VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

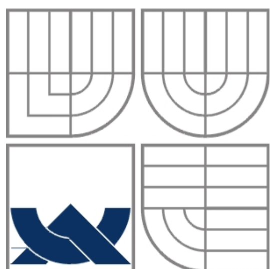
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SKRIPTOVACÍ JAZYK PRO ZPRACOVÁNÍ OBRAZU

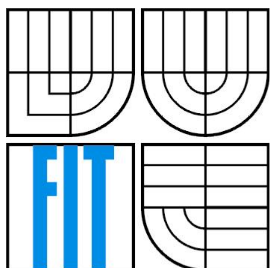
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE | RADEK CRLÍK
AUTHOR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SKRIPTOVACÍ JAZYK PRO ZPRACOVÁNÍ OBRAZU

SCRIPT LANGUAGE FOR IMAGE PROCESSING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE RADEK CRLÍK
AUTHOR

VEDOUCÍ PRÁCE Prof. Dr. Ing. PAVEL ZEMČÍK

SUPERVISOR

BRNO 2015

Abstrakt

Tato bakalářská práce se zabývá možnostmi skriptovacích jazyků a jejich druhů. Konkrétně pak návrhu takového jazyka pro obor zpracování obrazů. Cílem bylo vytvoření jazyka, který je malý, ale lehce použitelný. Cílem bylo využít knihovny OpenCV, pro kterou by výsledný jazyk umožňoval co nejkratší zápis kódu. První část se zabývá popisem a zpracováním skriptovacích jazyků počítačem. Druhá část obsahuje popis navrhovaného jazyka. Třetí a poslední část, dokumentuje možnosti dalšího rozšíření.

Abstract

This bachelor thesis deals with capabilities of scripting languages and their types. Specifically with design of one such language suitable for image processing. The goal was to create a language small enough, but easy to use. One of the requirement was to utilize OpenCV library, for which the resulting code would be as short as possible. First part deals with descriptions of scripting languages, their main philosophy and options of language processing by the computer. Second part contains description of the designed language. Last part documents achieved goals and possible ways how to extend the language even further.

Klíčová slova

Skript, syntaxe, lexikální analýza, syntaktická analýza, kompilace, kompilátor, interpret, Lua, zpracování obrazu

Keywords

Script, syntax, lexical analysis, syntax analysis, analysis, compilation, compiler, interpreter, Lua, image processing

Citace

Radek Crlík: Skriptovací jazyk pro zpracování obrazu, bakalářská práce, Brno, FIT VUT v Brně, 2015

Skriptovací jazyk pro zpracování obrazu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Pavla Zemčíka, prof. Dr. Ing.
Další informace mi poskytli...
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Radek Crlík
15.5.2015

Poděkování

Rád poděkoval svému vedoucímu, panu Pavlu Zemčíkovi, který mi pomohl tuto práci vytvořit a ostatním vyučujícím na fakultě, kteří vždy rádi poradili a přispěli novými zdroji informací.

© Radek Crlík, 2015
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod....1
2 Skriptovací jazyky....3
2.1 Charakteristiky skriptovacích jazyků....3
2.2 Jazyky používané pro zpracování obrazu....5
2.3 Překladače....6
2.4 Lexikální analýza....7
2.5 Syntaktická analýza....8
2.6 Ostatní části překladače....8
3 Specifikace jazyka....11
3.1 Datové typy....12
3.2 Operátory....15
3.3 Řízení toku programu....17
3.4 Funkce....20
3.5 Třídy....22
3.6 Jmenné prostory....23
3.7 Zpracování chyb....24
3.8 Komentáře....25
4 Implementace....26
4.1 Architektura systému....26
4.2 Lexikální a syntaktická analýza....27
4.3 Jazyk Lua....28
4.4 Interpretace a spouštění kódu....29
4.5 Funkce pro zpracování obrazu....30
5 Testování....31
5.1 Porovnání délek holých konstrukcí....31
5.2 Porovnání délek ostatních typů kódu....32
5.3 Interpretace testů....32
6 Závěr....34

Seznam tabulek

Tabulka 1: Datové typy....12
Tabulka 2: Operátory....15
Tabulka 3: Přiřazovací operátory....15
Tabulka 4: Srovnání základních konstrukcí....31
Tabulka 5: Srovnání konstrukcí s uživatelským kódem....32
Tabulka 6: Srovnání délky kódu krátkých programů....32
Tabulka 7: Escape sekvence....41

Seznam obrázků

Obrázek 1: Architektura překladače....26
Obrázek 2: Vzor Návštěvník pro abstraktní syntaktický strom....27

1 Úvod

Počítače jsou ve své podstatě hloupá zařízení, která nedokáží samostatně myslet. Dokáží jen provádět úkony, které mu člověk explicitně zadá. Ovšem počítače dnes nejsou na takové úrovni, aby rozuměli lidské řeči nebo textu. Důvodem je, že takové jazyky obsahují mnoho nejednoznačností s kterými si počítač nedokáže poradit. Je proto nutné konstruovat speciální programovací jazyky, které tyto nejednoznačnosti neobsahují a lze jimi přesně a výstižně popsat pracovní postup, který od počítače vyžadujeme vykonat. Stejně však jako různé lidské přirozené jazyky mají různou strukturu, různý počet slov a gramatiku, která se hodí pro popis různých pojmů a činností, existují i různé druhy programovacích jazyků hodících se pro popis různých algoritmů. Skriptovací jazyky jsou pak speciální programovací jazyky, které umožňují zápis kódu tak, aby člověk nemusel znát detaily počítače, který programuje a potřebný kód byl co nejkratší a nejjednodušší.

Cílem této práce bylo vytvořit jazyk, který by měl obsahovat konstrukce a funkce typické pro zpracování obrazových dat, tak aby vývoj programů byl co nejkratší a nezatěžoval programátora složitými detaily, jak který algoritmus pracuje. Cílem tedy není vytvořit jazyk, který by měl mít velkou vyjadřovací schopnost a různé techniky pro popsání obecných algoritmů. Měl by se primárně zaměřovat na zpracování obrazových dat a podporovat řadu funkcí pro toto zpracování a zároveň mít dostatečnou flexibilitu pro vytváření vlastních algoritmů a funkcí pro tuto činnost, které nejsou součástí jazyka. Oproti jiným jazykům by měl mít navíc jednodušší syntaxi s příkazy, případně operátory, které se často používají u zpracování číselných dat v obraze. Příkladem můžou být netypované proměnné, bezpečné `foreach` cykly, podpora bezpečných dynamických polí a různé další konstrukce, které se prosazují u moderních jazyků.

Je samozřejmé, že existuje mnoho technologií a jazyků, které se snaží řešit zpracování obrazu. Existují řešení například pro Matlab, který se snaží nad obrazovými daty pracovat pomocí matematických operací. Dále existují specializované aplikace jako ImageMagick, který se zaměřuje na zpracování statických obrázků. Obsahuje i vlastní jazyk MSL pro tvorbu jednoduchých skriptů pro automatizaci zpracování. Nevýhoda je popis pomocí XML, který se hodí spíše pro nenáročné uživatele. Existuje i řada knihoven pro jazyky jako C, zde můžeme zmínit knihovnu OpenCV. Avšak jazyk C jako systémový jazyk vyžaduje určitou znalost jazyka a programování. Jako poslední si můžeme uvést systém Mira, což je komerční nástroj a staví na technologii Lua. Tento systém je rozšiřitelný a také velmi rychlý. Je zde i různá podpora nástrojů jako editor zdrojových textů nebo nástroje pro zobrazení obrazových dat. Systém byl původně navržen pro zpracování velkého objemu dat. Podle výrobců je používán například pro zpracování dat z vesmírných teleskopů nebo různých medicínských přístrojů jako rentgen nebo CT. Z toho se také dá odvodit zaměření nástroje Mira.

Cíl této práce bylo vyzkoušet si tvorbu takového skriptovacího jazyka, který by navíc byl open-source a byl dostatečně jednoduchý a umožňoval co nejkratší zápis kódu pro časté operace.

Následující kapitola obsahuje obecný popis skriptovacích jazyků, jejich filozofii a různé příklady jejich použití nebo konkrétních jazyků. Je zde také nastíněn rozdíl mezi skriptovacím jazykem a kompilovaným jazykem. Další kapitola se zabývá obecným postupem zpracování programovacích jazyků a jejich převodem do formy srozumitelné počítačem. Je zde představena funkce překladače a jsou zde popsány typické fáze zpracování jazyka a jejich popis. V další kapitole je pak popsán navrhovaný jazyk a jeho syntaxe a konstrukce. Tato kapitola slouží k seznámení s možnostmi jazyka a syntaxe, po jejímž přečtení by měl být uživatel se znalostmi programování schopen začít tvořit první jednoduché programy. Poslední kapitola popisuje návrh a implementaci interpretu pro navrhovaný jazyk. Je zde také popsána řada problémů v aktuální verzi a možnosti dalšího rozšíření.

2 Skriptovací jazyky

V této části je uveden popis skriptovacích jazyků, jejich výhody a přehled běžně používaných jazyků pro různé účely. Avšak uvedený seznam není vyčerpávající a nesnaží se mapovat všechny kategorie. Takový seznam by byl příliš dlouhý a z důvodu existence řady specializovaných skriptovacích jazyků, by bylo složité vytvořit kompletní seznam. Jsou zde popsány jen hlavní kategorie, které byli uvažovány při návrhu jazyka této práce.

2.1 Charakteristiky skriptovacích jazyků

Skriptovací jazyky jsou interpretované programovací jazyky, navrženy s ohledem na snadné použití a ovládnutí jazyka, které umožňují rychlý a pohodlný vývoj programů. Tyto jazyky jsou také často tvořeny s cílem odstínit programátora od hardwaru počítače. Na rozdíl od systémových jazyků jako je C nebo assembler. Obsahují spíše vysokoúrovňové příkazy a funkce, které nevyžadují znalost funkci hardware nebo přesnou znalost operačního systému a jeho nástrojů, ale pouze popis rozhraní. Zbytek je právě na interpretu jazyka, aby požadované rozhraní dodržel. Skriptovací jazyky tak mohou mít velice vysokou vyjadřovací schopnost, která umožňuje jinou úroveň programování. Díky použití interpretu ke spuštění kódu, jako programu, lze tak zajistit i přenositelnost na různé platformy. Pak je jen potřeba mít dostupný interpret jazyka a výsledný skript může zůstat nezměněn.

Existují samozřejmě i nevýhody. Takové jazyky mohou být ve výsledku i často špatně čitelné. Jako příklad můžeme uvést jazyk Perl, který někteří lidé používají stylem „napiš a zapomeň“ [PERL, str.491]. Síla Perlu spočívá mimo jiné v krátkých, často jednořádkových programů, které provádí malou činnost, ale provádí ji dobře (tzv. o*ne-liner*). Takové programy se však často používají bez toho, aby člověk musel vědět jak přesně fungují [ONEL]. Je snadné napsat i složitý program na pár řádků, ale díky syntaxi jazyka je jeho následné čtení obtížné. Jako další nevýhodou může být rychlost zpracování. A to hlavně pokud je potřeba pracovat s velkými datovými strukturami nebo velkým množstvím dat. I když se dnes za použití techniky *just-in-time* kompilace, dokáží některé úseky kódu převézt na nativních instrukcí dané architektury [JIT], obecně to nelze provést pro celý skript. Interpretované jazyky tak nejspíše nikdy nepředčí rychlost systémových jazyků jako C nebo assembler.

