VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

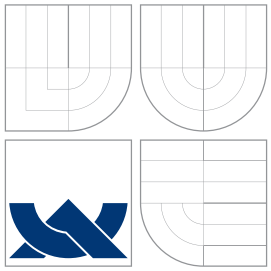
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘEKLAD DO RŮZNÝCH ASEMBLERŮ

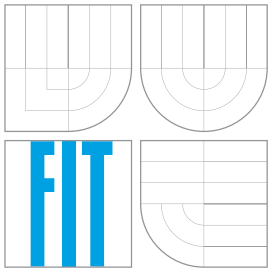
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JAN HRANÁČ
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘEKLAD DO RŮZNÝCH ASEMBLERŮ

COMPILATION INTO VARIOUS ASSEMBLERS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN HRANÁČ
AUTHOR

VEDOUCÍ PRÁCE — Prof. RNDr. ALEXANDER MEDUNA, CSc.
SUPERVISOR

BRNO 2008

Abstrakt

Cílem tohoto projektu je vytvořit překladač schopný přeložit vstupní jazyk do více různých asemblerů, dle volby uživatele. Toho bude dosaženo rozšiřitelností o moduly implementujících výstavbu zdrojových souborů konkrétních typů asemblerů.
Překladač bude sloužit jako generátor částí asemblerovských zdrojových souborů pro usnadnění práce programátora v asembleru.
Vstupní jazyk je odvozen od Pascalu. Má ale blíže k asembleru, než běžný Pascal.

Klíčová slova

překladač, asembler

Abstract

The goal of this project is to create a compiler capable of compilation of the input language into various assemblers (by the choice of the user). This will be achieved by expandibility of the compilator by modules implementing the building of the source files of the concrete types of assembler.
The compilator will serve as a generator of parts of assembler source codes to make the work of assembler programmer easier.
The input language is derived from Pascal but is closer to assembler then canonical Pascal.

Keywords

compiler, assembler

Citace

Jan Hranáč: Překlad do různých asemblerů, bakalářská práce, Brno, FIT VUT v Brně, 2008

Překlad do různých asemblerů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Prof. Alexandra Meduny.

.....
Jan Hranáč
3. května 2008

Poděkování

Děkuji Prof. RNDr. Alexanderu Medunovi, CSc. za vypsání této práce a vymyšlení jejího ústředního tématu, jakožto i za jeho podporu a trpělivost.

Obsah

Kapitola 1

Úvod

Tento dokument popisuje projekt zabývající se překladem do více assemblerů. To znamená, že vytvořený překladač načte a zkompiluje vstupní zdrojový soubor a poté vygeneruje zdrojový soubor v uživatelem zvoleném assembleru. Tento překladač může sloužit jako utilita pro assemblerovského programátora, který si bude moci z předpisu v modulárním jazyce vygenerovat nekritické části programu. Poněvadž je překladač teoreticky schopný generovat jakýkoliv assembler (je rozšiřitelný o modul implementující generování jakéhokoliv assembleru), není použití této pomůcky limitováno jen na uživatele GASu (jako je tomu v případě překladače *gcc*) nebo např. jen NASMu.

Následující kapitoly se zabývají bližším popisem projektu a práce na něm. Konkrétně se jedná o specifikaci cílů tohoto projektu (kapitola 2) a popis etap projektu a současného stavu projektu (kapitola 3). Kapitola 4 se zabývá základními rysy jazyka a některými jeho speciálními odlišnostmi. Nepopisuje však detailně syntax a sémantiku jazyka, ta je vysvětlena v přílohách. Kapitola 5 se zabývá hlavním tématem této práce - generováním více typů asemblerů. Je vysvětlena jen základní myšlenka, detailní popis je až v přílohách. Kapitola 6 se stručně zmiňuje o podobě výstupních souborů a pravidlech, které dodržují. Tyto věci jsou sice velkou měrou v rukou modulů generujících asemblery, avšak pro úplnost byly uvedeny.

Přílohy obsahují detailní popis projektu. Jedná se o detailní OO návrh v příloze A, která navazuje na kapitolu 5. Příloha B navazuje na kapitolu 4 a obsahuje detailní popis jazyka. Obsahuje i schémata konečných automatů, avšak nezabývá se analýzou výrazů. Těmi se zabývá samostatná příloha D, kde je uvedena gramatika pro výrazy, LR tabulka a její výpočet. Příloha C navazuje na kapitolu 6 a detailněji se zabývá generovanými assemblerovskými soubory, jakožto i technikami v nich použitými. Příloha E obsahuje návod k použití překladače. Seznam příloh a obsah CD přiloženého ke zprávě je na konci zprávy.

1.1 Návaznost na semestrální projekt

V rámci semestrálního projektu byl vytvořen jazyk pro vstupní soubory překladače a byl navržen a naimplementován tento překladač. Schopnost generovat různé typy assemblerů byla demonstrována na assemblerech NASM a GAS. V rámci této práce je zapotřebí vytvořený překladač testovat pro různé vstupní soubory a případně rozšířit vstupní jazyk o další možnosti, které by programátorovi ulehčovaly práci nebo rozšiřovaly jeho možnosti.

Kapitola 2

Cíle projektu

2.1 Generování různých assemblerů

Aby byl překladač použitelný pro uživatele všech typů assemblerů, musí být schopen generovat teoreticky jakýkoliv typ assembleru. V procesu překladu tedy bude figurovat jakýsi obecný assembler. Tím není myšlen nový typ textového zápisu assembleru, ale taková interní reprezentace, ze které je možno vytvořit výstupní soubor v assembleru užívajícího jakékoliv direktivy. Překladač také musí být snadno rozšiřitelný o nové konkrétní assemblery. Tím se zabývá kapitola 5.

2.2 Vazba jazyka na assembler

Poněvadž překladač má sloužit především jako utilita pro assemblerovské programátory, musí mít programátor už v okamžiku psaní předpisu pro program v modulárním jazyce moc nad tím, co bude vygenerováno. Kvůli tomu musí mít jazyk překladačem přijímaný větší vazbu na assembler než běžné modulární jazyky. Nástin tohoto jazyka je v kapitole 4.

2.3 Přehlednost výstupních souborů

Uživatel pravděpodobně nebude užívat tento program ke generování celých modulů, ale pouze jeho nekritických a rutinních částí (úvodní a ukončovací rutiny funkcí, volání funkcí a přerušení, obsluha I/O operací, řídicí konstrukce). Části kódu, které např. pomocí MMX či SSE jednotky budou zpracovávat data (šifrovací algoritmy, audiovizuální kodeky, kritické části 3D enginů, atd.), si asi bude chtít napsat sám. To však klade požadavek na to, aby byly vytvořené soubory editovatelné. Tímto se zabývá kapitola 6.1.

2.4 Shrnutí

Výše uvedené prvky jsou tedy hlavním přínosem této práce, neboť se jedná o věci, kterými se tento překladač liší od běžných překladačů, jako je např. *gcc*, nebo *fpc*, které jsou schopny generovat jen jeden assembler (např. GAS), programátor je od generovaného kódu téměř odříznut a generování assemblerovských souborů slouží spíše pro kontrolu, než jako základ pro další práci (neboť se nedají dost dobře upravovat a rozšiřovat).

Kapitola 3

Etapy projektu, současný stav

3.1 Etapy projektu

Etapy 3.1.1, 3.1.3, 3.1.4 a 3.1.5 jsou součástí semestrální části projektu.

3.1.1 Seznámení se s různými Assemblery

Tato fáze byla předpokladem pro zdárné zpracování projektu. Jednalo se pouze o assemblery NASM (vyučován v předmětech IAS a PAS) a GAS (používán překladačem *gcc*). Nebylo považováno za efektivní se učit větší množství různých assemblerů (FASM, TASM).

3.1.2 Formulace cílů

Vytvořit jazyk s velkou vazbou na assembler (sekce 2.2) a překladač s přehlednými výstupními soubory (sekce 2.3) bylo cílem již od počátku. Původním záměrem však bylo vytvořit generátor pouze NASMu. Nejdůležitější cíl projektu (sekce 2.1) byl zadán až vedoucím práce. Tato etapa proběhla v zimním semestru školního roku 2006/07.

3.1.3 Přibližný návrh jazyka a překladače, volba přístupu

V této fázi byl zhruba definován jazyk přijímaný překladačem (na úrovni nejvyšších neterminálů EBNF) a vytvořen přibližný návrh překladače (na úrovni návrhových vzorů). Byl zvolen objektově orientovaný přístup. Tato etapa proběhla v letním semestru školního roku 2006/07.

3.1.4 Detailní definice jazyka a návrh překladače

V této fázi byla dokončena definice syntaxe a sémantiky vstupního jazyka. V UML byl navržen překladač. Byla vytvořena schemata konečných automatů zpracovávajících regulární části jazyka. Byla nadefinována gramatika pro zpracování výrazů a vypočtena SLR tabulka pro jejich LR analyzátor. Tato etapa proběhla po letním semestru školního roku 2006/07.

