FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

Překladač PL/0 do x86

KIV/FJP - Semestrální práce

tým:	x86kaři	
studenti:	Martin Červenka	~~Zdeněk Ziegler~~
studijní čísla:	A17N0029P	~~A18N0104P~~
emaily:	cervemar@students.zcu.cz	~~zieglerz@students.zcu.cz~~
odevzdání:	3. ledna 2019	
github:	`https://github.com/cervenkam/fjp`	

Obsah

1 Zadání

Cílem práce bude vytvoření překladače zvoleného jazyka. Je možné inspirovat se jazykem PL/0, vybrat si podmnožinu nějakého existujícího jazyka nebo si navrhnout jazyk zcela vlastní. Dále je také potřeba zvolit si pro jakou architekturu bude jazyk překládán (doporučeny jsou instrukce PL/0, ale je možné zvolit jakoukoliv instrukční sadu pro kterou budete mít interpret).

Jazyk musí mít minimálně následující konstrukce:

- definice celočíselných proměnných
- definice celočíselných konstant
- přiřazení
- základní aritmetiku a logiku (`+`,`-`,`*`, `/`, `AND`, `OR`, negace a závorky, operátory pro porovnání čísel)
- cyklus (libovolný)
- jednoduchou podmínku (if bez else)
- definice podprogramu (procedura, funkce, metoda) a jeho volání

Překladač který bude umět tyto základní věci bude hodnocen deseti body. Další body (alespoň do minimálních 20) je možné získat na základě rozšíření, jsou rozděleny do dvou skupin, jednodušší za jeden bod a složitější za dva až tři body. Další rozšíření je možno doplnit po konzultaci, s ohodnocením podle odhadnuté náročnosti.

Jednoduchá rozšíření (1 bod):

- každý další typ cyklu (`for`, `do ... while`, `while ... do`, `repeat ... until`, `foreach` pro pole)
- `else` větev
- datový typ `boolean` a logické operace s ním
- datový typ `real` (s celočíselnými instrukcemi)
- datový typ `string` (s operátory pro spojování řětezců)
- rozvětvená podmínka (`switch`, `case`)
- násobné přiřazení (`a = b = c = d = 3;`)
- podmíněné přiřazení / ternární operátor (`min = (a < b) ? a : b;`)
- paralelní přiřazení ({`a, b, c, d` = 1, 2, 3, 4};)
- příkazy pro vstup a výstup (`read`, `write` – potřebuje vhodné instrukce které bude možné využít)

složitější rozšíření (2 body):

- příkaz `GOTO` (pozor na vzdálené skoky)
- datový typ `ratio` (s celočíselnými instrukcemi)
- složený datový typ (`Record`)
- pole a práce s jeho prvky
- operátor pro porovnání řetězců
- parametry předávané hodnotou
- návratová hodnota podprogramu
- objekty bez polymorfismu
- anonymní vnitřní funkce (lambda výrazy)

Rozšíření vyžadující složitější instrukční sadu než má PL/0 (3 body):

- dynamicky přiřazovaná paměť - práce s ukazateli
- parametry předávané odkazem
- objektové konstrukce s polymorfním chováním
- `instanceof` operátor
- anonymní vnitřní funkce (lambda výrazy) které lze předat jako parametr
- mechanismus zpracování výjimek

Vlastní interpret (řádkový, bez rozhraní, složitý alespoň jako rozšířená PL/0) je za 6 bodů.

2 Teoretický úvod

Překladače jsou nedílnou součástí dnešních počítačů. Mezi překladače lze řadit cokoliv, do čeho vstupuje nějaký předem definovaný formát a na výstupu je jiný, taktéž definovaný formát. Dnešní chápání pojmu překladač se však posunulo směrem k překladu programovacích jazyků, ať už mezi sebou nebo například do strojového kódu procesoru.

2.1 Jazyk PL/0

Jazyk PL/0 v dnešní době není tak rozšířen, protože je příliš jednoduchý pro reálné použití, avšak vychází z jazyku Pascal a je vhodný pro vytvoření jednoduchého překladače.

Tento jazyk disponuje pouze jednoduchou celočíselnou aritmetikou, jednoduchým větvením a jedním druhem cyklu. Oproti dnešním běžným jazykům (C-like, Java atd.) ale naopak podporuje plnohodnotné zanořené procedury.