Nejstarší skriptovací jazyky údajně vznikali jako pomůcka při automatizaci počítačů pracující s děrnými štítky a při konfiguraci operačních systémů [HIST]. A to nejvíce na Unixových systémech, kde jejich původní filozofie byla používat textové soubory pro konfiguraci programů. Zároveň jsou operační systémy Unix a Linux konstruovány pomocí velkého množství malých programů, které umí jednu zadanou funkci, ale dělají ji velice dobře. Tyto programy pak lze skládat do větších celků a provádět složitější úlohy. Přirozeně tak vznikali potřeby co nejvíce automatizovat provádění veškerých úloh s tímto spojené a urychlit vývoj takových programů.

Hlavní rozdíl mezi kompilovanými a interpretovanými jazyky je způsob jakým se výsledný program spouští na cílové výpočetní jednotce. Zatímco kompilované jazyky jsou převedeny přímo na instrukce spustitelné přímo na procesoru, interpretované jazyky je potřeba provádět – interpretovat speciálním programem, nazývaného interpret. Způsobů návrhu interpretu může být několik [HYDE01, str. 63]. Zde si jako hlavní kategorie uveďme:

- *Čisté interprety*: Nejjednodušší forma, kdy se pracuje přímo nad textovým souborem. Interpret čte text a rozpoznává lexémy jazyka jeden po druhém. Postupně se také vykonávaní příkazy, jak jsou rozpoznávány. Tato forma se však hodí jen pro velice jednoduché skriptovací jazyky nebo velice jednoduché shelly. Existují zde nevýhody se složitým vykonáváním skoků, kdy je potřeba znovu číst textový soubor a znovu interpretovat dané příkazy. Interpret tak může velkou část strávit prováděním lexikální analýzy, místo samotným prováděním kódu.
- *Interprety:* Zde se nejprve textový soubor převede na nějakou vnitřní reprezentaci kódu, například na sadu tokenů, uspořádaných ve vhodné datové struktuře. Token je pak kompaktní uložení informace o jednotlivých lexémech jazyka usnadňující jeho rychlé zpracování, oproti jeho textové podobě. Interpretace je pak o mnoho rychlejší. Je však nejprve nutné převést text na tuto reprezentaci, která může chvíli trvat, tento čas je ale velice často zanedbatelný oproti samotnému provádění skriptu.

- *Inkrementální kompilátor*[1]: je jakýsi hybrid mezi interpretem a kompilátorem. Tento postup spočívá v převedení skriptu do určitého mezikódu. Tato reprezentace je pak většinou posloupnost instrukcí, které jsou spustitelné na *virtuálním stroji*. To znamená, že neexistuje reálný procesor, který by tyto instrukce uměl spouštět. Pro provádění instrukcí je však relativně jednoduché napsat program, který dokáže tyto instrukce interpretovat a spouštět. Tento způsob interpretace je daleko rychlejší než předešlé dva způsoby. Příkladem jazyka používající tuto techniku může být Java a jeho *Java byte code engine*. Pro tento speciální druh interpretu a jeho virtuálním instrukcím je pak také relativně snadné použít techniku *just-in-time* kompilace.

Zde je seznam několika tříd jazyků. Následující seznam však není v žádném případě vyčerpávající a existuje celá řada [ZUZ].

Shelly

Shelly [SHEL] jsou nástroj pro ovládání systémových programů a jejich efektivní použití. První operační systémy byli pouze textově orientované bez grafického režimu. Proto mimo jiné vznikly právě shelly, které dokázali automatizovat spouštění programů v operačním systému, výpis textu na obrazovku a práci se soubory. Dokázali tak uživatele ušetřit častého a opakovaného psaní a manuální práce.

Zpracování textu

Tato skupina zástupce jazyků, které vznikla pro specifický účel a to pro zpracování textu. Jak již bylo zmíněno, první verze systému Unix používali textové konfigurační soubory nebo jiná data. Není tak překvapením, že byli snahy zpracování textu zautomatizovat a ulehčit co nejvíce. Díky tomu vznikla celá řada systémových utilit jako je sed nebo grep, ale řada skriptovacích jazyků speciálně určených pro zpracování textu. Můžeme zmínit AWK... Jako náhrada jazyka AWK (a pár

1 Tento název je dnes spíše nepřesný a označuje kompilátor, který je schopný zpracovat pouze část kódu při jeho změně, místo toho aby prováděl zdlouhavou kompilaci celého programu. Tímto se však v této práci nebudeme zabývat.

dalších) vznikl i jazyk Perl [PERL5], který se vyvinul v univerzální jazyk s obrovskou vyjadřovací schopností. Ten byl dokonce velmi populární pro vývoj webových aplikací.

Interní skriptovací jazyky

Velké množství specializovaných aplikací obsahuje i nějaký druh skriptovacího jazyka, který umožňuje měnit chování aplikace nebo ulehčení práce s programem. Může jít o různé modelovací programy nebo 3D grafické editory (například Blender nebo Maya 3D), které místo pracné manuální manipulace a modelování umožňují napsat jednoduchý skript, který dokáže požadovaný objekt vygenerovat. Do této kategorie můžeme zařadit i různé jazyky pro popis maker. Například ECMAScript, VBScript.

Dynamické webové stránky

Později se díky internetu a bezstavovému protokolu HTTP rozmach skriptovacích jazyků pro generování dynamického obsahu webových stránek. Jako první technologie pro zpracování programů na serveru a následné generování dynamického obsahu byla technologie CGI. Která definovala rozhraní pro komunikace aplikace s webovým serverem [CGI, str. 7]. V té době byl populární opět jazyk Perl pro programování dynamických webových stránek. Později vznikl jazyk PHP, který se používá dodnes a dá se označit za de facto standard. Na tomto jazyku je dnes postaveno velké množství frameworků a je také jeden z nejčastěji používaných spolu s webovým serverem Apache [LINS]. Dále to byli jazyky třeba jako ASP, zmíněný Perl, Ruby a řada dalších. Popularita takových skriptovacích jazyků byla možnost využití databáze. Na straně prohlížeče, které mimo dynamický obsah umožnil i dynamické chování zobrazených stránek byl JavaScript nebo VBScript.

Skriptovací jazyky jako úplné programovací jazyky

Dnes jsou v oblibě i skriptovací jazyky, které se dají použít pro univerzální programování. Mají podporu různých datových struktur, knihovny pro přístup k databázi, síťovým rozhraním, tvorbě grafických uživatelských rozhraní a mnoho dalších věcí. Populární jsou například Python, který má také podporu pro více programovacích paradigmat nebo jazyk Ruby.

2.2 Jazyky používané pro zpracování obrazu

Pro tento účel je možné použít několik jazyků. Například v úvodu byl zmíněn Matlab (a jeho open-source náhrada GNU Octave) jako jazyk pro matematické výpočty, který využívá toho, že obrázek lze chápat jako matici bodů a potřebné operace nad ním lze definovat pomocí matematických operací. Jazyk systému Mira, který je sice komerční, ale je uzpůsobený k práci nad velkým množstvím dat. A dále sem můžeme zařadit speciální jazyky větších aplikací pro práci s grafikou jako Photoshop nebo Gimp. Ty však často dokáží pracovat jen pomocí operací obsažených v těchto programech.

Jako skriptovací jazyk můžeme označit i jazyk Processing, který obsahuje sadu API pro tvorbu grafiky, případně i správy uživatelského vstupu. Jeho přednost je implementace pomocí jazyka JavaScript, takže je možné jej použít i ve webových prohlížečích. Poslední dobou se začíná prosazovat i standard WebGL, což je knihovna pro OpenGL pro jazyk JavaScript použitelná v prohlížeči pro práci s 3D grafikou.

Zajímavý je i jazyk MSL aplikace ImageMagick, umožňující zpracování statických obrázků. Konkurentem ImageMagick je pak nástroj G'MIC, který vznikl jako otevřená náhrada. Mezi speciální nástroje umožnující vývoj algoritmů a filtrů pro obraz a signály je Cassandra Studio, která nepoužívá primárně zdrojový text, ale grafickou specifikaci algoritmu. Každá operace je pak brána jako filtr, který má vstup a výstup. Ty jsou pak graficky spojovány. Úprava obrázků je často potřeba i ve webových aplikacích, proto existuje i řada rozšíření nebo pluginů pro jazyk PHP a to včetně pluginu aplikace ImageMagick. Výhodou PHP je jeho rozšířenost a velikost dokumentace. Avšak pro zpracování obrazu je závislý na možnostech těchto pluginů, které jsou často vyvíjeny pro zpracování pro webové aplikace. Neobsahují pokročilé algoritmy nebo možnost tvorby jednoduchého GUI.

Velice příjemné je zpracování v aplikaci Wolfram Mathematica. Což je matematický nástroj pro symbolické výpočty a pro zpracování výpočtů nebo dat pro různé inženýrské odvětví. Jeho výhodou je velké množství předdefinovaných funkcí a možnost provádět nad daty různé matematické operace. Nástroj obsahuje i speciální skriptovací jazyk, který mimo jiné dokáže zadávat matematické symboly i v jejich originálním tvaru. Vývojové prostředí obsahuje například symboly pro integrály nebo derivace. Proto je vývoj skriptů, které takové matematické operace využívají velice intuitivní. Nevýhodou jsou však složitější konstrukce pro běžně používané operace jako v konvenčních programovacích jazycích. Sem patří například provádění kódu na základě podmínek nebo opakované provádění a tak dále.

Jako příklad menšího jazyka můžeme zařadit jazyk Terra a na něm postavený skriptovací jazyk Darkroom. Ten obsahuje funkce speciálně pro práci s obrazem. Je zvláštnost je ta, že je to staticky typovaný jazyk, který ovšem dokáže spolupracovat s jazykem Lua, který je sám o sobě dynamicky typovaný. Zajímavou vlastností je, že se snaží využít hardware pro paralelní zpracování dat. Nevýhodou je pak slabší dokumentace a omezená sada funkcí pro složitější úlohy jako jsou rozpoznávání v obrazu a podobně.