3.1.5 Implementace překladače

Podle vytvořeného návrhu byl překladač naimplementován v jazyce C++. Byly naimplementovány dva generátory konkrétních assemblerů: NASMu a GASu. Překladač byl předběžně testován. Tato etapa proběhla před zimním semestrem školního roku 2007/08.

3.1.6 Testování, ladění a rozšiřování

Jazyk byl rozšířen o některé doplňující funkce, které nebyly dříve stiženy. Program byl dále testován a byly odstraňovány chyby (převážně v generátorech konkrétních assemblerů). Tato etapa proběhla mezi zimním a letním semestrem školního roku 2007/08.

3.2 Současný stav

V současné době je již překladač hotový (včetně později přidávaných rozšíření). Jazyk umožňuje vytvářet jednoduché algoritmy i pracovat s pamětí. Zatím je však podporována jen znaménková celočíselná aritmetika a operátorová sada je omezena. Překladač také poskytuje podporu pro editování vygenerovaných souborů (pomocí komentářů a maker).

Provedená rozšíření však přivedla projekt na hranici rozšiřitelnosti a před dalšími budou nutné rozsáhlé změny v kódu programu.

Kapitola 4

Hrubý popis jazyka

Vstupní jazyk je odvozen od jazyka Pascal a některé jeho prvky jsou dokonce shodné (řídicí struktury, příkaz přiřazení...). Mnoho aspektů jazyka ale bylo upraveno pro zajištění větší vazby na assembler. Vygenerovaný kód se navíc řídí jinými standardy, než Pascal (tímto se podrobněji zabývá sekce 6.2). Následující text upozorní na několik vybraných odlišností od Pascalu. Celkově by se dalo říci, že na rozdíl od Pascalu tento jazyk dělá za uživatele mnohem méně práce. Podrobnější popis jazyka je v příloze B.

4.1 Základní struktura

Na nejvyšší úrovni se zdrojový soubor skládá z hlavičky modulu a definic funkcí.

Hlavička modulu obsahuje deklarace a definice symbolů v modulu obsažených - funkce, globální proměnné a konstanty. Pomocí klíčových slov převzatých z assembleru je možno symboly exportovat a importovat.

Definice funkce se skládá z hlavičky definující pseudosymboly funkce (lokální proměnné a konstanty) a těla. Hlavička funkce je podobná hlavičce modulu. Tělo funkce má podobnou strukturu jako v Pascalu.

4.2 Datové typy

Datové typy vytvořeného překladače se liší od datových typů Pascalu. Označují pouze velikost vyhrazeného datového prostoru, což je zatím jen slabika, slovo a dvojslovo. Nedefinují způsob práce s těmito daty - tedy zde nejsou znaménkové a neznaménkové proměnné, ale pouze operace. V současné době jsou všechny operace pouze celočíselné se znaménkem, rozšíření o neznaménkové operace je plánováno.

Pole a řetězce jsou také řešeny jinak, než v Pascalu.

Důvodem pro neposkytnutí možnosti pracovat se čtyřslovy bylo to, že jazyk není určen pro práci s daty. Na vytvoření nekritických řídicích struktur a rutin postačují slabikové a slovní proměnné. Dále se dá předpokládat, že uživatel si bude chtít připravit i nějaké adresy, a proto byla přidána i možnost pracovat s dvojslovy.

Také stojí za zmínku, že pro jazyk je integer a ukazatel naprosto stejný datový typ (způsob užívání adres je popsán v příloze B).

4.3 Přerušení

Jazyk umožňuje volat programová přerušení (instrukce `INT`). Uživatel však nemůže určit vektor přerušení, ten je napevno definován v generátoru konkrétního assembleru a měl by ukazovat na obslužné rutiny operačního systému (např. 80h v Linuxu). Volání je provedeno speciálním operátorem. Parametry volání přerušení se zadávají do závorek stejně jako u volání funkce a jsou nahrány do registrů procesoru.

4.4 Kontroly

Kontroly prováděné v průběhu překladu jsou minimální. Je zajištěno jen to, aby byl vygenerovaný kód dále přeložitelný assemblerem. Především pak chyba v napsaném zdrojovém kódu nesmí způsobit takovou chybu ve vygenerovaném kódu, která by způsobila nepřeložitelnost tohoto kódu assemblerem ani po jeho doplnění uživatelem.

Na druhou stranu tato volnost nejde tak daleko, aby bylo např. možné při volání funkce místo slovního parametru použít dva slabikové (což by tak jako tak kolidovalo se zarovnáním - viz 6.2).

Většina kontrol, které se provádějí před spuštěním programu, zůstává na linkeru, případně se provedou za běhu. Například pokus o zápis do parametru funkce způsobí pád programu za běhu (SIGSEGV), ačkoliv nezpůsobí chybu při žádném ze dvou překladů ani při linkování (zde vyvstává otázka, proč toto neodhalí již překladač a nepracuje s parametry jako s konstantami - toto souvisí s generováním asemblerovského kódu, viz sekce C).

4.5 Složený příkaz

V současné době není možné do těl řídicích struktur umístit jednoduchý příkaz, ale pouze složený. Tedy i kdyby uživatel chtěl např. pomocí **if** podmínit jen jeden příkaz, musí jej uzavřít mezi klíčová slova složeného příkazu.

Kapitola 5

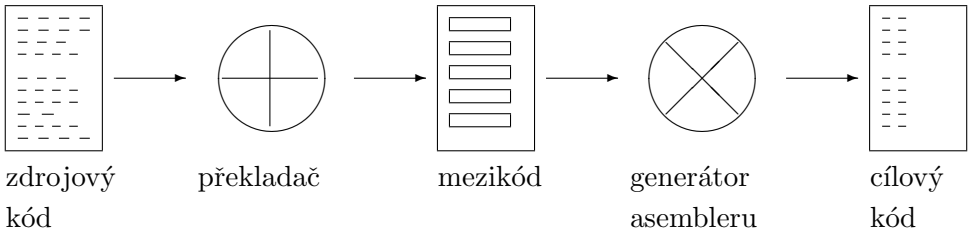
Generování různých assemblerů

Schopnost generovat různé typy assemblerů je ústředním tématem této práce. Zvládnutí tohoto aspektu projektu je předpokladem pro to, aby byl překladač použitelný pro uživatele všech typů assemblerů.

Na doporučení vedoucího práce v překladači figuruje *obecný assembler*. Tím není myšlen takový formát zdrojového assemblerovského souboru, který by byl kompatibilní se všemi existujícími překladači assembleru (což je prakticky nemožné), ale obecný generátor assemblerovského kódu, který by byl jednoduše rozšiřitelný o generátory konkrétní.

5.1 Poznámka k průběhu překladu

Stejně jako většina překladačů, ani tento neprovádí přímý překlad ze zdrojového jazyka do cílového. Ze zdrojového kódu je napřed vytvořen trojadresný kód a tabulky symbolů. Teprve poté jsou výsledky překladu předány generátoru assembleru, jak ukazuje obr. 5.1.



Obrázek 5.1: Nákres průběhu překladu.

Na tomto samozřejmě není nic netriviálního, nicméně je nutné si uvědomit, že se jedná o další prvek, který je nezávislý na zvoleném typu assembleru. Obecnost zápisu programu v mezikódu je v tomto projektu jedním ze stavebních kamenů obecnosti generátoru assembleru.

Další možností by bylo ještě před generováním assembleru převést trojadresný kód na reprezentaci více korespondující s assemblerem a teprve tu předat generátoru assembleru (což by bylo výrazné ulehčení práce implementátorů konkrétních assemblerů). To by však vedlo na omezení volnosti generátoru assembleru a mohlo by vést k omezení obecnosti a nezávislosti na konkrétním assembleru, která je primárním cílem této práce. Také by to vedlo k přidání další fáze překladu a tím pádem i prodloužení jeho trvání. Rozhodně by však tento přístup zjednodušil práci.

Ještě další možností je oba přístupy zkombinovat - tedy mezikód druhé úrovně negenerovat, ale rozšířit rozhranní *abstraktního stavitele* (viz další část).

5.2 Obecný generátor

5.2.1 Volba přístupu

Po zvážení všech v tomto případě použitelných paradigmat byl zvolen objektově orientovaný přístup.

Pro ospravedlnění tohoto rozhodnutí je zapotřebí zvážit otázku: Vyžaduje řešení použití dědičnosti a polymorfismu? Ano, je zapotřebí. Řešení generování různých assemblerů pomocí množiny podobných objektů je z hlediska doby vývoje a odolnosti proti chybám efektivnější, než např. strukturované řešení (použití přepínačů nebo ukazatelů na funkce by bylo příliš nemotorné a snadno by generovalo chyby).

