C COMPILER IN PYTHON

Tomáš Fiedor
Bachelor Degree Programme (3), FIT BUT
E-mail: xfiedo01@stud.fit.vutbr.cz

Supervised by: Zdeněk Vašíček
E-mail: vasicek@fit.vutbr.cz

Abstract: Currently, there is no link between hardware and compiler oriented courses during bachelor study. This paper demonstrates, that it is possible to link these courses with a relatively low effort. Suggested framework allows to experiment with various hardware architectures and shows the impact on output code.

Keywords: high level compiler, processor design, processor architecture

1 ÚVOD

Kurzy zaměřené na hardware a na koncepci překladačů jsou zpravidla vyučovány izolovaně, bez širší návaznosti. Neexistuje mezi nimi žádné logické propojení, které by zaručilo zachycení veškerých souvislostí mezi překladovou fází programu a jeho následném zavedení a spuštění na procesoru. Ne každý student tak dokonale pochopí probíranou problematiku. A přitom, je to opravdu tak nemožné přání? Mým cílem je demonstrovat, že vytvořit most mezi těmito kurzy a spojit související problematiky do jednoho projektu – překladače programovacího jazyka – není nic těžkého.

Vytvořený překladač je cílen pro výukové, demonstrační a experimentální účely. K dosažení tohoto cíle, jsem se při výběru prostředků zaměřil převážně na látku probíranou během bakalářského studia na Fakultě informačních technologií. Vstup překladače je v jazyce C a výstup v jazyce assembler, který je následně převeden do strojového kódu jednoduchého cílového procesoru využívaného pro výukové účely v předmětu INP (Návrh počítačových systémů). Tento procesor, s minimální výpočetně úplnou instrukční sadou o velikosti asi dvaceti instrukcí, však zároveň představuje i mnoho výzev – omezenou paměť, malý počet instrukcí a absenci hardwarového zásobníku. Tímto vytváří prostor pro využití optimalizací a experimentování s jednotlivými moduly překladače. Samotná struktura překladače pak vychází z běžných postupů uvedených v doporučené literatuře kurzů orientovaných na překladovou činnost [3].

2 ZVOLENÉ PROSTŘEDKY

Vhodným prostředím pro tvorbu překladačů cílených pro podporu výuky je objektově orientovaný skriptovací jazyk Python. Součástí jeho standardní knihovny je nespočet modulů obsahujících již vytvořené pomocné funkce a kolekce využitelné při implementaci překladače; například pro realizaci tabulky symbolů. Objektově orientovaný přístup Pythonu současně umožňuje jednoduše zapouzdřit logiku jednotlivých překladových částí. Tím získáme nezávislost modulů a snadné experimentování s různými implementacemi, za předpokladu, že dodržíme dané rozhraní metod a vlastností. Zde se využívá přednosti jazyka zvané „duck typing“ [1], tedy dynamické typování, kdy se metody a objekty nezabývají hierarchií dědičnosti, ale přítomností metod a vlastností v objektu. V jazyce Python je navíc vše objektem a parametery jsou vždy předávány odkazem. Takto odpadá veškerá starost o správu alokované paměti i práce s ukazateli a lze se tak zaměřit na důležitější aspekty struktury a implementace překladače, než je správné uvolňování paměti.

Velikost výrazu [znaky]	**ANTLR** [s/cyklus]	**PyParsing** [s/cyklus]	**PLY** [s/cyklus]	**Vlastní nástroj** [s/cyklus]
300	0.00696	0.01411	0.00514	0.00245
1200	0.03172	0.06570	0.01627	0.00965
20000	0.50598	0.97399	0.23571	0.15383
100000	2.84151	5.09380	1.28975	0.88024

Tabulka 1: Srovnání rychlostí parsovacích nástrojů pomocí testovacího skriptu.

2.1 PROSTŘEDKY PRO TVORBU PŘEKLADAČŮ

Existuje nespočet nástrojů v Pythonu, které usnadňují vytváření překladačů. Použití generických lexikálních a syntaktických analyzátorů je vhodnou volbou pro urychlení vývoje. Z nabídky dostupných modulů jsem se zaměřil na tři nejpoužívanější (PLY, ANTLR, PyParsing) a ty jsem dále podrobil podrobnějšímu vyhodnocení z pohledu výkonnosti v porovnání s vlastní implementací analyzátorů.

PLY [1] je implementace klasických unixových nástrojů Lex a Yacc, která klade důraz na použití regulárních výrazů a bezkontextové gramatiky pro popis syntaxe zpracovávaného jazyka. Podobný přístup má i nástroj ANTLR [2], který ze vstupního generického popisu syntaxe pomocí gramatiky vytvoří dva moduly pro syntaktickou a lexikální analýzu použitelné jako přední část překladače. Poslední zvažovanou možností je pak PyParsing [1] s odlišným způsobem analýzy pomocí tvorby parsovacích elementů a přiřazování funkcí k těmto elementům. Z výsledků uvedených v tabulce 1 lze vidět, že nejrychlejším z vybraných nástrojů je modul PLY, který dosahuje v porovnání s vlastními analyzátory šitými gramatice na míru dostatečné rychlosti a zároveň umožňuje rychlý vývoj aplikace.

2.2 CÍLOVÝ PROCESOR

Výstupem překladače je kód v assembleru v jednoduché instrukční sadě, který je následně konvertován do strojového kódu cílové architektury. Ta je realizována jako FPGA (Field Programmable Gate Array) projekt s jednoduchým 16-bitovým procesorem s registrovou architekturou typu Load-Store. Instrukční sada procesoru je výpočetně úplná a sestává z osmnácti instrukcí o šířce 16 bitů (čtyři skoky, práce s porty, pamětí, registry, sčítání, inkrement, dekrement). Je celkem možné adresovat 1024 paměťových míst a čtyři registry. Procesor nemá hardwarový zásobník a proto musí být implementace volání podprogramů realizována pomocí dostupných instrukcí simulací zásobníku v paměti za pomoci ukazatele. Tímto je omezena schopnost vlastní i nevlastní rekurze a hloubka zanoření volání funkcí.