2.3 Překladače

Jak již bylo zmíněno, napsaný program v některém vyšším programovacím jazyce v textové podobě čitelném člověkem, je potřeba převézt na reprezentaci vykonatelné počítačem. Jednotlivé fáze překladu se samozřejmě mohou lišit podle překládaného jazyka nebo podle možností překladače, zde si však uveďme možné fáze, které je možné aplikovat obecně pro kompilované i interpretované jazyky:

- Lexikální analýza
- Syntaktická analýza
- Sémantická analýza
- Překlad do mezikódu (tzv. intermediárního kódu)
- Optimalizace
- Generování kódu cílové platformy
- Sestavení výsledného programu *linkerem*

2.4 Lexikální analýza

Lexikální, znamená v tradičním překladu „týkající se slov“ [BASC, str. 9]. Lexikální analýza je první částí překladače a provádí ji lexikální analyzátor (někdy zvaný *scanner* nebo *lexer*). V podstatě se jedná o jednoduchý textový procesor. Tento analyzátor má za úkol rozdělit vstupní text na posloupnost lexému (jednotlivá slova). Pro programovací jazyky to mohou být například identifikátory, čísla, operátory nebo textové řetězce. Ty reprezentují jednotlivé terminální symboly syntaktické analýzy. Jeho dalším úkolem je ignorovat přebytečné bílé znaky nebo komentáře nebo zpracovat různé formáty znaku nového řádku.

Bývá to jedna z nejjednodušších části překladače. Striktně řečeno je lexikální analýza dobrovolná část překladače a syntaktický analyzátor by mohl pracovat přímo se zdrojovým textem. Protože je však často nutné zpracovávat dané lexémy několikrát, je z důvodu efektivity výhodnější převézt text na posloupnost *tokenů* lexikálním analyzátorem. Jak již bylo zmíněno výše, jedná se o strukturu popisující daný lexém pro snadnější zpracování. Je to malá struktura reprezentující lexém jazyka a umožňuje jeho číselné zpracování, namísto textového. Což je daleko časově náročnější (to je také jeden z důvodu, proč jsou čisté interprety neefektivní, pracují totiž pouze s textem, který musí někdy i opakovaně zpracovávat [HYDE01, str. 62]). Token ukládá data jako typ a třídu tokenu, případně jeho další hodnoty pro dané lexémy. Syntaktický analyzátor využívá lexer jako takovou službu. Kdy žádá o další token, pokud potřebuje a lexer jej extrahuje z textu a předá volajícímu syntaktickému analyzátoru.

Algoritmy zprostředkovávající lexikální analyzátor nejčastěji využívají ke své práci deterministický konečný automat. Využívá se předpokladu, že jednotlivé lexémy lze popsat regulárním výrazem (jazyk typu 3 v Chomského hiearchii [PARST, str. 30]). Teorie jazyků pak popisuje množství technik a algoritmů, jak mechanicky převést tento popis na daný konečný automat [BASC, str. 22].

Protože je teorie lexikální analýzy velice dobře popsána, lze nalézt řadu nástrojů, které dokáží takový analyzátor automaticky generovat. Stačí vysokoúrovňový popis lexémů a to nástroji stačí k vytvoření konečného automatu. Takové nástroje je vhodné použít ze dvou důvodů. První je udržovatelnost takového analyzátoru, než kdyby byl jeho kód psán ručně. Druhý důvod je ten, že i přes relativní jednoduchost analyzátoru se jedná o slabé místo, co se týče rychlosti provádění překladu kódu. A to zejména v případě, že se používají dlouhé zdrojové soubory nebo pokud může být zdrojovým souborů větší množství. Některé interprety pak tráví většinu času lexikální analýzou než samotným spouštěním programu [HYDE01, str. 64]. Práce lexikálního analyzátoru je mimo výše uvedené i čtení dat z pevného disku. Tyto nástroje pak umí využít speciálních technik, jak takový přístup na pevný disk optimalizovat. Patří sem použití dvojité vyrovnávací paměti a podobně. Implementace takových technik je sice jednoduché, ale pracné. V automatických nástrojích jsou pak dobře odladěné a odzkoušené a odpadá možnost, že v nich tvůrce vytvoří chybu, pokud by takový analyzátor psal ručně.

2.5 Syntaktická analýza

Je proces analýzy a strukturování lineární posloupnosti formálních prvků – lexémů, podle předem dané formální gramatiky jazyka. Tento popis je ponechán abstraktní v tom smyslu, že jako „lineární posloupnost“ můžeme chápat jak seznam příkazů programovacího jazyka, tak větu přirozeného jazyka, posloupnost hudebních not nebo třeba i posloupnost pracovního postupu. V každém případě každý element vytváří omezení na typ nebo tvar elementu následujícího. Pro některé uvedené příklady jazyků jsou gramatiky známé, pro některé však nemusejí mít formální podobu.

Zatímco úkolem lexikální analýzy je rozdělit text na jednotlivá slova, syntaktická analýza má za úkol je složit tyto lexémy zpět do formy, která nemusí být lineární, ale určitým způsobem musí popisovat danou strukturu textu [BASC, str. 53].

V oblasti programovacích jazyků pak syntaktická analýza zároveň provádí kontrolu kódu. Pokud analyzátor objeví syntaktickou nebo i sémantickou chybu, ukončí analýzu a zobrazí chybové hlášení. Také posloupnost tokenů převádí na další vnitřní reprezentaci kódu (často abstraktní syntaktický strom, zkr. AST).

V informatice nyní existují solidní teoretické základy algoritmů provádějící syntaktickou analýzu většiny typů jazyků z Chomského hiearchie. Jsou popsány možnosti těchto algoritmů, nedostatky, případně složitost analýzy daného typu jazyka []. Programovací jazyky se omezují jako nejvýše kontextové nebo bezkontextové jazyky, jejichž zpracování je zvládnutelné počítačem [PARST, str. 73].

2.6 Ostatní části překladače

Dá se říci, že lexikální a syntaktická analýza je prováděna dnes u všech typů programovacích jazyků. Mimo to může překladače obsahovat řadu dalších volitelných fází, kde jejich použití záleží na konkrétním programovacím jazyce. Jejich účel a použití je popsáno níže. Pro některé fáze je i uvedeno pro jaké programovací jazyky se nejčastěji používají.

Sémantická analýza

Tato část analýzy prochází symboly a struktury získané se syntaktické analýzy a přiřazuje jim význam. Při zpracování programovacího jazyka pak sémantická analýza kontroluje použití proměnných, přesněji jestli byly deklarovány, použití operátorů s validními typy operandů. V určitých případech dokáže analyzátor vytvořit správné přetypování nebo vyvolat chybu. Tato část překladu je důležitá pro typované jazyky. Často je tato fáze spojená s fází syntaktické analýzy. Pro dynamicky typované jazyky, kdy nemůže překladač v době překladu mít dostatek takových informací, je tato fáze spojena s fází spouštění kódu.

Překlad do mezikódu

Tato fáze převádí například AST z předchozí fáze na mezikód. Někdy také pojmenovaný intermediární kód [BASC, str. 147], je speciální jazyk nebo kód již připomínající architekturu hardwaru. Ten popisuje program na úrovni jednoduchých instrukcí, ty však jsou obecné a ještě nejsou svázány s žádnou konkrétní cílovou architekturou. V případě skriptovacího jazyka může být tento kód již spustitelný interpretem. Důvodem generování mezikódu mohou být dva. První, že lze provádět určité optimalizace [HYDE01, str. 69]. Druhý, pro překladače, které dokáží generovat spustitelný program pro více platforem (angl. c*ross-compiler*) je výhodné používat tuto mezifázi, kdy všechny operace nezávislé na platformě může být součástí překladače. Zatímco generování a další operace závislé na cílové platformě může být nezávislá a často je také vyvíjena jiným týmem lidí [HYDE01, str. 70].

Optimalizace

V této fázi, pokud máme mezikód, který je často dostatečně jednoduchý, že lze provádět velkou řadu optimalizací kódu. Mezi nejčastější optimalizace patří odstranění nepotřebných nebo dočasných proměnných, odstranění mrtvého kódu, výpočet a propagace konstant nebo zjednodušení logických výrazů [HYDE01, str. 76]. Mezi složitější optimalizace patří rozbalování cyklů, zjednodušení výpočtů pomocí matematických zákonů, inline funkce, eliminace tvorby zásobníku pro listové funkce a mnoho dalších. Některé optimalizace lze provádět i na výsledném binárním kódu celého procesu překladu, jako je nahrazování instrukce za jejich rychlejší ekvivalenty. Tyto optimalizace jsou závislé na cílové architektuře.

Druhů optimalizace je hned několik. Lze optimalizovat pro rychlost kódu nebo pro jeho velikost. Navíc, nelze snadno říct, jak optimální kód vlastně vypadá. Problémem také je, že optimalizace je obecně problém typu *NP-neúplný*. Proto se využívá mnoha heuristik a podmíněných algoritmů (*case-based algorithms*). Většinou je tak počet a typ optimalizací závislý na překladači a výrobci se v nich snaží konkurovat [HYDE01, str. 71].

Generování kódu

V tomto momentě je již možné (optimalizovaný) mezikód přeložit na nativních strojových instrukcí cílové architektury. Výše používaný intermediární kód totiž stále obsahuje některé vysokoúrovňové instrukce jako například jednoduché instrukce pro volání funkcí, speciální vysokoúrovňové instrukce pro vykonávání podmínek, používání neomezeného počtu registrů atd. [BASC, str. 179]. Takto vygenerovaný výsledný kód je již možné přímo spustit na procesoru. U kompilovaných jazyků se často neprovádí překlad přímo do binární podoby, ale do assembleru, který je dále zpracován specializovaným překladačem. Výhodou je, že se překladač nemusí zaobírat návěštími a relativními adresami. Vše za něj vyřeší překladač assembleru. Tuto část překladače pak nejčastěji píší vývojáři hardware dané architektury, protože pro generování efektivního kódu, je potřeba znát možnosti daného procesoru, jeho instrukční sady a obecně i dalšího hardwaru [LLVM]. V případě interpretovaných jazyků se v této fázi může jednat již o spuštění kódu interpretem, nebo generování kódu pro virtuální stroj.

Linkování

Tato konečná fáze je typická u kompilovaných jazyků. U mnoha jazyků je možnost nevytvářet přímo spustitelnou aplikaci ze zdrojového souboru, ale vytvořit tak zvaný objektový soubor. Ten již obsahuje výsledný binární kód a další metadata, kód je ale tzv. relokatibilní. To znamená, že adresy skoků, případně referencí dat, nejsou absolutní, ale lze je dále upravit. To umožňuje tvorbu modulů. Sada několika takovýchto modulů je pak možné spojit v jeden celiství program pomocí programu, který se nazývá *linker* [LDRS]. Pokud je dodržena konvence a formát objektových souborů, moduly mohou být dokonce vytvořeny každý v jiném jazyce. U interpretovaných jazyků se něco podobného prosazuje i dnes. Příkladem může být .NET technologie a různé množství jazyků, které tuto architekturu podporují.

3 Specifikace jazyka

V této kapitole jsou představeny existující řešení, jejich nedostatky a důvody pro návrh jazyka. Je představena základní syntaxe jazyka a jeho možnosti. Tato kapitola slouží jako úvod k syntaxi jazyka a nejedná se o úplnou specifikaci. Jsou popsány základní datové typy, konstrukce pro řízení kódu a práci s pamětí.