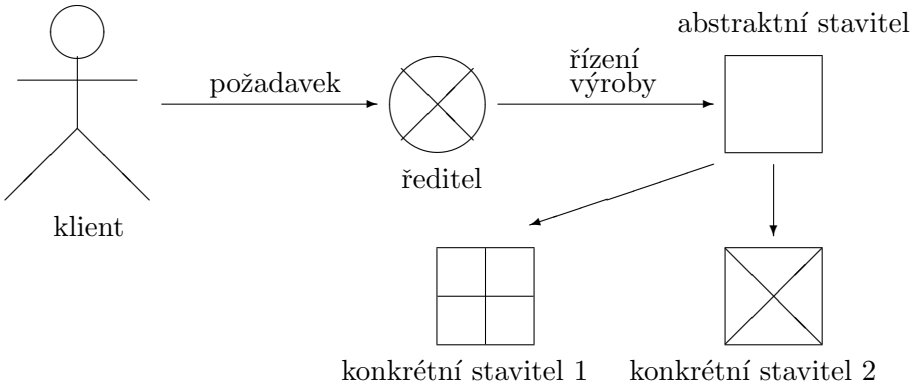
Další otázkou je požadavek na výkon. Poněvadž překladač není aplikací pracující v reálném čase, není v tomto ohledu důvod OO přístup nepoužít. Při překladu většího množství zdrojového kódu by sice modulární přístup mohl přinést kratší dobu překladu, avšak narazil by na problémy zmíněné výše.

Také je nutno zvážit odraz na dobu návrhu a implementace. Krom výhod dědičnosti a polymorfismu poskytne OO přístup i velké množství již hotových šablon jak pro návrh, tak pro implementaci, což podstatně zkracuje dobu vývoje.

5.2.2 Návrhový vzor

Návrhem celého programu se zabývá příloha A, v této části je jen popsán princip generování více assemblerů.

Při projití běžných návrhových vzorů je vidět, že v tomto případě bude nejlépe použít návrhový vzor č. 97 zvaný **stavitel**. V tomto návrhovém vzoru figuruje *klient*, který vznáší požadavek na výrobu produktu. Požadavek je přebrán *ředitelem*. Za *ředitelem* se nachází *stavitel abstraktního produktu* a množina *stavitelů konkrétních produktů*. Proces výroby produktu je po celou dobu řízen *ředitelem*. Ten k tomu využívá rozhraní *abstraktního stavitele*, avšak samotný proces výroby je proveden *konkrétním stavitelem*.



Obrázek 5.2: Schéma NV č. 97 - „Stavitel“.

V případě tohoto projektu je ředitelem objekt, který řídí celý překlad. Klientem není žádný objekt, ale strukturovaný kód samotného programu.

Abstraktním stavitelem je pak abstraktní třída obsahující čistě virtuální metody, jejichž implementace v dětských třídách této abstraktní třídy provádějí základní obecné úkony společné všem assemblerům.

Navíc, jak již bylo řečeno, tu figuruje mezikód, který je též nezávislý na konkrétním assembleru.

Začlenění tohoto návrhového vzoru je detailně ukázáno na diagramu A.1 v příloze A.

Kapitola 6

Generované soubory

Poznámky ke generátorům konkrétních assemblerů jsou v příloze C.

6.1 Editovatelnost

Vygenerovaný kód má lepší odsazování a na rozdíl od *gcc* využívá i odřádkování. Také využívá dva druhy komentářů: automaticky generované a uživatelské. Automaticky generované komentáře vkládá do vygenerovaného kódu sám generátor assembleru a slouží k přehlednějšímu rozdělení textu (na jednotlivé funkce). Uživatelské komentáře jsou kopie některých komentářů vložených do původního zdrojového souboru. Zkopírují se pouze ty komentáře, které byly uvnitř nějaké funkce a současně vně nějakého příkazu (pokud uživatel napíše komentář např. na začátek souboru, nezkopíruje se; pokud napíše začátek příkazu přiřazení, komentář a pak zbytek příkazu přiřazení, nezkopíruje se).

Další pomocí programátorovi je generování **maker**. Jedná se o makra lokálních symbolů, která udávají jejich offset vzhledem k `ebp`. Bez této možnosti by pro programátora bylo obtížné pracovat s lokálními symboly. Je však nutné, aby programátor tuto možnost využíval opatrně, neboť může dojít k různým interferencím s (konkrétním cílovým) assemblerem (například lokální proměnná s názvem „eax“). Vzhledem k tomu tato možnost není standardně zapnuta a uživatel ji musí explicitně zapnout při volání překladače. Avšak i v situaci, kdy generování maker není povoleno, je programátorovi poskytnuta informace o pozicích lokálních symbolů alespoň formou komentářů. Detailněji se makry zabývá příloha C.

Zajímavou možností by bylo vybavit překladač postprocesorem, který by do výstupních souborů automaticky vložil již předem připravené části assemblerovských kódů. Tuto funkce však zatím není naimplementována.

6.2 Standardy

Vygenerovaný kód se řídí běžnými standardy (*cdecl*).

Volání funkcí, zarovnání parametrů: Parametry jsou vkládány do zásobníku zleva doprava (takže nejpravější bude po zavolání funkce nejblíže k bázi). Funkce v rámci vstupní rutiny uloží bázový ukazatel na zásobník a pak do něj (bázového ukazatele) uloží zásobníkový ukazatel (nejpravější parametr je tedy od báze vzdálen +8 slabik, levější více). Následně je na zásobníku vytvořeno místo pro lokální proměnné.

Při vkládání parametrů jsou tyto zarovnávány na čtyři slabiky bez ohledu na velikost parametru. Tedy i kdyby měla funkce jako parametry slabiku, slabiku a slovo, nezaberou tyto parametry jen jedno dvojslovo, ale tři dvojslova.

Po skončení funkce je zásobník uklizen volajícím.

Zarovnání lokálních proměnných: Lokální proměnné a konstanty jsou zarovnávány tak, aby adresa byla násobkem velikosti aktuálně přidávaného symbolu. Tedy proměnné v pořadí slabika, slovo, slabika, slovo zaberou čtyři slova, zatímco při pořadí slabika, slabika, slovo, slovo zaberou jen tři slova.

Kapitola 7

Závěr

Podařilo se vytvořit jazyk a překladač, který umožňuje psát programy a moduly nebo vytvářet jejich kostry. Překladač však zatím není vhodný pro psaní celých programů a zvláště jejich mnohokrát opakovaných částí, neboť vygenerovaný kód není dostatečně efektivní (dvojice instrukcí PUSH a POP hned za sebou).

Na projektu se také negativně projevilo, že v době návrhu si řešitel stanovil nižší požadavky, které teprve později rozšířil. Návrh a implementace byly sice provedeny takovým způsobem, aby rozšíření (do určitých směrů) bylo možné, ale poslední přidaná rozšíření pravděpodobně přivedla projekt na hranici rozšiřitelnosti, která by byla snadno proveditelná a odolná proti chybám.

Také je diskutabilní, zda mělo být implementacím konkrétních assemblerů ponecháno tolik volnosti. Ta sebou totiž nese více práce. Implementátor se např. musí seznámit se strukturou tabulek symbolů a trojadresného kódu. Také musí dělat hodně práce, které by mohl udělat již překladač, ale za cenu části svobody generátoru.

7.1 Možnosti rozšíření a změn

V dalším pokračování projektu je nutno rozšířit sadu výrazových operátorů. To však povede na rozšíření LR tabulky do takové míry, že by byla jen složitě vypočítávána ručně. Pokud nebude požadováno doložení výpočtu, mohl by být použit nějaký poloautomatický nástroj. Samotný LR analyzátor je však naimplementován docela kvalitně a krom nové LR tabulky (a přídavných redukcí) se vylepšovat nemusí.

Pro zajištění další rozšiřitelnosti by bylo vhodné rozšířit strukturu objektů podílejících se na překladu, což však může vést k vyšším paměťovým nárokům.

Dále by bylo dobré přidat další fázi překladu a (optimalizovaný) trojadresný kód ještě převádět do abstraktní reprezentace assemblerovského kódu. Tím však bude omezena obecnost - respektive se může stát, že bude využívána jen podmnožina možností konkrétních assemblerů. Naproti tomu případní implementátoři konkrétních generátorů budou ušetřeni tvůrčí činnosti.

Užitečné by také bylo vybavit překladač preprocesorem a postprocesorem (automatické kombinování výstupních souborů s předem připravenými assemblerovskými soubory místo ruční editace).

Syntaxe obsahu složeného příkazu by mohla být rozšířena za hranice regulárních jazyků. Pak by bylo třeba vyměnit současný analyzátor těchto úseků kódu (LL analyzátor místo konečného automatu).

Také je možno ještě dále zvyšovat vazbu jazyka na hardware (možnost práce s registry, možnost využití jednotek MMX a SSE).

