla obecnou optickou soustavou.

2.3 Typické nároky

Paměťové a časové nároky jsou dány hlavně dvěma skutečnostmi. Optické členy je nutné modelovat s velikostí alespoň cm^2. Druhou podmínkou je vzorkovací vzdálenost u optického členu, která musí splňovat Nyquistovu podmínku. Nyquistovu podmínku lze vyjádřit vzorcem $\frac{1}{d_{vzorkovaci}} \geq 2 \cdot f_{max}$, kde $d_{vzorkovaci}$ je vzorkovací vzdálenost a f_{max} je maximální prostorová frekvence vyskytující se v simulaci. To většinou vede k nutnosti mít vzorkování o vzdálenosti srovnatelné s vlnovou délkou (tedy několik μm nebo méně). Tyto podmínky vedou k nutnosti pracovat s 2D poli s počtem vzorků v řádech desítek milionů, nebo více.

Například pro reprezentaci optického členu s velikostí 1 cm^2 se vzorkovací vzdáleností 10 μm, je nutné mít 2D pole o rozměrech 1000×1000. V jiném případě, ve kterém chceme vypočítat bezpečnostní hologram 1×1 cm, je nutné mít vzorkovací vzdálenost 0.5 μm, čili rozměry 2D pole budou 20000×20000

2.3.1 Paměťové nároky

Hlavní datová struktura pro reprezentaci optických členů je 2D pole. Jelikož je nutné v každém místě (pixelu) optického členu uchovávat informaci jak o amplitudě, tak o fázi, musí být k reprezentaci použito komplexní číslo. Komplexní číslo je v počítači reprezentováno jako struktura dvou reálných (double) čísel. Jedno reálné číslo (s přesností double) je uloženo do 8 B a tedy jedno komplexní číslo zabírá v paměti 16 B. Typicky se počítá s 2D poli o rozměrech minimálně v řádů tisíců prvků. Pokud tedy budeme uvažovat velikost 1000×1000 pak výsledná paměťová náročnost tohoto členu je $1000 \cdot 1000 \cdot 16 \approx 16$MB. Při většině úloh je nutné volit i několikrát větší optické členy a tedy paměťová náročnost jednoho členu může být i v řádech stovek MB. Navíc je většinou nutné mít několik členů najednou v paměti a paměťové nároky mohou vystoupat až do řádů desítek GB paměti.

Při výpočtech je nutné mít veškerá nutná data v paměti, jelikož odsun do souborů (případně swapu) by celou aplikaci zpomalil. Při optických simulacích je tedy nutné správně hospodařit s pamětí. V opačném případě se trvání výpočtů může i několikanásobně zvýšit.

2.3.2 Časové nároky

Zdlouhavost výpočtů simulací je hlavně dána velikostí 2D polí, se kterými se pracuje, a nutností provést Fourierovy transformace nad těmito 2D poli. Výpočet jedné simulace může trvat až desítky minut. V mnoha případech nás zajímá několik konfigurací simulace (různé úhly, vzdálenosti, ...) a tedy výpočet celé úlohy může trvat i dny.

2.4 Provoz knihovny

Knihovna je naprogramovaná v jazyce ANSI C. Existuje jako dynamická knihovna (tedy sama o sobě není spustitelná). Aktuálně je možné využít knihovnu jen v případě, že je naprogramovaná aplikace v nějakém nízkoúrovňovém jazyce (jako je C++) a k této aplikaci je přiložena dynamická knihovna. V takto vytvořené aplikaci je poté nutné poskládat simulační úlohu pomocí volání funkcí z knihovny a nakonec pomocí dalších funkcí knihovny zahájit simulaci a případně uložit získaná data do souboru.

Tento přístup je velmi nevhodný pro neprogramátory, jelikož samotné programování v nízkoúrovňových jazycích vyžaduje určité zkušenosti (hlavně co se týká správy paměti). Také samotná příprava prostředí (provázání s dynamickou knihovnou včetně hlavičkových souborů knihovny, nastavení překladače a někdy i zprovoznění překladače) může být náročná. Cílem této práce je hlavně zjednodušení užití této knihovny přidáním uživatelského rozhraní.

2.5 Možnosti zpřístupnění knihovny

Možností, jak uživatelům zpřístupnit knihovnu, je několik. Je možné přidat grafické uživatelské rozhraní a změnit knihovnu na plnohodnotný spustitelný program. Další možností je rozdělení knihovny do velkého množství jednoduchých programů, kde každý bude určen na specifickou část knihovny (jednoúčelové programy) a postupným spouštěním těchto malých programů dojít k požadovanému výsledku. Další možnost je přidání skriptového rozhraní, které bude dle vstupního skriptu volat požadované funkce z knihovny.

Grafické uživatelské rozhraní umožňuje ovládat program prostřednictvím interaktivních ovládacích prvků. V tomto případě bývá program většinou okenní aplikace, kde jednotlivá okna jsou určena pro jednotlivé části programu. V případě knihovny Rayleigh by bylo nutné poskládat scénu umístěním prvků na uživatelskou plochu. Jelikož se jedná o 3D scénu, musela by být vyřešena transformace z 3D na 2D okno. Kromě umístění prvků by také muselo být možné definovat vlastnosti jednotlivých prvků a operace mezi jednotlivými prvky.

Pro případ řešení rozdělení knihovny na dílčí programy, by muselo být vyřešeno vhodné rozdělení do programů. Též by muselo být vyřešeno předávání hodnot (pokud by se zvolila možnost ukládání dat do souborů, vedlo by to kvůli velkému množství dat, ke zpomalení vypočtu). Výpočet v tomto případě by probíhal způsobem postupného volání jednotlivých programů.

Další z možností je připojení skriptového rozhraní (tedy překladače a interpretu) ke knihovně. Uživatel by poté napsal textový soubor (představující skript), ve kterém by popsal scénu umístěním objektů a následně určením funkcí, které se nad objekty budou volat. Překladač by musel skript přečíst a zjistit, co uživatel popsal. Následně by interpret vykonal požadovanou činnost.

Rozhodl jsem se pro naprogramování skriptového rozhraní ke knihovně. Skriptové rozhraní nebude pravděpodobně tak atraktivní jako grafické rozhraní, ale ve skriptech bude možné popsat scénu tak, jak chceme, a též zde bude přesně určeno, co se má vykonat. Dalším přínosem skriptového rozhraní může být možnost zautomatizování (průběh stejného zpracování nad různými daty) a také bude moci spustit skripty přes noc (jelikož optické simulace jsou zdlouhavé).

3 Nástroje pro optické simulace

Kvůli práci na tomto projektu jsem prozkoumal i některé současné nástroje pro simulaci optiky. Vycházel jsem především z informací, které se o optických simulátorech vyskytují v časopisech SPIE Professional [14] a Optics & Photonics News [13]. Zde se zmíním o programu GLAD, který má skriptové rozhraní. Kromě tohoto programu jsem zkoumal i jiné programy, ale vzhledem k tomu, že nemají skriptovací rozhraní, zde další uvádět nebudu. Mimo programy zaměřené přímo na optiku zde ještě uvedu program Matlab, jelikož má též skriptovací rozhraní, a navíc je zde i možnost využít prostředí Matlabu jako skriptového prostředí pro knihovnu.

3.1 GLAD

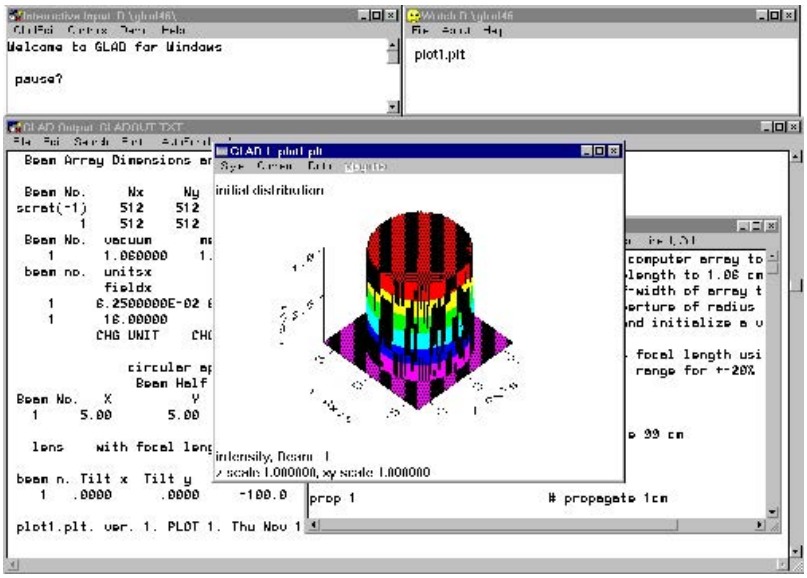
Program je jednou z komerčních aplikací zaměřených na optiku. Tento program je určen hlavně pro oblast fyzikální optiky a laserové analýzy. GLAD je zkratka pro General Laser Analysis and Design (obecná analýza a design laserů). Hlavním účelem je modelování systémů, kde se vyskytuje koherentní světlo. Víc informací o programu lze nalézt na stránkách `http://aor.com/`.

Ovládání programu je zpřístupněno pomocí příkazového rozhraní, případně lze příkazy zapsat do skriptů a spouštět celé skripty. Program umí uživateli zobrazit výsledky ve formě grafů.

Zde se pokusím porovnat skripty psané pro program GLAD a mnou navrhované pro knihovnu Rayleigh. Kompletní ukázkový skript pro program GLAD [1] lze nalézt v příloze C. Tento ukázkový skript slouží k vygenerování gausovského paprsku [10].

Ve skriptech pro program GLAD je možné využít příkazy (například `units/set 1 .01`), podmínky (například `if [x==y] x = z`) a samozřejmě i přiřazení hodnot proměnným v podobě výrazů `x = x + 3`. Ve skriptech je též možné vytvářet komentáře (řádky začínající `c`). Ve skriptech je možné definovat funkce (v jazyce nazývané makra), které lze později volat. Pomocí těchto funkcí (maker) lze docílit i chování podobné, jako u cyklů.

Jednou z odlišností je syntaktická odlišnost. Další z odlišností je mož-



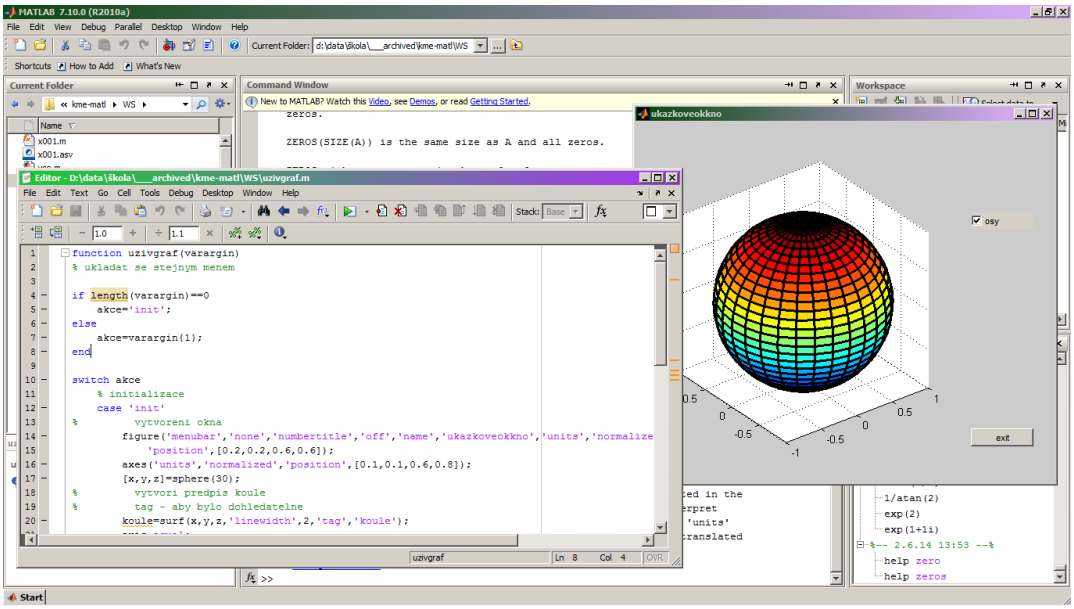
Obrázek 3.1: Prostředí programu GLAD.

nost definice vlastních funkcí (v navrhovaném jazyce pro knihovnu Rayleigh není možné definovat vlastní funkce). Co se týká cyklů, ty lze v navrhovaném jazyce vytvářet pomocí příkazu `for`. V případě jazyka pro GLAD i pro knihovnu Rayleigh je nutné hlavně připravit parametry a umožnit volání jednotlivých funkcí pro ovládání programu.

Další odlišností mezi programem GLAD a plánovaným uživatelským rozhraním knihovny Rayleigh je, že program GLAD má vlastní prostředí (obrázek 3.1), zahrnující příkazové okno, editor skriptů a také možnost zobrazovat výsledky formou grafů. Rozhraní pro knihovnu Rayleigh je prozatím naplánováno na čistě skriptovací, tedy program bude spuštěn s parametrem vstupního skriptu a veškeré výsledky je možné předat pouze v konzoly, případně uložit do souboru (ve formě obrázků, k jejichž zobrazení je nutný některý z prohlížečů obrázků).

3.2 Matlab

Matlab [6] je komerční program určený pro matematické výpočty. Domovská stránka programu je `http://www.mathworks.com/products/matlab/`. Ačkoli je matlab komerční, existují i další vhodné opensource alternativy (jako je GNU Octave).



Obrázek 3.2: Prostředí programu Matlab.

Matlab je programové prostředí (ukázáno na obrázku 3.2), které je primárně ovládáno přes příkazy (podobně jako příkazová řádka). Prostředí umožňuje zadávání příkazů (hlavní okno), psaní skriptů (v editoru skriptů) a i zobrazení výsledků (jako například grafů, ale umožňuje i vytváření oken s interaktivním rozhraním). Umožňuje výpočty s maticemi, vykreslování grafů, počítačové simulace a mnoho dalšího.

Příkazy je možné psát buď rovnou do příkazového okna (kde budou rovnou vykonány), nebo do skriptů (které lze později spustit). Tyto skripty jsou uloženy v souborech v příponou `.m` (někdy nazývané m-files) a lze je do prostředí nahrát. Ukázkový skript je možné prohlédnout v příloze D, jehož výsledkem je vytvoření jednoduchého okna, kde je zobrazen graf a ovládací prvky pro zapnutí/vypnutí os a zavření okna (část skriptu a výsledek vykonání skriptu jé též vidět na obrázku 3.2).

Nyní napíši něco ke srovnání skriptů pro Matlab a pro navrhovaný jazyk ke knihovně Rayleigh. Hlavní rozdíl je již při užití skriptovacích jazyků. Skriptovací jazyk Matlabu lze využít pro mnoho účelů (matematické úlohy, vytváření grafů, vytváření uživatelských rozhraní, ...). Navrhovaný jazyk pro knihovnu Rayleigh je mnohem úžeji zaměřený - slouží hlavně k rozmístění optických členů v simulaci a definováni vlastností těchto členů. I přes tento rozdíl v užívání lze porovnat jazykové schopnosti obou jazyků. Ve skriptu

Matlabu je možné využít komentáře (vše za znakem %), definice funkcí (`function uzivgraf(varargin)`), podmíněné akce, volání funkcí a přiřazování hodnot proměnným. Také je možné využívat cykly. Kromě definování vlastních funkcí umožňuje navrhovaný jazyk ke knihovně Rayleigh všechny tyto možnosti. Funkce v navrhovaném jazyce budou prozatím pouze předem definované (navržené tak, aby umožnily využít funkce knihovny). Další z odlišností mezi jazyky je definice proměnných. V Matlabu je proměnná automaticky definována při přiřazení hodnoty do proměnné. V případě jazyka knihovny Rayleigh je nutné napřed definovat proměnnou, včetně typu, a až následně přiřadit hodnotu.

