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

SCRIPT PRO ZPRACOVÁNÍ OBRAZU

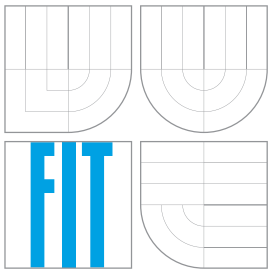
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ ZUZAŇÁK

BRNO 2007

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

SCRIPT PRO ZPRACOVÁNÍ OBRAZU

SCRIPT LANGUAGE FOR IMAGE PROCESSING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. JIŘÍ ZUZAŇÁK
AUTHOR

VEDOUCÍ PRÁCE — Doc. Dr. Ing. PAVEL ZEMČÍK
SUPERVISOR

BRNO 2007

Zadání diplomové práce

Řešitel: **Zuzaňák Jiří, Bc.**
Obor: Počítačová grafika a multimédia
Téma: **Script pro zpracování obrazu**
Kategorie: Uživatelská rozhraní

Pokyny:

1. Prostudujte základní informace o zpracování obrazu a dostupnou literatůru na téma script jazyky a interpretery script jazyků
2. Navrhněte syntax jazyka pro popis operací nad obrazy
3. Implementujte překladač jazyka do mezikódu a interpreter (nutno konzultovat)
4. Diskutujte dosažené výsledky

Literatura:

- Žára, J. Beneš B. Felker P.: Moderní počítačová grafika, Computer Press, 1998
- Dále dle pokynů vedoucího

Při obhajobě semestrální části diplomového projektu je požadováno:

- Body 1 a 2 zadání

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese
`http://www.fit.vutbr.cz/info/szz`

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problému a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 az 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programu. Informace v elektronické podobě budou uloženy na standardním pamětovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Zemčík Pavel, doc. Dr. Ing.**, UPGM FIT VUT
Datum zadání: 28. února 2007
Datum odevzdání: 22. května 2007

Licenční smlouva

Licenční smlouva v kompletním znění je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Výňatek z licenční smlouvy:

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávňení (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.

2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.

3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:

   ■ ihned po uzavření této smlouvy

   □ 1 rok po uzavření této smlouvy

   □ 3 roky po uzavření této smlouvy

   □ 5 let po uzavření této smlouvy

   □ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací).

4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona c. 111/1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Abstrakt

Tato práce pojednává o návrhu skriptovacího jazyka, určeného pro efektivní zpracování obrazu. Úvod této práce se zabývá studiem a osvojením si metod návrhu překladačů a interpretů, včetně jejich následné aplikace při návrhu skriptovacího jazyka a jeho interpretu. Práce dále popisuje metody návrhu a implementace interpretu, včetně automatizovaných metod využitých při návrhu implementovaného programu. Další část práce se zabývá popisem struktury a implementace navrženého programu, určeného pro generování překladače libovolného jazyka, popsaného jako vstup tohoto programu. Konec práce podrobněji popisuje navržený skriptovací jazyk, jehož implementace je založena na výše popsaných metodách.

Klíčová slova

Skript, skriptovací jazyk, překladač, interpret, zpracování obrazu, lexikální analýza, syntaktická analýza, sémantická analýza

Abstract

This thesis deals with design of scripting language, especially specified for effective image processing. Introduction of this thesis is focused on studying and also appropriation of methodology of compilers and interpreters design, include their following application in design of the scripting language and as well its interpreter. Another point of my work is showing the methods of design and implementation of the interpreter including automated methods used in the design of the implemented program. Next part deals with description of structure and implementation of the designed program, intended for generating compiler of any language which is described in input of this program. The conclusion of this work is more detailing description of the scripting language design; its implementation is based on the methods mentioned before.

Keywords

Script, scripting language, compiler, interpreter, image processing, lexical analysis, syntactic analysis, semantic analysis

Citace

Jiří Zuzaňák: Script pro zpracování obrazu, diplomová práce, Brno, FIT VUT v Brně, 2007

Script pro zpracování obrazu

Prohlášení

Prohlašuji že jsem tuto práci vypracoval samostatně na základě uvedené literatury pod vedením vedoucího práce. Uvedl jsem všechny literární prameny a publikace ze kterých jsem čerpal.

.....
Jiří Zuzaňák
22. května 2007

Poděkování

Děkuji vedoucímu práce Doc. Dr. Ing. Pavlu Zemčíkovi za vedení a cenné rady při zpracování diplomové práce.

Obsah

Kapitola 1

Úvod

Stejně jako projektový manažér ve firmě řídí své podřízené za účelem dokončení projektu, aniž přesně rozumí detailům jejich práce, tak skript zapsaný ve skriptovacím jazyku řídí spouštění systémových programů za účelem vykonání požadované akce. Skript nemusí znát implementační detaily jednotlivých systémových programů, postačuje mu znalost rozhraní přes které program přijímá a vrací data. Takový byl původní účel skriptovacích jazyků, ale v dnešní době jsou již skriptovací jazyky rozšířený téměř v každém odvětví informačních technologií a slouží k vykonávání velkého množství různorodých úkolů.

Způsob práce skriptů však zůstal zachován, jen jednotkami se kterými skript pracuje již nemusí být nutně jen systémové programy, ale mohou jimi být například samostatné funkce, objekty nacházející se v nějakém typu enginu, nebo kus hardwaru provádějící nějaké operace. Výhodou a zároveň důvodem rychlého rozšíření těchto jazyků je snadné programování, debugování a údržba skriptů.

Zpracování obrazu je ze svého principu vhodným kandidátem pro realizaci pomocí skriptovacího jazyka. Obrazové operace, mezi které patří například: načítání obrazu, úprava barev, různé typy filtrů apod. patří mezi výpočetně náročnější operace, tvořící množinu operací aplikovatelných na obraz. Pokud budeme prvky této množiny považovat za jednotlivé na sobě nezávislé operace je možné vztahy a posloupnost jejich provádění popsat právě skriptem. Tento skript pak bude obsahovat posloupnosti provádění jednotlivých operací, jejich návaznost a bude umožňovat i řízení sekvence provádění operací v závislosti na výsledcích jednotlivých operací.

Tato práce se zabývá návrhem skriptovacího jazyka určeného pro snadný zápis operací pracujících nad obrazovými daty. Cílem práce je studium metod návrhu a konstrukce interpretů, a na základě nabytých znalostí návrh skriptovacího jazyka a jeho překladače, případně interpretu. Jsou rozpracovány převážně metody návrhu a návrh interpretovaných skriptovacích jazyků. Samotná implementace jazyka pro zpracování obrazu ještě není dokončena.

Podrobnějším popis skriptovacích jazyků je obsažen v 2. kapitole nazvané Skriptovací jazyky. Nachází se v ní popis důvodů vzniku tohoto typu jazyků, porovnání s kompilovanými jazyky a výčet některých známějších typů skriptovacích jazyků a jejich využití.

Kapitola v pořadí 3. nazvaná Kompilátory, překladače a interprety se zabývá teorií o návrhu a implementaci překladačů, kompilátorů a interpretů. Budou v ní rozebrány základní pojmy této teorie, kterými jsou lexikální analýza, syntaktická analýza, sémantická analýza, intermediární kód a generování cílového kódu. Pojmy popsané v této kapitole budou v následujících kapitolách hojně používány.

Kapitola zabývající se návrhem samotného překladače se nazývá Navržený překladač

a je v pořadí 5. V této kapitole je podrobně rozebrán přístup k problému, nachází se zde popis metody, které se využívají k automatizované tvorbě překladačů. Mezi tyto metody patří nalezení konečných automatů, generování rozkladových tabulek syntaktické analýzy a zpracování sémantických podprogramů.

6. kapitola s názvem Navržený jazyk sémantických podprogramů se zabývá (jak vypovídá název) popisem jazyka určeného k popisu sémantických úkonů v navrhovaných jazycích. V této kapitole jsou popsány důvody k zavedení takového jazyka, popis jeho datových typů, jeho složitější příkazy a řízení toku programu tohoto jazyka.

Poslední kapitola v pořadí 7. nazvaná Navržený jazyk pro zpracování obrazu, se věnuje popisu navrhovaného skriptovacího jazyka. V této kapitole se nachází ucelený pohled na účel jazyka, popis datových typů jazyka a popis navrhovaného způsobu zápisu toku programu.

Diplomová práce navazuje na práci vykonanou v rámci semestrálního projektu. V tomto projektu byly prostudovány techniky návrhu překladačů a interpretů, na kterých je založena kapitola 3 Překladače, kompilátory a interprety. V semestrálním projektu byl dále navržen překladač a skriptovací jazyk pro práci s obrazem. Návrh překladače a skriptovacího jazyka byl v rámci diplomové práce přepracován zcela od začátku.

Kapitola 2

Skriptovací jazyky

Tato kapitola se zabývá popisem skriptovacích jazyků, důvodem jejich vzniku a oblastmi ve kterých se skriptovací jazyky využívají.

2.1 Popis skriptovacích jazyků

Původní skriptovací jazyky byly vyvinuty za účelem rychlého vývoje jednoúčelových programů, které řešily složitější administrativní úlohy v operačním systému. U těchto byla důležitější rychlost vývoje programu[1] než efektivita spouštění[2]. Tyto jazyky sloužily jako jednoduchý nástroj spouštějící množství jiných často specializovaných jednoúčelových programů[3] za účelem vykonání nějaké komplexní operace. Příkladem programů v takovýchto jazycích mohly být inicializační skripty některých operačních systémů, zálohovací skripty a podobně.

Anglický název skript byl odvozen od psaného předpisů pro divadelní herce, kteří jej měli "interpretovat" tj. předvádět scénu popsanou ve skriptech divákům. Ve světě informačních technologií zastupuje roli herce program který hraje svou roli popsanou ve skriptu, všechny programy využité ve skriptu tedy nakonec vykonají požadovanou operaci.

Skriptovací jazyky jsou podmnožinou interpretovaných jazyků. Dnes se však i interpretované jazyky řadí do skriptovacích jazyků, kde roli jednoúčelových programů zastoupily jednotlivé funkční jednotky spouštěné interpretem podle skriptu.

Hlavní rozdíl mezi interpretovanými jazyky a kompilovanými jazyky je způsob jakým se program spouští na výpočetní jednotce. Zatímco program zapsaný v kompilovaném jazyku se musí programem nazvaným překladač (angl. compiler) přeložit do strojového kódu cílové platformy, tak program zapsaný v interpretovaném jazyku zůstává v původní textové podobě, dokud není požadováno jeho spuštění. Spouštění skriptu se ujme program nazývaný *interpretační překladač* (angl. interpret). Interpret může ponechat zdrojový program v původní podobě a interpretovat příkazy jeden po druhém, nebo může celý zdrojový program přeložit do struktury, která už sice nebude mít čitelný textový zápis ale její spuštění bude efektivnější s možnými optimalizacemi při překladu.

Interpretované jazyky se mohou považovat za další úroveň abstrakce nad kompilovanými jazyky. Vyšší úroveň abstrakce interpretovaných jazyků je výhodná pro programátora, ten se většinou nemusí starat o správu paměti, typ cílové architektury, nebo dokonce typ

[1] Často několik minut.
[2] Často několik sekund.
[3] Takovýmto programem může být i jiný skript.

operačního systému na kterém bude jeho program spouštěn. Tyto a další problémy již za něj řeší interpret jazyka. Odstínění programátora od hardwarové realizace jeho výpočtů je výhodné i z pohledu bezpečnosti vykonávání těchto programů. Je nemožné za předpokladu že interpret je sám o sobě bezpečný program napsat skript který by ohrozil hardware, nebo umožnil nepovolené operace na cílovém stroji. Z tohoto důvodu jsou interpretované jazyky vhodné pro nasazení například na webových serverech, kde slouží k zpracování dynamických webových aplikací.

Vyšší abstrakce interpretovaných jazyků nad jazyky kompilovanými má však i některé nevýhodné důsledky. Programátor využívající interpretovaný jazyk nebude schopný sám optimalizovat svůj kód nad rámec jazyka, v některých jazycích nemůže rozhodnout v jakém místě chce přidělit případně uvolnit paměť. Absence ukazatele do paměti tj. nejmocnější programátorské struktury z důvodů bezpečnosti je vítaná, ale na druhou stranu může někomu chybět. Je pravidlem že interpretovaný kód je pomalejší než kód kompilovaný. Tyto problémy se sice řeší např. kompilováním kritických sekcí[4] v programu do strojového kódu aktuální architektury, ale nikdy nebude možné dosáhnout stejné efektivity jako u jazyků kompilovaných.

2.2 Známé skriptovací jazyky

Tato část se zabývá stručným popisem některých druhů skriptovacích jazyků.

Job control jazyky a shelly

První skriptovací jazyky nevznikly za účelem nahrazení kompilovaných jazyků vyšší úrovně, ale spíše jako nástroje pro jednodušší údržbu složitých systémů. Zjednodušovaly a automatizovaly opakující se úkony, například ve správě operačních systémů. Část jazyků anglicky nazvaná shelly vznikla na platformě operačních systému rodiny Unix, kde je celá správa systému v podstatě řešená zpracováním textových konfiguračních souborů. Shelly také slouží pro komunikaci uživatele s počítačem, každý příkaz zapsaný uživatelem z klávesnice je interpretován jako část kódu skriptu. Do této skupiny patří například: `bash`, `csh`, `ksh`, `tcsh`, `sh`, `zsh`, `Winbatch`.

Skriptování GUI

Po rozšíření různých grafických uživatelských rozhraní vznikl nový druh specializovaných skriptovacích jazyků určený pro kontrolu počítače přes prvky grafického uživatelského rozhraní. Skripty operují se stejnými prvky rozhraní tj. tlačítky, výběry, zatrhávacími tlačítky atd. jako uživatel. Používají se k automatizaci opakujících se akcí, nebo například k nastavení výchozích stavů aplikací. Využití takových jazyků je podmíněno podporou v aplikaci obsahující ovládané grafické uživatelské rozhraní a v operačním systému. Takovýmto jazykům se také říká makrovací jazyky.

Aplikačně-specifické

Jako součást některých velkých aplikaci figurují aplikačně-specifické skriptovací jazyky. Tyto jsou zaměřeny na řízení programovatelných součástí jednotlivých aplikací. Jazyky tohoto druhu jsou navrhovány pro použití pouze v jedné aplikaci a obvykle nebývají vhodné

[4] Části kódu které se provádějí při spuštění programu nejčastěji.

pro obecnější použití. Mezi ně patří například i herní skripty pro řízeni herních objektů, které se nemusí zabývat detailnější úlohou objektu v enginu. Do této kategorie jazyku patří například: `Action Code Script`, `AutoLisp`, `Emacs Lisp`, `QuakeC`, `UnrealScript`, `VBScript`.

Dynamické webové aplikace

Důležitým typem aplikačně-specifických skriptovacích jazyků jsou jazyky pro implementaci dynamických aplikací v prostředí internetu. Tyto jazyky jsou specializované na tvorbu dynamických webových stránek a jiných internetových aplikací a jsou nasazeny na serverech poskytujících tyto služby. Tyto jazyky jsou zvoleny z důvodu bezpečnosti a bývají rozšířeny o prvky pro snadný přístup k databázím apod. Nejmodernější webové programovací jazyky jsou již tak sofistikované že jimi lze v určitých oblastech nahradit i standardní kompilované programovací jazyky. Do této skupiny jazyků patří například: `ColdFusion`, `JavaScript`, `Lasso`, `Miva`, `PHP`, `SMX`, `XSLT`.

Zpracování textu

Velké množství skriptovacích jazyků bylo primárně určeno na zpracování obyčejného textu. Na-rozdíl od shellů většina z nich nebyla určena pro zpracování příkazů zadávaných přímo z klávesnice, ale spíše ke zpracování celistvých programů. Motivace jejich vzniku je však podobná jako u shellů tj. snadnější správa operačních systému založených na textových konfiguračních systémech. Některé z těchto jazyků tvoří dodnes nejefektivnější systémy pro zpracování textových informací. Velké množství rozšíření těchto jazyků z některých vytvořilo plně použitelné aplikační jazyky, jako například u `Perlu`. Do této skupiny jazyků patří například: `AWK`, `Perl`, `sed`, `XSLT`.

Programovací skriptovací jazyky

Některé skriptovací jazyky se navzdory svému původnímu účelu, vyvinuly v plně použitelné programovací jazyky použitelné pro tvorbu větších aplikací. Tyto těží ze své výhody rychlejšího vývoje aplikace proti kompilovaným jazykům, automatické správy paměti atd. Pokud programátor nepotřebuje psát přímo ovladače na hardware, nebo nějaký evoluční algoritmus tak výhoda rychlosti vývoje aplikace předčí nevýhodu menší rychlosti programu. I přes to že se většinou používají k tvorbě aplikací jako dynamické aplikační jazyky, často se označují jako vyspělejší skriptovací jazyky. Tyto jazyky jejich uživatelé většinou neoznačují jako skriptovací jazyky.

Extension/embeddable jazyky

Jazyky z této skupiny jsou vyvíjeny za účelem nahrazení aplikačně specifických jazyků, tak že se vkládají přímo do aplikačních programů. Programátoři vytvářející aplikaci v některém ze systémových jazyků, do této aplikace vloží na určité místa kód umožňující její ovládání pomocí skriptovacího jazyka. Jedním ze známých případů takového použití skriptovacího jazyka je JavaScript vyvinutý za účelem spouštění v internetovém prohlížeči na straně klienta. Dále do této skupiny patří jazyky: `Ch`, `ECMAScript`, `Python`, `Tcl`, `RBScript`, `Windows PowerShell`.

Kapitola 3

Překladače, kompilátory a interprety

V angličtině se pro pojem překládání v oblasti programovacích jazyků používá slovo "compilation", které má spíše význam skládání, nebo shromažďování. Samotný pojem kompilátor zavedl počátkem padesátých let Grace Murray Hopper. Překlad se v té době totiž prováděl jako skládání sekvencí podprogramů ve strojovém kódu, které byly uloženy v knihovně.