Existující řešení

Jak již bylo zmíněno v úvodu, pro práci s obrazem lze použít řadu jazyků. Mimo ně, existuje řada nástrojů specializujících se na zpracování statického obrazu jako GIMP[2], který umožňuje i dávkové zpracování pomocí skriptu. Jeho syntaxe má však tvar funkcionálního jazyka a nemusí být příliš jednoduchá na použití. Jazyk Python spolu s knihovnou OpenCV, který má velké možnosti v tomto ohledu, je však pořád spíše univerzální jazyk. A i když díky tomu pro něj existuje velká řada modulů umožňujících psát efektivní kód, často se nelze obejít bez rozsáhlé dokumentace. Systém Mira využívá jazyk Lua jako skriptovací jazyk, pro který dodává velké množství funkcí pro zpracování dat. Tyto funkce jsou charakteristické pro systém Mira. Nevýhodou je zdlouhavá syntaxe jazyka Lua a jejích konstrukcí, které jsou dnes k vidění spíše u starších jazyků nebo u systémů s kterými nepracují zkušení programátoři. Podporované funkce jsou pak často odlišné než u jiných systémů (například než právě u knihovny OpenCV) a při přechodu na tento systém je potřeba se vše naučit znovu. Jazyk MSL aplikace ImageMagick je také relativně silný nástroj, ovšem jako hlavní nevýhoda je použití systému XML pro specifikace operací nad obrázky. Jeho konkurent, aplikace G'MIC je o něco příjemnější na popis operací, avšak tyto příkázy jsou zadávány jako série parametrů na příkazové řádce. Psaní delších skriptů nebo jejich použití tak může být poněkud zvláštní nebo nepřehledné. Dalším jazykem, který lze pro různé účely použít je jazyk Terra, který je staticky typovaný jazyk, využívající zároveň systém Lua. Tento jazyk je pak použit pro doménově specifické jazyky jako je jazyk Darkroom pro zpracování obrazových dat. Mezi jeho charakteristiky patří konstrukce pro tvorbu funkcí speciálně zpracovávající nebo vracející obrazová data. V takovém případě pak automaticky osvobozuje programátora od přetypování hodnota a podobně, protože v takovém případě lze typy odvodit. Nevýhoda je stejná jako u jazyka Lua a to délka některých konstrukcí a používání klíčových slov. Software Mathematica obsahuje relativně jednoduchou syntaxi co se používání funkcí týče, avšak je komerční program a je relativně složité tvořit obecnější nebo vlastní algoritmy. Jedná se totiž o specifický jazyk, jehož některé konstrukce se nepodobají žádnému obyčejnému jazyku.

Jazyk Lua, stejně jako jazyk Python je použit ve velkém počtu aplikací, které tyto jazyky interně využívají. Zatímco podpora pro knihovnu OpenCV je v Pythonu relativně velká, pro jazyk Lua dlouho neexistovala ekvivalentní podpora. Nyní lze použít malou knihovnu LuaCV, která je portem knihovny OpenCV pro jazyk Lua. Návrh jazyka je proto řízen snahou využít interpret jazyka Lua spolu s knihovnou LuaCV (a tedy OpenCV jako často používanou knihovnou pro zpracování obrazu a počítačového vidění). Navrhovaný jazyk je pak nadstavbou samotného jazyka Lua.

2 The GNU Image Manipulation Program

Návrh syntaxe je navíc řízen snahou o co nejkratší kód, potřebný pro vytvoření jednoduchých až středně složitých programů, kde syntaxe jazyka Python nebo Lua, může obsahovat zbytečně dlouhé konstrukce plynoucí z obecnosti těchto jazyků. Dále je pak snaha o podobnou syntaxi s jazykem C++, aby by byl později možný alespoň poloautomatický překlad do tohoto kompilovaného jazyka z důvodu rychlosti vyvíjených funkcí nebo programů.

Pro shrnutí se jedná o dynamicky typovaný jazyk, u kterého je snahou aby byl dostatečně univerzální pro různé druhy úloh související s prací s obrazovými daty. Avšak zároveň mít speciální syntaktické konstrukce pro často se opakující vzory nebo pro úlohy objevující se při tvorbě programů pro zpracování obrazu.

3.1 Datové typy

Jak již bylo zmíněno v úvodu, jazyk je dynamicky typovaný. Což znamená, že proměnné nejsou svázány s jediným datovým typem, ale lze je v průběhu programu měnit. Typ jako takový mají jen hodnoty, proměnných. I když má typovaný systém řadu výhod, jako bezpečnost, kdy dovoluje řadu chyb objevit při překladu a často i rychlejší provádění programu, netypovaný systém umožňuje rychlejší vývoj u vhodných typů úloh a často je i programový kód kratší. Je však potřeba hlídat typovost a validitu operací za běhu programu. Zodpovědnost za platnost operací se tedy pouze přenáší z překladače na interpret.

Primitivní datové typy

Jsou datové typy, které nejsou definovány pomocí jiných datových typů. Jsou to tedy atomické hodnoty. Tak jako v jiných jazycích podporuje i navrhovaný jazyk číselný typ. Spolu s nimi jsou pro zvýšení bezpečnosti kódu implementovány jako primitivní datové typy i řetězce a tabulky (dále uvidíme, že implementace polí se provádí právě pomocí tabulek).

Zde je seznam základních datových typů jazyka:

`number`	Číslo
`bool`	Booleovská hodnota
`string`	Textový řetězec
`table`	Tabulka
`function`	Odkaz na funkci
thread	Odkaz na korutinu
userdata	Uživatelský datový typ
`null`	Neinicializovaná proměnná

Tabulka 1: Datové typy

Každá proměnná musí být explicitně inicializována a musí jí být přiřazena hodnota. Při referenci neinicializované proměnné je vrácena hodnota `null`.

Číselný typ

Je k dispozici pouze jeden datový typ pro číslo, nerozlišuje se tedy mezi celým číslem a číslem s plovoucí desetinou čárkou. Nejčastěji jsou čísla interně reprezentována pomocí datového typu `double`. Tato reprezentace však přináší potíž v tom, že binární operace lze provádět jen za pomoci rozšíření.

```
A = 128;
b = 1'000;        // Lze použít oddělovač pro velké čísla
c = 1.5;
```

Bool

Je typ, který může nabývat pouze hodnoty `true` nebo `false` reprezentující tak klasické booleovské hodnoty. Používá se u logických operátorů nebo podmínek. Avšak v podmínkách lze použít i jiné typy než pouze `bool`. Viz. Kapitola 4, Podmíněné provádění.

String

Jedná se o textový řetězec. Z pohledu programátora je bezpečnější mít speciální datový typ pro textové řetězce a s ním spojené operace. V jazyce C je řetězec uchováván jen jako pole znaků, reprezentováno pomocí číslic nějaké kódové tabulky a celé toto pole je pak ukončeno speciální hodnotu `NULL` (tzv. *sentinel*). To sice umožňuje větší flexibilitu, ale zároveň je to zdroj velkého množství chyb.

Jazyk dovoluje zadat textový řetězec klasicky jako seznam znaků uvozených do uvozovek. V tomto řetězci jsou také interpretovány klasické escape sekvence. Seznam používaných escape sekvencí je uveden v příloze.

Table

Lua používá univerzální typ tabulky, který je vlastně typ asociativního pole. Lze ho použít jako jednoduchou hašovací tabulku nebo v případě po sobě jdoucích celočíselných indexů jako obyčejné pole. S tím, že indexy mohou začínat od jakéhokoliv čísla. Pokud jsou navíc indexy po sobě jdoucí a zvyšující se o jedničku, může interpret použít i některé optimalizace a výsledné provádění může být při některých operacích urychleno.

Tabulky mohou sdružovat jak hodnoty, tak i funkce. Pomocí tabulek jsou tedy interně reprezentovány moduly nebo knihovny funkcí a lze jimi emulovat i objekty.

Bylo zmíněno, že při přístupu k nedefinované proměnné je vrácena hodnota `null`. Při přístupu do tabulky na nedefinovaný index je však vyvolána chyba.

Null

Hodnota `null` je speciální hodnota, která vyjadřuje neinicializovanou proměnnou. se využívá u funkcí, které nevrací hodnotu. Místo speciálního případu tyto funkci vrací hodnotu `null`. Při pokusu o čtení nedefinované proměnné je také vráceno `null`, místo vyvolání chyby.

Pozn.: Jako literál pro tento typ lze použít jak `null` tak `nil`. Záleží na preferenci programátora.

Speciální datové typy

Vedle primitivních datových typů jsou podporovány speciální typy, pro které pro jednoduchost zápisu jsou definovány operátory s určitým chováním. Jsou to typy pro odkazy na funkce a korutiny, ale hlavně na uživatelsky definované datové typy. Protože jazyk a interpret je rozšiřitelný externími moduly, slouží tento typ k identifikaci typů napsaných v jazyce C. K velké různorodosti takových typů jazyk neobsahuje a ani nemůže obsahovat dané syntaktické konstrukce. Přístup a operace s takovými typy je pak na funkcích daného modulu.

Momentálně jediným vestavěným speciálním datovým typem, pro který jazyk obsahuje literál je typ barvy. Ten je možné zapsat stejně jako v CSS

```
blue = #0000ff;
alpha_red = #ff000099;
```

3.2 Operátory

Výrazy jsou základní prvek jazyka jak specifikovat požadované výpočty. Je potřebné mít jasně definovanou sémantiku operátorů, asociativitu, případně aritu. Zde je seznam všech podporovaných operátorů jazyka

Popis	**Symbol**
Sčítání	+
Odčítání	-
Násobení	*
Dělení	/
Mocnina	**
Modulo	%
Zaokrouhlení	%%
Logický součin	and
Logický součet	or
Logická negace	not, !
Délka pole	#
Rovnost	==
Nerovnost	!=, ~=
Větší nebo rovno	>=
Menší nebo rovno	<=
Menší než	<
Větší než	>

Tabulka 2: Operátory

Přiřazení	=
Přiřazení s přičtením	+=
Přiřazení s odečtením	-=
Přiřazení s násobením	*=
Přiřazení s dělením	/=
Přiřazení s modulo	%=
Přiřazení se zaokrouhlením	%%=
Vložení do pole	<=
Provede přiřazení pokud, proměnná není definována. Jinak ponechá starou hodnotu.	?=

Tabulka 3: Přiřazovací operátory

Za zmínku stojí operátory <= a ?=. Jde o zkratku často využívaných idiomů

```
array <= value;
// to samé jako
array[#array+ 1] = value;

v ?= 100;
// to samé jako
v = v or 100;
```

Přiřazení se zaokrouhlením je také vhodná zkratka neobjevující se u běžných jazyků, dovoluje rychle zaokrouhlit hodnotu na daný počet desetinných míst

```
value = 3.141592;
value %%= 3;

print(value);
// Vytiskne 3.141;
```

U obyčejného přiřazení lze použít i vícenásobné přiřazení

```
a, b, c = 1, 3.5, null;

// a=1, b=2
a, b = 1, 2, 3, 4;

// a=1, b=2, c=null, d=null
a, b, c, d = 1, 2;
```

Toho lze pak dále využít u funkcí, které vrací více hodnot. Pokud na pravé straně není stejný počet výrazů jako počet proměnných, ostatní proměnné jsou nastaveny na `null`. Pokud je zde více výrazů, přebývající hodnoty se ignorují.