Mezi další odlišnosti mezi prostředím Matlabu a plánovaným rozhraním knihovny Rayleigh patří i to, že program Matlab poskytuje i grafické výstupy přímo na obrazovku (mezi něž patří i vytvoření vlastních oken). Ačkoli výstupy knihovny Rayleigh jsou obrázkové, musí stačit pouze ve formě souborů, a není tedy možné uživateli zobrazit grafický výstup přímo na obrazovku pomocí oken (uživatel si ale může obrázek otevřít ve svém oblíbeném prohlížeči obrázků).

Mimo to, že program Matlab má skriptové prostředí, poskytuje i možnost propojení s ostatními programy. Toto představuje potenciální možnost, jak přidat skriptové rozhraní ke knihovně Rayleigh.

Pro integraci Matlabu a knihovny Rayleigh jsem se ale nerozhodl. Hlavním důvodem je paměťová náročnost optických simulací, a v Matlabu je omezená podpora pro ruční správu paměti. Dalším důvodem je, že jazyk pro knihovnu Rayleigh jsem začal vyvíjet již během semestrální práce z FJP a v této práci jsem se snažil jej dokončit. Dalším důvodem je, že Matlab je komerční (ačkoli to by bylo možné vyřešit použitím alternativy Octave, ale na dalších důvodech se nic nemění). Posledním důvodem je, že vedoucí práce požadoval, aby ke knihovně existoval nezávislý skriptovací prostředek.

4 Možnosti implementace vlastního skriptovacího jazyka

Tato kapitola se bude věnovat možnostem přidání skriptovacího jazyka do programu. To zahrnuje dva hlavní způsoby - využít předpřipravený jazyk, a nebo navrhnout zcela vlastní jazyk.

4.1 Skriptovací jazyk

Skriptovací jazyk [11] je počítačem rozpoznávaný jazyk, který slouží k zapsání posloupnosti příkazů (akcí) jež řídí činnosti programu. Skriptovací jazyk většinou poskytuje jazykové konstrukce pro vytváření proměnných, přiřazení hodnot proměnným, volání funkcí, podmínky, cykly.

Existují předpřipravené jazyky vhodné pro obecné použití. Pokud chceme mít ve vlastním programu skriptové rozhraní, je někdy výhodné využít některý z předpřipravených jazyků. Obecně použitelné jazyky mívají k dispozici vlastní překladač a interpret, které lze začlenit do vlastního programu (například formou dynamické knihovny).

Jednotlivé jazyky se mezi sebou mohou odlišovat. Například některé jazyky vyžadují definice proměnných, jiné automaticky vytváří proměnné při přiřazení hodnoty. Vytvořené proměnné lze poté použít jako parametry při volání funkcí.

Některé jazyky jsou přímo vykonávány a jiné provádějí napřed překlad do vnitřního jazyka (většinou nazývaného bytecode). U jazyků využívajících překlad do mezikódu může být interpret a překladač oddělen. Interpret jazyka, může být propojen s programem, zatímco překladač bude oddělený, nebo je do programu začleněn jak interpret, tak i překladač. Některé jazyky nabízejí obě možnosti, tedy možnost mít jak překladač, tak i interpret začleněný do programu a umožnit uživatelům psát vlastní skripty, nebo mít pouze interpret a spouštět pouze předem připravené skripty.

Využití předpřipraveného jazyka může obnášet i některé negativní vlastnosti. Pokud je skriptovací jazyk poskytnut pomocí knihovny, je nutné mít k dispozici takovou verzi knihovny, která bude fungovat na cílovém systému.

Další komplikace může způsobit ukončení vývoje jazyka a následné zastarání knihovny (například již nebude fungovat na nových systémech a tím se stane i program, do kterého byl překladač a interpret jazyka přiložen, nepoužitelným na novém systému). Dále může být komplikací i nevyhovující struktura jazyka.

Pro vlastní použití je nutné rozhodnout, který jazyk bude použit. Já jsem se zde rozhodl uvést jeden příklad předpřipravených jazyků, tedy jazyk Lua. Ostatní skriptovací jazyky mají podobné vlastnosti a zde tedy uvedu jen jeden příklad.

4.1.1 Jazyk Lua

Skriptovací jazyk Lua [7] představuje jednu z možností implementace skriptovacího rozhraní ke knihovně Rayleigh. Syntakticky se tento jazyk podobá jazyku Pascal, ale sémanticky je více podobný JavaScriptu. Skriptovací jazyk je možné přímo propojit s jiným programem (psaným například v C++) a to jak s možností využití překladače i interpretu, tak připojení pouze interpretu (který bude interpretovat bytecode, viz dále). Poté lze v programu spouštět skripty v jazyce Lua (a nebo předkompilované verze těchto skriptů, pokud je využit pouze interpret bez překladače). V těchto skriptech lze pak popsat požadované chování včetně možnosti volat funkce z propojeného programu.

Jazyk podporuje proměnné typu booleovské hodnoty, čísla a řetězce. Pro složitější datové struktury je nutné využít typ tabulka, což je obdoba asociativního pole. Pomocí tabulek lze i vytvářet jmenné prostory a objekty.

Skripty psané v jazyce Lua nejsou přímo interpretovány, ale nejprve jsou předkompilovány do bytecode (což je mezistupeň mezi skriptovacím jazykem, ve formě textu, a přímo vykonatelným strojovým kódem). Tento bytecode je poté vykonáván pomocí virtuálního stroje, který slouží jako interpret. Proces kompilace je pro uživatele transparentní a ani není nutné, aby se tímto uživatel zabýval, jelikož proces proběhne před samotným vykonáváním.

Pro jazyk Lua také existuje mnoho frameworků, které mohou posloužit ke snadnějšímu provázání s programovacím jazykem.

Jeden ukázkový skript pro jazyk Lua je možné nalézt v příloze E.

Ještě je nutné probrat souvislost s knihovnou Rayleigh. Jazyk Lua před-

stavuje jedeno z možných řešení, jak přidat skriptovací rozhraní ke knihovně Rayleigh. Mezi výhody by jistě patřilo, že existují nástroje pro snadné zaintegrování skriptovacího jazyka Lua ke kódu v C++. Překladač i interpret by k existujícímu programu šly připojit formou knihovny bez většího zásahu. Toto řešení má ale i některé nevýhody. Jednou ze zásadních nevýhod je správa paměti. K tomu je využito garbage collectoru, který se pro tuto práci nehodí. Mezi další nevýhodu patří nutnost využití externí knihovny (a tedy práce by byla vázána použitím této knihovny a v budoucnu by mohly vzniknout problémy při pozdějších změnách).

Obecné skriptovací jazyky využívající garbage collector (mezi které patří i jazyk Lua) drží v paměti všechno, na co se lze ve skriptu odkázat. Teprve jakmile odkaz (např. proměnná) zanikne, je možné paměť uvolnit. Proměnná zaniká po konci bloku, ve kterém byla definována. Naproti tomu v optických simulacích je vhodné, aby se paměť uvolnila hned poté, co již ve skriptu nebude potřeba. K tomu je ovšem nutné provést před spuštěním skriptu jeho analýzu, kterou obecné skriptovací jazyky neprovádějí.

Další důvod, proč jsem jazyk Lua (ani některý jiný předpřipravený jazyk) nezvolil je, že pro knihovnu Rayleigh jsem již během semestrální práce z předmětu FJP začal vyvíjet jazyk, jehož interpret by provedl analýzu skriptu, při které by byly nalezeny místa, kde jsou proměnné naposledy použity a následně lze tedy uvolnit paměť, která již nebude nadále potřebná. Dalším důvodem je, že vedoucí práce požadoval, aby ke knihovně existoval nezávislý skriptovací prostředek.

4.2 Vlastní jazyk

Kromě možnosti využití předpřipraveného jazyka je zde možnost vytvořit zcela vlastní jazyk. V tomto případě je nutné zabývat se všemi detaily jazyka, to znamená gramatikou, překladačem a běhovým prostředím (virtuálním strojem). Tento způsob je náročnější, než využití předpřipraveného jazyka, ale umožňuje volnost v možnostech přizpůsobení jazyka konkrétním situacím.

V první řadě je nutné zaměřit se na vytvoření gramatiky pro náš jazyk. K popisu lze například využít Backus-Naurovu formu [8]. V tomto popisu je nutné zaměřit se na jednotlivá přepisovací pravidla, jejichž přepisováním dostaneme požadované výrazy.

Po vytvoření popisu jazyka je nutné vytvořit překladač [4] (který se skládá z lexikální a syntaktické části). Překladač je možné vytvořit manuálně, a tedy vytvořit lexikální analyzátor, který převádí vstupní text na lexikální symboly, s nimiž následně pracuje syntaktický analyzátor. Jeden z možných popisů lexikálních symbolů je pomocí regulárních výrazů. Po vytvoření lexikálního analyzátoru musíme ještě vytvořit syntaktický analyzátor. K tomu se většinou užívá rekurzivní sestup (pro který je nutné mít gramatiku LL(1)).

Konstrukce překladače manuálně je poměrně náročná a lze v ní udělat chyby. Avšak existují nástroje pro generování překladačů a ulehčují nám vytvoření překladače. Samotných druhů generátorů překladačů existuje velké množství [9]. Některé jsou určeny pro specifické jazyky, jiné podporují větší množství jazyků. Nástroje mohou mít i vlastní IDE (vývojové prostředí) ulehčující práci s nimi. Při užití nástrojů pro generování překladače je nutné zapsat lexikální a syntaktickou část gramatiky ve formě, jaká je nástroji požadována.

V následujících kapitolách se budu zabývat konkrétními nástroji právě pro vytváření překladačů. Jak jsem zmínil výše, existuje mnoho různých generátorů překladačů. Vzhledem k tomu, že knihovna je v jazyce C, hodí se, aby i generátor překladače jako výstup generoval zdrojové soubory v jazyce C (případně C++).

4.2.1 LEX a YACC

Tyto nástroje jsou poměrně dobře známé v oblasti generátorů překladačů. Pomocí těchto programů je možné vygenerovat zdrojový kód v C pro překladač.

LEX jako vstup požaduje popis jednotlivých slov (uložených v souboru s příponou `.l`). Tato slova bývají popsána regulárními výrazy. Z tohoto popisu vygeneruje program lexikální část překladače v jazyce C.

YACC jako vstup požaduje popis gramatiky (uložené v souboru s příponou `.y`). V gramatice jsou popsána jednotlivá přepisovací pravidla a navíc jsou k nim připsány akce, které mají být vykonány pro daná pravidla. Z tohoto vstupu je vygenerován zdrojový kód v C pro syntaktický analyzátor.

Z obou programů se získají zdrojové soubory v programovacím jazyce C pro celý překladač. Tyto zdrojové soubory stačí zkompilovat kompilátorem

pro jazyk C a získáme program, který rozpoznává námi popsaný jazyk a vykonává námy popsanou činnost v pravidlech gramatiky. Touto činností může být přímé provádění instrukcí, nebo vytvoření mezipřekladu jazyka. V případě mezipřekladu jazyka je nutné tento mezijazyk navrhnout a i interpret si musíme navrhnout a vytvořit zcela sami.

Od programů LEX a YACC jsou odvozeny další modifikace těchto programů sloužící témuž účelu, jako jsou Flex a Bison.

4.2.2 Flex a Bison

Programy Flex[3] a Bison[2] jsou odvozené od programů LEX a YACC. Vstupní soubory pro lexikální popis i gramatiku jsou podobné jako pro nástroje LEX a YACC popsané v kapitole 4.2.1. Výstupem je poté zdrojový kód v C (s možností využití konstrukcí z jazyka C++ pro akce). Po kompilaci zdrojových souborů získáme překladač.

Pro realizaci jazyka jsem se rozhodl využít právě tyto programy. Jedním z důvodů je, že jazyk navržený pro knihovnu Rayleigh je dosti komplikovaný a tedy manuální vytvoření překladače by bylo obtížné. Bylo tedy nutné použít generátor překladače. Dalším důvodem je, že program Flex a Bison existuje pro mnoho platforem. Dalším důvodem je kompatibilita vstupů těchto programů s LEXem a YACCem, a tedy v budoucnu je velká šance, že budou existovat nějaké podobné (nebo i stejné) programy, které budou schopny přeložit gramatiku.

5 Skriptovací jazyk ke knihovně Rayleigh

Rozhodl jsem se pro vytvoření vlastního skriptovacího jazyka ke knihovně Rayleigh. Důvod, proč jsem zvolil vlastní jazyk a ne některý z obecně použitelných skriptovacích jazyků je hlavně kvůli existenci návrhu jazyka (uskutečněného v rámci semestrální práce FJP). Navíc jsem měl připravený způsob předzpracování u tohoto jazyka, který by interpretu umožnil uvolňovat paměť, která již není zapotřebí. Právě včasné uvolnění paměti je důležité pro optické simulace, jelikož jejich paměťová náročnost (zmíněno v kapitole 2.3) je v řádech až desítek GB.

Pro realizaci překladače jsem se rozhodl užít nástroje Flex a Bison (zmíněné v kapitole 4.2.2). Jedním z důvodů užití těchto nástrojů je rozšířenost užití těchto nástrojů. Tyto nástroje existují již delší dobu a jsou poměrně známé v oblasti generátorů překladačů a lze tedy předpokládat, že i v budoucnu budou některé alternativy dostupné. Dalším důvodem je, že výstupem těchto nástrojů je kód v C, který lze přeložit překladačem pro C++. Tím odpadá závislost na další knihovně, kterou by bylo nutné mít v případě, že by se využilo předpřipraveného jazyka.

Nyní přistoupím k realizační části samotného jazyka. V prvním kroku je nutné podrobně navrhnout gramatiku jazyka (a navíc ji později zapsat v takové formě, v jaké je nástroji Flex a Bison požadováno). V dalším kroku je nutné rozhodnout o interpretu, tedy zda jazyk bude přímo interpretován, nebo zda se nejprve přeloží do vnitřního jazyka a až poté je interpretován. Kvůli analýze kódu pro zjištění, kdy jsou proměnné ještě zapotřebí, je mnohem vhodnější zvolit překlad do vnitřního jazyka a až ten interpretovat. Je tedy nutné ještě navrhnout vnitřní jazyk. Ten by měl být jednoduchý (formou jednoduchých instrukcí). Jako referenční model pro interpret jsem zvolil zásobníkový interpret (tedy všechna data jsou ukládána do zásobníku a jednotlivé instrukce pracují primárně se zásobníkem). Hrubý model je již navrhnut a nyní zbývá důkladněji prozkoumat jednotlivé části.