Dodatek A

Návrh

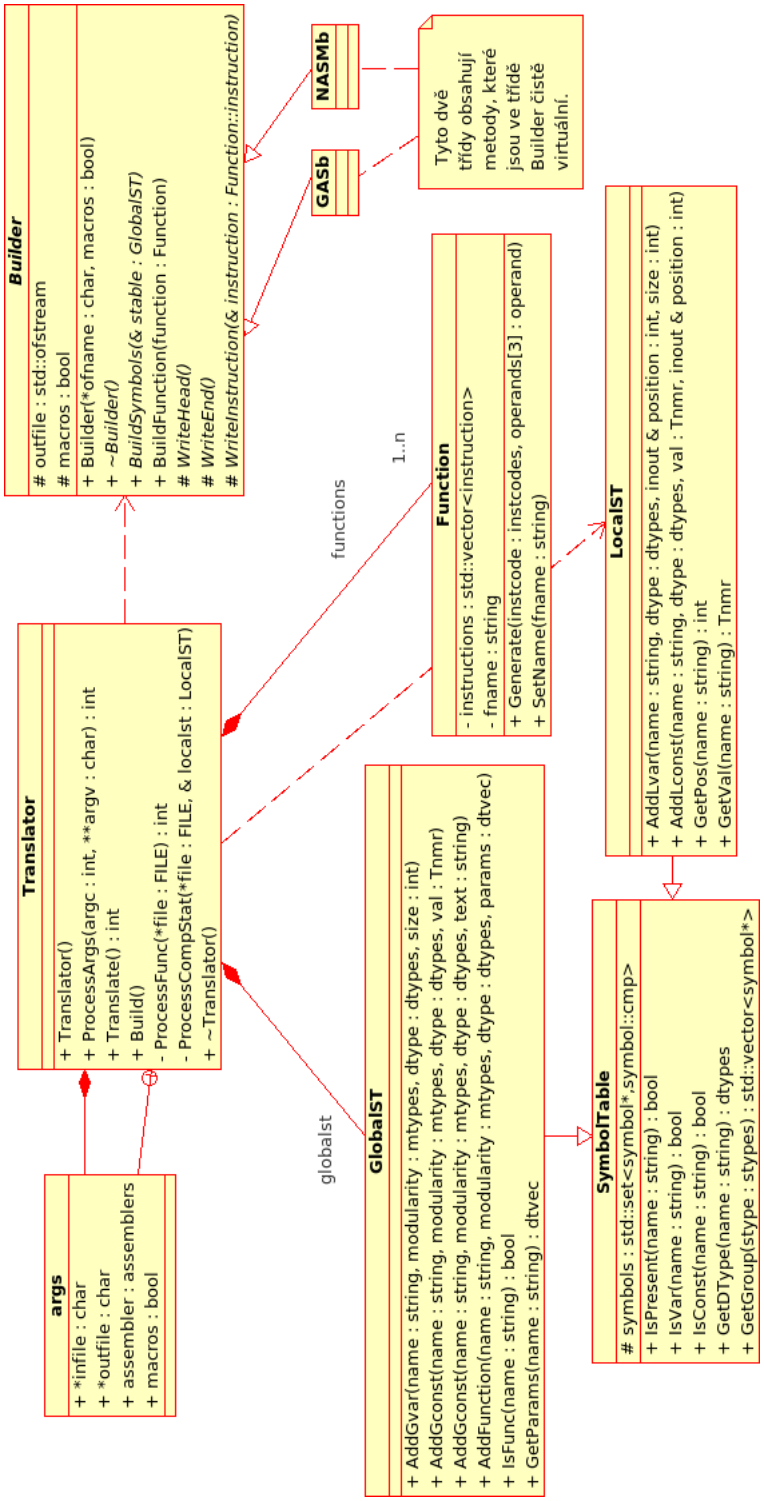
Tato část se zabývá pouze OO návrhem projektu. Schémata konečných automatů a výpočet LR tabulky jsou v dalších přílohách.

V části 5.2.2 bylo popsáno užití návrhového vzoru **stavitel**. Na schématu A.1 je uveden hlavní třídní diagram programu. V tomto schématu je návrhový vzor stavitel již bezešvě integrován.

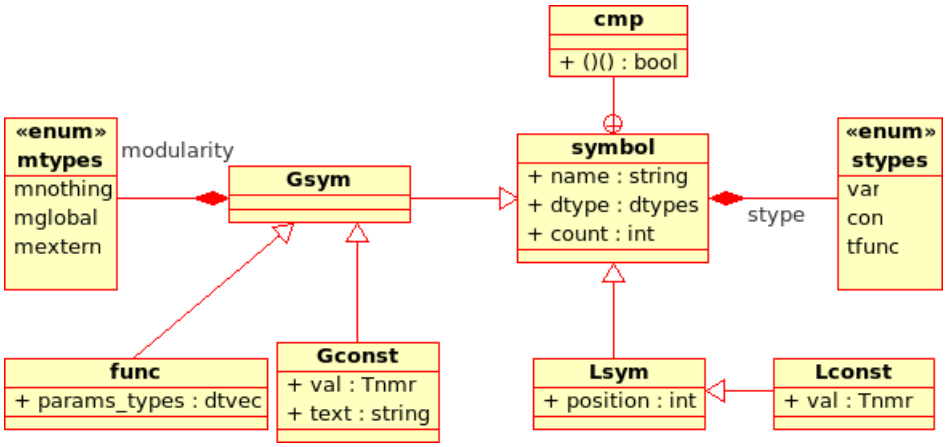
Jak je vidět, abstraktní stavitel je zde abstraktní třída *Builder*. Ta obsahuje čistě virtuální metody, které jsou pak ředitelem (třídou *Translator*) užity k řízení výstavby kódu. Třídy *GASb* a *NASMb* jsou zde uvedeny jako příklad a jejich obsah ani není ve schématu znázorněn.

Důležitým prvkem jsou také tabulky symbolů. Ty, jak vidět, mají jediný atribut a sice množinu symbolů. Třídou *symbol* se zabývá diagram A.2.

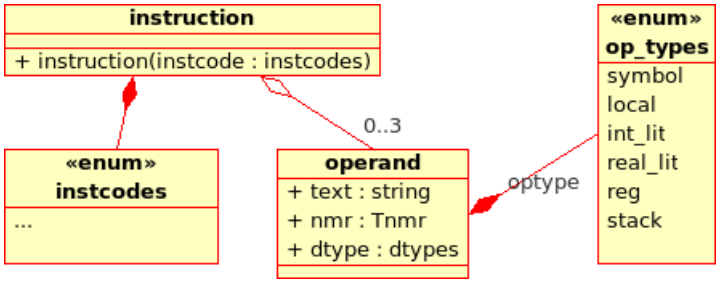
Neméně důležitá je třída *Function*, do jejichž objektů jsou ukládány instrukce v trojadresném kódu. Tyto instrukce jsou reprezentovány objekty třídy *instruction*, jejichž třída je vyznačena na schématu A.3.



Obrázek A.1: Hlavní třídní diagram překladače.



Obrázek A.2: Třída *symbol* a jeho dětské třídy.



Obrázek A.3: Třída *instruction*.

Dodatek B

Popis jazyka

B.1 Popis syntaxe

Zdrojové texty vstupního jazyka překladače se skládají z hlavičky a množiny definic funkcí. V hlavičce jsou definice a deklarace globální konstanty, proměnné a funkce.
Funkce se skládá z hlavičky a těla. Hlavička obsahuje definice lokálních proměnných a konstant. Tělo je velmi podobné Pascalu.
EBNF popis syntaxe jazyka je následující:

HeadBegin
[**const:**
<*dek/def konstant*>**;**]
[**var:**
<*dek/def proměnných*>**;**]
[**func:**
<*deklarace funkcí*>**;**]
HeadEnd;
<*funkce*> {<*funkce*>}

<*dek/def konstant*> = <*dek/def konstanty*> {**,** <*dek/def konstanty*>}
<*dek/def konstanty*> = [**global** | **extern**] <*typ*> <*identifikátor*> **:=** <*literál*>
<*dek/def proměnných*> = <*dek/def proměnné*> {**,** <*dek/def proměnné*>}
<*dek/def proměnné*> = [**global** | **extern**] <*typ*> <*identifikátor*>
<*deklarace funkcí*> = <*deklarace funkce*> {**,** <*deklarace funkce*>}
<*deklarace funkce*> = [**global** | **extern**] <*typ*> <*identifikátor*> (<*parametry*>)[1]
<*parametry*> = <*typ*> <*identifikátor*> {**,** <*typ*> <*identifikátor*>}

<*funkce*> =
<*typ*> <*identifikátor*> (<*parametry*>)
HeadBegin
[**const:**
<*definice lokálních konstant*>**;**]
[**var:**
<*definice lokálních proměnných*>**;**]

[1] Pro omezení použití zpětných lomítek tučná kulatá závorka značí kulatou závorku ve zdrojovém textu (zatímco obyčejná značí seskupení).