Jedním z prvních skutečných překladačů by dobře známý FORTRAN. Hlavní prioritou těchto prvních překladačů byla vzhledem k malé výpočetní a paměťové kapacitě soudobých počítačů optimalizace a efektivita překládaných algoritmů.

S příchodem jazyka Algol 60 nastal zlom v podstatě překladu. Ten zavedl problémově orientovaný přístup, to znamená že překladače již nesloužily jen pro umožnění přehlednějšího zápisu strojového kódu, ale důležitým faktorem se stal řešený problém, kterému se přizpůsobila syntaxe i filosofie jazyka.

Novější jazyky mezi které patří například Pascal, Modulo a Ada jsou vytvořeny nezávisle na cílové architektuře. Dnes se oddělení od cílové architektury umocňuje vytvářením tzv. virtuálních strojů, které jsou určeny ke spouštění cílových aplikací. Poslední popsaný přístup je charakteristický například pro platformy Java nebo .NET.

3.1 Úvod do překladačů

Překladač je v podstatě program realizující textový procesor umožňující používat programovací jazyk pro výpočty na počítači. Překladač načítá program zapsaný ve zdrojovém kódu, ten zpracuje a zapíše sémanticky ekvivalentní kód ve výstupním cílovém jazyce. Když je zdrojovým jazykem vyšší programovací jazyk a cílovým jazykem jazyk symbolických instrukcí nazýváme překladač kompilátorem a proces překladu kompilací. Překladač, který zdrojový kód nepřekládá do cílového jazyka, ale místo toho jej spouští a zobrazuje výsledky výpočtu se nazývá interpret, nebo také interpretační překladač.

Kompilátory a interprety tvoří důležitou součást dnešních počítačů a jsou nedílnou součástí prostředí většiny operačních systémů. Techniky jejich návrhu a zpracování představují stále aktivní oblast výzkumu. Podstatnou roli ve vývoji techniky realizace překladačů sehrála teorie formálních jazyků a gramatik. Výsledky této teorie lze využít k efektivnímu návrhu syntaktické analýzy. Syntaktická analýza sice tvoří jen část problematiky řešené při tvorbě překladače, ale fakt že syntaktická analýza může převzít řízení celého překladače velmi usnadní návrh zbylých komponent překladače.

Proces kompilace je vhodné rozdělit do více podprocesů nazvaných fáze, kde se každá fáze stará o převod jedné reprezentace zdrojového kódu na jinou. Těmito fázemi jsou:

- *lexikální analýza* - slouží k nalezení lexikálních jednotek jazyka ve zdrojovém programu. Jedna lexikální jednotka odpovídá například identifikátoru, konstantě, operátoru, klíčovému slovu apod. Lexikální analýza v podstatě převádí kód z reprezentace ve zdrojovém souboru na posloupnost lexikálních elementů.
- *syntaktická analýza* - přijímá na vstupu lexémy[1] a provádí rozklad programu podle gramatiky jazyka, tzn. buduje derivační strom zdrojového programu.
- *intermediární generování kódu* - využívá výstupu syntaktické analýzy a produkuje zdrojovému kódu ekvivalentní posloupnost akcí. Hlavní rozdíl mezi intermediárním kódem a strojovým kódem produkovaným interpretem je ten že intermediární kód neobsahuje specifikaci úložišť proměnných v paměti ani použité registry.
- *optimalizace kódu* - volitelná fáze, která provádí optimalizace z pohledu časové a paměťové náročnosti nad intermediárním kódem. Často se implementuje jako součást fáze generování intermediárního kódu.
- *generování kódu* - generuje výsledný strojový kód, nebo v případě interpretu spouští intermediární kód na virtuálním stroji. Generování kódu v případě kompilátoru spočívá v přiřazení paměťových míst datům, určení způsobu přístupu k nim a výběru registrů pro jednotlivé operace.

Z popisu fází překladače je zřejmé že řídící fází kompilátoru je syntaktická analýza, je hlavní fází při analýze zdrojového kódu. Při syntéze cílového kódu má hlavní postavení fáze generování intermediárního kódu.

Další důležitou částí překladače je *tabulka symbolů*, do této struktury kompilátor zaznamenává informace získané o objektech programu v průběhu analýzy. Tyto informace se získávají na základě kontextu v programu, nebo na základě deklarací.

Pokud v průběhu kompilace dojde k nalezení chyby ve zdrojovém kódu, mělo by dojít minimálně k vypsání zprávy upozorňující programátora na chybu. Obecně se však požaduje aby kompilátor odhalil při kompilaci pokud možno co největší množství chyb, proto by měla část kompilátoru nazvaná *zpracování chyb* provést takové zásahy aby se pokud možno mohlo pokračovat v kompilaci.

3.2 Lexikální analýza

Lexikální analýzu v překladači provádí část nazvaná *lexikální analyzátor*. Lexikální analyzátor je v podstatě jednoduchý textový procesor, který slouží k odstranění redundantních informací ze zdrojového souboru, tím že nalezne jednotlivé lexikální elementy překládaného jazyka. Tyto elementy reprezentují terminální symboly syntaktické analýzy. Z tohoto důvodu je výhodné nechat řízení (ve smyslu požadavku na nový symbol) lexikálního analyzátoru syntaktickému analyzátoru.

[1] Lexikální jednotky produkované syntaktickou analýzou.

Tvar výstupu lexikálního analyzátoru

Výstup lexikálního analyzátoru se zapisuje ve tvaru, jenž je vhodný pro zpracování dalšími částmi kompilátoru. Výstup se vyznačuje následujícími vlastnostmi.

- neobsahuje z hlediska významu programu redundantní informace[2].
- nejmenším nositelem informace již není jeden znak[3] ale obraz prvku patřící do seznamu syntaktických jednotek jazyka, nazývaný lexikální jednotka, nebo také lexém.
- informace o lexikální jednotce obsahuje složku syntaktickou identifikující typ lexikální jednotky a na složku sémantickou specifikující její aktuální hodnotu.(např. u lexému s typem `identifikátor` bude tato hodnota obsahovat název identifikátoru.).

Lexikální jednotky

V kontextu lexikální analýzy se pro lexikální jednotku používá synonymní název symbol, nebo také lexém. Název symbol je odvozen z funkce lexikálního analyzátoru, který na požádání syntaktického analyzátoru mu předává symboly vstupní věty. Rozlišují se dva základní typy symbolů.

- jazykově specifické řetězce a znaky(do této skupiny patří například klíčové slova programu, operátory apod.).
- řetězce znaků popisující například identifikátory, číselné nebo řetězcové konstanty apod.

Zatímco první skupina symbolů je plně popsána svým typem, tak u druhé je nutné kromě typu symbolu zaznamenat i hodnotu symbolu přečtenou ze vstupního souboru. Tuto hodnotu může vždy popsat řetězec který tvoří konkrétní symbol ve zdrojovém souboru, nebo již přímo hodnota získaná z tohoto řetězce na základě znalosti typu symbolu.

Implementace lexikálního analyzátoru

Výstavba lexikálních analyzátorů je založena na principu vycházejícím z předpokladu, že lexikální jednotky implementovaného programovacího jazyka tvoří formální jazyk typu 3 Chomského hierarchie. Pak je lze teoreticky popsat regulární gramatikou, nebo regulárním výrazem a v podstatě mechanicky získat konečný automat, určený k rozpoznání symbolů jazyka.

Implementace konečného automatu pro rozpoznávání lexikálních jednotek tvoří jádro celého analyzátoru. Kromě rozpoznávání symbolů musí lexikální analyzátor předávat informace o vstupním souboru ostatním jednotkám překladače(do těchto informací patří například řádek ve zdrojovém souboru na kterém se překlad právě nachází, nebo text části kódu ve kterém se nacházela chyba apod.).

[2] např.: tzv. bílé místa v kódu, komentáře apod.
[3] Jak tomu bylo ve zdrojovém kódu.

3.3 Syntaktická analýza

Syntaktický analyzátor (angl. parser) provádějící syntaktickou analýzu má v překladači výsadní postavení. Jeho úlohou je analyzovat posloupnost symbolů, které produkuje na svém výstupu lexikální analyzátor, za účelem vytvoření derivačního stromu celého programu.

Z hlediska teorie formálních jazyku je vstupním řetězcem terminálních symbolů posloupnost prvních složek lexikálních jednotek jak je rozpoznal lexikální analyzátor. Avšak na rozdíl od teorie formálních jazyků a překladačů není nutné aby výstupem syntaktické analýzy byl přímo pravý nebo levý rozklad jazyka. Podstatná je dekompozice posloupnosti symbolů zdrojového programu na fráze odpovídající neterminálům, z kterých již překladač "pozná" význam překládané věty. Hlavní požadavek na takovouto dekompozici řetězce terminálů je deterministický průběh, pokud takovýto analyzátor realizuje deterministickou syntaktickou analýzu tak není nutné vytvářet explicitní reprezentaci derivačního stromu analyzované věty.

Syntaktická analýza v sobě pochopitelně zahrnuje automatickou kontrolu syntaktické správnosti překládaného programu. Po odhalení chyby by proto mělo následovat zpracování této chyby, takovým způsobem, aby syntaktická analýza mohla pokračovat.

Implementace syntaktického analyzátoru

Tvorba syntaktického analyzátoru je jedna z nejlépe teoreticky i prakticky zvládnutých oblastí problematiky překladačů. Teorie bezkontextových gramatik a jazyků dává k dispozici algoritmy pro konstrukci deterministických analyzátorů a dokonce i prakticky použitelné algoritmy pro vytvoření konstruktorů syntaktických analyzátorů. Velký význam v této problematice mají jazyky `LL(1)` a `LALR(1)`.

3.4 Intermediární kód

Intermediární kód je kód složitostí náležící mezi zdrojový kód a cílový kód. Jeho opodstatnění u překladačů tkví v požadavcích na překladač a efektivnost jeho výstupního kódu, u kterého se požaduje efektivní vykonávání programu a také efektivní hospodaření s vnitřní pamětí. Z těchto důvodů mnohé překladače negenerují výstupní kód přímo, ale místo toho vytvářejí meziprodukt překladu, kterým je právě intermediární kód.

Kromě toho že tento kód neobsahuje specifikaci paměťových míst a použitých registrů, je charakterizován pořadím operací, které je již shodné s pořadím operací v budoucím cílovém kódu. Tuto vlastnost(pořadí akcí) nemá většina vyšších programovacích jazyků. K zápisu intermediárního kódu se využívá množství struktur nejčastěji bývá využíván tzv. *3-adresový kód*.

Implementace fáze generování intermediárního kódu

Při řešení problému návrhu a implementace fáze generování intermediárního kódu je důležitý koncept *sémantických podprogramů*, který představuje strukturalizaci fáze generování intermediárního kódu na určité logické akce spojené s přepisovacími pravidly gramatiky.

V případě že syntaktický analyzátor rozpozná určitou syntaktickou konstrukci, která se pojí s některým neterminálem gramatiky, tak sémantický podprogram příslušející k tomuto neterminálu vykoná potřebné sémantické akce. Tyto akce mohou zahrnovat.

	op	ARG1	ARG2	result
(1)	neg	c		T1
(2)	+	T1	d	T2
(3)	*	b	T2	T3
(4)	=	T3		a

Tabulka 3.1: Tabulka čtveřic.

- sémantickou analýzu, v jejímž rámci překladač odhalí takový druh chyb, které nejsou popsány bezkontextovou gramatikou, jako je například nekompatibilita typů operandů, duplicita návěští apod.
- generování odpovídajících příkazů intermediárního kódu a využití výsledků sémantické analýzy, např. při převodů typů operandů.

Syntaxí řízené překladové schéma

Syntaxí řízené překladové schéma je schéma v němž je popis sémantických akcí generátoru intermediárního kódu v bezprostřední návaznosti na syntaktické struktuře jazyka. Neexistuje však všeobecně rozšířený a prakticky používaný formalismus pro zápis tohoto schématu. Protože většinou existuje velké množství různých druhů sémantických akcí, jež značně závisejí na charakteru implementovaných jazykových prostředků, je dosud prakticky nejpoužívanějším prostředkem pro zápis sémantických akcí programovací jazyk.

Specifikace fází realizujících překlad do intermediárního kódu pak může sestávat z výčtu přepisovacích pravidel gramatiky, ke kterým lze případně automaticky vytvořit příslušný syntaktický analyzátor a ze souboru pravidlům odpovídajících podprogramů v určitém vyšším programovacím jazyce, které popisují sémantické akce. Z tohoto pojetí syntaxí řízeného překladu vzniká pojem sémantického podprogramu.

Typy intermediárního kódu

Návrh a fáze implementace intermediárního kódu je závislá nejen na implementovaném jazyce, ale také na zvoleném typu intermediárního kódu. Seznam typů intermediárního kódu následuje.

- *3-adresový kód* - tento název je ve skutečnosti abstraktním označením dvou typů intermediárních kódu, a to tzv. čtveřic a trojic. Čtveřice se skládají ze záznamů o čtyřech položkách: `op`, `ARG1`, `ARG2`, `result`, kde první označuje typ operace, druhá a třetí operandy a poslední označuje jméno výsledku. Příklad se nachází v tabulce 3.1.

  Reprezentace trojicemi je podobná reprezentaci čtveřicemi s tím rozdílem že místo používání dočasných proměnných k propojení jednotlivých operací jsou využity odkazy přímo do tabulky trojic, na pozici kde se nachází výpočet jehož výsledek se požaduje na místě operandu. Příklad se nachází v tabulce 3.2.

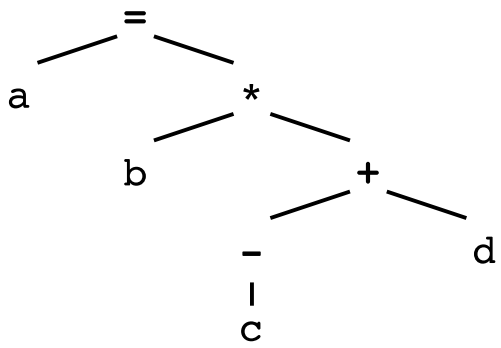
  V tabulkách je uveden příklad zápisu následujícího kódu: `a = b * (-c + d)`.

- *syntaktický strom* - tento způsob reprezentace má jistou příbuznost s trojicemi. Syntaktický strom je abstrakcí derivačního stromu. Koncovými uzly takového stromu jsou operandy a jeho vnitřními uzly jsou operátory operující nad operandy v jeho listech.

	op	ARG1	ARG2
(1)	neg	c	
(2)	+	(1)	d
(3)	*	b	(2)
(4)	=	a	(3)

Tabulka 3.2: Tabulka trojic.

Tento způsob zápisu je možné vyhodnotit průchodem typu postorder celým stromem. Příklad takového stromu je na obrázku 3.1.



Obrázek 3.1: Syntaktický strom.

- *postfixová notace* - podstatou postfixové, nebo také polské notace je umísťování operátorů za příslušné operandy. Tato notace, která byla původně vytvořena pro reprezentaci výrazů má známé výhody. Jsou jimi jednoznačný zápis operací i bez užití závorek a jednoduchý způsob interpretace s využitím zásobníku. Příklad zápisu výrazu v postfixové notaci: `a c neg d + b * =`.

Optimalizace kódu

Cílové programy určené k častému provádění by měly být rychlé a měly by šetřit místem v paměti. Proto některé překladače ještě před fází generování cílového kódu provádějí tzv. optimalizaci intermediárního kódu. Tato optimalizace aplikuje určité transformace na výstupu generování intermediárního kódu, za účelem získání jeho verze, která povede k menšímu a rychlejšímu kódu.

Termín optimalizace v tomto případě není adekvátní z toho důvodu že neexistuje algoritmická cesta, která by vedla k optimálnímu kódu pro daný zdrojový program. Proto produkt optimalizujícího překladače bude pokud možno menší a rychlejší kód, málokdy však kód optimální.

V dnešní době může být fáze optimalizování značně složitá, kromě klasických problémů se musejí potýkat s problémem různých instrukčních sad(MMX, 3DNow!, SSE, SSE2, hyperthreading apod.) a tak se fáze optimalizace v různých překladačích liší různou mírou propracovanosti.

3.5 Generování cílového kódu, interpretace

Tato fáze překladu programu se liší v závislosti na tom zda implementujeme kompilátor, nebo interpret. Zatím co kompilátor v této fázi generuje strojový kód z intermediárního kódu, tak interpret zde spouští jednotlivé příkazy intermediárního kódu na svém virtuálním stroji.

Generování cílového kódu

V této fázi transformuje překladač intermediární kód na strojový kód cílové architektury. Nejjednodušší způsob, kterým lze vygenerovat cílový strojový kód je jednoduchý přepis příkazů intermediárního kódu na strojové instrukce technikou rozvoje makroinstrukcí. Tento přístup ale většinou vede k neefektivnímu kódu, ve kterém vzniká příliš velké množství instrukcí přístupu do paměti. Aby se předešlo generování takového kódu musí generátor průběžně uchovávat informace o obsahu jednotlivých registrů a přistupovat do paměti jen pokud je to nezbytně nutné.

Pro návrh kvalitního generátoru strojového kódu je nezbytně nutná dobrá znalost cílové architektury. Každá architektura obsahuje několik registrů určených pro různé typy operací. Kvalitní generátor strojového kódu by měl umět využít tyto registry co nejefektivněji. Tento problém je obtížné řešit optimálně, obvykle se na jeho řešení aplikují různé heuristické přístupy. Mezi další úkoly generátoru patří pokud možno co nejlepší využití celého instrukčního repertoáru cílové architektury. Typickými příklady takových situací jsou například přičítání jedničky, nebo násobení celého čísla mocninou dvou.

Často se místo přímého generování strojového kódu volí metoda generování kódu v jazyce symbolických instrukcí, který se poté externím programem přeloží do strojového kódu. Tento přístup usnadní fázi generování kódu, protože se generátor nemusí zabývat detaily mezi které patří například výpočty relativních adres a adres skoků.

Spuštění kódu interpretem

Intermediární kód určený k interpretaci bývá často zapsán v postfixové notaci. S tím rozdílem že se základní myšlenka interpretace výrazu v postfixové notaci zobecní na celý soubor prostředků konkrétního jazyka. K interpretaci takového kódu postačí následující struktury.