5.1 Gramatika jazyka

Před samotným návrhem gramatiky by bylo vhodné ukázat příklad, jak může výsledný skript jazyka přibližně vypadat. Následující skript může ukazovat jedno z použití. V tomto skriptu je ukázán příklad vlivu vibrací na záznam objektové vlny na hologram (objektová vlna je propagována na hologramy v různých vzdálenostech, v tomto případě je celkový posun 0.1 μm, a jednotlivé výsledky jsou uloženy do souborů obrázků, které lze i pouze vizuálně porovnat).

```
1  int numLights, i;
2  numLights = 50;
3  rayleighSetEnvironment(532 *1e-9, 5 *1e-6);
4  lightsource lights;
5  lightsArrayGenerateLine(lights,
6      [-1e-3,0,-300e-3],
7      1.0 / numLights, 0,
8      [1e-3,0,-300e-3],
9      1.0 / numLights, 0,
10     1,0.5,
11     numLights, 1);
12 for(i = 0; i < 10; i++){
13     component hologram;
14     hologram.width = 512;
15     hologram.height = 512;
16     hologram.center = [0,0,0]+i*[0,0,1e-8];
17     hologram.sampling = 1;
18     lightsArrayPropagate(lights, hologram);
19     optfieldSavePNG(hologram, "object_wave_"+i);
20 }
21 //konec skriptu
```

Z tohoto příkladu je vidět, že jazyk bude podobný jazyku C. Jazyk bude muset umět definovat nové proměnné (řádky 1, 4 a 13), přiřazovat hodnoty jednoduchých proměnných (například řádek 2), přiřazovat hodnoty do složitějších struktur (řádky 14-17), volat funkce (například řádek 3 nebo řádky 5-11), podporovat cykly (cyklus for začínající na řádku 12 a končící na řádku 20). Celkem vhodná je podpora komentářů (zde jen jeden na řádku 21), které mohou být jednořádkové, jako ve skriptu, nebo víceřádkové (ohraničené mezi „/*“ a „*/“). Kromě těchto vlastností zmíněných v příkladu budou vhodné i další, jako například podmínky (jak jednoduché typu „if-akce“ tak i typu „if-akce-else-akce“), pole (proměnné umístěné do polí jednoduchých i více-

rozměrných). Kromě těchto podobností s jazykem C bude v jazyce i možnost zjednodušeného plnění polí, jako je ukázáno na následujícím příkladu:

```
1 complex content[100,100];
2 content = %row+%col*%i;
```

Pro tento účel jsou zavedeny předdefinované proměnné (`%row`, `%col`, `%x`, `%y`). Samotné přiřazení do pole poté probíhá formou cyklu tak, aby byl výraz vyhodnocen pro každý index pole. Při každém vyhodnocení pro jeden index pole jsou předdefinovaným proměnným nastaveny hodnoty a výraz je tak vyhodnocen s ohledem na hodnoty předdefinovaný proměnných vztahující se ke zpracovávanému prvku. Proměnné `%row` a `%col` jsou nastaveny na index řádku a sloupce v poli. Proměnné `%x` a `%y` lze použít pouze u polí, které jsou součástí optického členu, a jsou nastaveny na souřadnice prvku pole. V tomto naznačeném příkladu bude pole naplněné komplexními čísly, kde reálná část bude rovna číslu řádku a komplexní část bude rovna číslu sloupce v dvourozměrném poli. První index pole určuje číslo řádku, druhý určuje číslo sloupce. Pole bude naplněno následujícími hodnotami: v prvním řádku na indexu `[0,0]` bude $0+0i$, `[0,1]` $= 0+i$, `[0,2]` $= 0+2i$, ..., v druhém řádku budou čísla `[1,0]` $= 1+0i$, `[1,1]` $= 1+i$, `[1,2]` $= 1+2i$, ... a dle tohoto vzoru bude naplněno celé pole.

Z těchto příkladů tedy máme dostatečnou představu o struktuře jazyka. Nyní je nutné toto ještě formálně sestavit do gramatiky. Gramatika má zjednodušeně tento tvar:

```
block -> statementlist
       ;

statementlist -> statementlist statement
               ;

statement -> vardef ";"
           | if
           | for
           | expression ";"
           | "{" block "}"
           ;

vardef -> typ varlist
        ;
```

```
varlist -> promenna
         | promenna "," varlist
         ;

if -> "if" "(" expression ")" "{" block "}"
    | "if" "(" expression ")" "{" block "}" "else" "{" block "}"
    ;

for -> "for" "(" expression ";" expression ";" expression ")"
       "{" block "}"
       ;

expression -> hodnota
            | identifier
            | expression operace expression
            | "(" expression ")"
            ;

identifier -> promenna array
            | funkce "(" parametry ")"
            ;

array -> /* nic */
        | "[" indexy "]"
        ;

indexy -> hodnota
        | hodnota "," indexy
        ;

parametry -> hodnota
            | identifier
```

Symboly uvedené v uvozovkách jsou terminální symboly. Symboly `typ`, `hodnota`, `operace`, `promenna` a `funkce` představují též terminální symboly (pro zjednodušení nejsou rozepsány v gramatice). Typ představuje datový typ proměnné (například `int`). Hodnota představuje číselnou hodnotu (tedy celé nebo desetinné číslo) nebo řetězec. Operace představuje konkrétní typ operace, například sčítání, odčítání, ... Proměnná a funkce jsou identifikátory proměnné nebo funkce (tedy jejich název). Ostatní symboly jsou neterminální a dále se přepisují.

Toto je pouze zjednodušený popis gramatiky. Navíc je nutné formát zápisu přizpůsobit použitým nástrojům (tedy Flex a Bison).

Programu Flex je nutné dodat popis všech terminálních symbolů. Po nalezení terminálního symbolu musí být vrácena hodnota pro tento symbol. Kompletní vstupní soubor pro program Flex lze nalézt na přiloženém CD. Mezi terminální symboly patří řídící znaky (například pro označení bloku „{“ , „}“), klíčová slova („if“, „else“, „for“), označení datových typů („int“, „double“, „complex“, „string“, ...) a také hodnoty a identifikátory. Konstantní symboly stačí zapsat v podobě řetězců a těmto symbolům je nutné přiřadit návratovou hodnotu formou konstanty, která je následně využita programem Bison.

Pro popis jednotlivých hodnot a identifikátorů je již nutné využít popisu pomocí regulárních výrazů. Například regulární výraz pro popis desetinných čísel je následující: `([0-9]*\.[0-9]+|[0-9]+)((e|E)(\+|-)?[0-9]+)?`. Tento regulární výraz říká, že číslo obsahuje buď číslice, nebo číslice oddělené desetinnou tečkou a může obsahovat i exponent. Tento zápis čísel je podobný, jako zápis čísel v jazyce C. Kromě vrácení konstanty určující, že se jedná o číslo, je ještě nutné vrátit samotnou hodnotu čísla. Jelikož je zápis podobný, jako pro jazyk C, je možné využít knihovní funkce jazyka C pro převod řetězce na číslo a toto číslo předat programu Bison.

Kromě čísel je nutné popsat identifikátory (což mohou být názvy proměnných, nebo názvy funkcí). Popis identifikátorů je již jednodušší. Identifikátor začíná písmenem a poté může obsahovat další písmena a číslice. Regulární výraz pro popis identifikátorů je následující: `[a-zA-Z][0-9a-zA-Z_]*`. Kromě návratové hodnoty říkající, že další symbol je identifikátor, je nutné vrátit i jméno identifikátoru (tedy uložit si ho jako řetězec pro pozdější využití).

Tímto lze uzavřít popis lexikální části překladače (pro program LEX, případně Flex). Následující část se týká syntaktické části překladače (pro program YACC, případně Bison).

Gramatika pro program Bison vychází z výše popsané gramatiky, jen je nutné rozepsat všechny detaily. Kompletní gramatiku lze opět nalézt na přiloženém CD.

Každý skript začíná hlavním blokem `block`. Tento blok se poté může přepisovat na příkazy `statementlist`. Mezi příkazy patří: definování proměnných (`vardef`), podmíněný příkaz (`if`), příkaz cyklu (`for`), výraz (`expression`) a vnořený blok (`block`, jež je vymezen závorkami „{“ a

„}“).

Výraz `expression` je hlavní částí jazyka - zde se definují hodnoty proměnných, operace mezi nimi a volání funkcí. Výraz `expression` může být přepsán na mnoho možností - `hodnota` (vložení číselné nebo řetězcové hodnoty), `identifier` (odkaz na dříve definovanou proměnnou, nebo volání funkce), `expression operace expression` (kde `expression` jsou další výrazy a `operace` je oparace mezi nimi - například násobení, sčítání ...). Postupným přepisováním lze vytvořit libovolně složité výrazy.

Jednotlivým přepisovacím pravidlům gramatiky je ještě nutné přiřadit akce, které budou vykonány při průchodu pravidlem. Tyto akce nyní nebyly probírány, jelikož nejprve je nutné definovat vnitřní jazyk interpretu, který bude probrán v následující kapitole. Poté bude k pravidlům doplněn popis akcí (v kapitole 5.4).

5.2 Model virtuálního stroje

Jelikož jsem zvolil jazyk, který využívá vnitřní jazyk, musí být tento vnitřní jazyk generován překladačem (při zpracovávání pravidel gramatiky budou generovány instrukce). Až tento vnitřní jazyk bude vykonáván virtuálním strojem (interpretem). Dále je nutné zvolit nějaký referenční model. Jako referenční model jsem zvolil model zásobníkového automatu [12].

Interpret bude kromě pole instrukcí (vygenerovaných překladačem) mít k dispozici zásobník, který bude sloužit jako primární místo ukládání hodnot. Jelikož je nutné určit, jaká instrukce je vykonávána, je nutné mít ještě registr „pc“ (program counter). Pro zjednodušená plnění polí jsou ještě využívány pomocné registry (pro jednotlivé předdefinované hodnoty %row, %col, %x, %y), a také registr zpracovávaného indexu u pole.

Zjednodušené plnění polí ve skriptu je řešeno pomocí cyklu, při kterém jsou opakovány instrukce vygenerované pro daný výraz. Před prováděním těchto instrukcí jsou navíc nastaveny předdefinované proměnné na hodnoty odpovídající zpracovávanému prvku pole. Kvůli zjištění, kde začíná cyklus pro opakování instrukcí je nutné mít uložený začátek výrazu (tedy index první instrukce, kde se výraz zpracovává).

Při vykonávání cyklu přiřazení se nejprve nastaví pomocné registry. Po-

mocné registry %row a %col jsou nastaveny na index řádky a sloupce zpracovávaného prvku pole. V případě, že pole patří optickému členu, jsou nastaveny i registry pro %x a %y na souřadnice odpovídající umístění oblasti, kterou zpracovávaný prvek pole reprezentuje. Po nastavení registrů je možné interpretovat instrukce výrazu. Tím je vypočtena požadovaná hodnota s ohledem na aktuální stav registrů, jež reprezentují vlastnosti zpracovávaného prvku pole. Tato hodnota je následně uložena do pole (na zpracovávaném indexu). Následuje posun indexu na další a vrácení se na instrukci začátku výrazu (tedy změna registru „pc“). Poté může proběhnout opět celý cyklus výpočtu dalšího prvku pole. Tento cyklus probíhá, dokud není dosaženo konce pole.

Při vykonávání mezikódu jednotlivé instrukce primárně pracují se zásobníkem (kromě instrukcí skoku, které ještě mohou měnit registr „pc“). Jednotlivé instrukce budou probrány v kapitole 5.3. Zásobník umožňuje čtení všech hodnot, ale přímá manipulace (vložení nebo odebrání hodnoty) je přístupná pouze pro vrchol zásobníku (poslední hodnota zásobníku). Jednotlivé hodnoty v zásobníku jsou objekty odděděné od třídy `Promenna`. Tyto objekty mohou představovat jak jednoduché proměnné (například `int`), tak složité (například `lightsource`). Některé detaily implementace interpretu i proměnných lze nalézt v kapitole 5.6.

Nyní zde uvedu na příkladu, jakým způsobem bude muset interpret pracovat. Prozatím nebyly ještě uvedeny jednotlivé instrukce, ale v následujícím příkladě bude zapotřebí instrukcí `PUSH` (vložení hodnoty) a instrukce `ADD` (součet dvou hodnot). V tomto příkladě bude zpracováván výraz `5 + 12`. Prozatím nebyly probrány akce pravidel gramatiky, ale prozatím budeme předpokládat intuitivní překlad. Získáme tedy kód v podobě:

```
PUSH  5
PUSH 12
ADD
```

Interpret bude postupně instrukce vykonávat. Na začátku předpokládejme prázdný zásobník. Interpret vykoná první instrukci (`PUSH 5`) a vloží hodnotu 5 do zásobníku. Následně interpret vykoná druhou instrukci (`PUSH 12`) a vloží hodnotu 12 do zásobníku (v tomto okamžiku jsou v zásobníku tyto hodnoty, v tomto pořadí: `5, 12`). Dále je nutné vykonat poslední instrukci (`ADD`). Tato instrukce slouží k sečtení dvou čísel a musí tedy tyto dvě čísla získat ze zásobníku. Nejprve ze zásobníku vybere hodnotu 12, a poté hodnotu 5 (v této chvíli je zásobník opět prázdný). Jelikož požadovaná operace je `5 + 12`, musí být hodnoty vybrané ze zásobníku přeskládány, tedy první ze

zásobníku je vybrán druhý operand a následně je vybrán první (veškeré operace, které vyžadují parametry ze zásobníku je tedy musí získávat v tomto pořadí). Nyní již instrukce `ADD` má potřebné hodnoty a vykoná nad nimi operaci sečtení. Výsledek opět vloží do zásobníku (tedy v zásobníku bude po ukončení této instrukce hodnota 17).

Toto byl jen jednoduchý názorný příklad, jakým způsobem instrukce probíhají. Složitější skripty budou převedeny na větší množství instrukcí, ale jednotlivé instrukce vždy pracují na tomto principu, tedy mohou vybírat hodnoty ze zásobníku a poté tam přidávat hodnoty výsledků. V tomto ukázkovém příkladě byly použity jako hodnoty v zásobníku celá čísla. Ve skutečnosti jsou ale v zásobníku uloženy objekty (oděděné od třídy `Promenna`, která bude důkladněji probrána v kapitole 5.6), ze kterých lze získat jejich typ nebo hodnotu.

5.3 Jednotlivé instrukce

Zde se zaměřím na popis jednotlivých instrukcí, použitých ve vnitřním jazyce interpretu. Instrukce jsem se snažil navrhnout tak, aby rozložily složitější konstrukce na jednoduše vykonavatelné instrukce. Mezi mnou navrženými instrukcemi může být podobnost s instrukcemi pro procesor (ale instrukce pro procesor jsou značně jednodušší).

Jednotlivé instrukce budou vykonávány interpretem (tedy virtuálním strojem, zmíněném v kapitole 5.2). Každá z instrukcí může mít až dva parametry (přidělené při překladu ze zdrojového skriptu, například u instrukce `PUSH` je to typ hodnoty a vkládaná hodnota). Kromě těchto parametrů můžou instrukce přistoupit k obsahu zásobníku. Hodnoty ze zásobníku jsou vybírány (nebo přidávány) od vrcholu zásobníku. Hodnoty v zásobníku jsou objekty typu `Promenna`, ze kterých lze získat datový typ, který objekt představuje, hodnotu základního typu, nebo je přetypovat na objekty složitějších typů. Při vkládání nových hodnot do zásobníku se objekty proměnných vytvářejí konstruktorem, a po odebrání a zpracování jsou zničeny destruktorem (u složitějších typů se destruktor daného objektu postaré o uvolnění veškeré související paměti).

Nyní již k jednotlivým instrukcím:

- Instrukce ALLOC - slouží k vytvoření nové proměnné v zásobníku.

Objekt nové proměnné (s nastaveným datovým typem, který je parametrem instrukce) je zkonstruován a vložen do zásobníku.

- Instrukce PUSH - instrukce slouží k vložení konstantní hodnoty (jako je číslo, nebo řetězec), nebo odkazu na proměnnou. V případě konstanty je typ vkládané hodnoty prvním parametrem instrukce, a druhým je samotná hodnota. Podobně jako u instrukce ALLOC musí být vytvořen objekt proměnné, ale navíc je ještě nutné objektu nastavit požadovanou hodnotu. Objekt proměnné je poté vložen do zásobníku.

  V případě, že instrukcí má být vložen odkaz na proměnnou, je druhým parametrem instrukce index pozice proměnná, na kterou je odkázáno. V tomto případě je nutné do zásobníku vložit proměnnou typu odkaz, která odkazuje na jinou proměnnou uloženou v zásobníku.

- Instrukce PUSHV - instrukce vkládá na konec zásobníku předdefinované proměnné. Těmi mohou být předdefinované konstanty (jako je π nebo imaginární jednotka), nebo obsah pomocných registrů (jako je index řádky v poli %row). Typ toho, co má být vloženo, je určen dle parametru instrukce. Opět je nutné vytvořit objekt proměnné, kterému je nastavena hodnota, a vložit jej do zásobníku.

- Instrukce STORE slouží pro uložení hodnoty z jednoho objektu do druhého. Instrukce vyjme dva objekty ze zásobníku. Objekt na vrcholu zásobníku představuje hodnotu, které má být přiřazena. Druhý objekt (další v pořadí v zásobníku) představuje objekt proměnné, do které má být hodnota uložena. Tomuto druhému objektu proměnné je nastavena hodnota prvního objektu (v případě, že druhý objekt představuje odkaz na proměnnou, je možné uložit hodnotu i do dříve alokovaných proměnných, tedy nejen těm na vrcholu zásobníku).