3.3 Řízení toku programu

Jakýkoliv netriviální program vyžaduje možnost opakování určitých bloků příkazů nebo provádění kódu na základě podmínky. Navrhovaný jazyky pro tyto účely obsahuje podobné konstrukce jako jazyk C, ovšem nepodporuje žádnou formu `goto` příkazu s návěštími.

Bloky kódu

Je možné vytvořit blok kódu, který má vlastní kontext proměnných[3]. Po ukončení provádění tohoto bloku jsou pak dané proměnné automaticky smazány ze zásobníku.

```
{
   // seznam příkazů
}
```

Takové bloky jsou nejčastěji využívány u podmínek a cyklů, které se aplikují pro tyto konstrukce. Lze je však vytvářet i samostatně a tím tak zabránit tzv. znečištění velkým počtem proměnných.

Podmíněné provádění

Podmíněné provádění kódů a související konstrukce jsou alfou a omegou [HYDE01, str. 439] vyšších programovacích jazyků. Proto i navržený jazyk obsahuje konstrukce pro provádění kódu na základě podmínek. Syntaxe je podobná jazyku C, proto se nejedná o nic nového. Jsou použity klíčová slova `if`, `elif` a `else`. Příklad zápisu je uveden v příloze C.3.

Způsob vyhodnocování podmínky je však rozdílný. V případě, že se jako podmínka uvede pouze proměnná, vyhodnocení probíhá následovně. Pokud je proměnná `null` nebo `false`, vyhodnotí se podmínka jako false. Pro všechny ostatní hodnoty se podmínka vyhodnotí jako true. Pro relační operátory platí, že výsledek je vždy typu `bool`. Nejdříve se porovnává typ daných proměnných nebo hodnot. Pokud typ nesouhlasí, výsledek je `false`. Pokud souhlasí, potom se porovnávají samotné hodnoty.

Cykly

Velké množství programů stráví velkou část svého času při provádění svých instrukcí ve smyčce [HYDE01, str. 489] na moha místech. Proto je nutné mít podporu pro konstrukce, které toto dovolí. Jazyk obsahuje dvě klasické smyčky, známé z ostatních jazyků. Jazyk obsahuje syntaktickou podporu pro smyčky `while` a `for`, přičemž oproti jazyku Lua nebo C, přináší i některá vylepšení.

---

3 Anglický termín je *variable scope*

While

Je klasický cyklus, který provádí blok kódu tak dlouho, dokud je podmínka pravdivá. V základu se sentaxe nijak neliší od jazyka C (ukázku lze vidět v příloze C.4).

Pro tento cyklus lze zadat i blok `else`. Ten se provede, pokud podmínka selže hned na začátku a tělo cyklu by tak nebylo provedeno ani jednou.

```
while (false)    {
      print("won't be printed");
}else{
      print("OK");
}
```

Pomocí konstrukce `while` lze vytvořit i nekonečnou smyčku, stačí neuvést žádnou podmínku

```
   while   {

      action();

   }
```

for

Standardní cyklus, který má několik možností zápisu. Ty by měli zjednodušovat zápis některých běžně používaných operací pro tuto smyčku. První dokáže iterovat přes určitý číselný rozsah.

```
for (i in 1:3)    {
      print(i);
}
```

Jako dokáže iterovat nad přímo zadanou tabulkou.

```
for (i in {2, 4, 6})    {
      print(_key_ .. " => " .. i);
}
```

for2

Je zkratkou dvou zanořených smyček `for`. Tato konstrukce se hodí například pokud potřebujeme provést průchod maticí hodnot. Zde je příklad

```
for(i in {1, 2, 3})    {
      for(j in {4, 5, 6})    {
            print(i, j);
      }
}

// Jde nahradit kratším
for2(i, j in {1, 2, 3}, {4, 5, 6}) {
      print(i, j);
}
```

Příkaz break

U smyček `while` a `for`, lze použít tyto příkaz break pro řízení toku programu uvnitř smyčky. Ten způsobí ukončení provádění celé smyčky.

Momentálně je nutné mít příkaz break, stejně jako příkaz return jako poslední příkaz v bloku

```
for(i in {1, 2, 3 ,4})    {
   print(i);
   if(i > 2){
         break;
   }
}


// Vypise:
// 1
// 2
// 3
```

3.4 Funkce

Funkce jsou brány jako objekty první třídy. To znamená, že mohou být uloženy v proměnných, předávány jako argumenty nebo použity jako návratové hodnoty. Funkce jako takové tedy nemají jméno, jsou pouze spjaty s proměnnou, až ty mohou být pojmenovány. To sebou nese vysokou flexibilitu. Program dokonce může některé funkce dočasně odstranit a vytvořit tak bezpečné prostředí pro externí skripty.

Volání funkcí

Při volání funkcí lze použít standardní syntaxi, známou z jazyka C/C++.

```
func(arg1, arg2);

func();
```

Uživatelské funkce

Koncept programové abstrakce je známý již od 80.let. Dokonce i první Babbagův počítač umožňoval znovupoužití kartiček s instrukcemi na několika místech v programu [SEB, str. 388]. Tato snaha o znovupoužitelnosti kusů kódu je v dnešních jazycích implementována pomocí podprogramů nebo funkcí. Protože žádný jazyk nebo knihovna nemůže mít všechny možné funkce, které může programátor ve svém programu potřebovat, tak každý rozumně použitelný jazyk musí mít možnost definování vlastních uživatelské funkce.

Implementovaný jazyk používá následující syntaxi:

```
def process()    {

}

def funcName(param1, param2) {
      //seznam příkazů
}
```

Každá funkce může vracet hodnotu. Tu lze specifikovat pomocí příkazů `return`. Pokud funkce nepoužívá příkaz `return`, je implicitně vrácena hodnota `null`.

Návrat z funkce

Do kategorie řízení toku programu lze zařadit i příkaz `return`. Tento speciální příkaz umožňuje vyskočit ven z prováděné funkce a tak ukončit její provádění, případně specifikovat hodnotu, která se má vrátit po ukončení funkce volajícímu. Příkaz lze použít vícekrát v kódu funkce.

```
def foo(flag)    {
      if(flag == true) {
            return 100;
      }
      return null;
}
```

Funkce mohou vracet i více hodnot. Příkaz `return` tak může obsahovat seznam výrazů

```
…
if(not ok)  {
      return err_code, "index je mimo rozsah";
}
```

Z technických důvodů je možné příkaz `return` možné použít jen jako poslední příkaz v bloku. To mimo jiné slouží k eliminaci mrtvého kódu. Tím pádem může být interpret zjednodušen o tyto optimalizace.

Funkce s proměnným počtem parametrů

Jak vestavěné tak i uživatelsky definované funkce mohou obsahovat proměnný počet parametrů. Při předávání argumentů totiž platí podobná pravidla jako s mnohonásobným přiřazením. Pokud je funkce předán menší počet argumentů, jsou zbývají hodnoty implicitně brány jako `null`. Pokud je však naopak počet argumentů větší, než funkce vyžaduje, jsou přebývající argumenty jednoduše ignorovány.

Pro uživatelsky definované funkce, lze jako proměnný počet parametrů uvést speciální operátor "…" , který poté obsahuje seznam všech dalších argumentů. Lze s ním pracovat i jako s proměnnou.

```
def show(a, ...)  {
      print("a = ", a);
      print("zbytek argumentů: ", ...);
}
```

3.5 *Třídy*

Jsou podporovány základní konstrukce pro tvorbu tříd. Je zde však řada omezení oproti klasickým objektově orientovaným jazykům. Zde je příklad jednoduché třídy:

```
class Temperature      {
      celsius = null;

      // konstruktor
      def new()   {
            … Zjisti teplotu ...
      }

      def setNewTemperatur(t) {
            self.celsius = t;
      }

      def getCelsius()  {
            return self.t;
      }

      def getFarenheit()      {
            return self.t * 1.34;
      }

}
```

Pro třídy lze definovat 2 základní členské objekty. Jsou to atributy a metody. Bohužel nejsou podporovány specifikátory omezení přístupu.

Atributy a metody

Atributy jsou vytvářeny shora dolů, jak jsou deklarovány. Stejně jako obyčejné proměnné. Atributy jsou použitelné uvnitř metod až poté, co jsou deklarovány. To souvisí s tím, že na pozadí jsou třídy implementovány pomocí tabulek.

Metody oproti obyčejným funkcím je vhodné volat pomocí operátoru dvojtečka místo tečky.

```
obj = new someClass;
obj:someMethod();
// Stejné jako
obj.someMethod(obj);
```

To souvisí s tím, že jako první parametr se předává odkaz na samotný objekt. Operátorem dvojtečka se však toto předání provádí automaticky. Odkaz na vlastní objekt uvnitř metody a tedy pomocí něj na data konkrétního objektu lze zpřístupnit pomocí automatického parametru `self`.

3.6 Jmenné prostory

Jako jeden z požadavků na jazyk a jeho interpret, byla možnost následného rozšíření. Toto je zamýšleno pomocí externích modulů. Tyto moduly pak mohou přidávat vlastní funkce, napsané v nativním jazyce C/C++. Protože při takovémto rozšiřování velice snadno hrozí kolize jmen, jazyk podporuje pojem jmenného prostoru. Ten je však velmi omezený na rozdíl od jazyka C++. Obecná definice zní, že je to kontejner obsahující množinu identifikátorů, umožňující tak mezi nimi jednoznačně rozlišovat.

Každá knihovna vložená do skriptu pomocí direktivy `require`, nejčastěji vytváří další jmenný prostor se jménem knihovny, ve kterém jsou umístěny všechny identifikátory této knihovny (implementačně se jedná o tabulku). Všechny identifikátory se pak skládají ze dvou částí – ze jména prostoru a samotného názvu identifikátoru. Pro volání funkcí z modulu nebo přístupu k datům lze použít tečkovou notaci pro příjemnější syntaxi. Tento jazyk navíc zavádí operátor *::*, který se při překladu nahrazuje za obyčejnou tečku, avšak umožňuje ve zdrojovém souboru snadno rozlišit modul od obyčejné tabulky vytvořené v kódu.

```
transform::scale(img, 0.9);
// je stejné jako
// transform["scale"](img, 0.9);
// transform.scale(img, 0.9);
```

3.7 Zpracování chyb

Jazyk přináší velmi omezenou podporu pro zpracování výjimek. Syntaxe podporuje bloky `try` a `catch`. A však zpracování výjimek je velice odlišné od jiných jazyků, podporující pravé objektově orientované paradigma. Uveďme si nejdříve příklad

```
try  {
      // …
      //if error
      throw 34;

      // …

      //if error
      throw "Nastala chyba";
}catch(Number e)  {
      print(e);
}catch(String e)  {
      print(e);
}catch all{
      print("Neočekávaná výjimka");
}
```

Z technických důvodů a také z důvodu emulované podpory objektově orientovaného programování, lze zpracovávat výjimky jen na základně primitivních datových typů. Nelze tak pro zpracování výjimek jako například v jazyce C++ nebo Java využít dědičnost. Také nelze specifikovat typ výjimky podle uživatelsky definovaných typů. Ty totiž nejsou uvnitř interpretu nijak speciálně odlišovány.