- *program* - zdrojový program zapsaný v postfixové notaci, jehož prvky(instrukce) identifikují operace a operandy.
- *zásobník* - využívaný při interpretaci a poskytující pracovní paměť.
- *tabulka symbolů* - sloužící k uložení záznamů o objektech, atributem objektu může být i jeho hodnota.

Samotná interpretace probíhá následujícím způsobem. Kód je zpracováván postupně zleva doprava. Pokud interpret narazí na operand tak jej zkopíruje na zásobník. V případě že narazí na n-arní operátor tak načte prvních n hodnot ze zásobníku provede nad nimi operaci a výsledek uloží opět na zásobník. Na základě tohoto algoritmu lze jednoduše implementovat procesor realizující interpretační fázi.

Kapitola 4

Motivace, současný stav práce

V této části je popsána motivace k výběru a řešení tématu práce, dále obsahuje popis jednotlivých částí, které byly vypracovány v rámci projektu. U každé z popisovaných částí se nachází popis stavu ve kterém se nachází v čase vypracování tohoto dokumentu.

4.1 Motivace

Téma práce jsem zvolil na základě vlastních pokusů o implementaci programu pro snadnou manipulaci s obrazem. Tyto znalosti (tj. přístup k problematice zpracování obrazu) budou využity v následující práci na projektu, která bude zahrnovat detailní návrh jazyka určeného pro zpracování obrazu.

Dalším důvodem k volbě tohoto tématu bylo téma bakalářské práce, která se zabývala návrhem kombinované syntaktické analýzy aplikované při rozkladu vstupního řetězce. Při zpracování výše uvedeného zadání jsem získal základní zkušenosti z oblasti překladačů a interpretů, které jsem chtěl dále zužitkovat v této práci.

4.2 Současný stav práce

Následuje výčet jednotlivých navržených a implementovaných částí projektu a popis v jaké fázi rozpracování se nachází.

Skripty v jazyce `Perl`

Skripty používané pro automatické generování kontejnerových objektů. Implementační detaily popisující návrh a využití těchto skriptů jsou popsány v příloze. Návrh a implementace těchto skriptů jsou dokončeny a používají se při implementaci níže popsaných částí projektu.

Program generující překladač

Tento program tvoří velkou část celkové práce na projektu. Je určen k automatickému generování popisu překladače na základě popisu struktury jazyka překládaných programů. Návrh a implementace tohoto programu jsou dokončeny. Popis programu se nachází v kapitole 5. Navržený překladač. Navrhovaný skriptovací jazyk je realizován jako vstupní soubor tohoto programu.

Součástí programu je i jazyk určený k zápisu sémantických podprogramů. Návrh a implementace tohoto jazyka jsou již částečně dokončeny, ale stále dochází k drobným změnám,

v závislosti na požadované funkcionalitě při návrhu skriptovacího jazyka. Popis tohoto jazyka se nachází v kapitole 6. Navržený jazyk sémantických podprogramů.

Jazyk pro zpracování obrazu

Jazyk je zapisován jako vstup výše popsaného programu. Návrh programu a jeho zápis ještě není kompletně dokončen a závisí na detailní struktuře reprezentovaných obrazů. Popis dosud navržených částí skriptovacího jazyka je popsán v kapitole 7. Navržený jazyk pro zpracování obrazu.

4.3 Zhodnocení současného stavu

Byl navržen a implementován program generující popis překladače. Jeho implementace splňuje požadavky na něj kladené a je provedena s ohledem na efektivní výpočet a interpretaci vygenerovaným překladačem. Existuje možnost zápisu zdrojového kódu vygenerovaného překladače v jazyce `C`, která však zatím implementuje jen lexikální a syntaktickou analýzu tohoto překladače.

S využitím výše uvedeného programu je navrhován skriptovací jazyk pro zpracování obrazu. Je navrhován s ohledem na snadné použití a je snaha v rámci možností dodržovat základní syntaxi podobnou syntaxi jazyka `C`.

Kapitola 5

Navržený překladač

Tato kapitola se zabývá návrhem a implementací interpretačního překladače. Překladač je navržen pro zpracování libovolného jazyka, popsaného vstupním souborem obsahujícím popis lexikálních, syntaktických a sémantických pravidel tohoto jazyka. Úkolem překladače je na základě vstupního zdrojového programu vygenerovat jeho sémanticky ekvivalentní reprezentaci, kterou bude možné jednoduše dále upravovat, tj. spouštět na virtuálním stroji, nebo provádět nad ní další operace (Optimalizace apod.).

Princip funkce překladače

Z důvodů počáteční nezkušenosti s návrhem a implementací překladačů programovacích jazyků by bylo nevhodné implementovat překladač přímo v nízkoúrovňovém programovacím jazyce. Pokud by vyvstal v určité fázi implementace nějaký složitější problém, bylo by nutné provádět větší zásahy do kódu a mohlo by docházet k případnému zanášení chyb, apod.

Po prostudování problémů návrhu a implementace překladačů, dospěl autor k názoru že techniky automatizovaného generování důležitých pod-částí překladače jsou tak pokročilé a nepříliš složité na implementaci, že bude snadnější vytvořit systém vytvářející překladač na základě jeho popisu, a samotný překladač navrhnout jako vstupní popis do tohoto systému.

Na obrázku 5.1. je zobrazena struktura implementovaného programu, včetně zobrazení postupů při vytváření překladače a překladu zdrojových souborů. V následujících podkapitolách se nachází podrobnější popis jednotlivých pod-částí programu.

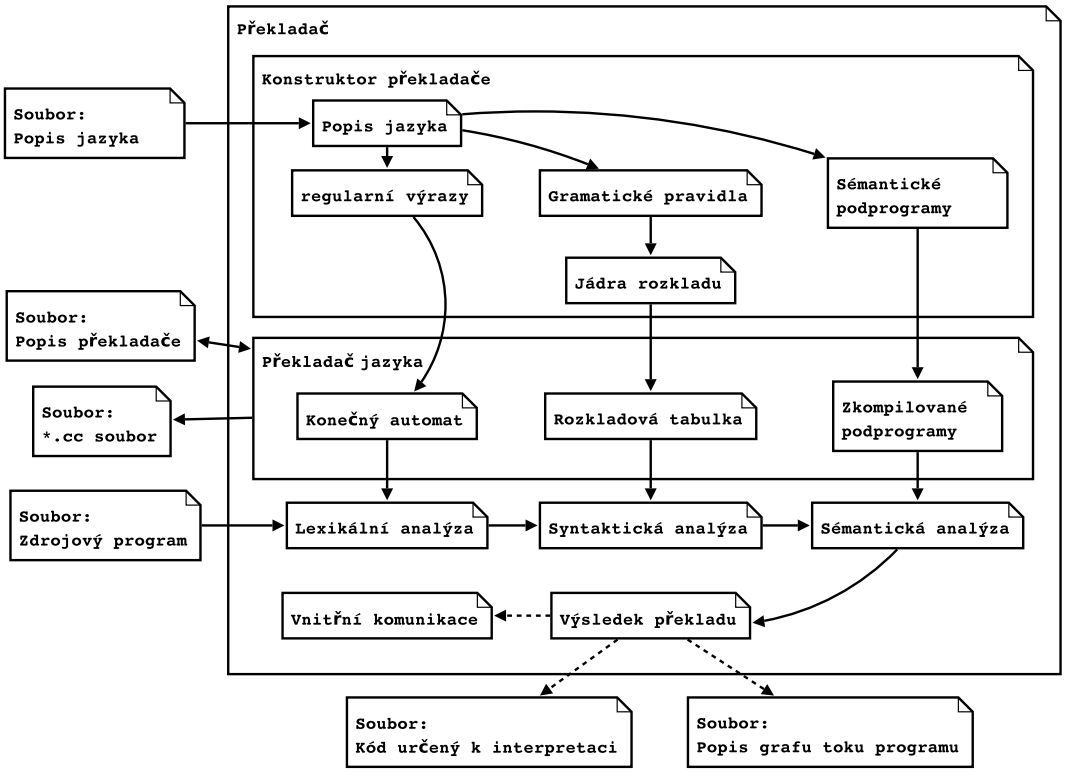
5.1 Popis souboru definujícího jazyk

Podle tohoto vstupního souboru implementovaný program vygeneruje strukturu popisující překladač. Tuto strukturu poté program využije k následnému přeložení vstupního zdrojového souboru. Popis překladače je potřeba vygenerovat pouze jednou, tento se může uložit na disk a při překladu dalšího zdrojového souboru pouze načíst ze souboru. V následující práci na projektu je zahrnuta i myšlenka generování zdrojového programu v jazyce `C` implementující překladač popsaný generovanou strukturou[1].

Formát souboru definujícího jazyk

Do tohoto souboru se zapisuje seznam terminálních a neterminálních symbolů syntaktické analýzy, popis tvaru terminálních symbolů a sémantické podprogramy jednotlivých pravidel.

[1] Nejedná se o žádnou převratnou myšlenku takových programů existuje celá řada.



Obrázek 5.1: Struktura překladače.

Základní struktura souboru je zobrazena v tabulce 5.1.

Následuje vysvětlení některých pojmů zapsaných v této struktuře.

- `<terminal_id>` - tento pojem popisuje identifikaci terminálního symbolu, která bude dále používaná v seznamu pravidel. Zapisuje se ve tvaru identifikátoru, jako např. v jazyce `C`. To znamená posloupnost čísel, písmen a podtržítek s podmínkou že nezačíná číslem.

- `<nonterminal_id>` - tento element popisuje identifikaci non-terminálního symbolu, pro použití v seznamu syntaktických pravidel. Zapisuje se jako identifikátor terminálního symbolu uzavřeny mezi znaky '`<`' a '`>`'.

- `<regular_expression>` - zastupuje regulární výraz určený k popisu tvaru terminálního symbolu ve zdrojovém programu jazyka. Není to regulární výraz jak jej známe z teorie formálních jazyků, ale spíše zjednodušená forma, ve které se mohou používat zástupné symboly pro množinu znaků, iterátor a zápis popisující výběr jednoho z množiny znaků. Seznam speciálních znaků následuje.

  - `\w` - "white" označuje tzv. bílé znaky mezi které patří mezera, tabulátor a konec řádku. Využije se například při označování terminálů, které syntaktický analyzátor nemusí zpracovávat. To se provede například zápisem `\w\w\*`.
  - `\d` - "digit" označuje jednu číslici z množiny `0-9`. Slouží k definici terminálů popisujících číselné konstanty, nebo i identifikátory proměnných.

```
init_code:
   {
      <semantic_code>
   }
terminals:
   <terminal_id>  {<regular_expression>}
   ...
nonterminals:
   <nonterminal_id>
   ...
rules:
   <nonterminal_id> -> <terminal_id> | <nonterminal_id> ... ->>
   {
      <semantic_code>
   }
   ...
```

Tabulka 5.1: Struktura souboru popisujícího jazyk.

  - `\l` - "letter" značí jedno písmeno abecedy z množin `a-z` a `A-Z`. Slouží hlavně k popisu terminálních symbolů označujících identifikátory.
  - `\!` - označuje skupinu znaků obsahující všechny znaky kromě znaků definovaného za tímto identifikátorem.
  - `\*` - popisuje 0-n iterací předcházejícího znaku. Například zápis `"\!"\*"` popisuje řetězec znaků.
  - `\[` a `\]` - označují závorky pro výběr jednoho znaku, který je v nich obsažen. Příkladem využití takového zápisu je popis terminálu označujícího identifikátor `\[_\l\]\[_\l\d\]\*`.

  Podrobnější popis využití těchto výrazů k rozpoznávání terminálů(lexémů) zdrojového programu bude uveden v podkapitole zabývající se lexikální analýzou překladače.

- `<semantic_code>` - tento symbol zastupuje program zapsaný v interpretovaném jazyce, který slouží pro definování sémantických operací prováděných při redukci podle pravidla, u kterého je tento kód definován. Jedinou výjimku tvoří kód zapsaný za klíčovým slovem `init:`, který se spouští vždy na začátku překladu. Popis jazyka pro zápis sémantických podprogramů se nachází v kapitole 6. nazvané Navržený jazyk sémantických podprogramů.

Zpracováním tohoto popisu jazyka získá program dostatek informací pro vygenerování překladače schopného zpracovat zdrojový soubor zapsaný v tomto jazyce.

5.2 Nalezení konečného automatu

Tato část se zabývá stručným popisem postupů využitých při hledání konečného automatu rozlišujícího od sebe jednotlivé terminální symboly, popsané ve vstupním definičním souboru regulárními výrazy.

```
0   1   2   3   4   5   6   7   8   9   10  11
<   \[  _   \l  \]  \[  _   \l  \d  \]  \*  >

          0:  '<'         ->   1
          1:  '_'         ->   5
              <letter>    ->   5
          5:  '_'         ->   5
              <letter>    ->   5
              <digit>     ->   5
              '>'         ->  12
```

Tabulka 5.2: Regulární výraz s množinou jeho přechodů.

Úprava regulárních výrazů

Prvním krokem je převod regulárních výrazů z textové podoby do zápisu v němž jsou speciální znaky(tj. znaky které zastupují množiny znaků jednoduchých) stejně jako jednoduché znaky reprezentovány jedním elementem. Většinou je tento zápis reprezentován polem, ve kterém je číslem jednoznačně identifikován jak jednoduchý znak, tak i speciální znak včetně jeho funkce. V tomto kroku se také zpracují tzv. escape sekvence zapsané v textové podobě regulárního výrazu.

V takto vytvořených polích lze pak na základě znalosti aktuální pozice ve výrazu[2] nalézt následující možné přechody. Tyto přechody jsou popsány symbolem, nebo množinou symbolů, které je provádějí a číslem označujícím následující pozici ve výrazu po provedení tohoto přechodu. Příklad přechodů v regulárním výrazu se nachází v tabulce 5.2.

Schopnost nalezení těchto přechodů v každém výrazu využije konstruktor konečného automatu určeného k rozpoznávání všech terminálních symbolů jazyka.

Nalezení stavů konečného automatu

Algoritmus hledající stavy konečného automatu je založen na výše popsané schopnosti vytvořit z regulárního výrazu, seznam přechodů se vstupními znaky v rámci tohoto výrazu. Úkolem takového algoritmu je spojit jednotlivé pod-automaty všech výrazů do jednoho celku, určeného k jejich rozpoznávání tj. konečného automatu lexikální analýzy.

K tomu bude využita struktura zaznamenávající pro každý zpracovávaný stav vytvářeného automatu množinu pod-automatů, které reprezentuje, včetně záznamu pozic(stavů), ve kterých se tyto pod-automaty nachází. Dále se využije struktura implementující obyčejnou řadu, do které se budou vkládat popisy stavů (popsané výše popsanou strukturou) určené k zpracování.

Na začátku algoritmus vloží do této řady stav obsahující seznam všech pod-automatů vytvořených z regulárních výrazů, které budou mít indikátor pozice nastavený na nulu. Samotny algoritmus je implementován jako cyklus pracující nad řadou popisů, dokud v řadě nějaký popis stavu "čeká". V cyklu se popis stavu, který je na řadě vyzvedne a porovná s množinou již zpracovaných stavů. Pokud se v této množině nachází stejný popis stavu, tak to znamená že tento stav již byl zpracován a upraví se jen přechody, které vedly k vytvoření tohoto nového stavu tak aby směřovaly na stav uložený v seznamu již zpracovaných. Když

[2] Pozice ve které se nachází, nějaký algoritmus, který tento výraz zpracovává.

se popis stavu v této množině nenachází následuje jeho zpracování. Při něm se naleznou všechny možné přechody z tohoto stavu založené na možných přechodech v pod-automatech.

Je nutné vybírat znaky pro následující přechod od nejzákladnějších po nejobecnější. Po výběru znaku pro následující přechod je nutné znovu zkontrolovat všechny ostatní přechody pod-automatů zda neobsahují přechod se stejným znakem, nebo se znakem obecnějším, který popisuje i znak přechodu. Na základě těchto porovnání se vytvoří nový seznam pod-automatů a jejich pozic popisující nový stav automatu. Tento nově vytvořený popis stavu se vloží do řady. Zpracování stavu skončí s vyčerpáním všech možností přechodů všech pod-automatů. Tento stav se po zpracování vloží do množiny stavů automatu a jeho popis se uloží do seznamu již zpracovaných stavů.

Tento algoritmus v podstatě implementuje paralelizaci více konečných automatu za účelem nalezení odpovídajícího konečného automatu vhodného k využití lexikálním analyzátorem jazyka. Automat je reprezentován vygenerovanou množinou přechodových stavů.

5.3 Generování rozkladové tabulky

Rozkladová tabulka je jednou ze základních struktur řídících syntaktický rozklad zdrojového programu. Protože ve většině překladačů je to právě syntaktický analyzátor, který řídí rozklad zdrojového programu je tvorba této struktury(syntaktické tabulky) stěžejním krokem tvorby překladače.

Automatizovaná tvorba této tabulky je velmi dobře popsána v mnoha publikacích proto se zde omezím jen na stručný popis základních kroků tohoto postupu.

Jádra rozkladu

Konverze přes tzv. jádra rozkladu je jedna z metod určených pro automatizovanou tvorbu rozkladové tabulky. Tato metoda je založena na nalezení a popsání všech možných stavů (jader), do kterých se může syntaktický analyzátor dostat při rozkládání zdrojového souboru popsaného jazyka. Na základě vygenerovaného seznamu těchto jader je možné opět automatizovaným postupem vytvořit požadovanou rozkladovou tabulku.

Metoda hledání jader rozkladu se dělí na několik fází, kde první dvě fáze slouží k přípravě dat vhodných pro jednodušší zpracování fáze poslední. Těmito fázemi jsou, výpočet množiny *firsts*, výpočet množiny *follows* a nalezení požadovaných jader rozkladu popsaných v množině *kernels*. Následující fází, která už ale nepatří mezi fáze hledající jádra rozkladu je generování samotné rozkladové tabulky na základě nalezených jader rozkladu.