  Parametrem instrukce je index první instrukce, kde začíná výraz přiřazení. To je nutné při zpracování zjednodušeného plnění polí, aby mohl být proveden cyklus instrukcí pro přiřazování pro každý index pole.

- Instrukce DEL - odstraní ze zásobníku daný počet objektů proměnných. Počet odstraňovaných proměnných je parametr instrukce.

- Skupina matematických instrukcí. Tyto instrukce mají za úkol provést matematickou operaci mezi dvěma hodnotami, které jsou získány ze zásobníku. Samotné instrukce nemají parametry. Jednotlivé instrukce musí poskytovat ještě modifikace, pro všechny podporované typy hodnot (určené z objektu proměnné). Typy proměnných mají priority, nejnižší je celé číslo, poté desetinné, komplexní a nejvyšší prioritu má řetě-

zec. Hodnoty jsou přetypovány na typ s nejvyšší prioritou z těchto dvou objektů. Pro některé situace mohou instrukce pracovat i se složitějšími typy (jako například vektory).

Například pokud oba objekty vyjmuté ze zásobníku představují proměnné celého čísla, hodnotami jsou též celá čísla a musí být provedeno sečtení dvou celých čísel. Výsledek je opět (ve formě objektu proměnné, v tomto případě s typem celého čísla) vrácen na zásobník. Pokud jeden z objektů proměnných nese typ komplexního čísla, musí být i hodnota z druhého objektu převedena na komplexní číslo (pokud má typ nižší prioritu) a výsledkem bude též komplexní číslo. O přetypování je rozhodnuto až během interpretace instrukce, která s objekty pracuje.

Jak již bylo zmíněno dříve, v zásobníku jsou objekty v opačném pořadí, a je nutné tedy nejprve vybrat objekt, nesoucí druhou hodnotu, a teprve poté objekt, nesoucí první hodnotu.

Jednotlivé matematické instrukce jsou:

- Instrukce ADD - sečte 2 hodnoty (čísla, řetězce). Instrukce může sečíst i dva vektory, v tomto případě je provedena operace sečtení pro jednotlivé složky vektoru zvlášť.
- Instrukce SUB - odečte 2 číselné hodnoty. Instrukce může odečíst i dva vektory, v tomto případě je provedena operace rozdílu pro jednotlivé složky vektoru zvlášť.
- Instrukce MUL - vynásobí 2 číselné hodnoty.
- Instrukce DIV - vydělí 2 číselné hodnoty.
- Instrukce INV - umocní 2 číselné hodnoty.
- Instrukce MOD - provede operaci modulo 2 číselných hodnot.

- Skupina booleovských instrukcí. Stejně jako matematické instrukce, i tyto vyjímají objekty proměnných ze zásobníku, ze kterých získají hodnotu, a vrací tam výsledek (ve formě objektu). Booleovský typ není přímo zaveden, ale je využito celočíselné reprezentace (číslo 0 představuje hodnotu `false` (nepravda), ostatní představují hodnotu `true` (pravda)).

  Jednotlivé booleovské instrukce jsou:

  - Instrukce EQU - porovná 2 hodnoty (čísla, řetězce). Pravdivé, pokud čísla mají stejnou hodnotu nebo řetězce stejný obsah.

  - Instrukce NEQ - porovná 2 hodnoty (čísla, řetězce). Pravdivé, pokud čísla nemají stejnou hodnotu nebo řetězce nemají stejný obsah.
  - Instrukce LS - porovná 2 číselné hodnoty. Operace je pravdivá, když první hodnota je menší než druhá.
  - Instrukce GT - porovná 2 číselné hodnoty. Operace je pravdivá, když první hodnota je větší než druhá.
  - Instrukce LSE - porovná 2 číselné hodnoty. Operace je pravdivá, když první hodnota je menší nebo rovna druhé.
  - Instrukce GTE - porovná 2 číselné hodnoty. Operace je pravdivá, když první hodnota je větší nebo rovna druhé.
  - Instrukce AND - booleovský součin 2 booleovských hodnot.
  - Instrukce OR - booleovský součet 2 booleovských hodnot.
  - Instrukce NOT - Inverze booleovské hodnoty.

- Skupina instrukcí matematických funkcí. Vybrané matematické funkce jsou součástí jazyka. Podobně, jako matematické instrukce, vybírají tyto instrukce jeden objekt proměnné ze zásobníku, z něhož získají hodnotu, a vrací tam výsledek (vypočtený dle příslušné funkce). Instrukce jsou implementovány pro podporu desetinných čísel i komplexních (a tedy instrukce dle typu objektu proměnné rozhodne o užití vhodného podtypu instrukce). Mezi implementované funkce (se stejnojmennými instrukcemi) patří: sin(), cos(), tan(), cotg(), asin(), acos(), atan(), acotg(), exp() a ln().

- Instrukce VECT - slouží k vytvoření proměnné typu vektoru. Hlavní použití proměnné vektoru je pro vytvoření matematického vektoru (tedy veličiny, která je určena nejen velikostí, ale i směrem). Tyto vektory lze použít při dalších matematických operacích, nebo volání funkcí. Instrukce vybere ze zásobníku určený počet objektů proměnných a umístí je do objektu vektoru, který je vložen do zásobníku. Dimenze vektoru (a tedy počet vybíraných objektů ze zásobníku) je parametrem instrukce.

- Instrukce CALLV - zavolání funkce. Parametrem instrukce je počet parametrů a index funkce. Při provádění instrukce je nalezena volaná funkce, funkci je předán počet parametrů (samotné parametry si musí funkce najít v zásobníku interpretu) a po dokončení provedení funkce je odstraněn ze zásobníku počet proměnných odpovídající počtu parametrů. Pokud funkce má návratovou hodnotu, je tato hodnota (po odstranění parametrů ze zásobníku) vložena do zásobníku.

- Instrukce JMP - nepodmíněný skok. Při provádění instrukce je změněn registr „pc“ na požadovanou hodnotu z parametru instrukce.
- Instrukce JMZ - podmíněný skok. Při provádění instrukce je ze zásobníku vybrán objekt proměnné, z něhož je získána booleovská hodnota. Pokud tato hodnota odpovídá nepravdě (`false`) je registr „pc“ změněn na hodnotu parametru instrukce.

5.4 Akce u gramatiky

Nyní již byl prozkoumán model virtuálního stroje (kapitola 5.2) a použité instrukce (kapitola 5.3). V této kapitole se zaměřím na akce, prováděné syntaktickým analyzátorem při zpracovávání vstupního skriptu. Pokusím se tedy některé základní akce doplnit do hrubé navržené gramatiky z kapitoly 5.1. Kompletní přehled akcí lze získat ze souboru gramatiky pro program Bison přiloženém na CD.

Výstupem bude kód vnitřního jazyka, proto nejdůležitější částí bude vygenerování instrukcí s parametry dle zpracovaného skriptu. Jednotlivé instrukce jsou reprezentovány jako struktura, nesoucí typ instrukce a dva parametry. Celý kód je tvořen polem těchto struktur, do kterého jsou postupně instrukce ukládány. Instrukce jsou vykonávány interpretem postupně, číslo další vykonávané instrukce určuje registr „pc“,který reprezentuje index další instrukce v poli instrukcí.

U některých konstrukcí (cykly, podmínky) je nutné doplnit některé parametry instrukcí až po vygenerování všech instrukcí pro všechny části pravidla. Je proto nutné uchovávat indexy těchto instrukcí, které mají být doplněny. Ale jelikož pravidla jazyka jsou rekurzivní, je nutné k uložení indexů užít zásobník (pro překlad).

Interpret již nezná jména proměnných, psaných ve skriptu, má k dispozici pouze zásobník, kde jsou uloženy. Je tedy ještě nutné uchovávat tabulku identifikátorů, které převede jméno proměnné na index do zásobníku (interpretu). Do této tabulky jsou záznamy přidávány při definování proměnné, a odebrány po opuštění bloku, ve kterém byly definovány tyto proměnné.

Nyní již k samotné gramatice. Každý skript začíná hlavním blokem (`block`). V bloku mohou být definované proměnné, volány příkazy a mohou zde být i vnořené bloky. V bloku je nejdůležitější informace o počtu definova-

ných proměnných v daném bloku (aby po opuštění bloku byly odstraněny ze zásobníku interpretu). Proto v pravidle definování proměnných (`vardef`) je nutné inkrementovat čítač proměnných v bloku. Po opuštění bloku je nutné vygenerovat instrukci `DEL` s parametrem počtu definovaných proměnných.

V pravidle definování proměnných (`vardef`) je nutné vymezit místo pro proměnnou v zásobníku (interpretu), provedené vygenerováním instrukce `ALLOC` s parametrem typu proměnné. Navíc je nutné aktualizovat tabulku identifikátorů přidáním záznamu o jméně a indexu v zásobníku interpretu. Tento index je dán počtem doposud definovaných proměnných v celém skriptu (je tedy nutné mít nejen čítač pro počet proměnných v bloku, ale i pro celkový počet). Aby čítač celkového počtu byl aktuální, je po opuštění bloku nutné odečíst počet definovaných proměnných v opouštěném bloku (jelikož v interpretu byl instrukcí `DEL` odstraněn stejný počet proměnných ze zásobníku).

Další složitost nastává při zpracování pravidla podmínky (`if`) a cyklu (`for`). Při nich je nutné vygenerovat instrukce skoku, kterým je nutné doplnit správné parametry, kam skočit. Postup ukáži na příkladu skriptu:

```
if(i == 0){
    i++;
}else{
    i--;
}
```

Z tohoto skriptu bude vygenerován mezikód (pouze část):

```
00005     PUSH typ odkaz na promennou: i
00006     PUSH typ int: 0
00007    EQUAL
00008      JMZ skok na 00013
00009     PUSH typ odkaz na promennou: i
00010      INC
00011      DEL uvolneni zasobniku: pocet 1
00012      JMP skok na 00016
00013     PUSH typ odkaz na promennou: i
00014      DEC
00015      DEL uvolneni zasobniku: pocet 1
```

Podmínkové části (`i == 0`) odpovídají instrukce 5-7, bloku `if` odpovídají instrukce 9-11, bloku `else` odpovídají instrukce 13-15. Dále je nutné vygenerovar instrukce skoku, které zajistí větvení. Instrukce na pozici 8 musí zajistit provedení bloku `if` pouze pokud je podmínka splněna, tedy pokud není podmínka splněna je přeskočeno do bloku `else` na instrukci 13. V případě splnění podmínky je nutné vykonat blok `if` a poté instrukcí 12 přeskočit blok `else`, tedy skok na 16.

Podobně je nutné doplnit instrukce skoků u cyklu. Konstrukce cyklů a

podmínky je též nutné důkladně otestovat, a konkrétní příklad, včetně doplněných instrukcí skoku, je uveden v kapitole 6.1.

Poslední z částí jazyka jsou výrazy (`expression`), kde dochází k přiřazení hodnot proměnným a volání funkcí. U operátorů ve výrazech většinou předpokládáme různé priority (tedy že některé operátory mají přednost před jinými, například přednost násobení před sčítáním). Pokud programu Bison správně tyto priority popíšeme, vygeneruje nám syntaktický analyzátor, který tyto priority dodržuje a u samotných pravidel gramatiky je již možné předpokládat provádění ve správném pořadí zpracování.

Při zpracovávání přiřazovacího výrazu (ve tvaru `identifier = expression`) je nejprve vložen odkaz na proměnnou na indexu odpovídající proměnné (zjištěné z tabulky identifikátorů), tedy pomocí instrukce `PUSH` s parametrem indexu pozice v zásobníku. Poté je zpracována přiřazovaná část výrazu `expression`. Nakonec je vygenerována instrukce `STORE`, která provede přiřazení hodnoty, jež vznikne jako výsledek provedení výrazu, do požadované proměnné.

Přiřazovaná část výrazu může představovat hodnotu, v tomto případě je instrukcí `PUSH` vložena hodnota (předaná přes parametr instrukce). Dále může výraz představovat identifikátor (tedy odkaz na proměnnou, případně funkci). Proměnná je též vložena přes instrukci `PUSH`, jen parametr typu je nastaven na odkaz a parametrem hodnoty je index v zásobníku, kde se nachází proměnná (index opět nalezen v tabulce identifikátorů). Dále může výraz představovat operaci (výraz ve tvaru `expression operace expression`), kde `operace` představuje konkrétní operaci matematickou nebo booleovskou. V tomto případě jsou nejprve vygenerovány instrukce jednotlivých výrazů (vyřešeno rekurzivně) a nakonec je vygenerovaná instrukce pro požadovanou operaci (například `MOD` pro operaci modulo).

V případě, že výraz představuje volání funkce (ve tvaru `funkce(parametry)`), je nutné zavést čítač počtu parametrů. Postupně jsou zpracovány jednotlivé parametry, které jsou další výrazy, a nakonec je vygenerována instrukce `CALLV`. Jedním parametrem instrukce je počet parametrů volané funkce a druhým parametrem instrukce je index na funkci (získané z tabulky předdefinovaných funkcí).

Tímto postupem jsou tedy při zpracovávání vstupního skriptu vygenerovány instrukce do pole instrukcí.

5.5 Předzpracování mezikódu

Jeden ze záměrů jazyka je automatická správa paměti. V této části tedy naznačím, jakým způsobem jsem tuto konkrétní část vyřešil.

Po vygenerování mezikódu (tedy vnitřního jazyka interpretu), je tento mezikód analyzován. Při této analýze jsou hledány instrukce pracující s proměnnými jak pro čtení, tak i zápis (jako instrukce přiřazení, matematické instrukce, ...). Samotná analýza probíhá ještě před zahájením interpretace. Při nalezení instrukcí, pracujících s proměnnou, je k této proměnné poznamenán index instrukce, která proměnnou použila (přesněji je nutné uchovávat rozsah, kdy je proměnná potřebná, tedy index instrukce prvního a posledního použití). Po analyzování všech proměnných je možné do připraveného mezikódu vložit pomocné instrukce pro uvolnění již nepotřebných proměnných. Proměnné jsou identifikovány indexem uložení v zásobníku. Při samotné interpretaci je nejprve prozkoumán spojový seznam pomocných instrukcí, a pokud jsou přítomné některé pomocné instrukce pro uvolnění proměnných, je nad objektem proměnné, která již není zapotřebí, zavolána uvolňovací metoda, která se dle typu proměnné postará o uvolnění paměti (jako je například struktura reprezentující optický člen). Tato uvolňovací metoda je ale rozdílná od destruktoru.

Předzpracování se týká pouze skriptu, nijak neovlivňuje funkce knihovny Rayleigh, které zabírají nejvíce času. Předzpracování ale pomůže uvolnit paměť a tím zabránit případnému swapování, čímž může být výpočet urychlen, pokud by začala docházet volná paměť v počítači. Ale samotná rychlost výpočtu přímo ovlivněná předzpracováním není.

5.6 Detaily implementace

Dosud jsem se zaměřoval na popsání způsobu fungování překladače a interpretu. V této části se zaměřím na popis implementace více z programátorského hlediska. Pro tuto část samozřejmě platí vše popsané dříve, a tedy pro plné pochopení je nutné přečíst i předchozí text (tato kapitola slouží pro ulehčení orientace ve zdrojových souborech vytvořeného programu).

Ačkoli jsem se zde vždy zmiňoval o části překladače a interpretu, jako oddělených části, ve skutečnosti tvoří dohromady jeden program. Část pro

překladač je vygenerována pomocí programů Flex a Bison. Druhou částí je část pro interpretaci, kterou jsem celou vytvořil sám. Program je vytvořen v jazyce C++.

Nejprve zde uvedu seznam souborů, z nichž se má práce skládá (ačkoli je vše spolu provázáno, pokusil jsem se jednotlivé soubory rozdělit do logických celků).