3 IMPLEMENTACE PŘEKLADAČE

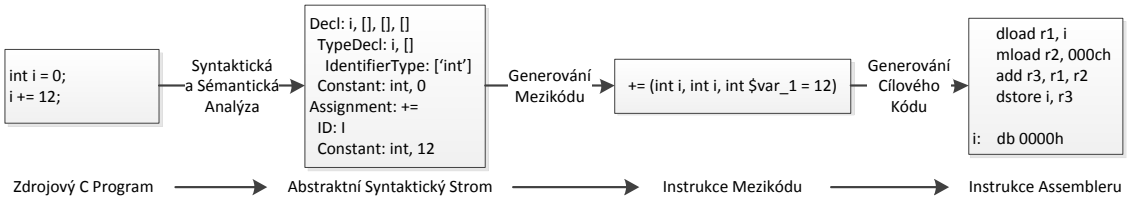
Vytvořený překladač funguje jako spustitelný skript ovládaný z příkazové řádky. Proces optimalizace výstupního kódu je plně modifikovatelný a lze zvolit jen konkrétní optimalizace nebo jednu z předpřipravených úrovní optimalizací (soubor optimalizací). Navíc jsou generovány statistiky překladu a veškeré výstupy mezikroků překladu. Příklad mezivýsledků lze vidět na obrázku 1.

Samotný překlad probíhá v sedmi krocích. Vstupní zdrojový soubor v jazyce C je nejprve zpracován textovým preprocesorem. V další části následuje vytvoření Abstraktního syntaktického stromu (dále jen AST) za pomocí lexikální a syntaktické analýzy, realizováné extérním modulem `pycparser` – parserem jazyka C implementovaného nástrojem PLY.

AST je vhodnou strukturou pro reprezentaci vstupního kódu a jeho analýzu. Uzly stromu jsou jednotlivé elementy jazyka C ze zdrojového souboru a hrany vyjadřují logickou sounáležitost mezi elementy. Implementačně je tato struktura realizována dědičností od bázové třídy `Node` podle objektově orientovaného idiomu pro reprezentaci AST (viz. [4]). Průchod stromem je umožněn návrhovým

vzorem `Visitor`, jenž odděluje algoritmus od datové struktury, nad kterou pracuje, bez nutnosti její modifikace v kódu. Takto jsou realizovány analýzy kódu kontrolující sémantickou validitu vstupního souboru. Vytvoří se tabulka symbolů a prověří se existence proměnných, zkontroluje se vnitřní tok instrukcí a aplikují se akce typového systému podle normy C99. Zanalyzovaný strom lze dále volitelně zredukovat rozvinutím konstant a následně je převeden do vnitřní reprezentace ve formě tří adresného kódu, ideálního pro další optimalizace.

Z důvodu výrazných omezení cílového procesoru jsem kladl hlavní důraz na poslední fázi překladu – optimalizaci kódu a efektivní generování výstupního assembleru. Překladač realizuje sadu optimalizací nad vnitřním tří-adresným kódem a umožňuje dva způsoby generování cílového kódu assembleru – tzv. „slepé“ a „kontextové generování“ (oba vychází z postupů uvedených v [3]). Při slepém generování jsou tří-adresné instrukce podle modelových šablon přepsány do instrukcí assembleru. Registry jsou instrukcím přidělovány rovnoměrně cyklicky. Oproti tomu kontextové generování je řízeno tabulkou základních bloků, kde základní blokem chápeme sekvenci instrukcí, které se vždy provedou společně. Z těchto bloků je generován cílový kód podle stavu učastnících se proměnných a jejich následných užití v kódu. Překladač se takto snaží ponechat proměnné v příslušných registrech co nejdelší dobu a negenerovat tak zbytečné instrukce přesunů mezi pamětí a registry.



Obrázek 1: Mezivýsledky fází překladu zdrojového program v jazyce C do instrukcí assembleru

4 ZÁVĚR

Implementovaný překladač úspěšně propojuje techniky návrhu procesoru i tvorby překladačů, čímž demonstruje možné využití při výuce. Při jeho tvorbě byl kladen důraz na použití zažitých postupů vyučovaných v předmětu Formální jazyky a překladače, který zahrnuje týmový projekt – tvorbu interpretu – společně s předmětem Algoritmy. Využitím programovatelných hradel a jazyka VHDL je pak možné vytvořit vlastní jednoduchý procesor a instrukční sadu, kterou lze využit jako cílovou množinu výstupního kódu překladače. Tím docílíme propojení souvislostí vyučovaných projektů a maximální pochopení probírané látky. Z pohledu gramatiky jazyka C podporuje překladač její většinu, s výjimkou datových typů, kdy cílový procesor zahrnuje pouze celočíselné operace.

REFERENCE

[1] Summerfield, M.: Programming in Python 3: A Complete Introduction to the Python Language, Addison-Wesley 2010, ISBN-13 978-0-321-68056-3.

[2] Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages, Terrence Parr 2007, ISBN-13 978-09787392-4-9

[3] Češka, M., Hruška, T., Beneš, M.: Překladače. Skriptum FEI VUT Brno 1993, ISBN 80-214-0491-4.

[4] Jones, J.: Abstract Syntax Tree Implementation Idioms. In: Proceedings of the 10th Conference on Pattern Languages of Programs, Illinois, 2003.