Výpočet množiny firsts

Množina firsts obsahuje pro každý symbol syntaktické analýzy, tj. pro každý terminál a neterminál seznam symbolů, které se mohou vyskytovat na začátku literálu rozepsaného z tohoto symbolu.

Z toho vyplývá že pro terminální symboly bude v tomto seznamu jen tento symbol samotný, protože žádný jiný literál se z něj vygenerovat nemůže. Množina first neterminálních symbolů se nalezne průchodem přes všechny pravidla u kterých neterminál tvoří hlavu. V každém kroku průchodu se do množiny first tohoto neterminálu přidá množná first prvního symbolu pravé strany pravidla.

Z důvodu jednoduchého nalezení množiny first pro terminální symboly tyto množiny nepočítám dopředu ale vypočítám (přiřadím index terminálu) až při potřebě této množiny

v dalším výpočtu.

Při výpočtu množiny first je pro fázi generování jader rozkladu nutné ke každému záznamu v množině first nějakého neterminálu vygenerovat seznam pravidel ze kterých vyplývá příslušnost tohoto záznamu do této množiny first. Takovému seznamu pravidel budeme dále říkat množina *first_rules*.

Výpočet množiny follows

Jak z názvu množiny follows vyplývá obsahuje tato pro každý neterminál[3] seznam symbolů které po něm mohou následovat v nějakém literálu rozepsaném ze startovacího neterminálu gramatiky.

Množina se opět pro neterminál nalezne průchodem přes všechny pravidla gramatiky jazyka, ale tentokráte se neterminál hledá v pravé straně pravidla. Pokud byl neterminál nalezen záleží na pozici ve které se v těle pravidla nachází, když se nenachází na poslední pozici přidá se do jeho množiny follow množina firsts symbolu vpravo od něj, v případě že se nachází na konci těla pravidla přidá se do jeho množiny follow množina follow neterminálu tvořícího hlavu pravidla.

Při tomto výpočtu se uplatní rekurze, protože množina follow, která se má přidat do množiny follow počítaného neterminálu ještě nemusí být spočítána. Avšak u této rekurze si musíme dát pozor a s každým voláním do hloubky předávat i informace o již výše počítaných neterminálech, tyto se už dále nesmí zanořovat z důvodů zacyklení algoritmu. Z důvodu efektivity výpočtu je vhodné množinu follow neterminálu počítaného kvůli požadavku jiného neterminálu zaznamenat jako spočítanou a v hlavním cyklu přes neterminály ji již zbytečně nepočítat. Toto opatření může být použito jen pokud ve výpočtu množiny follow nedošlo nikde k omezení zanořování kvůli opakovanému zpracování některého neterminálu.

Tato množina sehraje klíčovou roli při rozhodování u kterých terminálů na vstupu provést redukci podle pravidla gramatiky.

Výpočet jader rozkladu

Jádro rozkladu je částečně analogické ke stavu v konečném automatu. Je to struktura popisující jeden stav ve kterém se může nalézat syntaktický analyzátor při rozkladu vstupního řetězce. V jednom jádře jsou zaznamenány všechny pravidla ve kterých se může analyzátor "vyskytovat" při rozkladu, včetně pozic v těchto pravidlech, tato skupina se nazývá pravidla jádra rozkladu. Druhá skupina v jádře se nazývá skupina přechodů, ta obsahuje indexy jader ze kterých se do tohoto jádra vstoupilo a se kterým elementem (tj. terminálem nebo i neterminálem) byl přechod proveden.

Algoritmus výpočtu jader rozkladu začíná vytvořením prvního jádra, které obsahuje pouze první pravidlo s iterátorem na pozici nula, tj. před prvním symbolem těla tohoto pravidla, toto nově vytvořené jádro se vloží do dynamického pole, které slouží právě jako seznam jader rozkladu. Algoritmus prochází přes všechny jádra rozkladu v tomto dynamickém poli a postupně zpracovává každé jádro, skončí když dorazí na konec tohoto pole. Při zpracování jednoho jádra se prochází všechny jeho pravidla. Dále se pracuje pouze s těmi pravidly, které nemají iterátor na konci svého těla. U takového pravidla se v množině first nalezne seznam elementů, kterými může začínat element nacházející se za iterátorem v pravidle. Pro každý takový element se začne vytvářet nové jádro rozkladu (zatím se nevkládá do pole koncových jader rozkladu).

[3] Na rozdíl od množiny firsts není důvod vytvářet follow pro terminály

Do tohoto nového jádra se vloží seznam pravidel z množiny first_rules náležící k elementu se kterým se vytváří, iterátor v těchto pravidlech bude umístěn na pozici 1 (tj. za zpracovávaným elementem z množiny first). Dále se do seznamu pravidel tohoto jádra vloží pro element odpovídající samotnému elementu jehož množina first byla použita pravidlo aktuálního jádra, které bylo základem pro vznik nového jádra, s tím že se jeho iterátor posune o jednu pozici doprava (tj. za zpracovávaný element).

Do množiny přechodů nového jádra se vloží přechod ze zpracovávaného jádra s aktuálně zpracovávaným elementem z množiny first.

Po vytvoření nového jádra rozkladu se projdou všechny existující jádra a hledá se jádro se shodným seznamem pravidel a jejich iterátorů jako má nově vytvořené jádro, množina přechodů nehraje roli. Pokud není takové jádro nalezeno vloží se nové jádro do pole jader rozkladu. V případě že se takové jádro nalezne, přidá se množina přechodů nového jádra do nalezeného existujícího jádra.

Po dokončení tohoto algoritmu máme kompletní množinu jader rozkladu odpovídající předané gramatice.

Generování rozkladové tabulky

Každé jádro rozkladu popisuje právě jeden stav, ve kterém se může nalézat syntaktický analyzátor při zpracování zdrojového textu, proto také rozkladová tabulka jejíž řádky odpovídají jednotlivým stavům obsahuje právě tolik řádků kolik bylo vypočítáno jader rozkladu.

Rozkladová tabulka se skládá ze dvou částí, těmi jsou tabulka akcí a tabulka přechodů. Tabulka akcí slouží syntaktickému analyzátoru k nalezení následující akce, kterou by měl vykonat, zatímco tabulka přechodů popisuje do jakého stavu se syntaktický analyzátor přesune po redukci určitého pravidla. Mezi akce nalézající se v tabulce akcí patří akce `shift` a akce `reduce`.

Tabulka přechodů rozkladové tabulky se vyplní na základě množin přechodů jader rozkladu. Pokud jádro obsahuje přechod, který se provádí na základě neterminálního symbolu zapíše se do tabulky přechodů na řádek stavu z kterého přechod přechází a sloupec neterminálu s kterým se přechází index jádra ve kterém se tento přechod nachází.

Akce typu `shift` se v tabulce akcí vyplňují také na základě množin přechodů v jádrech rozkladu. To znamená, pokud jádro rozkladu obsahuje přechod, který se provádí na základě terminálního symbolu, zapíše se do rozkladové tabulky na řádek stavu ze kterého se přechází a sloupec terminálního symbolu akce `shift` s indexem jádra ve kterém se tento přechod nachází.

Pro vygenerování akcí `reduce` je potřeba projít všechny jádra rozkladu, ve kterých se hledají pravidla, která mají iterátor za posledním prvkem těla pravidla. Pokud se takovéto pravidlo nalezne prochází se všechny elementy z množiny follow hlavy tokového pravidla. Pracuje se pouze s terminály, a u každého z nich se znovu projdou všechny pravidla jádra z důvodu zjištění zda neexistuje nějaké, které by obsahovalo iterátor před terminálem z množiny follow s kterým by jsme v tomto případě chtěli předčasně redukovat pravidlo. Pokud takové pravidlo neexistuje vloží se do rozkladové tabulky na řádek aktuálního zpracovávaného jádra a sloupec prvku z množiny follow akce `reduce` s indexem zpracovávaného pravidla.

Výsledkem provedení všech těchto postupů je kompletní rozkladová tabulka připravená pro použití syntaktickým analyzátorem. Její výpočet není jednoduchý, ale díky němu bude její použití při rozkládání zdrojového kódu v syntaktické analýze o to jednodušší.

5.4 Překlad sémantických podprogramů

Sémantické podprogramy slouží k popisu reakcí překladače na redukci pravidla rozkladové gramatiky. Jedním z účelů těchto programů je kontrola správnosti sémantického zápisu kódu, do které patří kontrola typů proměnných a konstant. Dalším stejně důležitým úkolem sémantických podprogramů je generování intermediárního kódu překladu.

Překlad sémantických podprogramů je silně rekurzivní záležitost. Protože implementovaný program je v podstatě překladač jakéhokoli jazyka do jazyka jiného(pravděpodobně nějakého typu intermediárního kódu) je vhodný i pro překlad programů zapsaných v jazyce pro zápis sémantických podprogramů.

Jediné omezení oproti jiným jazykům je v absenci kódu pro zápis sémantických podprogramů. Proto se reakce na redukci pravidla v tomto jazyce redukuje na jednu instrukci.

Tato instrukce zapsaná v pravidlech syntaktické gramatiky sémantického jazyka se také interpretuje. Nejde však o interpretaci jako u sémantického kódu, ale jen nalezení indexu instrukce (v textové formě) pomocí konečného automatu (zpracovaného stejným způsobem jako konečný automat lexikální analýzy). Tento index se zapíše k popisu pravidla gramatiky. Na základě těchto indexů se provádí překlad programů sémantických podprogramů.

Pro implementaci lexikálního analyzátoru sémantických podprogramů se generuje konečný automat na základě terminálů popsaných v definičním souboru jazyka sémantických podprogramů.

Syntaktická analýza sémantických podprogramů se provádí stejným způsobem jako syntaktická analýza překládaného jazyka, s tím rozdílem že při redukci pravidla se spouští sémantický kód napevno "zadrátovaný" v programu, který je identifikovaný indexem zapsaným u redukovaného pravidla. Tyto napevno zapsané kódy v závislosti na posloupnosti indexů redukovaných pravidel gramatiky sémantického jazyka, generují spustitelný (interpretovatelný) kód sémantického podprogramu, který se přiřadí k pravidlu gramatiky překládaného jazyka.

Takto se zpracují všechny pravidla (jejich sémantické programy) ze souboru popisujícího jazyk.

5.5 Intermediární kód překladu

Tato část se v podstatě zabývá samotným překladem zdrojového souboru na intermediární kód, to znamená že využívá již výše popsaných struktur.

Překladač je navržen pro syntaxí řízené zpracování zdrojového programu, to znamená že překlad je řízen syntaktickým analyzátorem, který podle potřeby požaduje po lexikálním analyzátoru symboly a spouští sémantické podprogramy při redukci pravidel.

Lexikální analyzátor

Lexikální analyzátor je v překladači reprezentován jedinou funkcí. Tato funkce načítá postupně znaky ze vstupního souboru, které následně upraví(tzn. zpracuje escape sekvence, kontroluje ukončení souboru apod.). Funkce využívá výše popsaného konečného automatu k rozpoznávání symbolů ve vstupním zdrojovém kódu.

Na začátku inicializuje proměnnou popisující stav konečného automatu na nulu, každý terminál začíná ve stavu nula. Podle znaku načteného postupem popsaným výše se rozhoduje do jakého následujícího stavu automat přejde. V případě že neexistuje žádný stav do

kterého by mohl konečný automat přejít funkce vrací celočíselnou hodnotu definovanou u tohoto stavu.

Tuto hodnotu, která se vytváří při konstrukci konečného automatu obsahuje každý jeho stav. Hodnota je nastavená na index terminálu, který je v aktuálním stavu ze vstupu rozpoznán, pokud je hodnota nastavená na maximální celočíselnou hodnotu znamená to že tento stav nerozpoznal žádný terminál a syntaktická analýza podle této hodnoty odhalí chybu.

Po každém úspěšném přijetí znaku (tzn. automat přejde se znakem do dalšího stavu) se tento znak ještě jednou zpracuje z důvodů ukončení automatu pokud znak byl ukončovací symbol vstupu, počítání řádku pokud je znak konec řádku a posunutí indexu ve vstupním řetězci (u tzv. escape znaků se iterátor musí posunout o větší úsek).

Syntaktický analyzátor

Syntaktický analyzátor provádějící rozklad zdrojového řetězce, je v podstatě také jako lexikální analyzátor popsán jednou funkcí[4], která využívá výše popsané vygenerované rozkladové tabulky k rozkladu vstupního řetězce.

Syntaktický analyzátor pracuje se symboly, které získá z funkce implementující lexikální analýzu, nad těmito symboly se provádí rozklad zdola nahoru popsaný rozkladovou tabulkou.

Algoritmus syntaktické analýzy využívá pro svou práci dynamický zásobník popisující aktuální stav analyzátoru a u terminálních symbolů navíc i jejich pozici ve zdrojovém řetězci (Pro následující zpracování sémantickými podprogramy).

Před spuštěním samotného jádra analyzátoru se vloží na vrchol zásobníku stav 0, který reprezentuje počáteční stav syntaktického analyzátoru. Základem analyzátoru je nekonečný cyklus[5], ve kterém se provádí následující operace.

Na začátku cyklu se pomocí lexikálního analyzátoru získá nový symbol ze zdrojového řetězce. Pokud je tento terminál jeden z množiny tzv. vynechávaných terminálů provede se nové volání lexikálního analyzátoru. Předchozí krok se provede pouze v případě že předchozí symbol byl již zpracován (tzn. že na něj již byla aplikována akce `shift`).

Následujícím krokem je nalezení akce v rozkladové tabulce, kterou by měl syntaktický analyzátor provést. Tato akce se nalezne na základě stavu na vrcholu zásobníku a aktuálního terminálu načteného ze zdrojového souboru.

Další způsob zpracování se odvíjí od typu akce, která se nalezla v rozkladové tabulce, tato akce může nabývat typu `shift`, nebo typu `reduce`. Pokud je akce typu `shift`, vloží se se na zásobník element popisující nový stav (popsaný v akci `shift`) a pozice terminálu ve zdrojovém řetězci, získaná již při načítání terminálního symbolu lexikálním analyzátorem. V případě že tímto přesouvaným terminálem je koncový terminál gramatiky[6] ukončí se cyklus rozkladu. Akce `reduce` popisuje v podstatě pravidlo, podle kterého se má již přečtený vstup ze zdrojového souboru (jeho postfix) redukovat. Z toho vyplývá že z vrcholu rozkladového zásobníku se odstraní takový počet elementů, který odpovídá počtu prvků těla pravidla. Na základě hlavy pravidla a aktuálního stavu na vrcholu zásobníku se nalezne stav do kterého přejde automat po této redukci (tj. bude vložen na vrchol rozkladového zásobníku).

Takováto syntaktická analýza by neměla žádný význam pokud by se informace získané ze zdrojového souboru nevyužily k dalšímu zpracování. Toto zpracování provedou jednotlivé

[4] To neznamená že by nebylo možné jej rozložit do více funkcí.

[5] Ukončit se může jen na základě přijetí ukončovacího znaku gramatiky.

[6] Terminál ve vstupních pravidlech zapsaný na konci startovacího pravidla gramatiky.

sémantické podprogramy přiřazené ke každému pravidlu rozkladové gramatiky.

Sémantické podprogramy

Ve fázi překladu zdrojového souboru se sémantickým podprogramem rozumí už zkompilovaná posloupnost instrukcí intermediárního kódu.

Jazyk sémantických podprogramů vyžaduje typovaný zápis proměnných z důvodu rychlého zpracování jeho intermediárního kódu.

Pro potřebu algoritmu se alokuje místo v paměti, které bude sloužit jako paměťový prostor proměnným sémantického podprogramu. Tato operace se provede jen jednou a to už před spuštěním prvního sémantického podprogramu, na začátku syntaktické analýzy. Paměťový prostor se dále nemění protože všechny proměnné sémantických podprogramů jsou globální.

Interpretace jednotlivých podprogramů se provádí následujícím způsobem. Vytvoří se index popisující pozici v kódu podprogramu, z které se načítá následující instrukce programu, a nastaví se na hodnotu označující začátek kódu.

Algoritmus pracuje jako cyklus, který se opakuje dokud nedosáhne ukazatel programu konce kódu. V každém kroku tohoto cyklu se načte následující instrukce kódu z pozice, na kterou ukazuje programový ukazatel. Podle indexu načtené instrukce se nalezne kód k provedení a tento se provede. Při provádění kódu může dojít ke změně programového ukazatele (například při zpracování konstant, řetězců apod.), proto je nutné s tímto počítat.

Pokud by v této fázi nebyla implementována žádná forma výstupu překladač by sloužil jen k vytížení procesoru. Proto umožňuje jazyk sémantických podprogramů zápis příkazů vypisujících informace jak na standardní výstup programu, tak i do binární struktury popisující bytový tok instrukcí, nebo čehokoli jiného[7].

[7] Záleží na autorovi jakou zvolí strukturu výstupního kódu.

Kapitola 6

Navržený jazyk sémantických podprogramů

Tento skriptovací jazyk byl navržen za účelem snazšího definování sémantických operací prováděných při redukování pravidel v rámci překladu zdrojového programu. Způsob využití programů zapsaných v tomto jazyce je popsán v předchozí kapitole.

6.1 Základní popis jazyka

Vzhledem k obecnému návrhu programu, který umožňuje v rámci určitých mezí vytvořit překladač libovolného jazyka, by bylo nevhodné ponechat sémantickou analýzu programu pouze na složitějším zápisu rozkladových pravidel gramatiky jazyka.

V případě že by byl jazyk navrhován přímo v nějakém systémovém jazyku (pravděpodobně v jazyku `C`), by byly jednotlivé sémantické úkony zapsány přímo v tomto jazyce, ve formě podprogramů, které by se volaly při redukci sémanticky "citlivých" pravidel.

Protože sémantika jednotlivých programů může být velmi různorodá není možné vytvořit jednoduchý obecný přístup, jakým jsou například gramatiky v otázce syntaktické analýzy. Jediným zatím známým řešením je zápis sémantické analýzy jednotlivých částí jazyka v programovacím jazyce.