Mimo těchto konstrukcí lze pro zpracování chyb použít i funkce `pcall` a `error`[4]. Ty jsou také interně využívány překladačem pro generování kódu pro zpracování výjimek.

4 Viz. dokumentace jazyka Lua, kapitola Error Handling and Exceptions (http://www.lua.org/pil/8.4.html)

3.8 Komentáře

Jsou podporovány 2 typy komentářů, stejně jako v jazyce C/C++. Řádkový komentář, umožňující komentování krátkých úseků kódu a blokové, které mohou být umístěny na několika řádcích.

```
// komentáře
// Další komentář

/* Dlouhý komentář
na více
řadcích
*/
```

Jediné omezení, které jsou na komentáře kladeny, je nemožnost používat komentáře uvnitř textových řetězců.

Speciální blokové komentáře, začínající znakem `/**` se nazývají dokumentační komentáře. I když nejsou nijak využity interpretem nebo syntaktickým analyzátorem, lze je využít stejně jako u jiných jazyků pro automatické generování dokumentace. Příkladem takových nástrojů může být Doxygen, JavaDoc nebo PHPDocumenter. Problém je, že tyto nástroje většinou provádí alespoň částečnou syntaktickou analýzu zdrojového textu, takže je nelze snadno použít pro navržený jazyk.

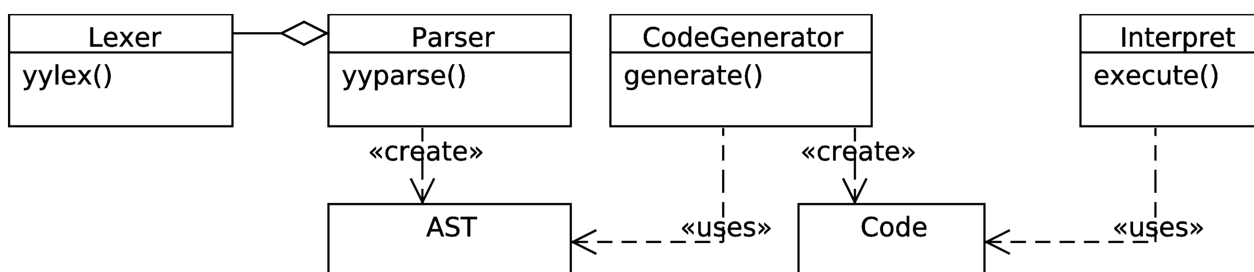
4 Implementace

V této kapitole je uveden popis jednotlivých nástrojů a postupů, které byly použity pro implementaci. Jsou zde také popsány důvody, které vedli k jejich použití. Postupně jsou popsány jednotlivé fáze překladače jazyka.

V první části je rozebrána základní architektura aplikace. V další části jsou popsány základní charakteristiky jazyka Lua, na kterém navrhovaný jazyk staví, v následující části je popsáno řešení lexikální a syntaktické analýzy, v předposlední části je popsáno řešení spouštění kódu a možnost dynamického rozšíření o další funkce interpretu. V poslední části je popsáno řešení funkcí pro zpracování obrazu a jejich možnosti.

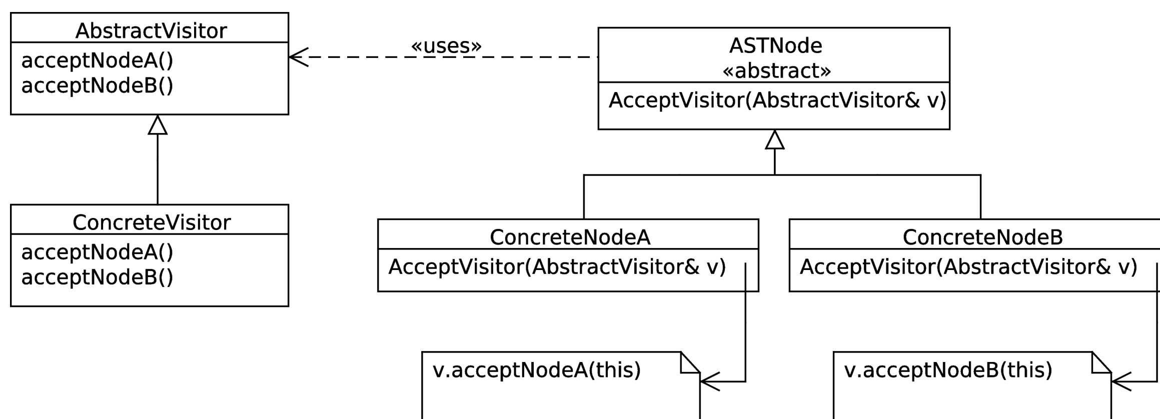
4.1 Architektura systému

Zde je popsána vysokoúrovňová architektura celého systému. Protože je pro implementaci použit jazyk C++ s využitím objektově orientované přístupu, lze systém popsat pomocí následujícího diagramu.



Obrázek 1: Architektura překladače

Jak je vidět, celý systém je koncipován jako řada po sobě jdoucích částí, kdy výstup jednoho subsystému je veden jako vstup dalšího. Tímto lze oddělit jednotlivé části a vyvíjet je tak samostatně. První část je lexikální analyzátor, který je navrhnut jako služba pro syntaktický analyzátor. Tvoří mezi sebou závislost. Výstupem syntaktického analyzátoru je abstraktní syntaktický strom, speciálně uzpůsobený pro navržený jazyk. Obecně platí, že pro každou konstrukci nebo klíčové slovo, existuje jedna třída. Ta si hlídá potřebné parametry, případně návaznosti na další potomky. Hlavním úkolem abstraktního syntaktického stromu je uchování struktury programu, tak jak byl napsán ve zdrojovém kódu. Veškeré další operaci se dějí jinde. Strom používá návrhový vzor Návštěvník. Třídy využívají tento vzor pro snadný průchod stromem. Takové třídy jsou pak využity například pro generování kódu spustitelného interpretem nebo jednoduchý přepis kódu pro snadné ladění aplikace. Výstup generátoru kódu je veden na samotný interpret, který vykoná cílový program. Výsledkem je buď chybové hlášení nebo akce specifikované ve zdrojovém kódu.



Obrázek 2: Vzor Návštěvník pro abstraktní syntaktický strom

4.2 Lexikální a syntaktická analýza

Pro tvorbu lexikálního analyzátoru byl zvolen open-source nástroj Ragel[5]. Je určený pro tvorbu obecných stavových automatů. Automat je možné popsat ručně, tedy specifikovat stavy a přechody mezi nimi, nebo pomocí regulárních výrazů, vhodných pro popis lexikálních jednotek programovacího jazyka. Jeho výhoda na rozdíl od známého nástroje *Flex* spočívá v příjemnější syntaxi a daleko větší flexibilitě při vyhodnocování stavového automatu. Zatímco nástroje Flex nejdříve vyhodnotí daný vstup a až poté co dojde ke shodě se vzorem, může být spuštěna událost, Ragel dovoluje přiřadit události i jednotlivým přechodům nebo stavům uvnitř automatu. Ne jen koncovým stavům jako u programu *Flex*. Pro účely vývoje a testování se ukázala výhodou i možnost generovat grafickou reprezentaci výsledného automatu. V příloze je poté uveden příklad syntaxe pro jednoduchý stavový automat v jazyce Ragel.

Výhodou bylo, že odpadla nutnost provádět složitější transformaci analyzovaných lexémů, jako například konverze escape sekvencí uvnitř textových literálů. A to hlavně z důvodu následné interpretace popsané v další kapitole.

Z důvodu nestálosti gramatiky jazyka hlavně v jeho počátcích byl pro syntaktickou analýzu zvolen volně dostupný nástroj pro tvorbu LR analyzátorů Bison. Jedná se o pokračovatele programu Yacc a přináší hlavně objektově orientovaný výsledný kód, který je snazší integrovat do architektury celé aplikace. V příloze je uveden příklad popisu gramatiky v tomto nástroji. Bylo snahou popsat jazyk takovou gramatikou, aby byla jednoznačná. Při analýze zdola-nahoru, mohou vznikat nejednoznačnosti, kdy analyzátor neví, jestli načíst další token, nebo redukovat symboly na zásobníku. V takovém případě dokáže Bison vygenerovat takzvaný GLR[6] analyzátor ,kdy se začne provádět několik analýz paralelně, pokud není jasné, kterou akci použít. Takový analyzátor však může být složitý a provádění nemusí být rychlé v případě velkého počtu nejednoznačností v gramatice. Řešení vedlo na zavedení dalších pomocných pravidel do gramatiky a to například u popisu výrazů nebo konstrukcí jako jsou smyčky. Kde je možné použít několik variací pro jejich

5 http://www.colm.net/open-source/ragel/
6 Generalized LR

zápis. Ve výsledku se však takový postup projevil jako výhodný, protože to vedlo na možnost jemnějšího řízení generovaného abstraktního syntaktického stromu.

Jedinou nevýhodou při použití generátoru Bison mohou být docela strohé chybové hlášení v případě syntaktických chyb. Momentálně při chybě zobrazí jen číslo řádku a případně jaké symboly analyzátor očekával. Nástroj umožňuje jednoduché zotavení z chyb, případně výpis vlastního popisu chyby, to však vede na další speciální pravidla v gramatice pro tento účel. Z důvodu nestálosti gramatiky, jak již bylo zmíněno výše, proto není tato vlastnost implementována.

4.3 Jazyk Lua

Je skriptovací dynamicky typovaný jazyk, speciálně uzpůsobený pro snadnou rozšiřitelnost, jednoduchou syntax a malému počtu atomárních datových typů [LUA01]. Snaží se tak dosáhnout síly a zároveň malé velikosti výsledného kódu. Je tak vhodný i do jednoduchých systémů nebo vestavěných zařízení. Díky těmto vlastnostem je použit ve velké řadě aplikací jako hry nebo aplikačně specifické programy s různým stupněm potřeby umožnit uživateli aplikaci upravit nebo automatizovat jeho chování pomocí skriptu.

Interpret jazyka Lua se pak hodí i pro tvorbu jiných jazyků. Díky jeho elementárním operacím, může být kód převeden do formy vhodné pro tento interpret, který se následně postará o jeho spuštění. Cože se děje prakticky běžně. I samotný jazyk Lua je použit jako základ, který je dále upraven pro potřeby konkrétní aplikace. Uživatel tak nemusí mít v první chvíli ani ponětí, že pracuje právě se systémem Lua. Tento přístup je také použít dále při implementaci navrženého jazyka této práce.

Interpret tohoto jazyka obsahuje některé pokročilé možnosti a samotní autoři udávají, že pro relativně jednoduché až středně pokročilé aplikace se jedná o jeden z nejrychlejších interpretů. To mimo jiné souvisí s tím, že prakticky všechny funkce nebo standardní moduly jsou volitelné. Pokud je tedy aplikace, která jazyk Lua využívá nepotřebuje, nemusí je vůbec použít.