HeadEnd;
<složený příkaz> ;

<definice lokálních konstant> = *<definice lokální konstanty>* {**,** *<definice lokální konstanty>*}
<definice lokální konstanty> = *<typ>* *<identifikátor>* **:=** *<literál>*
<definice lokálních proměnných> = *<definice lokální proměnné>* {**,** *<definice lokální proměnné>*}
<definice lokální proměnné> = *<typ>* *<identifikátor>*

<složený příkaz> =
begin
{*<jednoduchý příkaz>* | *<if-konstrukce>* | *<for-cyklus>* | *<repeat-cyklus>* | *<while-cyklus>* | *<komentář>* }
end

<if-konstrukce> =
if *<výraz>* **then** *<složený příkaz>*
{**elif** *<výraz>* **then** *<složený příkaz>*}
[**else** *<složený příkaz>*] ;

<for-cyklus> =
for *<přiřazení>* (**to** | **downto**) *<výraz>* **do** *<složený příkaz>* ;

<repeat-cyklus> =
repeat *<složený příkaz>* **until** *<výraz>* ;

<while-cyklus> =
while *<výraz>* **do** *<složený příkaz>* ;

<jednoduchý příkaz> = (*<přiřazení>* | *<volání funkce>* | *<přerušení>* | *<return konstrukce>*) ;
<přerušení> = **-|-** (*<výraz>* {**,** *<výraz>*})
<volání funkce> = *<identifikátor>*([*<výraz>*{**,** *<výraz>*}])
<přiřazení> = [^[1|2|4]]*<identifikátor>* \[<výraz> \] **:=** *<výraz>*
<return konstrukce> = **return** [*<výraz>*]
<komentář> = `//[^\n]*`[2]

<výraz> =
(*<výraz>*(|
<výraz> *<bin. operátor>* *<výraz>* |
<un. operátor> *<výraz>*[3] |
<literál> |
<identifikátor> |
<volání funkce>

[2]Toto je regulární výraz (stejně jako další texty psané tímto fontem).
[3]Pozor, operátor & je možné použít jen na symbol.

<identifikátor> =	`[_a-zA-Z][_a-zA-Z0-9]*`
<literál> =	*<celočíselný l.>* | *<znakový l.>*
<celočíselný l.> =	`[\d]+`
<znakový l.> =	`'([^'\]\\.)'`
<bin. operátor> =	`+ - * / == != > < => =<`[4]
<un. operátor> =	`- ^ &`
<typ> =	**int** | **short** | **char** | **void**

B.2 Vysvětlení sémantiky

Globální hlavička: Symboly zde nadefinované a nadeklarované se stanou skutečnými symboly i ve vygenerovaném assembleru. Nadefinované konstanty a proměnné se objeví v sekcích .data a .bss. Pomocí klíčových slov **global** a **extern** je možno provést export a import symbolů (například v hlavním modulu je nutno nadeklarovat _start, nebo jinou funkci, jako **global**, aby bylo možno modul slinkovat).

Statické pole proměnných se vytváří přidáním velikosti pole před jeho typ. Ze symbolu se poté však nestane ukazatel (stejně jako v assembleru) - daný symbol pak ukazuje na začátek pole stejně jako symbol skalární proměnné ukazuje na začátek této proměnné. Je nutno pochopit, že v assembleru (a tím pádem i ve vstupním jazyce) není rozdíl mezi skalárem a statickým polem.

Konstanty je možno definovat jako číselné, nebo řetězcové literály (zde už rozpozná samo a ošetří).

Definice funkce: V hlavičce jsou definice lokálních konstant a proměnných. Konstanty však už nyní nesmí být řetězce (i *gcc* řetězcové literály definuje v sekci data a v případě lokálních řetězců je jen nakopíruje do lokálního pole). Tělem funkce je složený příkaz.

Složený příkaz a jeho obsah: Zde je již jazyk velmi podobný Pascalu. Odlišností je možnost použít příkaz přerušení, který se používá podobně jako funkce a může mít jeden až čtyři parametry. Tyto parametry budou vloženy do registrů eax až edx.

K **ukazatelům** se přistupuje jako integerům, a to nejen ve vygenerovaném výstupním souboru, ale i v jazyce samotném. „Ukazatelem“ (nebo snad raději nositelem adresy) se tedy stává jakákoliv integerová proměnná, do které byl přiřazen výsledek operace reference. Následkem toho však nelze rozpoznat velikost cíle. Při zápisu do paměti přes ukazatel je tedy umožněno specifikovat velikost cílového úseku paměti přidáním čísla 1, 2, nebo 4 mezi operátor dereference a symbol. Implicitní hodnota je 4.

Hranaté závorky slouží pro snadnější přístup do polí pro zápis (není to operátor použitelný ve výrazu, tam je nutno použít operátory pro referenci, dereferenci, sčítání a odčítání). Jejich význam je takový, že nedojde k zápisu na místo definované před nimi, ale na to místo + offset (ve slabikách) určený výrazem uvnitř závorek. Nemá dereferenční význam jako v C. Tedy pokud by např. existovalo statické pole `arr`, tak `arr` nebo `arr[0]` na levé straně přiřazovacího příkazu povede na zápis do začátku pole, zatímco `arr[2]` povede na zápis do místa vzdáleného od začátku 2 slabiky. Pro zápis do dynamického pole je pak nutno použít operátor dereference, jak bylo popsáno výše.

Také stojí za povšimnutí, že chybí operátory logického a bitového and, or a not (stejně jako např. bitové posunu a další). Pro začátek byla totiž zvolena jen omezená sada operátorů

[4] Jedná se o výčet možností, i když není odděleno svislítkem.

pro jednodušší LR tabulku. Rozšíření je možné v dalším pokračování tohoto projektu. Mezitím je možno používat aritmetický operátor sčítání (a případně násobení, které je však pomalé).

B.3 Příklady

Příklad 1: Na následujícím jednoduchém příkladu je ukázána základní struktura vstupního kódu:

```
HeadBegin
  func:
    global void _start();
HeadEnd;

void _start()
HeadBegin
HeadEnd;
begin
  -|-(1,0);
end;
```

Za povšimnutí zde stojí jen dvě věci: export symbolu _start (aby byl viděn linkerem a mohl být použit jako vstupní adresa) a použití přerušení k ukončení programu (1 se nahraje do eax a značí službu OS exit, 0 se nahraje do ebx a značí návratový kód procesu).

Příklad 2: Na následujícím jednoduchém příkladu jsou ukázány řídící struktury jazyka:

```
HeadBegin
  func:
    global void _start();
  const:
    char hlaska := "ahoj ";
HeadEnd;

void _start()
HeadBegin
  var:
    int i;
HeadEnd;
begin
  // for cyklus
  for i:=0 to 10 do
  begin
    -|-(4,1,&hlaska,5);
  end;
  // while
  while (i>0) do
  begin
```

```
    -|-(4,1,&hlaska,5);
    i := i-1;
  end;
  // repeat
  repeat begin
    -|-(4,1,&hlaska,5);
    i := i+1;
  end until (i<3);
  // if-else
  if ((i==3)+(i==5)>0) then
  begin
    -|-(4,1,&hlaska,4);
  end
  elif ((i==6)+(i==8)>0) then
  begin
    -|-(4,1,&hlaska,3);
  end
  else
  begin
    -|-(4,1,&hlaska,2);
  end;
  // ukončení
  -|-(1,0);
end;
```

Jak je vidět, řídící struktury jsou až na klíčové slovo `elif` stejné jako v Pascalu. Dále stojí za povšimnutí to, že symbol *hlaska* jen ukazuje na první symbol řetězce, stejně jako symboly skalárních proměnných ukazují na tyto proměnné. Obyčejným použitím tohoto symbolu by byl pouze získán přístup k prvnímu znaku řetězce. Aby bylo možno přerušení předat adresu, musí být použit operátor reference (více viz C).

B.4 Konečné automaty

Některé části vstupního jazyka jsou regulární. Například globální hlavička se dá vyjádřit následujícím regulárním výrazem:

```
HeadBegin
  (var:
    ((global)|(extern))? ([1-9][0-9]*)? [_a-zA-Z][_a-zA-Z]*
    (,((global)|(extern))? ([1-9][0-9]*)? [_a-zA-Z][_a-zA-Z]*)*;)?
  (const: ... ;)?
  (func: ... ;)?
HeadEnd;
```

Z tohoto důvodu je možno části kódu zpracovávat konečnými automaty. Krom nich byl použit ještě rekurzivní sestup (složený příkaz, volání funkcí atd.) a LR analyzátor (výrazy).