Způsob implementace jazyka

Jazyk je navržen a zapsán stejnou metodou, která je určená pro návrh jazyků s využitím implementovaného programu. Jediný rozdíl mezi tímto jazykem a vnějším navrhovaným jazykem je logicky absence jazyka pro zápis sémantických podprogramů, proto jak jsem psal v úvodu této kapitoly je nutné sémantiku tohoto jazyka popsat přímo v programovacím jazyku `C`.

Zdrojový soubor popisující překladač sémantických podprogramů je zapsán přímo ve zdrojovém kódu implementovaného programu jako řetězec, a před použitím se v případě že se nenačítá přímo ze souboru musí jeho překladač vytvořit, stejně jako překladač vnějšího jazyka.

Tento zdrojový text má stejný tvar jako text popsaný v podkapitole 5.1., s tím rozdílem že symboly `<semantic_code>` nejsou nahrazeny sémantickým kódem, ale instrukcemi identifikujícími sémantický kód zapsaný přímo v jazyce `C`.

6.2 Datové typy jazyka

Z důvodu rychlé interpretace je tento jazyk navržen jako jazyk s typovanými proměnnými. Sémantická kontrola typů těchto proměnných se provede již při překladu a při interpretaci se s ní již nemusíme zdržovat. Jazyk navrhovaný za účelem provádění sémantické analýzy by měl umožňovat práci s celočíselnými typy a řetězci. Je zřejmé že bez celočíselných typů se neobejde žádný programovací jazyk. Zpracování řetězců je důležité z toho důvodu že jediným vstupem jazyka budou právě řetězce jednotlivých symbolů, které rozeznává lexikální analyzátor ve zdrojovém souboru programu.

Výčet jednotlivých datových typů

K základním datovým typům, je většinou potřeba přiřadit i složitější datové struktury, kterými jsou například pole, zásobníky a další. Z tohoto důvodu umožňuje tento jazyk zápis polí obsahujících posloupnosti základních typů. Následuje výčet datových typů použitelných v jazyce.

- `int` - popisuje celociselny typ, ktery je přijímán jako argument většiny základních operátorů programovacích jazyků.
- `int[]` - popisuje dynamické pole (zásobník), které obsahuje jako prvky proměnné výše popsaného datového typu.
- `int[][]` - definuje typ, popisující dynamické pole obsahující jako prvky výše popsané dynamické pole celých čísel.
- `string` - popisuje řetězec znaků, umožňuje nad těmito řetězci provádět základní operace, jakými jsou například konkatenace, získáni podřetězce, formátování na základě nastavených escape sekvencí atd.
- `string[]` - označuje dynamické pole, obsahující výše popsaný typ `string` jako elementy pole.
- `string[][]` - definuje datový typ, popisující dynamické pole, které obsahuje jako prvky výše popsané pole řetězců.

Tento výčet datových typu doposud byl a stále je dostačující pro zápis sémantických operací v navrhovaném překladači.

Datový typ `int`

Typ `int` je velmi podobný stejně nazvanému typu v jazyce `C`. Podobně jako v jazyce `C` je používán v navrženém jazyce jako celočíselný tak i logický typ. To znamená že výsledkem testu na pravdivost nějakého výrazu bude typ `int` s hodnotou 0 nebo 1.

Zápis konstanty typu `int` je možné provést několika způsoby. V řetězci definujícím překladač sémantických podprogramů jsou konstanty typu `int` popsány následujícím způsobem.

```
oct_int      {0\[01234567\]\[01234567\]\*}
dec_int      {\d\d\*}
hex_int      {0x\[\dabcdef\]\*}
char_int     {'\!''}
```

Z tohoto zápisu vyplývá, že je možné konstantu typu `int` zapsat jako číslo v oktalovém, dekadickém a hexadecimálním zápisu, nebo také jako znak, jehož číselnou hodnotu konstanta ponese.

Následuje seznam operátorů aplikovatelných na typ `int`. Protože jsou v kódu operace popsány dvojím způsobem, názvem operace a řetězcem rozpoznávaným v programu, přiřazeným k této operaci, budou u každého operátoru zapsány oba způsoby identifikace.

- `set =` - přiřadí levému operandu hodnotu pravého operandu, levý operand musí být proměnná, vrací odkaz na proměnnou.
- `and and` - logická operace aplikující logický and na operandy, vrací 1 pokud žádný z operandu není nulový, v opačném případě vrací 0.
- `or or` - logická operace aplikující logický or na operandy, vrací 1 pokud jeden z operandů není nulový, v opačném případě vrací 0.
- `compare ==` - porovná hodnoty operandů, vrací 1 pokud jsou shodné, v opačném případě vrací 0.
- `neg_compare !=` - porovná hodnoty operandů, vrací 1 pokud nejsou shodné, v opačném případě vrací 0.
- `less <` - porovná hodnoty operandů, vrací 1 pokud je první operand menší než druhý, v opačném případě vrací 0.
- `greater >` - porovná hodnoty operandů, vrací 1 pokud je první operand větší než druhý, v opačném případě vrací 0.
- `less_equal <=` - porovná hodnoty operandů, vrací 1 pokud je první operand menší nebo stejný jako druhý, v opačném případě vrací 0.
- `greater_equal >=` - porovná hodnoty operandů, vrací 1 pokud je první operand větší nebo stejný jako druhý, v opačném případě vrací 0.
- `add +` - sečte operandy, vrácí součet.
- `sub -` - odečte druhý operand od prvního, vrací rozdíl.
- `post_inc ++` - lze aplikovat jen na proměnné, zvýší hodnotu proměnné o 1, vrací původní hodnotu proměnné.
- `post_dec --` - lze aplikovat jen na proměnné, sníží hodnotu proměnné o 1, vrací původní hodnotu proměnné.
- `pre_inc ++` - lze aplikovat jen na proměnné, zvýší hodnotu proměnné o 1, vrací novou hodnotu proměnné.
- `pre_dec --` - lze aplikovat jen na proměnné, sníží hodnotu proměnné o 1, vrací novou hodnotu proměnné.

Datový typ `string`

Tento datový typ je určen k práci s řetězci znaků. Definuje základní operace, kterými jsou konkatenace, získání podřetězce apod. V jazyce sémantických podprogramů slouží tento typ v první řadě ke zpracování jmen proměnných, konstant, funkci apod., rozpoznávaných lexikálním analyzátorem ve vstupním řetězci. Příkaz jazyka `get_rule_body` slouží k získání právě těchto řetězců, se kterými je pomocí tohoto typu možné dále pracovat, například kontrolovat zda jsou jména funkcí jedinečná apod.

Konstanta typu `string` se zapisuje obvyklým způsobem, mezi uvozovky. Zápis konstanty popisující řetězec v popisu překladače vypadá následovně.

```
str     {"\!"\*"}
```

Tento popisuje text složený z počáteční uvozovky, posloupnosti libovolných znaků jiných než uvozovka a koncové uvozovky. Operace pracující s tímto datovým typem jsou zobrazeny v následujícím seznamu. Podobně jako v předchozím případě jsou zobrazeny oba druhy identifikace operací.

- `set =` - přiřadí levému operandu hodnotu pravého operandu, levý operand musí být proměnná, vrací odkaz na proměnnou.
- `swap <=>` - zamění hodnoty dvou operandů, žadný z operandů nesmí být konstantní, vrací novou hodnotu prvního operandu.
- `compare ==` - porovná hodnoty operandů, vrací 1 pokud jsou shodné, v opačném případě vrací 0.
- `neg_compare !=` - porovná hodnoty operandů, vrací 1 pokud nejsou shodné, v opačném případě vrací 0.
- `conc .` - konkatenace dvou operandů, vrací hodnotu nového dočasného řetězce.

Část příkazů jazyka přijímá řetězce jako své argumenty a slouží k další manipulaci s nimi. Do této skupiny patří příkazy pro zjištění velikosti pole, konverze na celé číslo apod.

Datový typ pole

Operace popsané v této části jsou aplikovatelné na jakýkoli datový typ z množiny: `int[]`, `int[][]`, `string[]`, `string[][]`. Tyto datové typy popisují dynamické pole proměnné velikosti. Většina operací nad těmito typy je definována jako příkazy, tj. funkce, které požadují pole jako argument.

Konstantní pole se v jazyce zapisuje s využitím příkazu `array`, nebo příkazu `empty` následujícím způsobem.

```
int[]       int_a = array(1,2,3),
            int_a_empty = empty();
int[][]     int_aa = array(
               array(1,2,3),
               array(4,5,6),
               array(7,8,9)
            );
```

```
string[]    str_a = array("str0","str1","str2");
string[][]  str_aa = array(
               array("str0","str1","str2"),
               array("str3","str4","str5")
            );
```

Z příkladu je zřejmé že příkaz `empty` definuje prázdné pole libovolného typu a jeho přiřazením do proměnné obsahující nějaké pole bude obsah pole proměnné odstraněn a nastaven na prázdné pole.

Konstantní pole je definováno příkazem `array`, který přijímá neomezený počet konstantních argumentů stejného typu, a po jeho uzavření je vytvořeno konstantní pole těchto přijatých typů.

Následuje seznam operací pracujících s datovými typy popisujícími pole. Stejně jako v předchozích případech jsou zobrazeny oba druhy identifikací operací.

- `set =` - přiřadí levému operandu hodnotu pravého operandu, levý operand musí být proměnná, vrací odkaz na proměnnou.
- `swap <=>` - zamění hodnoty dvou operandů, žadný z operandů nesmí být konstantní, vrací novou hodnotu prvního operandu.
- `compare ==` - porovná hodnoty operandů, vrací 1 pokud jsou shodné, v opačném případě vrací 0.
- `neg_compare !=` - porovná hodnoty operandů, vrací 1 pokud nejsou shodné, v opačném případě vrací 0.
- `get_element []` - vrací prvek pole na základě celočíselné hodnoty určující jeho pozici v poli.
- `last_of last` - vrací poslední prvek v poli.
- `tail_of tail[]` - vrací prvek pole na základě celočíselné hodnoty určující jeho pozici v poli počítanou od konce pole.

Velká část příkazů, zapisovaných jako funkce je určena pro práci s datovými typy reprezentující pole jednodušších datových typů.

6.3 Příkazy jazyka

Do této skupiny patří příkazy, které se v jazyce zapisují jako volání funkcí v jazyce `C`. Je nevhodné jim říkat funkce, protože jazyk nepodporuje žádnou metodu, kterou by bylo možné definovat funkce, a popisované zápisy jsou přímo vestavěny do syntaxe kódu. Syntaxi zápisu příkazů je jednoduché měnit[1] a několikrát se tak stalo v průběhu vývoje. Původní zápis byl podobný volání metod objektů v jazyce `C`, ale protože vzbuzoval dojem že jazyk nějakým způsobem podporuje objekty tak byl nahrazen dále popsaným zápisem.

Následuje seznam jednotlivých příkazu s jejich stručným popisem. V popisu příkazů bude použita konvence pro zápis požadovaných typů argumentů příkazu s následujícím významem: `<any>` - jakýkoli typ, `<array>` - typ obsahující pole prvků, `<array_type>` - typ

[1] Definice syntaxe je popsána ve znakovém řetězci.

obsažený v poli, `...` - libovolný počet argumentů předcházejícího typu, `|` - operátor or pro výběr jednoho typu z množiny.

- `out( <any> , ... )` - vypíše hodnotu všech argumentů v zadaném pořadí na standardní výstup.
- `assert( int )` - zkontroluje logickou hodnotu předaného argumentu, pokud je nepravdivá zobrazí na standardní výstup chybovou zprávu obsahující řádek na kterém se tento příkaz nachází a řetězec, kterým je zapsán jeho argument.
- `substr( string , int , int )` - vytvoří podřetězec z řetězce v prvním argumentu začínající na pozici popsané druhým argumentem o délce, kterou udává třetí argument.
- `format( string , int[] )` - formátuje tzv. escape sekvence v řetězci popsaném prvním argumentem. Druhý argument je pole typů `int`, které musí mít minimální délku 3 a počet jeho prvků musí být lichý. V tomto poli první číslo popisuje escape znak, a následující dvojice znaku znak vyskytující se za escape znakem a znak, kterým se tato dvojice nahradí. Pokud se nachází za escape znakem znak, který se dále nenachází v popisu ohlásí se chyba.
- `rule_body( int )` - získá řetězec popisující terminální symbol v pravidle u kterého se tento sémantický kód nachází na pozici určené parametrem příkazu.
- `line_cnt()` - získá číslo řádku rozkládaného kódu na kterém se provádí redukce pravidla, k němuž je přiřazen sémantický kód obsahující tento příkaz.
- `to_int( string )` - zkonvertuje řetězec na celé číslo.
- `to_string( int )` - zkonvertuje celé číslo na řetězec.
- `empty()` - vytvoří prázdné pole, přiřaditelné do pole libovolného typu.
- `array( <array_type> , ... )` - vytvoří konstantní pole, jehož typ se rozhodne podle předaných argumentů. Argumenty nesmí být vícerozměrné pole.
- `size ( string | <array> )` - získá velikost řetězce, nebo aktuální počet prvků v poli.
- `push ( <array> , <array_type> )` - vloží na konec pole popsaného prvním argumentem, hodnotu danou druhým argumentem.
- `pop( <array> )` - Vyzvedne z pole prvek nacházející se na jeho konci. Pole musí obsahovat nejméně jeden prvek.
- `remove( <array> , int )` - odstraní z konce pole popsaného prvním argumentem počet prvků určený druhým argumentem. Podobně jako u příkazu `pop` je nutné aby pole obsahovalo minimálně takový počet prvků, který se požaduje k odstranění.
- `get_idx( <array> , <array_type> )` - nalezne v poli obsaženém v prvním argumentu pozici prvního prvku, který má stejnou hodnotu jako druhý argument příkazu, pokud tento prvek v poli není obsažen vrátí příkaz hodnotu -1.
- `get_next_idx( <array> , <array_type> , int )` - plní stejnou funkci jako příkaz `get_idx` s tím rozdílem že vyhledává prvek od pozice určené třetím argumentem.

last	tail	int
string	if	fi
else	do	while

Tabulka 6.1: Seznam klíčových slov.

6.4 Řízení toku programu

Řízení toku programu v jazyce sémantických podprogramů podporuje obvyklé metody, kterými jsou podmínky a cykly. V jazyce není možné explicitně vytvářet bloky kódu, ty je možné vytvořit jen jako součásti podmínky, nebo cyklu. V tabulce 6.1. se nachází seznam klíčových slov jazyka, část z těchto slov je používaná při řízení toku programu. V seznamu klíčových slov se nenachází výše popsané příkazy jazyka.

Větvení toku programu

K rozdělování toku programu v závislosti na splnění podmínky se využívají klíčová slova `if`, `else` a `fi`. Následuje zápis obyčejné podmínky.

```
if <expression>
   <command>
   ...
fi
```

Tento zápis znamená že se posloupnost příkazů `<command>` provede pokud je podmínka `<expression>` platná. Stejně jako v jiných jazycích se klíčové slovo `else` v tomto jazyku používá k označení kódu, který se má provést při neplatnosti podmínky. Následuje zápis podmínky s klíčovým slovem `else`.

```
if <expression>
   <command>
   ...
else
   <command>
   ...
fi
```

Zápis smyček toku programu

Protože byl jazyk navrhován za účelem zápisu syntaktických podprogramů autorem, obsahuje pouze jeden druh zápisu cyklů. Tím je cyklus s testem podmínky na konci. Cyklus s testem na začátku a cyklus for je jednoduché pomocí tohoto cyklu nahradit. (Zápis těchto cyklů tímto způsobem v jazyce `C` vede v případě bez optimalizací k efektivnějšímu kódu, nemluvě o predikci procesorů při skoku zpět, proto je autor s nimi natolik sžitý že jiné typy smyček nepoužívá :). K zápisu cyklu v jazyce sémantických podprogramů slouží klíčové slova `do` a `while`. Následuje zápis cyklu s podmínkou na konci.

```
do
   <command>
```

```
   ...
while <expression>;
```

Tělo cyklu tvořené posloupností příkazů se provede pokaždé alespoň jednou, další průchody se provádějí dokud je podmínka `<expression>` pravdivá.

6.5 Příklad zápisu operace v jazyce

Následující příklad demonstruje použití jazyka sémantických podprogramů. V tomto zápisu je zobrazen popis pravidla přijímajícího řetězcovou konstantu využívaný v návrhu skriptovacího jazyka (viz. níže).

```
 0   <A> -> string ->>
 1      {
 2         c_name = rule_body(0);
 3
 4         if (c_idx = get_idx(string_consts,c_name)) == -1
 5            push(string_consts,c_name);
 6            c_idx = size(string_consts) - 1;
 7         fi
 8
 9        $out("const:string:",c_name,":",c_idx,"\n"); $FIXME
10        push(var_type_stack,vt_const);
11        push(tmp_exp_graph,i_string_const);
12        push(tmp_exp_graph,c_idx);
13     }
```

Tento kód popisuje jedno pravidlo, rozkladové gramatiky jazyka, kterým je redukce terminálu popisujícího řetězcovou konstantu na neterminál označující nejnižší prvek v hierarchii výrazů. Samotný sémantický kód je uzavřen mezi složenými závorkami (z tohoto důvodu nemohou být složené závorky používány v jazyce sémantických podprogramů).

Význam tohoto kódu je následující. Do proměnné `c_name` se načte řetězec popisující terminál na vstupu. Nalezne se pozice řetězce `c_name` v poli obsahujícím jména všech konstantních řetězců doposud načtených ze vstupního souboru, a uloží se do proměnné `c_idx`. Pokud nebyl řetězec v poli nalezen vloží se do něj a proměnná `c_idx` je nastavena na jeho pozici (tj. velikost pole mínus 1). Následuje zakomentovaný výstup na standardní výstup. Do pole uchovávající typy dat na zásobníku (pro kontrolu při redukci operací) se vloží hodnota označující konstantu. Nakonec se do pole popisujícího graf vytvářeného výrazu vloží informace identifikující řetězcovou konstantu a její pozice v poli řetězcových konstant.

Z příkladu je zřejmé že sémantické programy neslouží pouze k jednoduché kontrole sémantické správnosti zápisu, ale jsou využívány pro generování struktur popisujících samotný výsledek překladu programu.