Příjemná vlastnost jazyka Lua je velké množství přispívajících vývojářů. Jako externí modul pro tento jazyk je tak i volně šířitelná implementace just-in-time kompilátoru (LuaJIT). Ten provádí překlad částí kódu do nativního binárního kódu procesoru a v některých případech slibuje znatelné urychlení aplikace. Toho lze využít pokud je potřeba vysoký výkon aplikací. Jako ostatní části aplikace, i LuaJIT je dostupný na relativně velký počet platforem. Od Windows, Linux až po Playstation4 nebo XBox360.

4.4 Interpretace a spouštění kódu

Spouštění navrhovaného jazyka je řešeno tak, že z přeloženého kódu je vygenerován zdrojový kód v syntaxi jazyka Lua, který je použit jako vstup pro jeho interpret, který následně kód vykoná. Podmínkou je, aby zdrojové kódy v navrhovaném jazyce a následné kódy pro jazyk Lua byli sémanticky ekvivalentní. Protože je syntaxe jazyka Lua velmi jednoduchá a obsahuje spoustu elementárních konstrukcí, které lze použít pro tvorbu nových a složitějších, může navržený jazyk obsahovat řadu dalších a složitějších syntaktických konstrukcí, které usnadňují některé často používané operace nebo idiomy. Stejný postup používají i překladače pro jazyky jako C, kdy se kód generuje do assembleru. Prakticky je takovýto postup omezen pouze samotným interpretem, nebo obecně nižší vrstvou, která kód zpracovává, a jeho možnostmi. Což je na druhou stranu také nevýhodou. Není totiž snadné nebo někdy nemožné vytvořit konstrukce nad rámec interpretu bez toho abychom upravili i ten.

Druhým nedostatkem mohou být zmatečné chybové hlášení, vzniklé během provádění kódu. To je vzniklé hlavně tím, že řádky kódu se nemusí shodovat s řádky výsledného kódu pro interpret Lua. Proto aplikace nabízí možnost uložit mezikód v jazyce Lua, který je poté možné použít pro ladění aplikace.

Rozšiřitelnost

Jak již bylo zmíněno interpret jazyka Lua, lze rozšířit pomocí modulů. Uvnitř skriptu lze importovat moduly napsané také v jazyce Lua, ale běžnější je dané moduly napsat v jazyce C, kde kód daných funkcí běží nativně na procesoru. To je také důvod rychlosti tohoto jazyka.

Vytváření takových knihoven je také relativně jednoduché. Lua obsahuje dobře navržené API s malým počtem funkcí pro tuto úlohu. Stěžejním je výměna dat mezi interpretem a nativní funkcí psanou v C. To se děje pomocí zásobníku, takže práce je velice intuitivní. Zásobník slouží jak pro předávání parametrů funkce zevnitř interpretu, tak i návratových hodnot zpět do interpretu.

Moduly jsou ve výsledku řešeny dynamickými knihovnami, které lze programově nahrát uvnitř kódu. Tyto knihovny jsou pak nejčastěji implementovány v jazyce C, ale je možné je implementovat i v jiném jazyce, pro které překladač dokáže generovat dynamickou knihovnu pro danou platformu. Díky tomu je kód takového modulu velice efektivní a rychlý během provádění.

Jedna nepřímá výhoda řešení pomocí interpretu Lua je, že lze použít velké množství již existujících rozšiřujících knihoven pro tento jazyk i se syntaxí nově navrhovaného jazyka, který je do jisté míry syntakticky zpětně kompatibilní s jazykem Lua.

4.5 Funkce pro zpracování obrazu

Pro zpracování obrazu existuje velké různé množství knihoven pro různé jazyky a s různým počtem funkcí a možnostmi. Jedna z nejznámějších je knihovna OpenCV, pro jazyk C. Tato knihovna obsahuje velké množství funkcí pro zpracování obrazu a počítačové vidění a je také velice dobře optimalizována. Dokáže využít GPU kde je to možné a provádět i složité výpočty relativně rychle.

Další hlavní výhodou i rozšířenost OpenCV na velkém množství platforem. Protože je pak pro navrhovaný jazyk řešena jako externí modul, lze ho tak vytvořit na tyto různé platformy a tím zajistit relativně snadnou přenositelnost, celého systému.

Většina funkcí pro zpracování obrazu je interně pouze adaptér nad funkcemi knihovny OpenCV. Jedná se o multiplatformní knihovnu pro manipulaci obrazu. Její funkce dokáží využít akceleraci na GPU a tím běžet velice rychle.

Pro jazyk Lua je využita knihovna LuaCV šířena pod LGPLv3 licencí. Vznikla jako bakalářská práce pana Jiřího Prymuse. Jedná se o projekt naportování OpenCV knihovny právě pro jazyk Lua. Bohužel ne všechny funkce z OpenCV jsou v LuaCV podporovány. I když je knihovna přeložitelná s OpenCV 2.4.x, byla implementována pro OpenCV 2.2 a tak nedrží krok s nejonvějším OpenCV API. Není podporována například práce s XML soubory, chybí podpora pro feature2d modul a mohou vznikat odlišnosti kvůli odlišnosti jazyka C/C++ pro který je nové OpenCV primárně vyvíjeno. I přesto je podporován téměř celý modul core, dále knihovna obsahuje funkce z modulů calib3d, highgui, imgproc, video, legacy a základní funkce z modulu objdetect.

Při integraci do systému bylo nutné vyřešit kompatibilitu knihovny LuaCV s vývojovými knihovnami OpenCV. Originálně byla vyvíjena pro starší verzi OpenCV 2.2. Nyní novější verze OpenCV 2.4 však prepracovala použití některých datových struktur a funkcí, takže není zpětně kompatibilní bez zásahu do zdrojových kódů. Bylo tak na některých místech nutno přidat hlavičkové soubory `legacy.h` a `compat.h.` Dále je potřeba vyřešit přilinkování dynamických knihoven, pro tyto soubory. A to zejména pro systém Windows. Na systémech Linux při použití nástroje `pkg-config` jsou tyto knihovny zahrnuty implicitně.

5 Testování

Porovnávat programovací jazyky lze hned z několika hledisek. Jako dva hlavní aspekty, na které se zaměřují následující testy navrženého jazyka je rychlost, jakou dokáže programátor danou myšlenku zaznamenat, nebo ekvivalentně jak dlouhý text musí programátor napsat pro splnění daného úkolu. Čím dokáže programátor zapsat složitější program na co nejméně znaků nebo řádků, říkáme, že má jazyk větší expresivní syntax [KNESL]. Naproti tomu jde druhý aspekt, tedy jak snadno lze zapsaný kód číst. To však z velké části závisí na subjektivním pocitu a také na programovacích znalostech každého programátora. Tímto způsobem hodnocení se proto zde nebudeme dále zabývat, protože nelze snadno vytvořit objektivní nebo kvantitativní hodnocení.

Druhý způsob hodnocení návrhu může brát v potaz požadavek na možnost konverze kódu do kódu v jazyce C nebo C++. Protože dynamický jazyk lze převést na čistý kód staticky typovaného jazyka jako je C jen za pomocí určité podpory běhového prostředí nebo dalších knihoven, v mnoha jazycích se tak jedná spíše o experimentální projekty. Jinak je vždy nutná aspoň minimální ruční zásah do takto převedeného programu. Protože však tato vlastnost překladače nebyla dokončena, není momentálně možné vytvořit objektivní test nebo porovnání.

Všechny následující testy byly prováděny na neoptimalizovaných programech zachovávající zhruba stejný styl zápisu kódu, jako stejné řádkování počet bílých míst, tak aby byl kód v daném jazyce čitelný bez velkých problémů. Provedené testy tak nemusí být přesné a může se jednat o přibližné hodnoty.

5.1 Porovnání délek holých konstrukcí

Jako první test bylo porovnání délky kódu jednotlivých konstrukcí bez uživatelského kódu. Tedy jen minimální délky kódu takových konstrukcí. Všechny testy byli navíc prováděny i s ohledem na jazyk Python. V tabulce níže jsou pak uvedeny poměry délky kódy navrženého jazyka oproti jazykům Lua a Python. Výsledky jsou uváděny v procentech. Čím nižší číslo, tím byl kód v navrženém jazyce kratší. Příklad zápisu kódu pro takový test je uveden v příloze B.

	Lua	**Python**
If	91.5 %	153.6 %
While	68.1 %	100 %
For	77.6 %	88 %
Uživatelsky definovaná funkce	74.3 %	123.8 %
Třídy	24.4 %	101.4 %

Tabulka 4: Srovnání základních konstrukcí

5.2 Porovnání délek ostatních typů kódu

Jako druhý test bylo poté provedeno porovnání kódu konstrukcí s ostatním potřebným uživatelským kódem, jako volání funkcí, používání proměnných a dalších, jako simulace reálného kódu. Tento test má zjistit v jakém poměru jsou délky takového kódu s ostatním zdrojovým textem. Je potřeba uvést, že takový test se může lišit podle stylu zápisu kódu a konkrétní volbě identifikátorů.

	Lua	**Python**
If	94.8 %	113.3 %
While	85.6 %	104.1 %
For	90.6 %	102.8 %
Uživatelsky definovaná funkce	96.1 %	147.6 %
Třídy	49.6 %	116.3 %

Tabulka 5: Srovnání konstrukcí s uživatelským kódem

Posledních pár testů pro srovnání proběhlo za pomocí malých programů jako problém 8 dam, malé skripty pro úpravu obrázků a podobně. Zde byla snaha porovnat délku kódu, který lze reálně napsat.

	Lua	Python
Problém 8 dam	89.2 %	147.8 %
Zmenšení obrázků	101.9 %	106.1 %
Série filtrů nad obrázkem	104.4 %	106.7 %

Tabulka 6: Srovnání délky kódu krátkých programů

5.3 Interpretace testů

Z těchto pár testů vidět, že navržený jazyk umožňuje psát v průměru kratší kód než jazyce Lua, pokud jsou využívány konstrukce pro řízení běhu programu a jiné. Při jednoduchých volání funkcí z knihovny OpenCV již není rozdíl tak velký. Oproti tomu rozdíl oproti jazyk Python není v žádném testu tak markantní a to hlavně díky své syntaxi využívající bílé znaky. Oproti výše zmíněným konstrukcím, však je Lua výhodnější pro práci s programy využívající tabulky, kde navržený jazyk nepodporuje veškerou syntaxi pro práci s nimi.

Průkaznější statistiku lze však získat jen za pomocí velkého množství zdrojových textů. Navíc by všechny tyto zdrojové texty museli obsahovat sémanticky stejný program. V takovém případě by se do porovnání promítly i ostatní speciální konstrukce nebo speciální operátory a různé idiomy programovacího jazyka, které mají také značný vliv na délku potřebného kódu.

Z toho vyplývá, že navržený jazyk nemusí být vhodný ke všem účelům, ale pro práci s knihovnou OpenCV může být srovnatelný s jazykem Python. Pokud je pak využit překlad do čistého jazyka Lua, může se pro vývoj programů jednat o vhodnou alternativu, než psát algoritmy v čistém jazyce Lua.