Hlavní konečný automat (na obrázku B.1) přečte globální hlavičku a poté začne číst jednotlivé funkce. Lze si povšimnout, že v části zabývající se globálními konstantami, byla už naznačena (nedeterministická) možnost přeskočení definice konstanty. To je záležitost

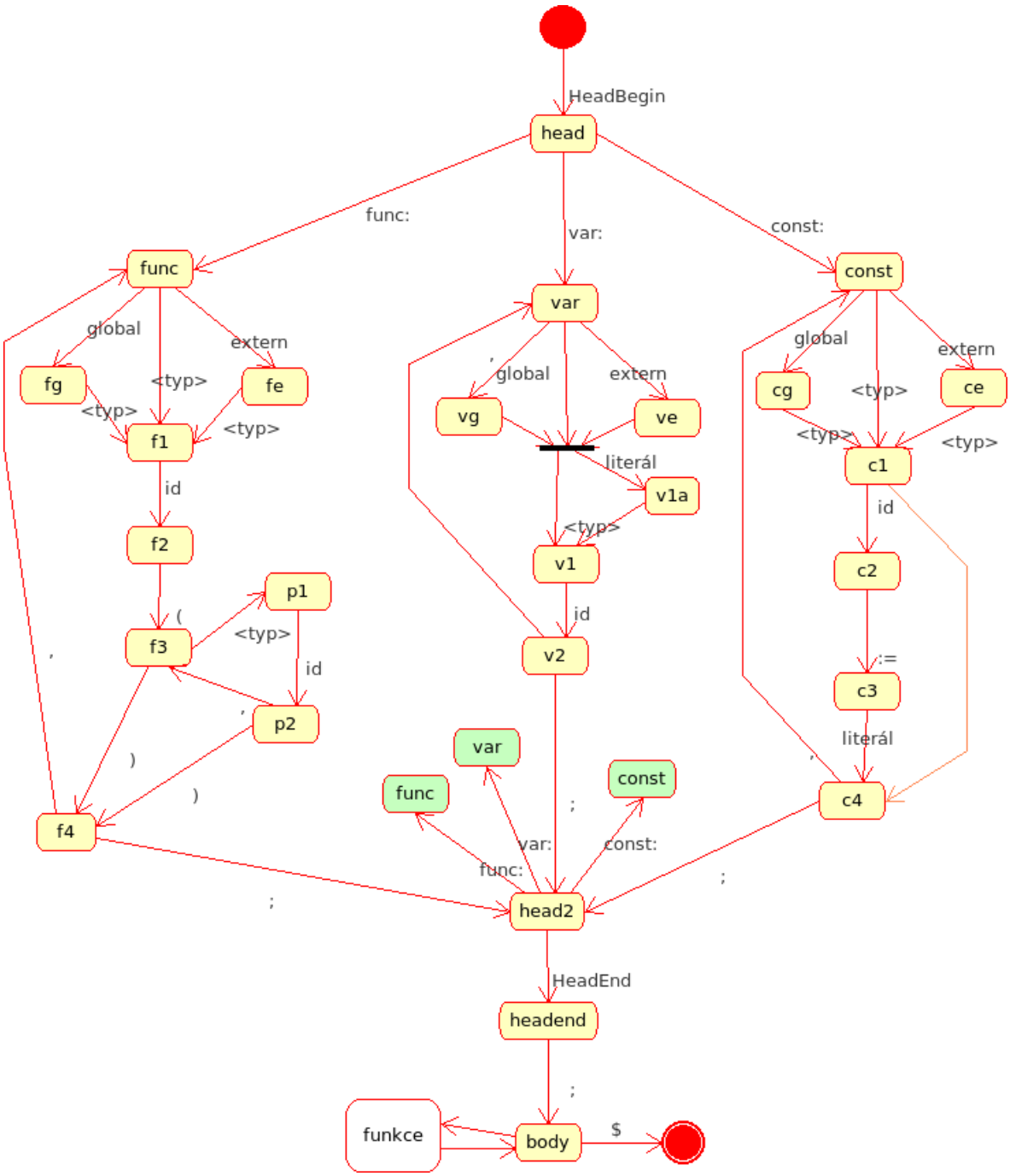
sémantické analýzy - při deklaraci externí konstanty se bude pochopitelně přeskakovat definice její hodnoty. Toto by sice mohlo být ošetřeno už na úrovni syntaktické analýzy, což by ale vedlo na složitější implementaci.

Také je třeba poznamenat, že ne všechny stavy vyznačené v konečném automatu mají svůj stav i v implementaci. Tam, kde nedochází k větvení automatu, se obvykle načte více tokenů za sebou bez přepínání stavu.

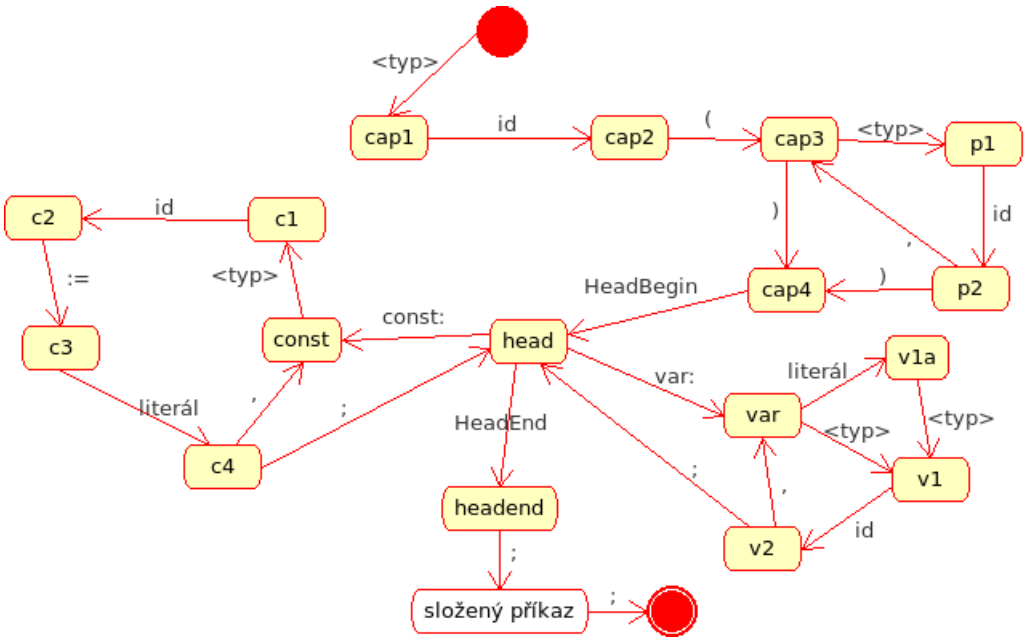
Značení přechodu a větvení (černá tlustá úsečka) slouží pouze pro zjednodušení grafu a nemá žádný zvláštní význam.

Další automat (obrázek B.2) má na starosti načtení jedné funkce. Sám však načte jen hlavičku funkce, poté předá řízení dalšímu automatu. Symboly počátečního a finálního stavu zde znamenají vstupní a výstupní bod.

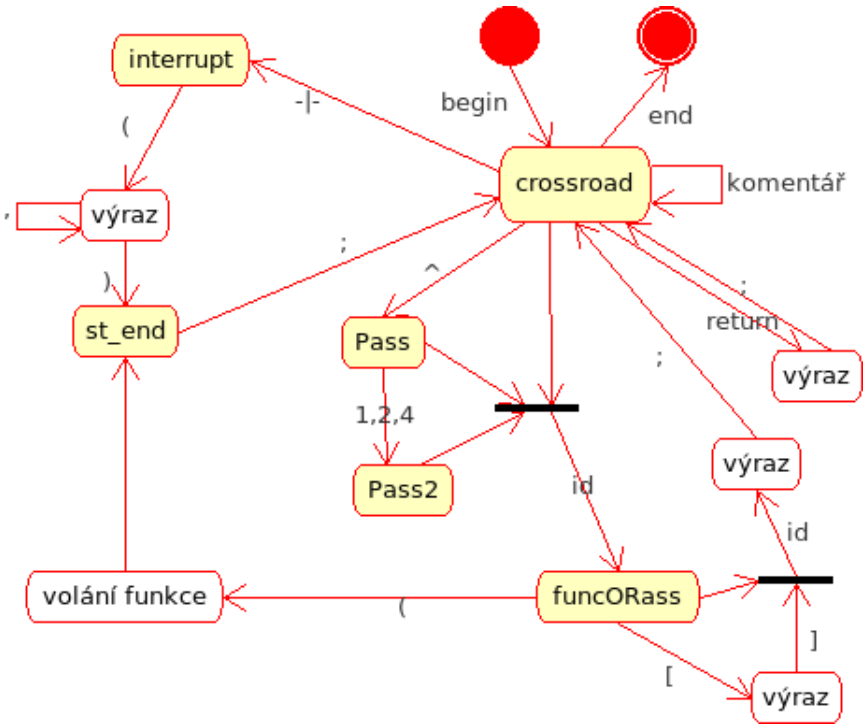
Složený příkaz je sice zpracováván jedním automatem, ale pro jednoduchost bylo schéma tohoto automatu rozděleno do dvou (B.3 a B.4). Nevyplněné stavy (podobně jako u předchozího automatu) značí ošetření speciální funkcí. V implementaci těmto stavům žádné opravdové stavy neodpovídají a patří ke stavu předešlému. Z toho pak dojde na přepnutí rovnou na stav následující.