Kapitola 7

Navržený jazyk pro zpracování obrazu

Skriptovací jazyk určený pro zpracování obrazu je navržen jen částečně, jeho popis je vytvářen jako vstupní soubor implementovaného programu, který je schopný překladač tohoto jazyka interpretovat, nebo vytvořit jeho zdrojový soubor v jazyce `C++`. V této fázi jsou v jazyce navrženy základní struktury pro řízení toku programu a popis základních datových typů.

7.1 Základní popis jazyka

Navrhovaný jazyk bude sloužit jako skriptovací jazyk pro zjednodušení zápisu a definování obrazových operací. Jazyk bude netypovaný. Výsledky překladu tohoto jazyka budou: bytový kód umožňující snadnou interpretaci, nebo popis grafového zápisu toku programu, nad kterým budou prováděny optimalizační úkony. Výsledkem návrhu tohoto jazyka by měl být jazyk umožňující snadný popis operací s obrazem, jakými jsou například: načítání obrazů z datových úložišť, operace nad obrazy (např: různé typy filtrů, kombinování více obrazů, statistika informací v obraze apod.), konverze obrazových formátů a jejich opětovné ukládání na datová úložiště, případně zobrazování na k tomu určených zařízeních. Jedná se o návrh jazyka umožnujícího zápis těchto operací ne jejich realizaci, výsledkem překladu jazyka, případně překladu optimalizovaného grafu bude kód popisující nízkoúrovňové operace realizované například na speciálním hardwaru, nebo jiným softwarem.

Popis implementace překladače jazyka

Překladač tohoto jazyka je realizován jako vstupní soubor implementovaného programu, obsahující popis jazyka na úrovni regulárních výrazů, rozkladových gramatik a sémantických podprogramů. Tvar tohoto vstupního souboru je popsán v podkapitole 5.1.

Z popisu překladače jazyka vytvoří (Za předpokladu bezchybnosti tohoto popisu) implementovaný program samotný překladač, reprezentovaný daty v paměti programu. Takto vytvořenou reprezentaci překladače je možné použít k překladu zdrojových souboru navrhovaného jazyka. Vytvořený překladač je možné také uložit do binárního souboru, ze kterého je možné tento překladač načíst při požadavku na přeložení jiného zdrojového souboru. Další možností jak využít takto vytvořený překladač je vygenerovat zdrojový soubor v jazyce `C++`, implementující popsaný překladač bez jakýchkoli nadbytečných funkcí. Poslední popsaná možnost není ještě kompletně implementována, zdrojový soubor a program, který

vznikne jeho přeložením popisují zatím jen lexikální a syntaktickou analýzu jazyka, bez příslušných sémantických operací.

7.2 Datové typy jazyka

Na rozdíl od jazyka určeného pro popis sémantických podprogramů, který byl navrhován s požadavkem na rychlou interpretaci, navrhovaný jazyk pro zpracování obrazu je vytvářen jako nástroj pro zjednodušení složitých operací. Z výše popsaného důvodu není tento jazyk typovaný a proměnné tohoto jazyka si v průběhu interpretace ponesou popis datového typu sebou.

V současné fázi návrhu jsou do jazyka zahrnuty následující základní datové typy.

- `int` - datový typ popisující znaménkové celé číslo. V jazyce popisuje i logickou hodnotu, která je pravdivá pokud není rovna nule.
- `float` - slouží k popisu čísla s plovoucí řádovou čárkou.
- `string` - popisuje znakový řetězec včetně základních operací nad řetězci.

Protože je jazyk netypovaný, není možné v průběhu překladu jazyka (do intermediárního kódu) provádět typovou kontrolu operací. Z tohoto důvodu nebude následovat popis operací přiřazený k jednotlivým typům, ale jen popis jednotlivých operací, kde se u každé z nich bude nacházet výčet typů, na které je možné tuto operaci aplikovat. Seznam základních operací nad datovými typy následuje.

- `set =` - přiřadí levému operandu hodnotu pravého operandu, levý operand musí být proměnná, vrací odkaz na proměnnou. Pracuje nad všemi datovými typy.
- `add_set +=` - přičte k levému operandu hodnotu pravého operandu, levý operand musí být proměnná, vrací odkaz na proměnnou. Pracuje nad datovými typy `int` a `float`.
- `sub_set -=` - odečte od levého operandu hodnotu pravého operandu, levý operand musí být proměnná, vrací odkaz na proměnnou. Pracuje nad datovými typy `int` a `float`.
- `mul_set *=` - vynásobí levý operand hodnotou pravého operandu, levý operand musí být proměnná, vrací odkaz na proměnnou. Pracuje nad datovými typy `int` a `float`.
- `div_set /=` - podělí levý operand hodnotou pravého operandu, levý operand musí být proměnná, vrací odkaz na proměnnou. Pracuje nad datovými typy `int` a `float`.
- `mod_set %=` - nastaví levý parametr na zbytek po dělení, levého parametru pravým parametrem, levý parametr musí být proměnná, vrací odkaz na proměnnou. Pracuje nad datovým typem `int`.
- `and &&` - provede logickou operaci and nad operandy, vrací hodnotu 1 v případě pravdivosti obou operandů, v opačném případě vrací 0. Pracuje nad všemi datovými typy.
- `or ||` - provede logickou operaci or nad operandy, vrací hodnotu 1 pokud je alespoň jeden operand pravdivý jinak vrací 0. Pracuje nad všemi datovými typy.

- `equal ==` - Porovná hodnoty operandů, vrací 1 pokud jsou shodné, v opačném případě vrací 0. Pracuje nad všemi datovými typy.
- `not_equal !=` - Porovná hodnoty operandů, vrací 1 pokud nejsou shodné, v opačném případě vrací 0. Pracuje nad všemi datovými typy.
- `greater >` - Porovná hodnoty operandů, vrací 1 když je první operand větší než druhý, v opačném případě vrací 0. Pracuje nad datovými typy `int` a `float`.
- `less <` - Porovná hodnoty operandů, vrací 1 když je první operand menší než druhý, v opačném případě vrací 0. Pracuje nad datovými typy `int` a `float`.
- `greater_equal >=` - Porovná hodnoty operandů, vrací 1 když je první operand větší nebo stejný jako druhý operand, v opačném případě vrací 0. Pracuje nad datovými typy `int` a `float`.
- `less_equal <=` - Porovná hodnoty operandů, vrací 1 když je první operand menší nebo stejný jako druhý operand, v opačném případě vrací 0. Pracuje nad datovými typy `int` a `float`.
- `add +` - sečte hodnoty operandů, vrací součet. Pracuje nad datovými typy `int` a `float`.
- `sub -` - odečte od prvního operandu druhý operand, vrací rozdíl. Pracuje nad datovými typy `int` a `float`.
- `conc .` - provede konkatenaci řetězců popsaných operandy, vrací výsledek konkatenace. Pracuje nad datovým typem `string`.
- `mul *` - vynásobí mezi sebou operandy, vrací součin. Pracuje nad datovými typy `int` a `float`.
- `div /` - vydělí hodnotu prvního operandu hodnotou druhého operandu, vrací podíl. Pracuje nad datovými typy `int` a `float`.
- `mod %` - vydělí hodnotu prvního operandu hodnotou druhého operandu, vrací zbytek po dělení. Pracuje nad datovým typem `int`.
- `post_inc ++` - lze aplikovat jen na proměnné, zvýší hodnotu proměnné o 1, vrací původní hodnotu proměnné. Pracuje nad datovými typy `int` a `float`.
- `post_dec --` - lze aplikovat jen na proměnné, sníží hodnotu proměnné o 1, vrací původní hodnotu proměnné. Pracuje nad datovými typy `int` a `float`.
- `pre_inc ++` - lze aplikovat jen na proměnné, zvýší hodnotu proměnné o 1, vrací odkaz na proměnnou. Pracuje nad datovými typy `int` a `float`.
- `pre_dec --` - lze aplikovat jen na proměnné, sníží hodnotu proměnné o 1, vrací odkaz na proměnnou. Pracuje nad datovými typy `int` a `float`.
- `invert -` - vrací opačnou hodnotu operandu. Pracuje nad datovými typy `int` a `float`.

Tento seznam definovaných operací bude rozšiřován v závislosti na nově přidaných datových typech.

`fun`	`if`
`else`	`do`
`while`	`break`
`continue`	

Tabulka 7.1: Seznam klíčových slov.

7.3 Řízení toku programu

Řízení toku programu v navrženém jazyce podporuje pokud je to možné (viz. funkce) shodné konstrukce jako jazyk `C`. Mezi tyto patří konstrukce umožňující větvení toku programu, konstrukce umožňující vytváření cyklů a konstrukce ukončení a pokračování v cyklu. V tabulce 7.1. se nachází seznam klíčových slov, které zatím navržený jazyk akceptuje.

Vytváření bloků kódu

V kódu je možné jakoukoli posloupnost příkazů uzavřít do složených závorek a tímto způsobem ji označit jako jednotný celek, který se dále považuje za jeden jednoduchý příkaz.

```
{
   <command>
   <command>
   ...
   <command>
}
```

Tento zápis je často využíván v podmínkách, nebo cyklech, když je potřeba vykonat složitější podmíněnou operaci, která by se nedala popsat jen jedním příkazem. Také je samozřejmostí při definici funkce, nemělo by velký smysl vytvářet funkci pro jeden příkaz.

Větvení toku programu

K rozdělení toku programu v závislosti na testované podmínce se využívá zápis obsahující klíčové slova `if` a `else`. Zápis jednoduché podmínky vypadá následovně:

```
if ( <expresion> ) <command>
```

Kde zástupné slovo `<expression>` bude nahrazeno výrazem určujícím podmínku k vyhodnocení. Slovo `<command>` zastupuje příkaz určený k provedení pokud bude podmínka vyhodnocena jako pravdivá. Stejně jako v jiných jazycích je možné definovat který příkaz se má provést i při neplatnosti podmínky, tento zápis následuje:

```
if ( <expression> ) <command>
else <command>
```

Význam zástupných slov v zápisu je stejný jako v předchozím případě. Příkaz zastoupený slovem `<command>` za klíčovým slovem `else` bude vykonán v případě neplatné podmínky.

Další možnosti větvení programu bude nabízet tzv. přepínač jehož zápis bude pokud možno stejný jako v jazyce `C`. Bude využívat klíčová slova `switch`, `case`, `default` a `break`. Tento zápis bude vypadat následovně.

```
switch( <expression> ) {
case <const_exp> :
   <command>
   ...
case <const_exp> :
   <command>
   ...
   break;
...
default :
   <command>
   ...
}
```

V tomto zápisu označuje slovo `<expression>` výraz, jehož hodnota se bude testovat. Tok programu povede přes první výskyt `case` (Pokud je alespoň jeden zapsán) postupně až k poslednímu, případně k slovu `default`. Pokud nebyl žádný z předcházejících výrazů `<const_exp>` shodný s testovaným výrazem tak se testuje hodnota konstantního výrazu u následujícího zápisu `case`. Pokud je rovnost pravdivá spouští se seznam příkazů definovaných pod tímto výrazem. Pokud již byl nějaký předchozí test pravdivý provádí se všechny kódy zápisů case již bez dalších testů na rovnost. Když dorazí tok programu k zápisu `default`, ke spuštění pod ním definované množiny příkazů dojde bez testu. Takový průchod zápisem přepínače je přerušen pokud tok programu narazí na klíčové slovo `break`, které značí že program bude pokračovat na prvním příkazu za přepínačem (uvažuje se o vytvoření odlišnosti od jazyka `C` a nahrazení příkazu `break` jiným klíčovým slovem, které bude umožňovat opuštění smyčky na základě testu v přepínači použitím právě slova `break`). Průchod příkazem `switch` popsaným v ukázce jeho zápisu je zobrazen na obrázku 7.1.

Tento příkaz zatím není v popisu jazyka definován.

Zápis smyček toku programu

Jazyk v této fázi definuje základní typy cyklů, kterými jsou cyklus s podmínkou na konci (tj. nastane minimálně jedno provedení těla cyklu), a cyklus s podmínkou na začátku. Při zápisu těchto cyklů se používají klíčové slova `while` a `do`. Nasleduje Zápis cyklu s podmínkou na začátku:

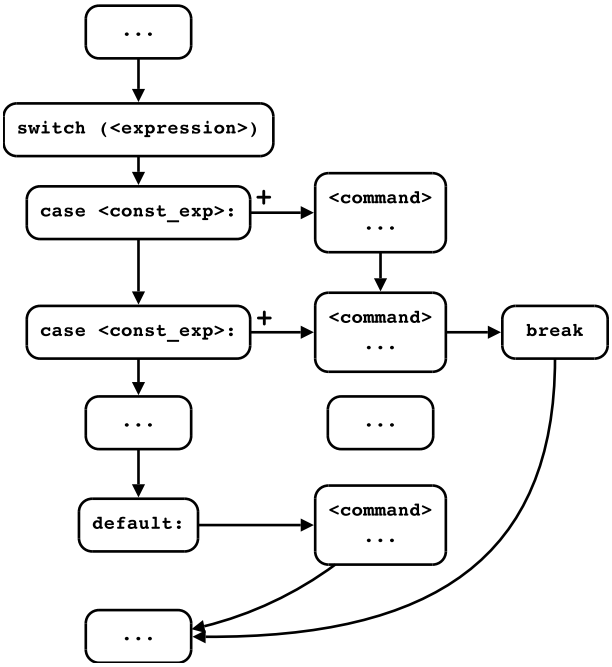
```
while ( <expression> ) <command>
```

Kde slovo `<expression>` nahrazuje výraz, jehož nesplněním cyklus končí, a slovo `<command>` popisuje podprogram prováděný jako tělo cyklu. Zápis cyklu s podmínkou na konci vypadá následovně:

```
do <command> while ( <expression> );
```

Kde mají zástupné slova `<command>` a `<expression>` stejný význam jako v předchozím typu cyklu.

Dalším cyklem, který bude do popisu jazyka přidán bude pravděpodobně oblíbený, tzv. počítaný cyklus s názvem `for`. Po přesnějším určení datových typů, implementujících kontejnery bude možné zavést i cykly `foreach` apod. známe z jazyka Perl.



Obrázek 7.1: Příkaz switch.

Každý cyklus stejně jako v jazyce `C` reaguje na klíčové slova `break` a `continue`. Při zápisu klíčového slova `break` je na jeho pozici řečeno že kód bude následovat bezprostředně za prvním cyklem ve kterém je tento zápis zanořený. Kód mezi slovem `break` a zbytkem kódu do konce cyklu bude přeskočen. Při zápisu slova `continue` bude následujícím spuštěným příkazem testovaná podmínka prvního cyklu, ve kterém je tento kód zanořen. Stejně jako v předchozím případě kód mezi klíčovým slovem a koncem cyklu nehraje další roli (Jedná se v podstatě o skok v programu).

Definování a volání funkcí

Mechanizmus definování funkcí není shodný s jazykem `C`, z toho důvodu že navrhovaný jazyk neobsahuje typované proměnné a nelze definovat typ návratové hodnoty funkce. V definici se využívá klíčové slovo `fun` a zápis definice vypadá následovně:

```
fun <name> ( <parm> , ... , <parm> ) <command>
```

V tomto zápisu je slovo `<name>` nahrazeno jménem funkce, každé slovo `<parm>` unikátním jménem parametru funkce a zástupné slovo `<command>` příkazem určeným k provedení při zavolání této funkce. Často bude příkaz funkce pomocí blokového zápisu rozšířen na blok příkazů.

Volání funkce se realizuje stejným způsobem jako v jazyce `C`, zápisem jména funkce a parametrů této funkce do kulatých závorek. Takový zápis tvoří součást výrazu v programu. Struktura zápisu následuje.

```
... <name> ( <parm> , ... , <parm> ) ...
```

S výsledkem takto zavolané funkce se dále počítá ve výrazu, ve kterém se toto volání nachází. Volání funkcí ještě není zaneseno v popisu jazyka.

7.4 Příklad programu v jazyce

V této části bude uveden přeložitelný příklad programu v jazyce. V rámci kurzu GJA byla implementována aplikace realizující vizualizaci grafu toku programu, jehož popis vypisuje debug výpis překladače na standardní výstup. Obrázek popisující graf toku programu vygenerovaného při překladu ukázkového programu bude rovněž následovat.

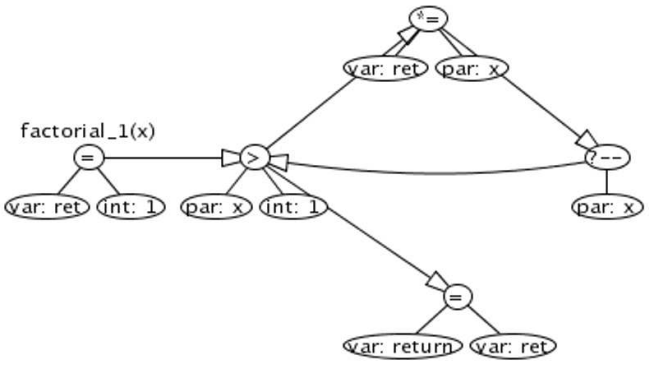
```
0   fun factorial(x)
1   {
2      ret = 1;
3      while(x > 1) {
4         ret *= x;
5         x--;
6      }
7
8      return = ret;
9   }
```

Je zřejmé že funkce realizuje výpočet faktorialu z parametru funkce x. V zápisu je místo klíčového slova `return` (které není ještě implementováno v popisu jazyka), využito přiřazení do proměnné nazvané `return`.

Odlaďovací výpis překladače při překladu tohoto kódu vypadá následovně.

```
1:1:1:0:0:1:11:factorial_1:0:1:1:x:2:3:ret:6:return:5:6:26:0:0:27:0:0:6:25:
0:0:27:0:10:7:26:0:0:25:0:0:3:4:25:0:0:21:7:26:0:1:26:0:0:0:16:0:0:3:1:1:
13:7:0:2:10:0:3:3:0:4:-1:
```

Jeho popis není příliš důležitý, Podstatné je jen že obsahuje popis konstant, funkcí s jejich parametry a proměnnými, grafů jednotlivých výrazů a grafu toku programu. Graf načtený z tohoto zápisu a zobrazený programem z kurzu GJA je zobrazen na obrázku 7.2.