- Soubor s hlavní funkcí programu `Main.cpp`
- Soubory související více s překladačem
  - `lex.yy.cpp` - lexikální část překladače
  - `y.tab.cpp` a `y.tab.h` - syntaktická část překladače
  - `funkce.cpp` a `funkce.h` - předpřipravené funkce
  - `gram.cpp` a `gram.h` - proměnné pro překladač (jako zásobník, pole mezikódu)
  - `konstanty.cpp` a `konstanty.h` - konstanty
  - `struktury_typy.h` - definice struktur používaných překladačem
- Soubory související více s interpretem
  - `Interpret.cpp` a `Interpret.h` - třída interpretu
  - `Instrukce.cpp` a `Instrukce.h` - třída instrukcí
  - `Promenna.cpp`, `Promenna_komponenty.cpp`, `Promenna.h` a `Promenna_komponenty.h` - třídy pro reprezentaci proměnných

5.6.1 Část překladače

V části překladače je soubor `lex.yy.cpp` vygenerován programem Flex ze zdrojového souboru gramatiky `lex.l`, kde je popsána lexikální analýza vstupního skriptu. Soubory `y.tab.cpp` a `y.tab.h` jsou vygenerovány programem Bison ze zdrojového souboru gramatiky `gram.y`, kde je popsána syntaktická analýza a překlad vstupního skriptu do vnitřního jazyka interpretu.

V souboru `funkce.cpp` jsou definovány předpřipravené funkce (tento soubor se také týká i překladače, jelikož obsahuje funkce, které provedou požadovanou činnost, kterou většinou je zavolání funkcí z knihovny Rayleigh).

Při přidávání dalších funkcí je nutné doplnit (ve funkci `priprav_fce()`) jména a vytvořené funkce. Další detaily o přidávání dalších funkcí lze získat z přílohy B.

V souboru `gram.cpp` je jednak ukládán mezikód vygenerovaný překladačem, je zde zásobník překladače a také je zde uchovávána tabulka identifikátorů. Pro přistoupení k jednotlivým částem jsou definovány funkce, jako funkce `gen(instrukce, param1, param2)` pro vygenerování instrukce (a zařazení na konec generovaného kódu). Funkce `pridej_identifikator`, `najdi_identifikator` a `odstran_identifikatory_bloku` slouží pro práci s tabulkou identifikátorů (tedy přidání, nalezení a odstranění identifikátorů). Pro přístup k zásobníku překladače slouží funkce `push_prekl` a `pop_prekl` (tedy vložení a vybrání hodnoty na vrcholu zásobníku).

Hlavičkový soubor `konstanty.h` slouží pro definování konstant instrukcí a typů proměnných. V souboru s kódem (`konstanty.cpp`) stojí za zmínku jen funkce `priprav_struktury`, která přidává popisy struktur (použitelných ve skriptu) obsahující názvy a typy podproměnných u strukturovaných typů. Tyto popisy jsou použity překladačem i interpretem.

Hlavičkový soubor `struktury_typy.h` obsahuje definice jednotlivých struktur použitých překladačem (například struktura pro uchování instrukce, uchování identifikátoru v tabulce identifikátoru a další).

5.6.2 Část interpretu

V této části je definován samotný interpret, tedy třída `Interpret` (soubory `Interpret.cpp` a `Interpret.h`). Třída obsahuje pole instrukcí, registry (registr „pc“ a další pomocné registry pro plnění polí), zásobník objektů proměnných a pomocný zásobník pro předzpracování. Je poskytnuta metoda `start`, ve které je nejprve provedeno předzpracování a poté interpretace vygenerovaného mezikódu. Interpretace probíhá způsobem volání metody `exec` nad objektem instrukce umístěné v poli na indexu „pc“. Po dokončení instrukce je registr „pc“ inkrementován (případně změněn instrukcí skoku). Po dosažení konce pole instrukcí (nebo v případě chyby) tato metoda končí.

Třída `Instrukce` slouží k implementaci všech instrukcí (popsaných v kapitole 5.3). Každá z instrukcí slouží k vykonání základní činnosti, čímž by měly být instrukce jednoduché. Kód každé z instrukcí se ale komplikuje, kvůli nutnosti zpracování všech možných typů proměnných (které mohou

nastat) a ošetření všech výjimek. Každá z instrukcí tedy musí obsahovat „podinstrukce“, které zpracují dané vstupní typy (například instrukce sčítání musí řešit sečtení čísel, řetězců i vektorů, a pro každé typy parametrů tedy musí být „podinstrukce“ zajišťující správné vykonání). O tom, která z podinstrukcí bude provedena, je rozhodnuto na základě typů proměnných v zásobníku (které instrukce vyžaduje) v době zpracovávání instrukce.

Třída `Instrukce` nabízí dvě hlavní metody: `exec` a `preexec`. Metoda `preexec` slouží pro provedení předzpracování. V této metodě je provedena jen simulace provádění instrukce. Například pokud se jedná o instrukci pracující s dvěma hodnotami ze zásobníku, jsou k těmto dvěma hodnotám poznamenány informace o užití (ale samotná činnost, jako sečtení, provedena není). Další z metod je `exec`, sloužící k samotnému vykonání instrukce. Tato metoda vykoná samotnou činnost příslušné instrukce. Obě tyto metody jsou využívány interpretem.

Třídy pro reprezentace proměnných tvoří poměrně rozsáhlou část práce. Veškeré tyto třídy dědí od třídy `Promenna`, která definuje základní rozhraní. Základní rozhraní umožňuje přístup k hodnotám základních typů (celé, desetinné a komplexní číslo a řetězec). Dále toto rozhraní umožňuje detekci a přetypování na složitější typy.

Typy proměnných jsem rozdělil do těchto skupin: Základní proměnné, proměnná odkazu, pole (oděděné od třídy `Promenna_array`), vektor a komponenty (oděděné od `Promenna_komponenta`).

Základní proměnné slouží k uložení hodnoty základního typu (tedy celé, desetinné, komplexní číslo a řetězec). Pro snadnější použití každý základní typ poskytuje metody získání hodnot i ostatních základních typů (na které budou hodnoty přetypovány).

Proměnná odkazu slouží k vytvoření „kopie“ proměnné. Implementuje veškeré metody třídy `Promenna` způsobem, že jsou přesměrovány na objekt proměnné, na který má být vytvořen odkaz. Důvod zavedení tohoto typu proměnné je, aby se zjednodušily činnosti při uvolňování a bylo s každým objektem zacházeno stejně, tedy zavolán destruktor příslušného objektu. Destruktor zavolaný na odkazu nezpůsobí vymazání původního objektu (tedy zavolání jeho destruktoru), na který je odkázáno.

Proměnná typu pole slouží k uchování většího množství hodnot stejného typu. Je možné mít pole jak pro základní typy, tak i složitější typy. Veškeré třídy proměnných typu pole implementují rozšiřující třídu `Promenna_array`,

která definuje metodu `odindexuj`. Tato metoda slouží k získání proměnné na daném indexu. Index je ve skutečnosti pole čísel (jedno pro každou dimenzi), ale veškeré indexy jsou převedeny pouze na jeden index, aby mohlo být každé pole vnitřně reprezentováno jen jednodimenzionálně. Od třídy `Promenna_array` jsou poté odděděny další třídy, které reprezentují jednotlivé typy (například `Promenna_array_int`).

Proměnná typu vektor slouží, podobně jako pole, k uchování více proměnných. Pokud složky vektoru jsou číselné proměnné, je možné s proměnnou vektoru pracovat jako s matematickým vektorem (tedy jako s veličinou, která je určena nejen velikostí, ale i směrem). K získání a nastavení proměnných, které tvoří složky vektoru, slouží getter a setter definovaný ve třídě `Promenna_vektor`. S vektory, na rozdíl od polí, také dokáží pracovat některé instrukce (sčítání, násobení, ...).

Proměnná typu komponenta slouží k uchování strukturovaných dat. V současné době proměnné typu komponenta reprezentují pouze struktury z knihovny Rayleigh. Konkrétně se jedná o optický člen a světelný zdroj. U jednotlivých podtříd, představujících typ optického členu nebo zdroje světla, je nutné provázat datové typy reprezentované ve strukturách knihovny Rayleight a převést je na objekty proměnných, jež využívá interpret a ke kterým lze přistoupit přes tečkovou notaci ve skriptech. Ve funkcích je možné využít i přímý přístup ke struktuře z knihovny Rayleigh, která je v proměnné též uložena.

Kromě tříd proměnných, se kterými se pracuje při interpretaci, jsem vytvořit ještě pomocnou třídu určenou pro předzpracování (třída `Promenna_preexec`). Instance třídy jsou využívány jen při předzpracování (a jsou ukládány v pomocném zásobníku interpretu určenému pouze pro předzpracování). Jejich účelem je shromažďovat informace o použití proměnné, tedy indexy instrukcí, kde byly poprvé a naposledy použity. Z těchto informací lze pak do kódu vložit pomocné instrukce pro uvolnění těch proměnných, které již nadále nebudou využity.

Tímto lze uzavřít přehled o proměnných. Ještě se drobně zmíním o funkcích, které lze ve skriptech volat. Jak bylo řečeno v části překladače, překladač potřebuje názvy funkcí (ze souboru `funkce.cpp`), které mohou být volány. Interpret poté potřebuje výkonný kód funkcí. Tyto funkce mají za úkol získat ze zásobníku interpretu objekty proměnných, které představují parametry funkce, ověřit jejich typy a poté provést požadovanou funkčnost. Ve většině případů je touto funkčností zavolání funkce z knihovny Rayleigh

s parametry (získanými z objektů proměnných), jež jsou vyžadovány. Právě tyto funkce je nutné vytvořit, aby bylo možné volat funkce ve skriptech. Některé z těchto funkcí jsou již nyní připravené a při vytvoření dalších funkcí lze využít podobných postupů. Další detaily o vytváření nových funkcí je možné získat z přílohy B.

5.6.3 Funkce main

Zde se ještě zmíním o poslední neprobrané části, tedy hlavní funkci `main` prováděnou při spuštění programu. V této funkci je nutné zajistit kompletní běh programu, tedy nejprve přeložit vstupní skript do mezikódu interpretu a poté tento mezikód analyzovat a interpretovat.

Program by měl být spuštěn přesně s jedním parametrem - souborem vstupního skriptu. Tento soubor je poté předán lexikální části překladače jako vstupní soubor. Následně je zavolána funkce pro zpracování skriptu (tedy syntaktická část, jež si jednotlivá slova bere od lexikální části a dle gramatiky generuje mezikód). Po zpracování skriptu a převedení do mezikódu je možné vytvořit interpret (pracující nad vygenerovaným mezikódem). Interpret poté bude provádět instrukce mezikódu (určené indexem registru „pc“). Interpretace pokračuje, dokud není interpretována poslední instrukce (tedy dokud index „pc“ ukazuje do pole mezikódu). Tímto celý program končí

6 Otestování funkčnosti

Tato kapitola je věnována otestování vytvořeného programu. Testování programu lze rozdělit do 3 kategorií: otestování překladu skriptu dle gramatiky do mezikódu, otestování správnosti vykonávání mezikódu a otestování provázání s knihovnou Rayleigh.

Pro samotné otestování jsem použil dva způsoby testování (nebo kombinace obou). Testování pomocí debugovacích výpisů a testování spuštěním skriptů a otestováním výsledků skriptů. Debugovací výpisy lze nyní v programu zapnout přidáním druhého parametru „d“ nebo „d2“. Parametr „d“ zajistí vypsání instrukcí vygenerovaného mezikódu a drobných detailů o předzpracování. Parametr „d2“ navíc zajistí rozšíření debugovacích výpisů o vypisování prováděných instrukcí.

6.1 Otestování překladu do mezikódu

Při otestování překladu do mezikódu jsem využíval debugovacího výpisu (konkrétně pro výpis vygenerovaného mezikódu). Snažil jsem se otestovat veškeré možné konstrukce a zkontrolovat, zda vygenerovaný mezikód odpovídá daným konstrukcím. Je nutné zaměřit se na udržení zásobníku v konzistentním stavu (tedy aby po každé konstrukci byl zásobník ve stejném stavu jako před vykonáním první instrukce konstrukce). Mezi to patří například odstranění proměnných vygenerovaných v bloku. Také po vykonání instrukcí výrazu musí zůstat pouze jedna hodnota (které je po „;“ odstraněna). Mezi další kritickou oblast patří generování cyklů a podmínek.

Zde uvedu příklad skriptu, ve kterém jsou tyto oblasti obsaženy, a jeho překladu do mezikódu. Do skriptu jsem zařadil některé konstrukce, které bude snadné identifikovat v mezikódu.

```
int i, j;
j=0;
for(i = 0; i < 10; i++){
    if(i%2 == 0) {
        int k[10];
        j++;
    }
}
```

Výpis mezikódu pro tento skript je následující:

```
00000    ALLOC typ int
00001    ALLOC typ int
00002     PUSH typ odkaz na promennou:
      promenna na pozici: 1
      pocet indexu do pole: 0
00003     PUSH typ int: 0
00004    STORE zacatek prirazeni: 00003
00005      DEL uvolneni zasobniku: pocet 1
00006     PUSH typ odkaz na promennou:
      promenna na pozici: 0
      pocet indexu do pole: 0
00007     PUSH typ int: 0
00008    STORE zacatek prirazeni: 00007
00009      DEL uvolneni zasobniku: pocet 1
00010     PUSH typ odkaz na promennou:
      promenna na pozici: 0
      pocet indexu do pole: 0
00011     PUSH typ int: 10
00012       LS
00013      JMZ skok na 00032
00014      JMP skok na 00019
00015     PUSH typ odkaz na promennou:
      promenna na pozici: 0
      pocet indexu do pole: 0
00016      INC
00017      DEL uvolneni zasobniku: pocet 1
00018      JMP skok na 00010
00019     PUSH typ odkaz na promennou:
      promenna na pozici: 0
      pocet indexu do pole: 0
00020     PUSH typ int: 2
00021      MOD
00022     PUSH typ int: 0
00023    EQUAL
00024      JMZ skok na 00031
00025     PUSH typ int: 10
00026    ALLOC typ: 41, indexu: 1
00027     PUSH typ odkaz na promennou:
      promenna na pozici: 1
      pocet indexu do pole: 0
00028      INC
00029      DEL uvolneni zasobniku: pocet 1
00030      DEL uvolneni zasobniku: pocet 1
00031      JMP skok na 00015
00032      DEL uvolneni zasobniku: pocet 2
```

Pro konstrukci definic proměnných `i` a `j`, vytvořených v hlavním bloku, slouží instrukce `ALLOC` s indexy 0 a 1. Uvolněné jsou instrukcí `DEL` s indexem 32. Hlavní blok je tedy konzistentní. Další bloky jsou blok cyklu for a blok podmínky. V bloku cyklu nejsou definovány žádné proměnné, tudíž není nutné nic uvolňovat. V bloku podmínky je definováno pole instrukcemi 25-26 a uvolněno instrukcí 30. I tento blok je tedy v pořádku.

Výraz `j=0;` začíná instrukcí 2 a končí instrukcí 5. Během této konstrukce jsou 2 hodnoty vloženy do zásobníku (instrukcemi 2 a 3) a též 2 odstra-

něny (instrukcemi 4 a 5). Tato konstrukce tedy také ponechává zásobník v konzistentním stavu. Další z výrazů je výraz podmínky `i%2 == 0`, kterému odpovídají instrukce 19-23. Zde je zanechána jedna hodnota, které bude využita instrukcí skoku 24, tedy celý výraz je v pořádku. Dalším výrazem je `j++;` (instrukce 27-29) též v pořádku.