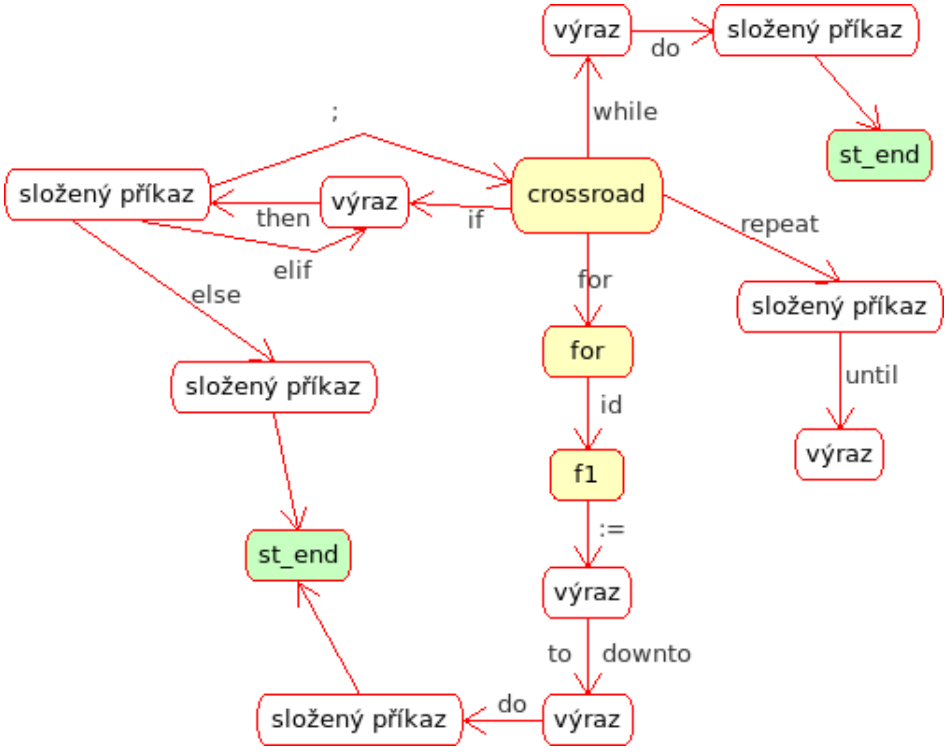
Obrázek B.1: Nejhornější konečný automat překladače.



Obrázek B.2: Automat pro zpracování funkce.



Obrázek B.3: Automat pro zpracování složeného příkazu. Načítá jen jednoduché příkazy a komentáře.



Obrázek B.4: Automat pro zpracování složeného příkazu. Načítá řídící konstrukce.

Dodatek C

Poznámky ke generovanému kódu

Tato příloha obsahuje jednak obecné dodatky a rozvedení věcí naznačených v hlavním textu (kapitoly 4 a 6) a věci, které je zapotřebí znát pro efektivní programování (kvůli vysoké vazbě jazyka na assembler) a jednak konkrétní poznámky k ukázkovým generátorům.

C.1 Obecné dodatky

Pole: **Statická** pole jsou proměnné, které mají za sebou volné místo (ať už se jedná o globální pole, kde je vynechání místa sděleno assembleru pomocí direktiv, nebo lokální, kde se jedná o vynechání místa na zásobníku). Běžným zápisem do symbolu pak dojde k zápisu do první položky tohoto pole. Jak již bylo řečeno ve vysvětlení sémantiky, hranaté závorky umožňují posunout místo zápisu, ale nenahrazují dereferenci (neplatí tvrzení z C, že pole a ukazatele jedno jsou). Začátek pole je na tom konci, které má nižší adresu.

Dynamická pole jsou již běžně známé naalokované úseky paměti, jejichž počáteční adresa se dá přiřadit do integeru, se kterým se poté dají provádět běžné aritmetické úkony (pozor, i násobení a dělení) a tím se po poli pohybovat (případně je možno zkombinovat ukazatel a hranaté závorky - i při jejich použití je však stále zapotřebí psát operátor dereference).

Vyhodnocení výrazů: Při vyhodnocování výrazů se nepoužívá běžná kombinace registry + pomocné proměnné, ale zásobník. Tedy, když je výsledek nějaké operace zapotřebí uschovat na později, neuloží se do překladačem dočasně vytvořené proměnné (případně, po optimalizaci, registru), ale vloží se na zásobník (což je sice po stránce fyzického umístění proměnné to stejné, ale v prvním případě se použije instrukce MOV, zatímco ve druhém PUSH). Takto je možno zpracovat jakýkoliv výraz neužívající levou asociativitu (v tomto případě splněno), neboť je možno tyto výrazy převést na postfixovou notaci (pozor: překladač ale pracuje s trojadresným mezikódem, postfixovou notaci neužívá).

Optimalizovanost kódu: V současné době nejen, že ještě není naimplementován optimalizátor mezikódu, ale i generátor cílového kódu je zatím slepý. Ve vygenerovaném kódu tak například v souvislosti s předchozí poznámkou můžeme najít příkazy PUSH a POP hned za sebou (a se stejným operandem).

Využití registrů: Pokud využití registrů není přímo určeno překladačem na úrovni trojadresného kódu (např. při přerušení překladač přímo určí, co přijde na který registr), je

využití registrů v kompetenci jednotlivých implementací konkrétních assemblerů. Nicméně oba generátory, které v současné době existují, používají registry eax, ebx a ecx (a také samozřejmě ebp). Tím zůstávají některé registry zcela nevyužity - jak již bylo řečeno, generování assembleru zatím probíhá slepě.

C.2 NASM

Makra: Offsety lokálních proměnných jsou definovány pomocí direktivy preprocesoru `%define`. Název makra je shodný s názvem proměnné a tělem makra je číselná hodnota offsetu. Pod kódem funkce jsou všechny použitá makra zrušena pomocí `%undef`.

Globální proměnné a konstanty: Jsou běžným způsobem definovány v sekcích .data a .bss.

C.3 GAS

Makra: Makra v GASu jsou trochu odlišná od maker NASMu a C. Vzhledem k tomu, že je zapotřebí definovat symbol a ne předpis pro vygenerování kusu kódu, byla v GASu použita direktiva `.equ`. Tyto symboly pak nejsou rušeny a proto je nutno tuto možnost využívat s rozvahou (nejlépe ji ponechat vypnutou a pak jen odkomentovat to, co je zapotřebí).

Globální proměnné a konstanty: Stejně jako v NASMu, i zde byly použity sekce .data a .bss. Řetězce jsou v sekci .data definovány pomocí direktivy .ascii. Běžné proměnné a konstanty jsou definovány běžným způsobem (typem a počáteční hodnotou). Pole jsou vytvářena pomocí direktivy .size.

Dodatek D

Zpracování výrazů

D.1 Definice gramatiky

G je bezkontextová gramatika. Do této gramatiky již bylo přidáno ukončovací pravidlo (i když je očíslováno jako 1).

G = { N, T, P, E' }
T = { `^`, `&`, `+`, `-`, `*`, `/`, `==`, `!=`, `>`, `<`, `=>`, `=<`, `id`, `lit`, `(`, `)` }
N = { A, B, C, D, E, E' }
P = {

1. E' → E
2. A → id
3. A → lit
4. A → (E)
5. B → - A
6. B → ^ A
7. B → & A
8. B → A
9. C → C * B
10. C → C / B
11. C → B
12. D → D + C
13. D → D - C
14. D → C
15. E → E ◇[1] D
16. E → D

}

D.2 Konstrukce LR tabulky

Při konstrukci LR tabulky byl použit SLR algoritmus.