Obrázek 7.2: Graf toku programu.

Kapitola 8

Závěr

Hlavním cílem této práce bylo navrhnout skriptovací jazyk určený pro zpracování obrazu a implementovat překladač, nebo interpret tohoto jazyka. Tento úkol byl splněn.

- Studiem literatury a popisem překladačů a podrobností jejich návrhu se zabývá kapitola 3. Překladače kompilátory a interprety.
- Návrhem syntaxe skriptovacího jazyka určeného pro zpracování obrazu se zabývá kapitola 7. Navržený jazyk pro zpracování obrazu. Zápis syntaxe tohoto jazyka je vytvářen jako vstupní soubor implementovaného programu.
- Návrh a popis překladače navrženého jazyka pro zpracování obrazu je popsán v kapitole 5. Navržený překladač. Další nedílná součást překladače, kterou je jazyk pro zápis sémantických podprogramů je popsána v kapitole 6. Navržený jazyk sémantických podprogramů.
- Shrnutím dosažených výsledků a popisem případných dalších možností rozvoje projektu se zabývá zbytek této kapitoly.

Popis současného stavu práce následuje. Byl plně implementován program umožňující popis překladače pomocí textového souboru. S využitím tohoto programu je možné vytvořit překladač libovolného (v jistých mezích) jazyka, popisem jeho lexému, syntaktické analýzy a zápisem sémantických podprogramů. Překladač vygenerovaný na základě tohoto souboru může výše popsaný program využít k překladu zdrojových souborů popsaného jazyka, nebo uložit jeho popis do binárního souboru, ze kterého jej může později načíst. Jediná nedokončená funkce tohoto programu je zápis vygenerovaného překladače jako zdrojového souboru jazyka `C++`, ve kterém je tento překladač implementován. Generovaný zdrojový soubor jazyka `C++` zatím implementuje lexikální a syntaktickou analýzu jazyka (tzn. při spuštění tohoto programu se na výstupu objeví indexy pravidel v pravém rozkladu vstupního řetězce).

Jako vstup výše uvedeného programu je implementován skriptovací jazyk určený pro zpracování obrazu. Tento se nachází ve fázi rozpracování. Jsou navrženy a implementovány popisy základních datových typů jazyka, zápis řízení toku programu a práce s funkcemi tohoto jazyka. V této fázi nejsou známy metody, které budou využívány pro zápis výstupu překladu programu tohoto jazyka. Jsou uvažovány následující: přímá komunikace s jiným programem, nebo programovou součástí, zápis bytového toku popisujícího instrukce k provádění, nebo výstup popisující graf toku programu umožňující snadnější optimalizace nad programem.

Další práce na projektu by měla sestávat z dokončení implementace skriptovacího jazyka ve vstupním souboru provádějícím jeho popis. Rozhodnutí výstupní metody v závislosti na projektu `RIPAC`, a její implementace. Dokončení funkce umožňující generování zdrojového souboru v jazyce `C++` popisujícího vygenerovaný překladač.

Literatura

[1] Češka M., Rábová Z., Hruška T., Máčel M.: *Překladače.* SNTL - Nakladatelství technické literatůry, 1986.

[2] Žára J., Beneš B., Sochor J., Felkl P.: *Moderní počítačová grafika.* Computer Press, 2004.

Seznam obrázků

Seznam tabulek

Příloha A

Základní popis programu

Program byl implementován jako konzolová aplikace (tzn. neobsahuje žádné grafické prostředí). Pracuje pouze jako textový procesor, který načte vstupní soubory tyto zpracuje a výsledek uloží opět do souboru, nebo zobrazí na standardní výstup.

Použité nástroje

Program je zapsán jako přenositelný mezi různými operačními systémy, z tohoto důvodu nepoužívá Žádné složitější systémové nástroje (žádné ani nepotřebuje). Program byl implementován a testován v prostředí operačního systému GNU/Linux.

Programovacím jazykem, ve kterém je program zapsán je jazyk `C++`. Některé nástroje využívané při implementaci programu jsou zapsány v jazyce `Perl`. V implementaci není použita knihovna `STL`, ani třídy jak bývá obvyklé, místo nich je využit vlastní způsob vytváření objektů obalujících data, popsaný v příloze nazvané Šablony v perlu.

Následuje seznam nástrojů použitých při zpracování programu.

- *bash* - interpret příkazů určený pro správu OS GNU/Linux.
- *vim* - textový editor, který byl použit k editování zdrojových souborů, jak v jazyce `C++` tak i dalších.
- *gcc* - GNU překladač programů zapsaných v jazyce c.
- *gdb* - GNU debugger určený pro debugovaní programů překládaných kompilátorem gcc.
- *cpp* - preprocesor jazyka C využívaný pro zjednodušení zápisu popisu překladače.
- *make* - program usnadňující překlad složitějších programů.
- *perl* - interpret skriptů zapsaných v jazyce `Perl`.
- *gnome-terminal* - virtuální terminál používaný v desktopovém prostředí GNOME.
- LaTeX - jazyk vytvořený pro zápis dokumentů, postavený nad systémem TeX.

Překlad programu se provádí obvyklým způsobem využitím programu make a předdefinovaného makefile souboru. Po spuštění programu make, dojde nejprve ke spojení všech zdrojových souboru do jednoho, následně k vygenerování struktur popisujících definované objekty a posledním krokem je přeložení výsledného souboru překladačem gcc.

Součásti programu

Projekt se skládá ze skriptů v perlu, které slouží k upravení zdrojových `.cc` souborů před překladem a samotných zdrojových souborů jazyka `C++`. V projektu jsou obsaženy následující soubory.

- skripty v perlu
  - `link_all.pl` - linkuje všechny `.cc` soubory do jednoho, na základě zápisu `%%include <filename>`.
  - `process.pl` - provádí zpracování již slinkovaného `.cc` souboru a generuje struktury objektů.
- zdrojové kódy
  - `main.cc` - hlavní soubor obsahující funkci `main()` ,do kterého se linkují ostatní zdrojové soubory.
  - `basic.cc` - soubor obsahující základní nástroje, kterými jsou například assert makra a správa paměti.
  - `struct.cc` - obsahuje definice struktur pro použití v objektech generovaných skriptem.
  - `definitions.cc` - obsahuje definice objektů určených pro roz-generování skriptem.
  - `lexical.cc` - soubor, který obsahuje funkce implementující lexikální analyzátor.
  - `parser.cc` - soubor obsahující funkce implementující samotný překladač.
  - `parse_sem_instr.cc` - obsahuje převod instrukcí sémantického překladače do instrukcí popisujících i datové typy operandů.
  - `run_sem_instr.cc` - popisuje kód spouštěný při zpracování instrukcí sémantického překladače.
  - `cc_creator.cc` - obsahuje kód, který generuje zdrojový kód vytvořeného překladače v jazyce `C++`.

Logické rozdělení programu v jazyce `C++` se skládá ze sedmi částí, jejich seznam a stručný popis následuje.

- `basic definitions` - v souboru `basic.cc`, zahrnuje vložení hlavičkových souboru knihoven jazyka `C` , seznam maker řídících překlad, definici konstant používaných v celém programu a makra assert pro kontrolu výpočtu.
- `memory control` - v souboru `basic.cc`, spravuje paměť tak, že zaznamenává informace o každém přiděleném úseku paměti a jeho velikosti. K tomuto účelu definuje funkce `cmalloc` a `cfree`. Kontrolu lze vypnout odstraněním makra `MEM_CHECK`.
- `basic structures` - soubor `struct.cc`, do této skupiny patří základní struktury definované bez použití šablon v Perlu. Jsou to struktury `string_s` a `buffer_s`, obě jsou zapsány takovým způsobem aby bylo možné jejich využití jako typů v šablonách.

- `lexical` - soubory `definitions.cc`, `lexical.cc`, obsahuje struktury a jejich metody pro konstrukci konečných automatů za účelem rozpoznávání elementů v souvislém textu.
- `parser generator` - soubory `definitions.cc`, `parser.cc`, obsahuje struktury a metody generující překladač na základě popisu překládaného jazyka.
- `semantic parser` - soubory `definitions.cc`, `parser.cc`, `parse_sem_instr.cc`, obsahuje struktury a jejich metody, reprezentující překladač sémantických podprogramů. Výsledky jeho překladu jsou dále použity v cílovém překladači.
- `parser` - soubory `definitions.cc`, `parser.cc`, `run_sem_instr.cc`, `cc_creator.cc`, definuje struktury a metody reprezentující překladač vytvořený ze souboru popisujícího tento překladač.

Příloha B

Šablony v perlu

Z důvodu přehlednosti a potřeby alespoň částečné ohebnosti vytvářeného kódu byl zvolen objektově orientovaný přístup k tvorbě programu. Předchozí verze programu byla implementována s využitím šablon, tak jak je popisuje jazyk `C++`. To vedlo k neefektivnímu kódu, což bylo možná zapříčiněno i mou neznalostí podrobností těchto nástrojů. Při požadavku na vetší kontrolu nad fungováním kódu, jsou tyto šablony nevhodné, proto byl zvolen jiný přístup k zápisu kontejnerů a podobných struktur. Mezi tyto požadavky patřil třeba požadavek na generování rozdílného kódu v závislosti na datovém typu obaleném strukturou.

Proto před prací na samotném překladači jsem vytvořil skript v jazyce Perl, který je určen pro zpracování právě takových zápisů objektových kontainerů, ve kterých se mnohé jejich metody opakují často jen s drobnými změnami. Tento skript přepisuje vstupní soubor na standardní výstup s tím že pokaždé když narazí na klíčové slovo (např. `%%define`, `%type`) zapíše na výstup místo těchto slov definici a funkce nového objektu.

Samotný skript není jen snůškou maker, které se hloupě přepisují jako náhrada klíčových slov, ale spíše program, který si každý vytvořený objekt zapamatuje (včetně typu a vlastností) a tento je možné použít v dalších definicích.

Vytvářené objekty

Vytvářené objekty je možné popisovat z pohledu skriptu a pohledu vytvářeného programu. Skriptu stačí když si uchovává informace potřebné pro další použití objektu v nových objektech. Program (programátor) by měl vědět jak s vygenerovaným objektem zacházet[1]

Objekt ve skriptu

Objekt není ve skriptu definován jako samostatný element, ale je spíše popsán svými vlastnostmi uloženými v množinách těchto vlastností pro všechny elementy. Při potřebě informací o určitém typu objektu se vyhledají v těchto množinách.

Základními třemi množinami objektů jsou.

- `@basic_types` - obsahujici objekty, které jsou jedním ze základních typů (např. `char`, `unsigned int` atd.).

[1] Obvykle funkce typu `push`, `swap` apod.

- `@no_dynamic_types` - obsahující vytvořené objekty, které ale nejsou dynamické (tzn. nealokují dynamickou paměť).
- `@dynamic_types` - popisující vytvořené dynamické objekty, kterými jsou například řada, zásobník apod.

Dalšími množinami popisujícími vlastnosti objektů jsou.

- `@flushable_types` - u všech objektů v této množině je možné omezit přidělenou paměť na nezbytné množství potřebné pro objekt.
- `@abbreviated_types` - množina objektů, které mají definované nějaké náhradní jméno (zkratku),určenou pro použití v cílovém programu.

Objekt je ve skriptu označen řetězcem zadávaným v definici nového objektu, z toho důvodu, aby nedocházelo k rozdílům mezi objektem `unsigned` a `unsigned int`, je potřeba před použitím jména objektu převézt toto jméno na jednotnou reprezentaci. Tento převod zajišťuje ve skriptu funkce jménem `edit_type`, která upraví jména základních proměnných a také přeloží náhradní jména vytvořených objektů na jejich základní zápis.

Další důležitou funkcí ve skriptu je funkce `get_type_type`, která z výše popsaných množin objektů vlastnosti vygeneruje celočíselnou reprezentaci typu předaného objektu. Vygenerované celé číslo v sobě obsahuje všechny informace o objektu v bitové reprezentaci. Předaný objekt musí být před zpracováním touto funkcí již převeden funkcí `edit_type`.

Poslední základní funkcí pro práci nad typem je funkce `get_type_name`, která slouží převodu jména typu na řetězec použitý v zdrojovém kódu výstupního programu. Funkce operuje pouze nad typy `basic` a převádí jejich jména na zkrácenou reprezentaci (např. `unsigned` na `ui`), z důvodu čitelnosti generovaného zdrojového kódu[2].

Objekt v programu

Objekt v programu je reprezentován vygenerovanou strukturou, obsahující funkce, které nad ní operují.

Z pohledu jazyka `C++` jde v podstatě o třídu a její metody. Vygenerované struktury však neobsahují žádné konstruktory, nebo destruktory ale obsahuji své vlastní API. Je požadován povinný výčet funkcí, které musí obsahovat každá struktura, která má být použita v nějakém dalším objektu.

Následuje výčet těchto základních funkcí s jejich stručným popisem.

- `void $type::init()` - připraví strukturu pro použití, na-rozdíl od konstruktoru nevytváří žádné dynamické data, spíše je nastaví na hodnotu `NULL` a inicializuje počítadla na nulovou hodnotu. Tato funkce se volá pouze jednou a to hned po vytvoření objektu (tj. jeho proměnných na zásobníku nebo alokování instance v paměti). Při vytváření instance objektu programátorem záleží její zavolání zase na programátorovi.
- `void $type::clear()` - slouží k uvolnění dynamických dat struktury z paměti. Po zavolání této funkce se bude struktura nacházet ve stejném stavu jako po zavolaní funkce `init()`. Opět pokud se nejedná o objekt spravovaný jiným objektem je potřeba tuto funkci zavolat ručně.

[2] Jména nových objektů se vytvářejí na základě jmen objektů, které obalují.

- `$type &$type::operator=($type &src)` - zkopíruje hodnoty objektu stejného typu popsaného proměnnou `src` do tohoto typu a zároveň vrátí referenci na tento objekt.
- `bool $type::operator==($type &second)` - porovná hodnoty této struktury a struktury popsané proměnnou `second` a vrátí hodnotu `true`, nebo `false` na základě toho jestli jsou shodné, nebo ne.
- `void $type::flush_all()` - provede zavolání této funkce u všech proměnných obalených strukturou, které jsou typu obsaženého v množině @flushable_types a poté omezí velikost svých dynamických struktur na minimální potřebnou.
- `void $type::swap($type &second)` - zamění data této struktury a struktury popsané proměnnou `second`, na úrovni prohození všech jejich proměnných (tj. prohozeni počítadel, ukazatelů a základních datových typů).
- `unsigned $type::save_size()` - vrátí velikost potřebné paměti pro uložení "obrazu" struktury.
- `void $type::save_to(char **b_ptr)` - uloží obsah struktury do bufferu znaků, a posune ukazatel na znaky (předaný parametrem) o počet zapsaných znaků (počet zapsaných znaků musí souhlasit s velikostí vrácenou funkcí `save_size()`, v případě že se obsah struktury nezměnil).
- `void $type::load_from(char **b_ptr)` - načte obsah struktury z bufferu znaků a posune ukazatel na znaky o přečtený počet znaků. Původní obsah struktury je uvolněn (většinou volání funkce `clear()`).
- `void $type::print()` - vytiskne obsah struktury na standardní výstup, v čitelném formátu.

Definice objektu

Zápis definice objektu se může nacházet na libovolném místě ve zdrojovém souboru. Je potřeba si však uvědomit že bude nahrazen zápisem struktury, za níž budou následovat její metody. Struktůra zápisu definice objektu má následující tvar.

```
%%define <name>
%<param> "<value>"
...
%%end
```

Následuje vysvětlení symbolů použitých v popisu struktury definice objektu.

- `<name>` - popisuje jméno struktury použité k vytvoření nového objektu (z pohledu skriptu jméno určuje, která funkce se bude volat pro vygenerování kódu) může nabývat jednu z hodnot `double_array`, `hash_array`, `array`, `queue`, `map` nebo `set`.
- `<param>` - definuje typ parametru předávaného funkci generující nový objekt. Tímto zápisem je možné jak předávat jméno objektu určených pro obalení objektem, tak například definovat náhradní jména pro vytvářenou strukturu. Parametr může nabývat následujících hodnot `name`, `type`, `variable`, `function` a `fundef`.

- `<value>` - nahrazuje řetězec který obsahuje hodnotu parametru.

Takováto definice objektu bude nahrazena kódem popisující strukturu objektu a funkce operující s tímto kódem. Následuje popis jednotlivých struktur generujících kód.

- `double_array` - struktura popisující dvourozměrné nedynamické pole. Potřebuje jeden parametr typu `type`.
- `hash_array` - popisuje dynamické pole, implementované tak, aby umožnilo rychlé vkládání prvků na prázdné místa (Tzn. udržuje záznamy posloupností vložených a prázdných polí). Požaduje jeden parametr typu `type`.
- `array` - popisuje dynamické pole, obsahující operace ekvivalentní zásobníku. Požaduje jeden parametr typu `type`.
- `queue` - struktura popisující řadu, s obvyklými operacemi jako jsou `insert` a `next`. Potřebuje jeden parametr typu `type`.
- `map` - struktura popisující objekt složený ze dvou objektu, ve kterém se porovnání provádí jen podle prvního objektu. Požaduje dva parametry typu `type`.
- `set` - popisuje objekt tvořící množinu objektů, které zahrnuje. Přijímá jeden nebo více parametrů typu `type`.

V seznamu struktur generujících kód jsou popsány jen argumenty, které jsou nutné k vytvoření kódu objektu. Následuje úplný seznam jednotlivých typů argumentů s popisem.