6 Závěr

Hlavním cílem práce bylo vytvořit skriptovací jazyk a možnost jeho spouštění pro zpracování obrazu s tím, že by jeho kód byl relativně dobře čitelný a umožňoval co nejkratší zápis kódu. To bylo splněno jazykem, jehož syntaxe připomíná jazyk C++, který zná většina programátorů, ale navíc obsahuje speciální konstrukce a funkce pro zpracování obrazu v podobě knihovny LuaCV. Jeho interpret je pak postaven na interpretu jazyka Lua. Ten umožňuje rychlé spouštění kódu a relativně malé množství silných konstrukcí, které je snadné se naučit a efektivně používat k rychlému psaní programů.

Jako první bod zadání bylo studium dostupných materiálů. Mezi ně patřily publikace o teorii programovacích jazyků, různých metod jejich zpracování a spouštění. Všechny podstatné části jsou uvedeny v druhé kapitole věnující se skriptovacím jazyků a překladačům. Navržený jazyk je pak popsán v následující kapitole, kde je uveden hrubý popis syntaxe a konstrukcí, které nejsou běžné u ostatních používaných jazycích. Způsob spouštění kódu jako třetí bod zadání práce je popsán v kapitole věnující se implementaci překladače jazyka. Z porovnání jazyků lze pak usoudit, že do určité míry byl splněn i cíl o zkrácení nutného kódu, při programování a to hlavně oproti jazyku Lua, na kterém je navržený jazyk z větší části postaven.

Jako poznatek z pozdější fáze při práci s jazykem, by mohla být možnost přidání deklarativních konstrukcí do jazyka, podobné jako používají kaskádové styly úspěšně používány na webu. I když hrozí určité omezení obecnosti jazyka, jednalo by se o zajímavý způsob jak zkrátit nebo zjednodušit programový kód. Takové deklarativní konstrukce by pak mohli fungovat jako filtry, které by byly znovupoužitelné. Pro často používané operace by se zabránilo repetetivnímu psaní stejného kódu.

V průběhu návrhu a implementace bylo zjištěno, že není snadné vytvořit zcela nový jazyk, který pro používané algoritmy využívá ve velké míře imperativního přístupu, který je prakticky nezbytný ve všech programech snažící se o nové postupy nebo řešící nestandardní přístupy. Imperativních jazyků existuje velké množství. Ty jsou dnes velice propracované a obsahují velké množství ověřených postupů a idiomů. Základním prvkem návrhu autorova jazyka tak nebyla reimplementace takovýchto jazyků, ale spíše jejich rozšíření a to na základně úpravy syntaxe a v první řadě zkrácení často používaných konstrukcí. Výsledný systém je pak vhodný spíše pro znalejší uživatele skriptovacího jazyka Lua, kde jeho znalost může zjednodušit vývoj v navrženém jazyce.

Myšlenek pro další vývoj je hned několik. Bylo by vhodné upravit knihovnu LuaCV tak, aby podporovala modernější Lua 5.3 a případně výše. Tato nová verze přináší některé vylepšení, odstraňuje chyby a obecně se snaží o lepší API. Není tak zpětně kompatibilní s verzí 5.1 a níže, pro které bylo LuaCV původně implementováno. Dále by bylo jistě vhodné doplnit nové funkce knihovny OpenCV nebo počkat na další verzi této knihovny.

Literatura

[SEB]
Sebesta, Robert W., Concepts of Programming Languages, 10th edition,

[LUA01]
Lua: about. [online]. [2015] [cit. 2015-04-09]. Dostupné z: http://www.lua.org/about.html

[HYDE01]
HYDE, Randall. *Write Great Code: Thinking Low-Level, Writing High-Level*. Volume 2. William Pollock, 2006. ISBN 1-59327-003-8.

[PERL]
WALL, Larry, Tom CHRISTIANSEN a Jon ORWANT. Programming Perl. 3rd Edition. 1005 Gravenstein Highway North, Sebastopol: O'Reilly Media, 2000. ISBN 0596000278.

[PERL5]
HUSAIN, Kamran a Robert F. BREEDLOVE. Perl 5 UNLEASHED [online]. 1996 [cit. 2015-04-25]. ISBN 0-672-30891-6. Dostupné z: http://ods.com.ua/win/eng/program/Perl5Unleashed/

[ONEL]
KRUMINS, Peteris. Introduction to Perl one-liners. Introduction to Perl one-liners [online]. 2007 [cit. 2015-04-25]. Dostupné z: http://www.catonmat.net/blog/introduction-to-perl-one-liners/

[JIT]
RAMANAN, Neeraja. JIT through the ages: Evolution of just-in-time compilation from theoretical performance improvements to smartphone runtime and browser optimizations [online]. 2012 [cit. 2015-04-25]. Dostupné z: http://www.cs.columbia.edu/~aho/cs6998/reports/12-12-17_Ramanan_JIT.pdf. Columbia University.

[HIST]
UNIVERSITY OF HAWAII. History of Scripting [[online]]. 2005 [cit. 4.2.2015]. Dostupné z: http://www2.hawaii.edu/~nreed/ics313/lectures/scriptHistory.pdf

[CGI]
COLBURN, Rafe. Teach Yourself CGI in 24 Hours. 2nd edition. United States of America: Sams Publishing, 2003. ISBN 0-672-32404-0.

[ZUZ]
ZUZAŇÁK, Jiří. Script pro zpracování obrazu. Brno, 2007. Diplomová práce. Vysoké učení technické Brno. Vedoucí práce prof. Dr. Ing. Pavel Zemčík.

[SHEL]
Shell. [online]. [cit. 2015-04-25]. Dostupné z: http://www.webopedia.com/TERM/S/shell.html

[LINS]
ZAJÍC, Petr. Historie jazyka PHP. In: PHP [online]. 2004 [cit. 2015-04-25]. Dostupné z: http://www.linuxsoft.cz/article.php?id_article=171

[BASC]
ÆGIDIUS MOGENSEN, Torben. Basics of Compiler Design. University of Copenhagen, Denmark, 2010. Anniversary edition. ISBN 978-87-993154-0-6. Dostupné z: www.diku.dk/~torbenm/Basics/basics_lulu2.pdf

[PARST]
GRUNE, Dick a Ceriel J.H. JACOBS. *Parsing Techniques: A Practical Guide*. Springer Science+Business Media, LLC, 2008. 2nd edition. ISBN 978-0-387-20248-8.

[KNESL]
KNESL, Jiří. Jak se porovnává kvalita programovacího jazyka?. [online]. 2013 [cit. 2015-04-25]. Dostupné z: http://www.knesl.com/articles/view/jak-se-porovnava-kvalita-programovaciho-jazyka

[LLVM]
LATTNER, Chris. The Design of LLVM. Dr.Dobbs's : The world of software development [online]. 2012 [cit. 2015-04-26]. Dostupné z: http://www.drdobbs.com/architecture-and-design/the-design-of-llvm/240001128

[LDRS
R. LEVINE, John. Linkers and Loaders. the United States of America: Academic press, 1999. ISBN 1558604960.

Příloha A

A.1 Ukázka syntaxe popisu FSM v nástroji Ragel

```
%%{
	machine pex;
	alphtype char;

	new_line = '\n' @{ ++curline; };
	ws = [ \t\r] ;
	id = [_a-zA-Z] [_a-zA-Z0-9]* ;
	integer = [0-9]+ ('\'' [0-9]+ )* ;

	# FSM Entry Point
	main := |*

	'0x' [0-9a-fA-F]+ => {
		ret = Token::INTEGER;
		fbreak;
	};
	integer => {
		ret = Token::INTEGER;
		fbreak;
	};
	'for' => {
		ret = Token::KEYWORD;
		fbreak;
	}
	any => {
		std::cerr << "[Lexer] Unknown character sequence '" << *ts << "' on
line " << curline << std::endl;
		ret = Token::END;
		fbreak;
	};

	*|;
}%%
```

A.2 Ukázka popisu syntaxe pomocí nástroje Bison

```
constant : INTEGER { $$ = new ast::UInt($1); }
      | DOUBLE { $$ = new ast::Double($1); }
      | STRING { $$ = new ast::String( std::move($1) ); }
      | BOOL { $$ = new ast::Bool($1); }
      | NIL { $$ = new ast::Null(); }
      | COLOR { $$ = $1; }
      | list { $$ = $1; }
;

function-call : IDENTIFIER LPAREN arg-list RPAREN
{
   $$ = new ast::FCall( std::move($1), static_cast<ast::NodeList*>($3));
}

coroutine : COROUTINE LPAREN parameter-list RPAREN block
{
   $$ = new ast::Coroutine(static_cast<ast::NodeList*>($3),
static_cast<ast::Block*>($5));
};
```

Příloha B

B.1 Příklad testů pro holé konstrukce

Konstrukce if/then/else v navrženém jazyce

```
if() {

}


if() {

}elseif() {

}else {

}
```

Konstrukce v jazyce Lua

```
if then

end


if then

elseif then

else

then
```

Uživatelsky definovaná třída v navrženém jazyce

```
class  {
      def ()      {
      }

      def ()      {
      }
}
```

```
class  extends  {
      def ()      {
      }
}
```

Uživatelsky definovaná třída v jazyce Lua

```
 = {}

function :()
end

function :()
      object = {}
      mt = {__index = }
      setmetatable(object, mt)
      return object
end

function :()
end

function :()
end

 = {}
setmetatable(, { __index =  })
function Derived:new()
      t = {}
      mt = {__index = }
      setmetatable(t, mt)
      return t
end

function :()
end
```

Příloha C

C.1 Číselný literálů

```
d = 0xFFFF88;     // Hexadecimální zápis
e = 0xab00d5;
g = 4.57e-3
h = 1'2'7;
```

C.2 Escape sekvence

Sekvence	Název
\'	Jednoduchá uvozovka
\"	Uvozovka
\\	Zpětné lomítko
\a	Zvonek
\n	Nový řádek
\r	Návrat vozíku
\t	Tabulátor
\b	Backspace
\f	Form feed
\v	Vertikální tabulátor
\0	Null character

Tabulka 7: Escape sekvence

C.3 Podmíněné provádění

```
   if (v == false)     {

      doSomethingElse();

   }
```

```
   if (s == 0)    {

      foo();

   }elif (s == 1) {

      bar();

   }else {

      skip();

   }
```

C.4 Smyčky

```
i = 5;

while (i > 0)  {

   action();

   --i;

}
```

Smyčka for se specifikací číselného rozsahu, používá hodnoty včetně.

```
for (i in 1..3)   {
      print(i);
}

//Vypíše:
// 1
// 2
// 3
```

Pro iteraci přes přímo zadanou tabulku je dostupná magická proměnná `__key__`, která obsahuje hodnotu klíče nebo indexu aktuální hodnoty.

```
for (i in {2, 4, 6})    {
      print(_key_ .. " => " .. i);
}

//Vypise
"1 => 2"
"2 => 4"
"3 => 6"
```