Nejprve je zapotřebí určit množinu Θ:

Θ_G = { $< \varepsilon >$,
<E>, <id>, <lit>, <(>, <->, <ˆ>, <&>, <A>, <B>, <C>, <D>,
<(E>, <-A>, <ˆA>, <&A>, <C*>, <C/>, <D+>, <D->, <E⋄ >,
<(E)>, <C*B>, <C/B>, <D+C>, <D-C>, <E⋄D> }

Poté byla vypočtena množina *Contents* pro každý prvek množiny Θ:

Contents(< ε >) = { E' → •E, E → •D, E → •E⋄D, D → •D+C, D → •D-C, D → •C, C → •C*B, C → •C/B, C → •B, B → •-A, B → •ˆA, B → •&A, B → •A, A → •id, A → •lit, A → •(E) }
Contents(< E >) = { E' → E•, E → E•⋄D }
Contents(< D >) = { E → D•, D → D•+C, D → D•-C }
Contents(< C >) = { D → C•, C → C•*B, C → C•/B }
Contents(< B >) = { C → B• }
Contents(< A >) = { B → A• }
Contents(< - >) = { B → -•A, A → •id, A → •lit, A → •(E) }
Contents(< ˆ >) = { B → ˆ•A, A → •id, A → •lit, A → •(E) }
Contents(< & >) = { B → &•A, A → •id, A → •lit, A → •(E) }
Contents(< id >) = { A → id• }
Contents(< lit >) = { A → lit• }
Contents(< (>) = { A → (•E), E → •D, E → •E⋄D, D → •D+C, D → •D-C, D → •C, C → •C*B, C → •C/B, C → •B, B → •-A, B → •ˆA, B → •&A, B → •A, A → •id, A → •lit, A → •(E) }
Contents(< E⋄ >) = { E → E⋄•D, D → •D+C, D → •D-C, D → •C, C → •C*B, C → •C/B, C → •B, B → •-A, B → •ˆA, B → •&A, B → •A, A → •id, A → •lit, A → •(E) }
Contents(< D+ >) = { D → D+•C, C → •C*B, C → •C/B, C → •B, B → •-A, B → •ˆA, B → •&A, B → •A, A → •id, A → •lit, A → •(E) }
Contents(< D- >) = { D → D-•C, C → •C*B, C → •C/B, C → •B, B → •-A, B → •ˆA, B → •&A, B → •A, A → •id, A → •lit, A → •(E) }
Contents(< C* >) = { C → C*•B, B → •-A, B → •ˆA, B → •&A, B → •A, A → •id, A → •lit, A → •(E) }
Contents(< C/ >) = { C → C/•B, B → •-A, B → •ˆA, B → •&A, B → •A, A → •id, A → •lit, A → •(E) }
Contents(< -A >) = { B → -A• }
Contents(< ˆA >) = { B → ˆA• }
Contents(< &A >) = { B → &A• }
Contents(< (E >) = { A → (E•), E → E•⋄D }
Contents(< E⋄D >) = { E → E⋄D•, D → D•+C, D → D•-C }

[1]Tento znak zastupuje porovnávací operátor, které budou pro svůj počet vnímány LR parserem jako jeden neterminál. Díky způsobu implementace toto nečiní později žádných problémů.

$Contents(< \text{D+C} >) = \{ \text{D} \rightarrow \text{D+C}\bullet, \text{C} \rightarrow \text{C}\bullet\text{*B}, \text{C} \rightarrow \text{C}\bullet\text{/B} \}$
$Contents(< \text{D-C} >) = \{ \text{D} \rightarrow \text{D-C}\bullet, \text{C} \rightarrow \text{C}\bullet\text{*B}, \text{C} \rightarrow \text{C}\bullet\text{/B} \}$
$Contents(< \text{C*B} >) = \{ \text{C} \rightarrow \text{C*B}\bullet \}$
$Contents(< \text{C/B} >) = \{ \text{C} \rightarrow \text{C/B}\bullet \}$
$Contents(< (\text{E}) >) = \{ \text{A} \rightarrow (\text{E})\bullet \}$

Pro konstrukci tabulky jsou též zapotřebí množiny *Follow* všech neterminálů:

$Follow(\text{E'}) = \{ \$ \}$
$Follow(\text{E}) = \{ \$, \diamond,) \}$
$Follow(\text{D}) = \{ \$, \diamond,), +, - \}$
$Follow(\text{C}) = \{ \$, \diamond,), +, -, *, / \}$
$Follow(\text{B}) = \{ \$, \diamond,), +, -, *, / \}$
$Follow(\text{A}) = \{ \$, \diamond,), +, -, *, / \}$

S těmito podklady je možno vytvořit LR tabulku. Následující tabulka obsahuje akční část LR tabulky[2]:

#	$< x >$	id	lit	^	&	-	+	*	/	◇	(	)	$
1	$< \varepsilon >$	s<id>	s<lit>	s<^>	s<&>	s<->					s<(>		
2	<E>									s<E◇>		$	$
3	<D>					s<D->	s<D+>			r16		r16	r16
4	<C>					r14	r14	s<C*>	s<C/>	r14		r14	r14
5	<B>					r11	r11	r11	r11	r11		r11	r11
6	<A>					r8	r8	r8	r8	r8		r8	r8
7	<id>					r2	r2	r2	r2	r2	f	r2	r2
8	<lit>					r3	r3	r3	r3	r3		r3	r3
9	<^>	s<id>	s<lit>								s<(>		
10	<&>	s<id>	s<lit>								s<(>		
11	<->	s<id>	s<lit>								s<(>		
12	<(>	s<id>	s<lit>	s<^>	s<&>	s<->					s<(>		
13	<E◇>	s<id>	s<lit>	s<^>	s<&>	s<->					s<(>		
14	<D+>	s<id>	s<lit>	s<^>	s<&>	s<->					s<(>		
15	<D->	s<id>	s<lit>	s<^>	s<&>	s<->					s<(>		
16	<C*>	s<id>	s<lit>	s<^>	s<&>	s<->					s<(>		
17	<C/>	s<id>	s<lit>	s<^>	s<&>	s<->					s<(>		
18	<-A>					r5	r5	r5	r5	r5		r5	r5
19	<^A>					r6	r6	r6	r6	r6		r6	r6
20	<&A>					r7	r7	r7	r7	r7		r7	r7
21	<(E>									s<E◇>		s<(E)>	
22	<E◇D>					s<D->	s<D+>			r15		r15	r15
23	<D+C>					r12	r12	s<C*>	s<C/>	r12		r12	r12
24	<D-C>					r13	r13	s<C*>	s<C/>	r13		r13	r13
25	<C*B>					r9	r9	r9	r9	r9		r9	r9
26	<C/B>					r10	r10	r10	r10	r10		r10	r10
27	<(E)>					r4	r4	r4	r4	r4		r4	r4

[2]Písmeno f v políčku [7,(] značí volání funkce. Na políčku [2,)] je $, poněvadž znak) může mít i význam jako $.

Následující tabulka obsahuje přechodovou část LR tabulky:

#	$< x >$	A	B	C	D	E
1	$< \varepsilon >$	<A>	<B>	<C>	<D>	<E>
2	<E>					
3	<D>					
4	<C>					
5	<B>					
6	<A>					
7	<id>					
8	<lit>					
9	<ˆ>	<ˆA>				
10	<&>	<&A>				
11	<->	<-A>				
12	<(>	<A>	<B>	<C>	<D>	<(E>
13	<E⋄ >	<A>	<B>	<C>	<E⋄D>	
14	<D+>	<A>	<B>	<D+C>		
15	<D->	<A>	<B>	<D-C>		
16	<C*>	<A>	<C*B>			
17	<C/>	<A>	<C/B>			
18	<-A>					
19	<ˆA>					
20	<&A>					
21	<(E>					
22	<E⋄D>					
23	<D+C>					
24	<D-C>					
25	<C*B>					
26	<C/B>					
27	<(E)>					

Dodatek E

Návod k použití

Kompilace: Provádí se příkazem make v adresáři se zdrojovými kódy. Je nutno mít nainstalovaný *flex*. Příkazem make doxy bude vygenerována programová dokumentace (již přiložena).

Užití: `jhp2asm zdroj [-o cíl] [-a assembler] [-m]`

Parametrem `-a` se provádí výběr cílového assembleru. Rozpoznávané možnosti jsou NASM a GAS.

Parametr `-m` zapíná generování (vypíná zakomentování) pomocných maker pro lokální proměnné a konstanty.

Výstupní soubory je možno dále zpracovávat programy nasm/as a ld.

Literatura

[1] Pokyny k vypracování bakalářských prací.
http://www.fit.vutbr.cz/info/szz/pokyny.bp.html.

[2] Popis návrhových vzorů. http://www.vincehuston.org/dp/.

[3] Šablona pro vypracování bakalářské práce v prostředí LaTeX.
http://www.fit.vutbr.cz/info/szz/sablona07.zip.

[4] Alexandr Meduna a Lukáš Roman. Materiály k předmětu ifj.
https://www.fit.vutbr.cz/study/courses/IFJ/public/materials/.

[5] Alexandr Meduna a Lukáš Roman. Materiály k předmětu vyp.
https://www.fit.vutbr.cz/study/courses/VYP/public/materials/.

[6] Filip Orság. Materiály k předmětu pas.
https://www.fit.vutbr.cz/study/courses/PAS/private/.

[7] František Zbořil. Materiály k předmětu ias.
https://www.fit.vutbr.cz/study/courses/IAS/private/.

Seznam příloh

A Návrh

B Popis jazyka

C Poznámky ke generovanému kódu

D Zpracování výrazů

E Návod k použití

Obsah CD Na přiloženém CD jsou důležité následující soubory:

- report.pdf - elektronická forma tohoto dokumentu
- src/
  - doxy/index.html - programová dokumentace projektu
  - *.cc *.h *.l - zdrojové soubory projektu
  - Makefile - makefile projektu, příkazy:
    - make - kompilace projektu
    - make clean - vyčištění zkompilovaných souborů
    - make doxy - znovuvygenerování programové dokumentace