Dále je nutné, aby cykly a podmínky generovaly skoky na správné indexy instrukcí jednotlivých částí. U podmínky musí podmíněný skok přeskočit blok (19-30) při nesplněné podmínce, tedy podmíněný skok z instrukce 24 na 31 je v pořádku. U cyklu je nutné identifikovat jednotlivé části: inicializace 6-9, podmínka 10-14, inkrementace 15-18 a blok 19-31. Při nesplnění podmínky je skočeno na konec bloku (z instrukce 13 skok na 32), při splnění skočeno do bloku (z instrukce 14 skok na 19). Za koncem bloku musí následovat inkrementace (skok z 31 na 15) a po inkrementaci podmínka (skok z 18 na 10). Všechny skoky odkazují na správná místa a kód je tedy v pořádku.

Obdobným způsobem jsem otestoval i další možné konstrukce (volání funkcí, přiřazení do polí, ...).

6.2 Otestování správnosti vykonávání

Testování správnosti vykonávání jsem provedl pomocí napsání a spuštění většího množství skriptů, zaměřených na různé činnosti, s výpisem (pomocí funkce `print`) výsledných hodnot a jejich porovnání s očekávaným výsledkem. Některé z těchto skriptů jsou umístěny na přiloženém CD. Některé z příkladů jsem testoval i rozšířením debugovacích výpisů (které jsou již nyní z programu odstraněny), například pro otestování volání funkcí.

U příkladu z kapitoly 6.1 na test překladu, je možné předpokládat hodnotu proměnné `j` rovnu 5 (jelikož celkem je cyklus vykonán 10x a každý sudý průchod inkrementuje proměnnou). Je tedy možné za konec skriptu přidat řádek

```
print(j);
```

a vykonáním tohoto skriptu a porovnáním hodnoty toto ověřit. Na podobném principu jsem testoval mnoho dalších situací (přiřazení hodnot, správnost vyhodnocení výrazů, přiřazení do polí, zjednodušené plnění polí, ...).

6.3 Otestování provázání s knihovnou

Pro otestování správného provázání s knihovnou jsem napsal skript, popisující scénu, a poté jsem napsal program v jazyce C, který popisoval stejnou scénu. V této scéně se jednalo o záznam interference od zdrojů světla. Následně byla tato interference uložena do obrázkového souboru. Konkrétně se jednalo o tento program (funkce `main`):

```
t_lightsArray *lights = NULL;
t_optfield *hologram = NULL;
int numLights = 50;
rayleighSetEnvironment(532 NM, 5 UM);
lightsArrayCreate(&lights, 1000);
lightsGenerateLine(lights,
    -1 MM, 0, -300 MM,
    1.0 / numLights, 0,
    1 MM, 0, -300 MM,
    1.0 / numLights, 0,
    1,
    0.5,
    numLights, 1);
optfieldNew(0, 0, 0, 512, 512, 1, SAMPLE_TYPE_FFTWDOUBLE, BUFFER_TYPE_MAIN
            , NULL, &hologram);
lightsArrayPropagate(lights, hologram);
optfieldSavePNG(hologram, (char*)"01_object_wave", PICTURE_REALIMAG, 1.0, 0
            , (char*)"The object wave.");
```

a tento skript:

```
int numLights;
numLights = 50;
rayleighSetEnvironment(532 *1e-9, 5 *1e-6);
lightsource lights;
component hologram;
lightsArrayGenerateLine(lights,
    [-1e-3,0,-300e-3],
    1.0 / numLights, 0,
    [1e-3,0,-300e-3],
    1.0 / numLights, 0,
    1,0.5,
    numLights, 1);
hologram.width = 512;
hologram.height = 512;
hologram.center = [0,0,0];
lightsArrayPropagate(lights, hologram);
optfieldSavePNG(hologram, "object_wave", "PICTURE_REALIMAG", "The object wave.");
```

Skript jsem spustil v interpretu a program jsem přeložil pomocí překladače C++ a následně spustil. Od interpretu i od programu v C jsem získal stejný výsledek a tedy interpret správně připravil proměnné a provedl správné volání funkcí. Skripty, stejně jako výsledné obrázky, lze nalézt na přiloženém CD. Tímto jsem tedy ověřil správnost provázání s knihovnou Rayleigh.

7 Vytváření vlastních skriptů

Tato kapitola je věnována možnostem navrhovaného jazyka knihovny Rayleigh a základním postupům při vytváření vlastních skriptů.

Nejprve zde popíšu strukturu skriptů. Každý skript se skládá z bloků. Hlavní blok je tvořen celým skriptem. Lze vkládat i vnořené bloky ohraničené znaky „{“ a „}“. Do bloků lze zapisovat jednotlivé příkazy. Těmi může být definování nových proměnných, výrazy, volání funkcí, podmínky a cykly. Jednotlivé příkazy jsou ukončeny středníkem (`;`). V celém skriptu lze též využít komentáře.

Ve skriptech lze využít dva druhy komentářů: řádkové a blokové. Řádkový komentář začíná znaky `//` a končí koncem řádku. Blokové komentáře jsou vymezeny mezi znaky `/*` a `*/` a mohou být i víceřádkové.

Před použitím proměnných je nutné ji nejprve definovat. Při definování nových proměnných je nutné zapsat typ a za něj pojmenování jednotlivých proměnných. Příklad užití ve skriptu je následující:

```
int i,j,k;
```

Tímto jsme si připravili 3 proměnné (`i`, `j` a `k`) typu celé číslo pro následné využití.

Definované proměnné mají ovšem určitou platnost. Začít využívat proměnné (ve výrazech) lze po jejich definici a jejich platnost končí po ukončení bloku, ve kterém byly definovány. Zde názorný příklad:

```
int i;//deklarace v hlavním bloku
{int j;
//nějaké operace s proměnnými i a j
}
//zde již proměnná j není přístupná
{int j;
/* další operace s proměnnými i a j, j je ale jiná proměnná
   než v bloku výše */
}
```

Celý tento skript je složen ze 3 bloků, první je hlavní blok (kde je definována proměnná `i`), dále je zde první vnořený blok (definována proměnná `j`) a druhý vnořený blok (opět definována proměnná `j`, ale jedná se o rozdílnou proměnnou než tu, užitou v prvním vnořeném bloku). Proměnnou `i` lze

použít v celém skriptu. Proměnnou `j` z prvního vnořeného bloku lze použít pouze v tomto bloku (tedy po ukončení již tato proměnná neexistuje). Ve druhém vnořeném bloku je proměnná `j` opět definována, ale jedná se o jinou proměnnou s odlišnou hodnotou než v předchozím bloku. Komentáři je naznačeno, že s proměnnými lze v jednotlivých blocích dále pracovat.

7.1 Proměnné a výrazy

Ve výše zmíněných ukázkových příkladech jsem využíval pouze celočíselné proměnné (`int`), ale typů proměnných je samozřejmě více. Na typu proměnné také záleží, jak lze s proměnnou pracovat (do jakých výrazů ji lze použít). Dále se tedy zaměřím na další typy proměnných, které lze ve skriptech užít, a také ve kterých výrazech je možné proměnné použít.

7.1.1 Základní typy

Ve skriptech lze použít 4 základní typy proměnných - celé číslo (`int`), desetinné číslo (`double`), komplexní číslo (`complex`), řetězec (`string`). Proměnnou je nutné nejprve definovat a následně ji lze použít ve výrazech. Za výraz lze považovat i pouhé přiřazení hodnoty. Dle typu proměnné lze přiřadit i hodnotu - proměnným typu `int` celá čísla, typu `double` desetinná čísla, typu `complex` komplexní čísla a typu `string` řetězce. Příklad přiřazení do celočíselné proměnné:

```
int cislo;
cislo = 12;
```

Podobným způsobem lze přiřadit i desetinná čísla. Desetinná čísla lze zapisovat i použitím exponenciálního tvaru, například:

```
double desetinne;
desetinne = 1.89e-3;
```

Přiřazení hodnoty komplexnímu číslu je podobné, jako přiřazení desetinného čísla, pouze imaginární část je nutné vynásobit imaginární jednotkou (ve skriptech definována pomocí `%i`). Příklad:

```
complex komplexni;
komplexni = 11+22*%i;
```

Ze základních typů zbývají ještě řetězce. Hodnotu řetězce je nutné uzavřít do uvozovek. Příklad:

```
string retezec;
retezec = "tohle je retezec";
```

V přiřazovacích výrazech je možné použít i složitější výrazy použitím operátorů matematických operací. Pro číselné základní typy (`int`, `double`, `complex`) jsou definovány základní matematické operace: sčítání (`+`), odčítání (`-`), dělení (`/`), násobení (`*`), umocnění (`**`). Pro celočíselný typ je definována i operace modulo (`%`). Pro číselné typy jsou též definovány některé matematické funkce (jejichž výsledkem je desetinné, nebo komplexní číslo, dle vloženého parametru). Definované funkce jsou: sin(), cos(), tan(), cotg(), asin(), acos(), atan(), acotg(), exp() a ln(). Kromě matematických funkcí lze ve výrazech volat i ostatní funkce (zmíněné v kapitole 7.3), které mají návratovou hodnotu.

Matematické operace mají různé priority: nejvyšší prioritu má umocnění, poté násobení a dělení, a nejnižší prioritu má sčítání a odčítání. Ve výrazech je možné využít závorky, a tím zajistit požadované pořadí operací. Složením výše zmíněných operací je možné vytvořit jakkoli složitý přiřazovací výraz obsahující jak konstanty, tak i proměnné. Příkladem přiřazovacího výrazu může být:

```
komplexni = (cislo + 2.5 * cislo2 ** (cislo+1))
      + ((cislo2 % cislo) - sin(5.5 / desetinne)) * %i;
```

Kromě matematických operací lze využít i booleovské operace. Booleovské operace slouží k porovnání dvou hodnot a získání booleovské (pravdivostní) hodnoty. Booleovské hodnoty mají typ `int`. Kladná hodnota (true) je rovna číselné hodnotě 1, záporná hodnota (false) je rovna číslu 0. Pro číselné základní typy jsou definovány booleovské operace: je rovno (`==`), není rovno (`!=`), větší než (`>`), větší nebo rovno (`>=`), menší než (`<`), menší nebo rovno (`<=`). Mezi booleovskými hodnotami jsou poté definovány operace: logické násobení (`&&`), logický součet (`||`) a logická negace (`!`). Booleovské hodnoty jsou hlavně nutné pro podmínky a cykly (zmíněné v kapitole 7.2).

Ještě nebyly zmíněné operace mezi řetězci. S řetězci lze provádět pouze operaci sloučení (`+`) a porovnávat dva řetězce lze pouze na rovnost (`==`) či nerovnost (`!=`).

Při operacích mezi různými typy hodnot se uplatňují priority typů v pořadí (vzestupně): `int`, `double`, `complex`, `string`. Pokud například bude operace sčítání mezi celým číslem a komplexním, výsledkem bude komplexní číslo. V případě, že jedním z operandů bude řetězec, bude i druhý operand převeden na řetězec. V případě zapsání operací mezi nekompatibilními typy (např. dělení řetězců), dojde při interpretaci instrukcí představující operaci chybě, které je uživateli oznámena, a interpretace je přerušena.

7.1.2 Vektory

Proměnné typu vektor slouží hlavně k použití jako matematické vektory (tedy veličiny, které mají kromě velikosti i směr). Typ vektor se definuje pomocí klíčového slova `vector`. Vektor lze vytvořit tak, že se mezi hranaté závorky (`[` a `]`) napíše seznam hodnot oddělených čárkou. Příklad:

```
vector v;
v = [25.5, 26.6, 27.7];
```

Mezi dvěma vektory jsou definovány operace sčítání (+) a odčítání (−). Mezi vektorem a číslem jsou definovány operace násobení (∗) a dělení (/). Příklad operací s vektory:

```
v = 2*[1,2,3] + [0.1,0.2,0.3]/2;
```

Pokud je nutné získat jednotlivé složky vektorů, je možné přistoupit k jednotlivým složkám přes index zapsaný v hranatých závorkách. Index je celé číslo (začínající od 0, takže první složka má index 0). Příklad získání složek z vektoru:

```
vector v;
double x, y;
v = [25.5, 26.6, 27.7];
x = v[0];
y = v[1];
```

7.1.3 Optické proměnné

Optické proměnné jsou základem při optických simulacích. Ve skriptech lze využít dva druhy optických proměnných: světlo (`lightsource`) a optický člen (`component`).

Proměnná typu světlo představuje pole bodových světelných zdrojů v simulaci. U těchto světelných zdrojů je nutné definovat jejich vlastnosti (jako amplituda, fáze). Pro nastavení potřebných parametrů světelným zdrojům jsou využity funkce (například `lightsArrayGenerateLine`) zmíněné v kapitole 7.3.

U proměnné typu světlo je jeden parametr - maximální počet světelných zdrojů. Tento parametr je původně nastaven na 1000. V případě nutnosti, lze parametr změnit přes podproměnou `maxlights`, ke které je možné přistoupit přes tečkovou notaci. Parametr lze změnit pouze před prvním použitím proměnné ve funkcích.

Druhým typem je proměnná optického členu. Optický člen představuje rovinu, kde jsou zaznamenány vlastnosti světla. Tyto vlastnosti lze definovat buď přímo, nebo je lze vypočítat z ostatních zdrojů světla. Optický člen může také dále působit jako nový zdroj světla. Pro nastavení parametrů u optického členu slouží podproměné, ke kterým lze přistoupit přes tečkovou notaci. Použitelné podproměné jsou: umístění středu (`center`), umístění rohu (`corner`), skutečná velikost (`size`), vzorkování (`sampling`), výška a šířka v pixelech (`height` a `width`), obsah optického členu (`content`).

U optického členu je nutné nastavit výšku a šířku (`height` a `width`) 2D pole, reprezentující optický člen. Tyto parametry je nutné nastavit jako první, a až po nastavení těchto parametrů je možné nastavit další. Některé parametry jsou vzájemně provázány (například po nastavení pozice středu je změněna i pozice rohu). Parametry umístění středu (`center`) a umístění rohu (`corner`) jsou typu vektor se třemi složkami (odpovídajícím souřadnicím). Parametr skutečné velikosti (`size`) je typu vektor se dvěma složkami - šířka a výška. Parametr skutečné velikosti se odvíjí od velikosti 2D pole, reprezentující člen, a vzorkovací vzdálenosti. Vzorkovací vzdálenost u optického členu je celočíselným násobkem základní vzorkovací vzdálenosti nastavené při inicializaci knihovny (pomocí funkce `rayleighSetEnvironment` zmíněné v kapitole 7.3). Parametr vzorkovací vzdálenosti (`sampling`) v optickém členu je tedy celé číslo, určující násobek základní vzorkovací vzdálenosti. Poslední podproměnnou je `content`, přes kterou lze přistoupit ke 2D poli, kterým je optický člen reprezentován. Obsah tohoto pole lze změnit jako u ostatních proměnných typu pole (zmíněné v kapitole 7.1.4).