- `name` - definuje nové jméno objektu, které je možné použít při definici dalších objektů. Těchto jmen je možné definovat neomezený počet. Parametr není povinný protože objekt je možné identifikovat jeho jedinečným jménem složeným ze jmen typů které obaluje. Jedinečné jméno objektu se také zapíše při definici struktury ve výsledném zdrojovém kódu. Definované jména se v kódu jazyka `C` realizují pomocí klíčového slova `typedef`.
- `type` - definuje datový typ, který bude vytvářený objekt obalovat. Mohou zde být použity základní datové typy, nebo již vytvořené objekty. Počet požadovaných parametrů typu `type` je určen typem struktury generující kód objektu.
- `variable` - popisuje jména proměnných při generování kódu strukturami, které přijímají proměnný počet parametrů typu `type`. Počet parametru `variable` by měl být minimálně stejný jako počet parametrů typu `type`.
- `function` - popisuje definici funkce, která se vloží do zápisu struktury. Vloží se jako obyčejný řetězec nekontroluje se správnost zápisu apod. Tělo funkce se musí definovat dále v kódu, kdekoli pod definicí objektu.
- `fundef` - označuje parametr určený k výběru dalších funkcí pro roz-generování u vytvářené struktury. Do těchto patří zatím jen funkce `save_file` a funkce `load_file`, které umožňují ukládání struktury do souboru a její načtení z takto vytvořeného souboru.

Na takto vygenerovaných strukturách (třídách) je založená celá implementace překladače. Definice všech objektů překladače se nacházejí v souboru `definitions.cc`. Základními objekty v tomto souboru jsou `final_automata_s`, `p_create_descr_s`, `semantic_parser_s` a `parser_s`.

Příloha C

Popis implementace programu

Obsah této přílohy se skládá z výčtu logických částí programu, kde u každé takové části je uveden stručný popis základních tříd, jejich dat a funkcí určených pro práci s nimi.

Kontrola paměti

V této části bude popsán systém provádějící kontrolu uvolňování a velikosti přidělené paměti za běhu programu. Jedná se v podstatě o vytváření záznamu o každém přiděleném úseku paměti, jeho pozici a velikosti.

Pro alokaci paměti, o které má být vytvořen záznam se používá funkce jménem `cmalloc` (Check malloc), ta zajistí zavolání globálního objektu spravujícího záznam přidělených bloků. Stejně jako funkce `malloc` přijímá jako argument požadovanou velikost paměti v bytech a vrací ukazatel na přidělenou paměť.

Pro uvolnění takto alokované paměti je určena funkce `cfree` (Check free), která stejně jako předchozí funkce pomocí globálního objektu uvolní záznam o bloku přidělené paměti. Stejně jako funkce `free` požaduje jako argument ukazatel na přidělenou paměť.

Před použitím těchto funkcí je nutné kontrolu paměti inicializovat zavoláním `mc_init()`. Před koncem programu je vhodné zavolat funkci `mc_clear()`, která před samotným uvolněním objektu kontrolujícího paměť provede průchod přes seznam přidělených bloků paměti a pokud tento není prázdný zobrazí zprávu na chybový výstup.

Celý systém kontroly paměti lze z programu odstranit ve fázi překladu zakomentováním, nebo odstraněním makra s názvem `MEM_CHECK`. Po jeho odstranění se budou funkce `cmalloc` a `cfree`, při překladu nahrazeny funkcemi `malloc` a `free`, a funkce `mc_init()` a `mc_clear` prázdným kódem.

Základní struktury

Zde se nachází popis struktur, k jejichž definici se nevyužívá systém generující kód skriptem v jazyce Perl. Tyto struktury se nazývají základní, protože jsou určeny k definování složitějších objektu s využitím výše zmíněného skriptu. Do množiny těchto struktur patří struktury `string_s` a `buffer_s`. Následuje detailnější popis těchto struktur.

Struktura `string_s`

Tato struktura popisuje řetězec znaků, zakončený terminálním znakem `'\0'`. Struktura umožňuje základní operace pro práci s řetězci, kterými jsou například: přiřazení, porovnání, konkatenace, formátování apod. Definice této struktury se nachází v souboru `struct.cc` na řádku `12`.

Seznam proměnných struktury následuje.

- `unsigned size` - obsahuje aktualní velikost retězce.
- `char *data` - obsahuje ukazatel na paměť popisující obsah řetězce.

Následuje seznam metod aplikovatelných na tuto strukturu, jsou zde popsány jen ty metody, které jsou odlišné od standardních metod objektu (viz. popis objektu).

- `set` - nastaví řetězec podle předaných parametrů popisujících délku řetězce a ukazatel na znaky.
- `set_format` - nastaví řetězec podle předaných parametrů, a provede formátování podle escape sekvencí tak jak se používají v jazyce `C`.
- `set_format_parm` - nastaví řetězec podle předaných parametrů, a provede formátování podle escape sekvencí popsaných třetím a čtvrtým argumentem funkce.
- `conc_set` - nastaví obsah řetězce, jako výsledek konkatenace dvou řetězců předaných parametry.
- `compare_char_ptr` - porovná řetězec s řetězcem popsaným předanými argumenty a vrátí výsledek porovnání.
- `load_text_file` - nastaví řetězec podle obsahu textového souboru předaného argumentem.
- `save_text_file` - uloží obsah řetězce do textového souboru, jehož jméno popisuje argument.

Struktura `buffer_s`

Tato struktura popisuje posloupnost znaků, určenou pro reprezentaci kódů, textových souborů apod. Obsahuje funkce pro vkládání a načítání různých základních datových typů, řetězců a typů popisujících jednotlivé sémantické instrukce (`unsigned short`). Tyto funkce vkládají prvky na konec posloupnosti, nebo je načítají ze zadané pozice s posunutím ukazatele do kódu. Struktura se používá k popisu přeloženého kódu sémantických podprogramů. Definice struktury `buffer_s` se nachází v souboru `struct.cc` na řádku `297`.

Následuje seznam proměnných této struktury.

- `unsigned size` - popisuje velikost alokované paměti v bytech.
- `unsigned used` - obsahuje počet bytu využitých v alokované paměti.
- `char *data` - ukazatel na alokovanou paměť obsahující data.

Následuje seznam metod této struktury, obsahující pouze metody, které nejsou obsaženy v popisu základních metod objektu.

- `push_sem_instruction` - vloží do posloupnosti identifikátor sémantické instrukce, popsaný argumentem.
- `get_sem_instruction` - získá z posloupnosti na pozici popsané argumentem identifikátor sémantické instrukce.
- `push_string` - vloží do posloupnosti řetězec, popsaný předanými parametry.
- `push_format_string` - vloží do posloupnosti řetězec popsaný předanými argumenty, a zároveň v něm zformátuje escape sekvence tak jak se používají v jazyce `C`.
- `push_char` - vloží do posloupnosti znak popsaný argumentem.
- `get_char` - získá z posloupnosti znak na pozici popsané argumentem.
- `push_short` - vloží do posloupnosti `short int` popsaný argumentem.
- `get_short` - získá z posloupnosti `short int` nacházející se na pozici popsané argumentem.
- `push_int` - vloží do posloupnosti integer popsaný předaným argumentem.
- `get_int` - získá z posloupnosti integer nacházející se na pozici popsané argumentem.
- `push_unsigned` - vloží do posloupnosti unsigned int popsaný předaným argumentem.
- `get_unsigned` - získá z posloupnosti bezznaménkový integer nacházející se na pozici popsané předaným argumentem.
- `push_data` - vloží do posloupnosti jinou posloupnost znaků popsanou předanými argumenty.
- `push_terminated_string` - vloží do posloupnosti řetězec ukončený znakem `'\0'`, předaný parametrem.

Lexikální kód

Zahrnuje popis základních struktur, implementujících práci s regulárními výrazy a konečnými automaty. Je zde uveden výčet jen stěžejních struktur a jejich základní popis, k získání podrobného popisu programu slouží zdrojový kód.

struktura `ushort_reg_exp_s`

Tato struktura slouží k popisu regulárních výrazů a k následné manipulaci s nimi. Výraz se do této struktury načte konverzí z textové reprezentace. V takto vytvořeném výrazu je možné snadno vyhledávat s jakým znakem a na jakou pozici je možně ve výrazu přejít ze zadané výchozí pozice v tomto výrazu. Definice této struktury se nachází v souboru `definitions.cc` na řádku `98`.

Následuje seznam důležitých funkcí této struktury. Vyjma základních funkcí, které obsahují všechny objekty.

- `process_reg_exp_char` - nalezne číselnou reprezentaci jednoho následujícího znaku při načítání výrazu z textového řetězce.

- `load_ascii_reg_exp` - načte regulární výraz ze struktury popisující jeho textovou podobu.
- `load_char_ptr_reg_exp` - vytvoří regulární výraz z textového řetězce ukončeného znakem `'\0'`.
- `next_chars` - spočítá množinu znaků, se kterými je možné ze zadaného stavu přejít do nějakého následujícího stavu. Ke každému takto vypočítanému znaku funkce generuje množinu pozic, ve kterých se bude regulární výraz nacházet po provedení přechodu s tímto znakem.

struktura `final_automata_s`

Tato struktura popisuje konečný automat určený k rozpoznávání lexému ve vstupním zdrojovém souboru programu. Používá se při čtení souboru popisujícího jazyk a při implementaci lexikální analýzy sémantického i obecného překladače. Definice této struktury se nachází v souboru `definitions.cc` na řádku `224`.

Data popsané touto strukturou mají následující význam.

- `state_idx` - aktuální stav automatu při rozpoznávání symbolu v řetězci.
- `states` - množina struktur popisujících všechny stavy konečného automatu.
- `state_moves` - seznam možných přechodů mezi jednotlivými stavy automatu.

Obsahuje funkce generující tento automat na základě vstupní množiny výše popsaných regulárních výrazů. Seznam podstatných funkcí a jejich stručný popis následuje.

- `create` - vygeneruje obsah konečného automatu na základě předané množiny výše popsaných regulárních výrazů.
- `recognize` - rozpozná jeden následující lexém v předaném řetězci na zadané pozici a vrátí jeho index. Posune index označující pozici ve vstupním řetězci čímž označí konec rozpoznaného lexému.

Generátor překladačů

Kód v této části popisuje struktury a jejich metody určené k vygenerování překladače na základě textového popisu překládaného jazyka.

struktura `p_creat_descr_s`

Základní struktura využívaná při generování překladače. Využívá se při generování překladače sémantických podprogramů a také při generování obecného překladače. Definice této struktury se nachází v souboru `definitions.cc` na řádku `360`.

Struktura obsahuje proměnné potřebné k uchování dat popisujících vytvářený překladač. Do této množiny patří.

- `final_automata_set` - množina konečných automatů uznadňující čtení souboru popisujícího strukturu jazyka přijímaného generovaným překladačem.

- `init_sem_prog` - textový řetězec načtený ze vstupního souboru popisující inicializační sémantický podprogram překladače.
- `terminals` - seznam terminálních symbolů načtených ze vstupního souboru.
- `nonterminals` - seznam neterminálních symbolů načtených ze vstupního souboru.
- `rules` - množina pravidel načtených ze vstupního souboru. Pravidla jsou popsány indexy terminálních a neterminálních symbolů uvedených výše.
- `firsts` - množina seznamů first pro každý neterminální symbol.
- `first_rules` - množina pravidel, na základě kterých byly přidány prvky do množiny firsts, pro každý neterminální symbol.
- `follows` - množina seznamů follow pro každý neterminální symbol načteny ze souboru.
- `kernels` - seznam jader rozkladu, vygenerovaných na základě dat načtených ze souboru.

Obsahuje funkce umožnující vygenerování překladače na základě popisu terminálních symbolů regulárními výrazy a popisu gramatiky jazyka seznamem přepisovacích pravidel této gramatiky. Seznam důležitých funkcí této struktury následuje.

- `load_final_automata_set` - načte množinu konečných automatů usnadňujících načítání a rozklad souboru popisujícího strukturu jazyka.
- `load_from_rule_char_ptr` - načte data ze znakového řetězce popisujícího strukturu překládaného jazyka.
- `load_from_rule_string` - provádí stejnou činnost jako předcházející funkce, s tím rozdílem že data načítá ze struktury `string_s`.
- `find_key_terminals` - nalezne v seznamu terminálních symbolů klíčové terminální symboly. Do množiny klíčových terminálních symbolu patří terminál označující konec zdrojového programu a terminály označující přeskakované úseky programu.
- `create_final_automata` - vytvoří konečný automat rozlišující terminální symboly na základě jejich popisu načteného ze souboru popisujícího jazyk.
- `compute_firsts` - spočítá množinu first všech neterminálních symbolů, obsahující seznam elementů kterými může začínat nějaký literál rozepsaný z tohoto neterminálu.
- `compute_follows_of_nonterminal` - spočítá množinu follow jednoho neterminálního symbolu popsaného předaným argumentem. Množina follow obsahuje všechny symboly, které se v nějakém literálu rozepsaném ze startovacího symbolu gramatiky nacházejí za prošetřovaným neterminálem.
- `compute_follows` - spočítá množinu follow všech neterminálních symbolů načtených ze souboru popisujícího jazyk.
- `compute_kernels` - spočítá jádra rozkladu založená na množinách firsts a follows spočítaných výše popsanými funkcemi.
- `create_lalr_table` - vygeneruje rozkladovou tabulku určenou k použití v rámci syntaktické analýzy. Tabulka se vygeneruje na základě dříve spočítaných jader rozkladu.

Překladač sémantických podprogramů

Tato část obsahuje popis struktur definujících překladač, určený k překladu sémantických podprogramů a jejich stručný popis.

struktura `p_semantic_parser_s`

Struktura `p_semantic_parser_s` popisuje překladač sémantických podprogramů, který se využívá při generování obecného překladače k vytvoření interpretovatelného kódu na základě zdrojových řetězců sémantických podprogramů. Obsahuje data umožňující provádění lexikální a sémantické analýzy nad zdrojovým programem. Definice této struktury se nachází v souboru `definitions.cc` na řádku `477`.

Data obsažená v této struktuře popisují následující prvky.

- `end_terminal` - index terminálního symbolu označujícího konec zdrojového souboru.
- `skip_terminals` - množina symbolů stejná jako v předchozím případě, označujících ignorované terminální symboly.
- `sem_rule_descrs` - seznam prvků popisujících pravidla rozkladové gramatiky. Záznam obsahuje index levé strany pravidla, velikost pravé strany pravidla a identifikátor sémantického kódu určeného ke spuštění při redukci podle tohoto pravidla.
- `final_automata` - popisuje konečný automat rozpoznávající terminální symboly ve vstupním zdrojovém kódu. Tvoří základ lexikální analýzy překladače.
- `lalr_table` - rozkladová tabulka na které je založena syntaktická analýza překladače.

Struktura definuje základní funkce pro vygenerování sémantického překladače, jeho uložení do souboru a funkci pro překlad řetězce obsahujícího zdrojový program. Seznam důležitých funkcí struktury a jejich stručný popis následuje.

- `load_or_create` - vygeneruje překladač sémantických podprogramu z řetězce popisujícího jeho jazyk, nebo načte již vygenerovaný popis překladače ze souboru. (záleží na aktuálním nastavení maker při překladu programu.).
- `parse_sem_string` - přeloží zdrojový kód reprezentovaný řetězcem znaků. Výsledkem překladu je bytový kód spouštěný v rámci překladu zdrojového programu navrhovaného jazyka.
- `save_file` - uloží popis vygenerovaného překladače sémantických podprogramů do binárního souboru, ze kterého je možné tento překladač později načíst.
- `load_file` - načte dříve uložený popis překladače sémantických podprogramu z binárního souboru.

Překladač jazyka

V této části se nachází popis struktur definujících vygenerovaný překladač libovolného jazyka.

struktura `parser_s`

Tato struktura popisuje vygenerovaný překladač určený k překladu libovolného jazyka. Definice této struktury se nachází v souboru `definitions.cc` na řádku `634`.

Data popsaná strukturou zahrnují konečný automat rozpoznávající lexikální symboly ve zdrojovém programu, rozkladovou tabulku využívanou v rámci syntaktické analýzy a množinu přeložených sémantických podprogramů prováděných při redukci podle pravidel, ke kterým jsou přiřazeny. Následuje seznam datových prvků obsažených v této struktuře.

- `terminal_cnt` - počet terminálních symbolů gramatiky jazyka.
- `end_terminal` - index terminálního symbolu označujícího konec zdrojového souboru.
- `skip_terminals` - množina indexů terminálních symbolů, které se ve zdrojovém kódu ignorují.
- `init_sem_code` - přeložený sémantický podprogram určený ke spuštění na začátku překladu zdrojového programu.
- `rule_descrs` - popis rozkladového pravidla gramatiky sestávající z indexu levé strany pravidla, velikosti pravé strany pravidla a přeloženého sémantického podprogramu přiřazeného k tomuto pravidlu.
- `sem_variable_cnts` - seznam počtu proměnných ruzných datových typů používaných při spouštění sémantických podprogramů jazyka.
- `final_automata` - konečný automat určený k použití v rámci lexikální analýzy překladače.
- `lalr_table` - stejně jako u překladače sémantických podprogramů, slouží rozkladová tabulka jako řídící prvek syntaktické analýzy překladače.
- `string_constants` - seznam konstantních řetězců. Tyto a všechny níže popsané konstanty se používají v sémantických podprogramech překladače a vytvářejí se při překladu těchto programů.
- `int_array_constants` - seznam konstantních polí celých čísel.
- `string_array_constants` - seznam konstantních polí řetězců.
- `int_array_array_constants` - seznam konstantních polí obsahujících pole celých čísel.
- `string_array_array_constants` - seznam konstantních polí obsahujících pole řetězců.

Struktura popisuje funkce pro vygenerování překladače na základě souboru popisujícího překládaný jazyk, provedení překladu zdrojového programu, uložení popisu překladače do souboru a vygenerování zdrojového kódu v jazyce `C` implementujícího tento překladač. Popis podstatných funkcí této struktury následuje.

- `create_from_rule_string` - vytvoří překladač jazyka na základě jeho popisu v textovém souboru.
- `run_semantic_code` - provede sémantický kód identifikovaný v předané struktuře.

- `parse_source_string` - přeloží zdrojový kód popsaný, řetězcem předaný parametrem funkce.
- `create_cc_source` - vygeneruje zdrojový soubor zapsaný v jazyce `C`, implementující funkce tohoto překladače.