Příklad vytvoření optického členu a nastavení některých hodnot ve skriptu:

```
component hologram;
hologram.height = 512;
```

```
hologram.width = 512;
hologram.center = [0,1.2e-3,0];
hologram.sampling = 1;
```

7.1.4 Pole

Pole slouží k uchování většího množství proměnných stejného typu. Při definici proměnné typu pole je nutné udat velikost pole. Pole mohou být i vícerozměrná. Velikosti pole se zapisují za jméno proměnné v definici mezi hranaté závorky (`[` a `]`), kde velikosti jednotlivých dimenzí jsou odděleny čárkou. Příklad definice dvourozměrného pole celých čísel:

```
int pole[10,20];
```

Jedním ze způsobů, jak do pole vložit hodnoty, je vkládání jednotlivých hodnot s použitím indexů do pole. Jednotlivé indexy jsou opět zapsány mezi hranaté závorky (`[` a `]`) oddělené čárkou. Tento způsob naplnění pole je možné kombinovat s cykly (zmíněné v kapitole 7.2). V polích se indexuje od indexu 0. Příklad přiřazení hodnot bez využití cyklů:

```
pole[0,0] = 11;
pole[0,1] = 12;
```

Druhým ze způsobů plnění polí je zjednodušené plnění. Při tomto plnění je možné vytvořit výraz s použitím předdefinovaných proměnných %row a %col. Pole je poté naplněno tak, že pro každý prvek pole je tento výraz vyhodnocen zvlášť a předdefinovaným proměnným je nastavena hodnota, která se vztahuje ke zpracovávanému prvku. Za proměnnou %row je dosazen index řádky prvku a za %col je dosazen index sloupce prvku. Tyto předdefinované proměnné jsou určené pro dvourozměrné pole, kde %row (řádka) odpovídá prvnímu indexu a %col (sloupec) odpovídá druhému indexu. U více rozměrných polí tyto předdefinované proměnné fungují pouze pro první dva indexy. Jelikož u jednorozměrných polí je pouze jeden index, je možné využít jak proměnnou %row, tak i %col, a obě mají nastavenu stejnou hodnotu - index prvku. Příklad užití předdefinovaných proměnných pro plnění pole:

```
complex komplexnipole[100,100];
komplexnipole = %row*0.1 + %col*0.1*%i;
```

U pole v optickém členu je možné navíc využít předdefinované proměnné %x a %y. Tyto předdefinované proměnné představují skutečnou pozici zpracovávaného prvku pole v rovině, která optický prvek reprezentuje. Pozice je odvozena z umístění optického členu a vzorkovací vzdálenosti u tohoto členu. U optického členu je velikost 2D pole odvozena od výšky a šířky optického členu. Pozice jednotlivých prvků na rovině je ovlivněna umístěním optického členu. Před přístupem k obsahu optického členu je nutné nastavit velikost a umístění. Příklad naplnění pole u optického členu s použitím předdefinovaných proměnných umístění:

```
component transmittance;
transmittance.height = 1000;
transmittance.width = 1000;
transmittance.center = [0,1.2e-3,0];
transmittance.content = sin(%x*2*%pi+%y*4*%pi);
```

7.2 Cykly a podmínky

Ve skriptech je možné větvit kód pomocí podmínek a opakovat části skriptu pomocí cyklů. V podmínkách a podmínkové části cyklů je nutné využít výrazy, jejichž výsledkem je booleovská hodnota (zmíněné v kapitole 7.1.1).

Podmínkový příkaz začíná klíčovým slovem `if`, následuje podmínkový výraz v závorkách a poslední částí je blok vykonávaných příkazů (ohraničený `{` a `}`). To tvoří jednoduchou podmínku, kde blok příkazů je vykonán při splněné podmínce a při nesplněné podmínce je přeskočen. Příklad jednoduché podmínky:

```
if((a % 2)==0){
    print("sude");
}
```

Podmínka může být i rozšířena o klíčové slovo `else` a blok `else`. U rozšířené podmínky je při splněné podmínce vykonán blok `if`, v opačném případě je vykonán blok `else`. Příklad rozšířené podmínky:

```
if((a % 2)==0){
    print("sude");
}else{
    print("liche");
}
```

Příkaz cyklu začíná klíčovým slovem `for`, v závorkách následují 3 části

oddělené středníkem - inicializace, podmínka a inkrementace. Poslední částí je blok akcí, prováděný dokud je podmínka splněna. Příklad cyklu `for` plnící pole čísel:

```
int i;
double pole[100];
for(i = 0; i < 100; i++){
    pole[i] = sin(2*%pi/100);
}
```

7.3 Funkce

Ve skriptech není možné vytvářet vlastní funkce, je možné pouze volat předpřipravené funkce. Tyto funkce jsou hlavně zaměřeny na poskytnutí funkčnosti z knihovny Rayleigh. V současné době je možné ve skriptech užít některé funkce, ale v budoucnu můžou být přidány další funkce (a i upraveny ty existující), aby byl poskytnut plný přístup ke knihovně Rayleigh.

Mimo funkce pokrývající funkčnost knihovny je možné ve skriptech užít funkci pro vypsání textu (`print`) a funkci pro zjištění velikosti pole (`size`). Parametry pro funkce jsou psány za názvem funkce v závorkách. Například u funkce `print` jednotlivé parametry představují, co se má vypsat (mohou to být konstanty i proměnné, je možné využít i výrazy):

```
print("text", 123, "promenna je: "+cislo);
```

Funkce `print` je určena pro všechny typy parametrů, a počet může být libovolný (každý je vypsán na jednu řádku), ostatní funkce ale většinou vyžadují předem daný počet a dané typy parametrů. U funkce `size` jsou vyžadovány přesně 2 parametry - prvním je pole, jehož velikost chceme zjistit, a druhým je číslo dimenze, jehož rozměr požadujeme. Číslo dimenze, stejně jako indexy pole, je číslováno od 0. Funkce s návratovou hodnotou (jako je funkce `size`) lze též použít jako parametry dalších funkcí, například:

```
int pole[10,20];
print( size(pole, 0) );
```

Výsledkem tohoto skriptu by bylo vypsání čísla 10 (první rozměr pole).

Dále následují funkce pro přístup k funkcím knihovny. Všechny tyto funkce jsou v současné době bez návratové hodnoty a vyžadují parametry daných typů, které budou upřesněny u každé funkce.

První z těchto funkcí je funkce pro inicializaci knihovny Rayleigh: `rayleighSetEnvironment`. Tuto funkci je nutné zavolat před voláním dalších funkcí a nebo před využíváním optických členů, jelikož funkce nastavuje základní vzorkovací vzdálenost pro celou simulaci. Jak jsem již zmínil v kapitole 7.1.3, u optických členů může být vzorkovací vzdálenost pouze celočíselným násobkem této základní vzdálenosti. Kromě vzorkovací vzdálenosti je také funkcí nastavena vlnová délka pro světlo v simulaci. Funkce vyžaduje 2 parametry typu desetinné číslo (`double`) - prvním je vlnová délka světla použitém při optických simulacích, druhým parametrem je základní vzorkovací vzdálenost. Příklad volání funkce ve skriptu:

```
rayleighSetEnvironment(532 *1e-9, 5 *1e-6);
```

Dále je zde funkce pro vygenerování světel umístěných na „úsečce“ `lightsArrayGenerateLine`. Pomocí této funkce je možné do proměnné světel vygenerovat několik bodových světelných zdrojů. Funkce vyžaduje 11 parametrů. Prvním parametrem je proměnná světel (typu `lightsource`), do které budou světla vygenerována. Dalším parametrem je pozice začátku „úsečky“, kterou budou světla tvořit. Tento parametr je typu vektor čísel o třech složkách, představující souřadnice umístění. Parametry 3 a 4 jsou amplituda a fáze světelných zdrojů na začátku „úsečky“, parametry jsou typu desetinné číslo. 5., 6. a 7. parametr jsou pozice, amplituda a fáze světelných zdrojů na konci „úsečky“, podobně jako pro začátek „úsečky“. 8. parametr je inicializační hodnota (`seed`) generátoru náhodných čísel. 9. parametr je náhodnost fáze u generovaných světel. 10. parametr je počet vygenerovaných bodových zdrojů. Poslední (11.) parametr je booleovská hodnota (tedy 1 pro ano, 0 pro ne), zda má být vygenerován i koncový bod. Příklad volání funkce ve skriptu:

```
lightsArrayGenerateLine(lights,//proměnná světel
    [-1e-3,0,-300e-3],//pozice začátku
    1.0 / 50, 0, //amplituda a fáze na počátku
    [1e-3,0,-300e-3], //pozice konce
    1.0 / 50, 0, //amplituda a fáze na konci
    1,0.5, //seed pro random a náhodnost fáze
    50, //počet vygenerovaných světel
    1); //má být vygenerován poslední bod?
```

Funkce pro zkopírování optického členu `optfieldCopy`. Funkce vyžaduje 2 parametry. Prvním parametrem je proměnná cílového optického členu,

druhým parametrem je proměnná zdrojového optického členu. Funkce zkopíruje vlastnosti zdrojového členu do cílového. Příklad volání funkce ve skriptu:

```
optfieldCopy(hologram2, hologram);
```

Funkce pro propagování světelných vln ze světel na optické členy `lightsArrayPropagate`. Funkce vyžaduje 2 parametry. Prvním parametrem je proměnná světel (`lightsource`) a druhým parametrem je cílový optický člen (`component`). Příklad volání funkce ve skriptu:

```
lightsArrayPropagate(lights, hologram);
```

Poslední z funkcí je funkce pro uložení 2D pole, reprezentující optický člen, jako obrázek - `optfieldSavePNG`. Funkce vyžaduje 2 až 4 parametry. Prvním parametrem je optický člen (`component`), jenž bude uložen do souboru. Druhým parametrem je název souboru (řetězec). Následující parametry jsou nepovinné (nemusejí být tedy zadány). Třetím parametrem je, jakým způsobem bude uložen obsah 2D pole. Bez zadání je uložena jak reálná, tak imaginární část. Parametrem může být jedna z následujících řetězcových konstant: `PICTURE_INTENSITY` (uloží do obrázku intenzitu), `PICTURE_PHASE` (uloží do obrázku fázi), `PICTURE_INTENSITYPHASE` (uloží do obrázku jak intenzitu tak fázi), `PICTURE_REAL` (uloží do obrázku reálnou část), `PICTURE_IMAG` (uloží do obrázku imaginární část) a `PICTURE_REALIMAG` (uloží do obrázku jak reálnou, tak imaginární část). Posledním parametrem je poznámka, uložená do popisu obrázku.

```
optfieldSavePNG(hologram, "01_object_wave",
    "PICTURE_INTENSITYPHASE", "Objektova vlna.");
```

Výběr implementovaných funkcí jsem zvolil náhodně a vybral jsem funkce tak, abych mohl otestovat interpret napsáním některých testovacích skriptů. Další funkce, které by poskytly přístup k dalším částem knihovny, je nutné doprogramovat do interpretu (při doprogramovávání funkcí lze využít návod z přílohy B).

7.4 Ukázkový skript

V této části uvedu jeden celkový skript. V následujícím skriptu je popsána scéna, kde jsou vygenerovány světelné zdroje umístěné na „úsečce“. Následně jsou tyto světla využity k osvíceni 10 hologramů, které jsou postupně oddalovány. Do jednoho souboru je uloženo 2D pole, reprezentující optický člen, a do druhého je uložena intenzita a fáze. Ze získaných výsledků lze vypozorovat, že fáze se i při posunu o $10\mu m$ změní (zatímco intenzita příliš ne). Na záznam skutečného hologramu, který trvá určitý čas, by ale tyto posuny (vibrace) způsobily poškození zaznamenávaného hologramu. V simulaci by bylo nutné nakonec všechny tyto vytvořené hologramy sečíst, aby se docílilo podobného účinku, jako při vibracích v reálném světě.

```
int numLights, i;
numLights = 50;
rayleighSetEnvironment(532 *1e-9, 5 *1e-6);
lightsource lights;
lightsArrayGenerateLine(lights,
    [-1e-3,0,-300e-3],
    1.0 / numLights, 0,
    [1e-3,0,-300e-3],
    1.0 / numLights, 0,
    1,0.5,
    numLights, 1);
for(i = 0; i < 10; i++){
    component hologram;
    hologram.width = 512;
    hologram.height = 512;
    hologram.center = [0,0,0]+i*[0,0,1e-8];
    hologram.sampling = 1;
    lightsArrayPropagate(lights, hologram);
    optfieldSavePNG(hologram, "object_wave_"+i,
            "PICTURE_REALIMAG", "Objektova vlna");
    optfieldSavePNG(hologram, "intensity_phase_"+i,
            "PICTURE_INTENSITYPHASE", "Intenzita a faze");
}
```

Tento skript, včetně vygenerovaných souborů, je možné nalézt na přiloženém CD.

7.5 Spuštění skriptů

V předchozích částech této kapitoly jsem se věnoval způsobu vytváření skriptů. Nyní ještě zbývá poslední věc - spuštění vykonávání vytvořeného skriptu.

V první řadě je nutné mít k dispozici program interpretu. V případě, že je nutné přeložit si interpret ze zdrojových souborů, je možné využít návod k přeložení z přílohy A. Je také možné využít program uložený na CD této práce. Poté, co již máme program interpretu a také vstupní skript, je možné spustit interpretaci. Jméno souboru skriptu je nutné vložit jako parametr při spouštění programu interpretu. Příklad spuštění z příkazové řádky:

```
interpret skript.txt
```

V tomto příkladu je skript, který chceme spustit, uložen v souboru `skript.txt`.

Po spuštění skriptu bude zahájena interpretace. Pokud ve skriptu není žádná chyba, při interpretaci bude vykonán celý skript. V případě chyby při překladu bude uživateli zobrazen popis chyby a číslo řádku, kde chyba nastala, a v tomto případě vůbec nezačne interpretace. Pokud ve skriptu bude taková chyba, která překladu nebude bránit, ale dojde k chybě až při vykonávání (například index pole bude mimo rozsah), po pokusu provedení této části skriptu dojde k přerušení celé interpretace (i v tomto případě bude vypsán popis chyby).

8 Závěr

Hlavním cílem této práce bylo zpřístupnění knihovny neprogramátorům. Jako způsob zpřístupnění jsem zvolil naprogramování skriptového rozhraní. Kvůli tomu jsem se seznámil s možnostmi návrhu vlastních jazyků a s nástroji pro vytváření překladačů. Následně jsem navrhl skriptovací jazyk pro knihovnu Rayleigh. Po návrhu jazyka jsem musel vytvořit překladač a interpret, jenž mají za úkol interpretovat vytvořené skripty v navrženém jazyce.

Jazyk jsem se snažil navrhnout tak, aby bylo jednoduché v něm psát skripty. Tento jazyk by ale bylo možné v budoucnu ještě více zjednodušit, jako například odstranit nutnost definovat typ při definicích proměnných, případně úplně odstranit nutnost definovat proměnné. Některá zjednodušení již ale poskytuje i stávající verze jazyka, jako je například zjednodušené plnění polí.

U interpretu jsem věnoval zvýšenou pozornost správě paměti, jelikož optické simulace vyžadují velké množství paměti. Pro tento účel jsem navrhl předzpracování vytvořeného mezikódu pro interpret, kde se zjistí rozsah, kdy jsou proměnné používány, a jakmile již nejsou proměnné zapotřebí, jsou uvolněny (včetně paměti, kterou potřebovali).

Samotný program interpretu jsem se snažil naprogramovat tak, aby byl snadno rozšiřitelný. Lze například doplnit nové funkce funkce (dle návodu v příloze B), které umožní přístup i k dalším funkcím knihovny Rayleigh, které nyní nejsou přístupné. Tím, že je program snadno rozšiřitelný, lze zajistit, že až se rozšíří knihovna Rayleigh o další funkčnost, bude možné rozšířit i skriptovací jazyk.

Kromě návrhu a vytvoření programu interpretu jsem musel vytvořený program otestovat a odladil všechny nalezené chyby. Při testování jsem některé skripty navrhl tak, aby odpovídali některým optickým úlohám (jako je záznam světelných vln od zdrojů).

Zadání práce jsem tedy naplnil. Program je i nadále možné upravovat, aby pokryl všechny možnosti knihovny, nebo více přizpůsobit potřebám optiků.

Literatura

[1] *GLAD Examples Manual*. [Online].
URL http://www.aor.com/anonymous/pub/examples.pdf

[2] *GNU Bison - The Yacc-compatible Parser Generator*. [Online].
URL http://www.gnu.org/software/bison/manual/

[3] *Lexical Analysis With Flex*. [Online].
URL http://flex.sourceforge.net/manual/

[4] Algoritmy.net: *Konstrukce překladače*. [Online].
URL http://www.algoritmy.net/article/100/Konstrukce-prekladace

[5] Benton, S.; Bove, V.: *Holographic Imaging*. John Wiley & Sons, 2008, ISBN 9780470068069.

[6] MathWorks: *MATLAB 7, Getting Started Guide*. [Online].
URL http://www.mathworks.com/academia/student_version/learnmatlab.pdf

[7] Root.cz: *Seriál Programovací jazyk Lua*. [Online].
URL http://www.root.cz/serialy/programovaci-jazyk-lua/

[8] Wikipedia: *Backusova-Naurova forma*. [Online].
URL http://cs.wikipedia.org/wiki/Backusova-Naurova_forma

[9] Wikipedia: *Comparison of parser generators*. [Online].
URL http://en.wikipedia.org/wiki/Comparison_of_parser_generators

[10] Wikipedia: *Gaussian beam*. [Online].
URL http://en.wikipedia.org/wiki/Gaussian_beam

[11] Wikipedia: *Skriptovací jazyk*. [Online].
URL http://cs.wikipedia.org/wiki/Skriptovac%C3%AD_jazyk

[12] Wikipedia: *Zásobníkový automat*. [Online].
URL http://cs.wikipedia.org/wiki/Z%C3%A1sobn%C3%ADkov%C3%BD_automat

[13] *Optics & Photonics News*. Washington: Optical Society of America, 1990. ISSN 1047-6938.
URL http://www.osa-opn.org/home/

[14] *SPIE Professional*. Washington: SPIE, 1994. ISSN 1994-4403.
URL http://spie.org/x4274.xml

A Překlad programu ze zdrojových souborů

Pro překlad programu ze zdrojových souborů je nutné mít k dispozici nástroje pro překlad. Pro překlad zdrojových souborů programu (v jazyce C++) je nutný překladač, já jsem použil MinGW GCC. Pro překlad gramatiky je nutné použít ještě nástroje Bison a Flex, před samotnou kompilací zdrojových souborů programu (tento krok není nutný, pokud není zasaženo do souborů gramatiky).

Programy Flex a Bison lze v prostředí Windows zprovoznit nainstalováním prostředí Cygwin. Ze stránek `http://cygwin.com/install.html` lze stáhnout instalační program (který následně stáhne všechny potřebné části). V instalačním okně je nutné nastavit instalaci programu Bison a Flex. Nejjednodušeji je to možné nastavit zadáním slova „Bison“ do vyhledávacího pole a vybrat `install` u tohoto programu. Stejným způsobem je nutné najít a nastavit instalaci programu Flex. Po vybrání těchto programů je nutné dokončit instalaci (instalační program bude muset stáhnout a nainstalovat potřebné programy).

Jakmile je prostředí Cygwin nainstalováno, je nutné přeložit soubory gramatiky. To se nejlépe provede zkopírováním souborů `gram.y`, `lex.l` a `_compile.sh` do domovského adresáře Cygwinu. Následné je nutné spustit konzoly Cygwinu, přesunout se do adresáře, kde jsou soubory, a spustit kompilaci pomocí připraveného skriptu `_compile.sh`. Po dokončení jsou vygenerovány zdrojové soubory s příponou `.c`, kterým je nutné změnit příponu na `.cpp` a přesunout k ostatním zdrojovým souborům programu. Také je nutné přesunout i vygenerovaný hlavičkový soubor s příponou `.h`. V případě následných komplikací při překladu, kvůli funkci `yywrap`, je nutné najít řádek

```
extern "C" int yywrap (void );
```

a nahradit ho za řádek

```
extern int yywrap (void );
```

Překlad gramatiky není nutné dělat vždy, ale překlad programu interpretu je nutný vždy. K tomu je nutné mít překladač MinGW GCC. Ten lze získat z `http://sourceforge.net/projects/mingw/files/`, kde je ke stažení instalátor. Tento instalátor se zeptá na instalační cestu, a zde stáhne

další instalátor (i automaticky spustí). V následujícím instalátoru je nutné vybrat instalaci jak `mingw-base`, tak `msys-base`. Poté instalátor stáhne a nainstaluje potřebné komponenty.

Po dokončení instalace je nutné, aby systémová proměnná `PATH` obsahovala cesty k programům z MinGW. Pokud nelze z příkazové řádky spustit programy `g++` a `make`, je nutné aktualizovat systémovou proměnnou `PATH`. Jakmile již je možné programy z MinGW spustit z příkazové řádky, je možné přejít k samotnému překladu.

Jelikož samotný program interpretu potřebuje knihovnu Rayleigh a ta vyžaduje i další knihovny, je nutné přeložit vše potřebné. Pro systém Windows a překladač MinGW je připraven skript pro překlad. Tím je buď `build_win32.bat`, nebo `build_posix64.bat`. Pokud jste nainstalovali program MinGW podle výše popsaného postupu, bude pravděpodobně nutné použít skript `build_win32.bat` (pouze pokud jste zvolili posixovou verzi MinGW a máte 64bitový operační systém, použijte `build_posix64.bat`). Tento skript provede přeložení všech nutných částí a nakonec přeloží i program interpretu. Výsledný program a knihovny budou umístěny v adresáři, kde jsou umístěny soubory `.bat`.

Takto přeložený program a knihovny je možné přesunout na požadované místo, nebo přímo spustit.

B Vytváření nových funkcí

Pro interpret jazyka ke knihovně Rayleigh je nutné vytvořit funkce, které zajistí získání a převedení hodnot ze zásobníku interpretu a předání je funkcím z knihovny Rayleigh. Tyto funkce je nutné dodat do zdrojového souboru `funkce.cpp`. Do funkce `priprav_fce()` v tomto souboru je nutné dodat popisek funkce, který udává jméno funkce, pod kterým je možné danou funkci volat ve skriptu, a pointer na samotnou funkci. Následně je nutné strukturu popisu vložit do vektoru `funkce`, aby popisek mohl být nalezen interpretem.

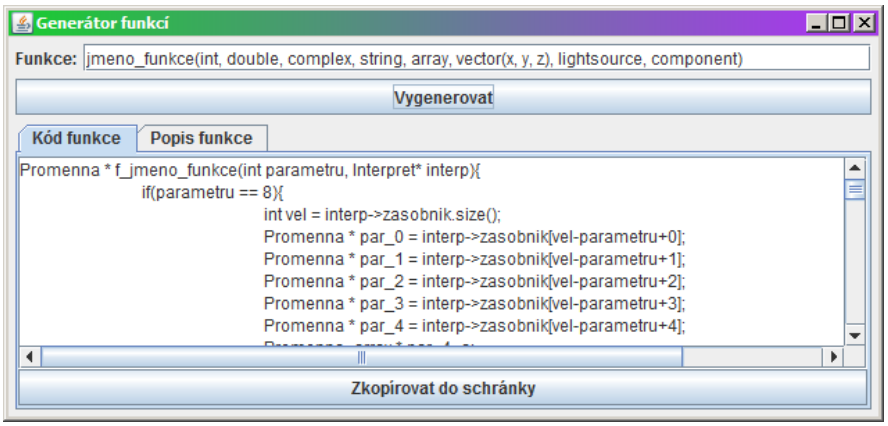
Kromě popisku je nutné vytvořit kód samotné funkce, která provede požadovanou činnost. Tuto funkci lze vytvořit podobně, jako funkce, které jsou již připravené. K ulehčení vytvoření funkcí je možné využít mnou připravenou aplikaci pro vygenerování části funkce.

Tato aplikace je vytvořena v jazyce Java a je nutné ke spuštění mít nainstalovanou Javu 1.7, nebo vyšší. Aplikaci je poté možné spustit poklepáním na program (pokud je nastaveno přiřazení přípony `.jar` k Javě), nebo přes konzoli příkazem:

```
java -jar generator_funkci.jar
```

Po spuštění aplikace bude otevřeno okno podobné jako na obrázku B.1. Po spuštění je nutné napsat popis funkce do horního textového pole. Tento popis je složen ze jména funkce a typů parametrů funkce. Po spuštění je zde již ukázkový příklad, ve kterém jsou uvedeny veškeré možné typy parametrů, které aplikace dokáže zpracovat. S použitím těchto typů je možné vytvořit požadovanou funkci. Po vytvoření popisu funkce je možné stisknout tlačítko vygenerovat. Pokud je popis funkce správně vytvořen, je vygenerován kód části funkce a popisek funkce.

V panelu `Kód funkce` bude vygenerována část kódu funkce v jazyce C++, kterou je nutné zkopírovat a vložit ke zdrojovým souborům interpretu, nejlépe do souboru `funkce.cpp`. Ve většině případů slouží takto vygenerované funkce pro zavolání funkce z knihovny Rayleigh. Ve spodní části vygenerovaného kódu funkce je naznačeno volání funkce s použitím typů parametrů, které byly v popisu napsány. Základní typy a optické typy je možné rovnou předat funkci z knihovny, ale například pole bude nutné převést ručně z objektu proměnné na pole v jazyce C (kód pro tento parametr aplikace nevytvoří a je nutné ručně vytvořit převedení tohoto parametru). V případě, že je cílem zavolat funkci z knihovny Rayleigh, je možné odkomentovat tuto část



Obrázek B.1: Pomocný generátor funkcí.

kódu a lehce ji upravit, aby přesně odpovídala funkci v jazyce C z knihovny. Funkce může provádět i jinou činnost, než volání funkce z knihovny Rayleigh. V tomto případě je nutné zakomentovanou část volání funkce nahradit jakýmkoli jiným užitečným kódem.

Kromě části funkce je vygenerován i popisek pro interpret, aby bylo možné funkci interpretem nalézt. Tento popisek je v panelu `Popisek funkce`, a obsah je nutné zkopírovat do funkce `priprav_fce()` umístěné v souboru `funkce.cpp`.

Tímto způsobem je tedy možné vytvořit kód pro další funkce, které bude možné ve skriptech volat.

C Ukázkový skript programu GLAD

Skript je převzatý z [1].

```
c## ex2a
c
c Example 2a: Gaussian and supergaussian beam generation, automatic
c units.
c
c Initialize two beams with array sizes of 128x128 and wavelengths
c of 10 microns.
c
array/set 1 128 128 # Set Beam 1 to 128 x 128
nbeam 2 # Expand to 2 beams, with Beam 1 size
wavelength/s 0 10.
units 1 .01
c
c Generate Beam 1 as a gaussian beam with radius .5 cm and peak
c fluence of 10. Generate Beam 2 as a top hat beam with a peak
c fluence of 10. and a radius of .5 cm.
c
c For a Gaussian beam default units are :
c Units = sqrt(pi*r0*r0/Nline)
c = .07833213 cm
c
c At the pupil the ratio of the field size to beam size is 9.87
c
c gaussian/[ushap]/[uscal] ibeams pkflu r0x r0y sgxp r0y decx decy
c
gaussian/cir/res 1 10. .5 1.
c
clear 2 1
mult 2 10.
c
c For top hat beams default units are:
c Units = sqrt((rad*rad)/(.61*nline))
c = .05658484 cm.
c
c At the pupil the ratio of the field size to beam size is 7.13
c
clap/cir/res 2 .5
c
c Show a sample of each intensity array
c
c field kbeam jyline ixstart ixend istep
c
field 1 65 65 128 5
pause
field 2 0 65 80 2
c
c Generate a single line slice plot of both beams. After each beam
c  is plotted to the screen GLAD will wait for a carriage return.
c
c PLOT/XSLICE/INTENSITY Kbeam Slice Left Right Fmin Fmax
c
c Define a title for the plots
c
title EX: 2 GAUSSIAN BEAM AT PUPIL
plot/w ex2_1.plt
```

```
plot/xslice/intensity 1
title EX: 2 TOP HAT BEAM AT PUPIL
plot/w ex2_2.plt
plot/xslice/intensity 2
pause
c
c Propagate both beams 100. cm and apply a lens of 100cm focal
c length, then propagate to the focus. Check beam status.
c
dist 100.
c
c LENS Ibeams Flx (Fly) Decx Decy
c
lens 0 100.
dist 100.
status/parax
c
c For a Gaussian beam the new beam size should be :
c r0' = Lambda*fl/(pi*r0)
c = .0010*100/(pi*.5)
c = .06366198 cm.
c
c For a top hat beam the first dark ring should have a radius :
c rd = 1.22*lambda*fl/(2.*rad)
c = 1.22*.0010*100./1.
c = .122 cm.
c
c Notice that the radii of the beams match the above predictions.
c
c At the focus the ratio of the field size to "beam" size for the
c Gaussian and top hat beam are the same as at the pupil.
c
c Generate a single line plot of each beam
c
title EX: 2 GAUSSIAN BEAM AT FOCUS
plot/w ex2_3.plt
plot/xslice 1 0 -.4 .4 0. 650.
title EX: 2 TOP HAT BEAM AT FOCUS
plot/w ex2_4.plt
plot/xslice 2 0 -.4 .4 0. 650.
pause
c
c Repeating the previous propagation sequence in reverse order
c should generate an image of the original beam. Repeating the
c sampling will provide data to check the accuracy of the
c propagator. Any differences will be a result of roundoff
c or other numerical errors.
dist 100
lens 0 100
dist 100
title EX: 2 GAUSSIAN BEAM AFTER FOURIER LENS
plot/w ex2_5.plt
plot/xslice 1
title EX: 2 TOP HAT BEAM AFTER FOURIER LENS
plot/w ex2_6.plt
plot/yslice 2
field 1 65 65 128 5
pause
field 2 0 65 80 2
pause
end
```

D Ukázkový skript programu Matlab

```
function uzivgraf(varargin)
if length(varargin)==0
    akce='init';
else
    akce=varargin{1};
end
switch akce
    % inicializace
    case 'init'
%         vytvoreni okna
        figure('menubar','none','numbertitle','off','name',...
            'ukazkoveokkno','units','normalized',...
            'position',[0.2,0.2,0.6,0.6]);
        axes('units','normalized','position',[0.1,0.1,0.6,0.8]);
        [x,y,z]=sphere(30);
%         vytvori predpis koule
        koule=surf(x,y,z,'linewidth',2,'tag','koule');
        axis equal;
        chck_osy=uicontrol('style','checkbox','units','normalized',...
            'position',[0.8,0.7,0.15,0.05],'string','osy',...
            'value',[1],'callback','uzivgraf(''osy_zapvip'' )', ...
            'tag','chck_osy');
        lt_exit=uicontrol('style','pushbutton','string','exit', ...
            'position',[0.8,0.1,0.15,0.05], ...
            'callback','uzivgraf(''exit'')','units','normalized');
% zajistujeme vypinani/zapinani os
    case 'osy_zapvip'
        h=findobj('tag','chck_osy');
        stav=get(h,'value');
        if stav==0
            axis off
        else
            axis on
        end
    case 'exit'
        delete (gcf);
end
```

E Ukázkový skript jazyka Lua

Skript je převzatý z [7].

```
function c1()
    for i=0, 3 do
        forward(280)
        left(90)
    end
end

function c2()
    for i=0, 360/8-1 do
        c1()
        forward(10)
        left(8)
    end
end

clean()
right(90)
penup()
forward(80)
left(90)
pendown()
c2()
```

Obsah CD

Na CD jsou přiloženy zdrojové soubory interpretu, spustitelný program interpretu, testovací skripty, aplikace generátoru funkcí a text této práce.

Zdrojové soubory interpretu jsou umístěny ve složce `zdrojove_soubory`. Překlad těchto souborů je popsán v příloze A.

Spustitelný program interpretu, včetně všech potřebných knihoven, je umístěn ve složce `spustitelny_program`.

Testovací skripty jsou ve složce `testovaci_skripty`. Je zde skript zmíněný v kapitole 6.1 (`skript_otestovani_prekladu.txt`), v podsložce `test_provazani` je skript, zdrojový i přeložený program pro otestování provázání s knihovnou zmíněný v kapitole 6.3 (také zde jsou vygenerované obrázky). V podsložce `ukazkovy_skript` je ukázkový skript zmíněný v kapitole 7.4, včetně vygenerovaných obrázků. V podsložce `netridene` jsou některé skripty, které jsem použil při otestování funkčnosti.

Pomocná aplikace pro vygenerování funkcí je umístěna ve složce `pomocny_generator_funkci`. Popis užití aplikace je uveden v příloze B.

Text práce ve formátu pdf je umístěn ve složce `text`. V podsložce `src` jsou umístěny zdrojové soubory pro TeX této práce.