Jazyk není sám o sobě použitelný, protože nemá žádný aparát pro vstup a výstup ani z konzole ani ze souboru ani odjinud. To znamená, že bude něco počítat, co musíme zadat při překladu a výsledek nemáme možnost běžnými prostředky získat. Proto je nutné tento jazyk rozšířit.

2.2 Procesor x86-64

Dnes jedním z velmi používaných procesorů v osobních počítačích je architektury x86-64. Tato architektura je dnes až 64bitová, ale je zpětně kompatibilní s 32bitovým a 16bitovým režimem.

Pravě v 16bitovém režimu procesor startuje. V tomto režimu dokáže teoreticky adresovat pouze 64kB paměti, ale díky segmentaci má vlastně adresaci 20bity, tedy 1MB paměti. Do některých částí této segmentované paměti se načte např. BIOS, nebo se rezervuje pro video paměť, což redukuje rozsah použitelné paměti na 640kB[1]. Tento režim byl dříve při představení i8086 jediným, s nástupem novější procesorů s chráněným režimem se tomuto 16bitovému režimu začalo říkat reálný.

Tato práce se konkrétně bude zabývat překladačem z jazyka PL/0 do strojového kódu instrukcí dnes běžně používaného procesoru x86 v 16bitovém reálném režimu.

[1] 640KB ought to be enough for anybody.

3 Platforma

U překladače se musejí definovat dvě platformy, jedna je ta zdrojová, tedy v jakém programovacím jazyce se program programuje, a druhá je cílová, tedy pro jaký systém se bude program překládat.

3.1 Zdrojová platforma

Překladač je napsán v programovacím jazyce Java, proto je kompatibilní jakékoliv zařízení, kde běží Java Virtual Machine (JVM) ve verzi alespoň 1.8.

3.2 Zdrojový program

Jako zdrojový (překládaný) programovací jazyk byl zvolen jazyk PL/0, který byl rozšířen oproti původní verzi. Jedná se konkrétně o tato rozšíření:

- větev `else`
- příkaz `read` a `write` pro čtení/zápis z/do konzole
- příkaz `execute`, který vykoná konkrétní zadaný strojový kód (podrobnosti dále)
- lokální `goto` skoky na zadaná návěští (`label`)
- jednořádkové komentáře začínající znaky \\
- logické operace (`AND`, `OR`, `XOR`, `NOT`) a bitové posuvy (`LEFT`, `RIGHT`), které taktéž v původní PL/0 chybí

Zjednodušeně zapsaná gramatika je také uvedena v příloze A.

3.3 Cílová platforma

Pro cílovou platformu jsem zvolil instrukce procesoru x86. Co se týče softwarové části, tak se program překládá do bootovatelné podoby, proto nepotřebuje žádný operační systém, jediný SW který v základní verzi potřebuje, je volání BIOSu pro čtení a zápis z konzole.

4 Uživatelská příručka

V této sekci je uvedna základní manipulace s překladačem.

4.1 Prerekvizity

Spuštění je možné provést děma způsoby. Doporučeným způsobem je použití emulátoru (např emulátor **QEMU**), kde prakticky nemůže dojít k tomu, aby přeložený program provedl nežádanou operaci typu mazání/přepisování souborů (např systémových). Druhý způsob (na vlastní riziko) je spuštění přímo na HW. Obě možnosti budou diskutovány.

4.2 OS Linux a emulátor QEMU

Překládá-li se program na OS Linux, jsou dodány skripty pro překlad a spuštění.

4.2.1 Překlad programu

Pro překlad se používá skript `compile.sh` a má dva nepovinné parametry. Prvním je cesta k překládanému programu a má přednastavenou hodntotu `hilbert.pl0`. Druhým je cesta k vyprodukovanému binárnímu souboru – přednastaveno je `out.bin`. Následující dva příkazy jsou tedy ekvivalentní:

```
./compile.sh
./compile.sh hilbert.pl0 out.bin
```

4.2.2 Spuštění

Pro spuštění je připraven skript `run.sh` a má jeden nepovinný parametr, kterým je přeložený soubor (s defaultní hodnotou `out.bin`. Následující dva příkazy jsou tedy opět ekvivalentní:

```
./run.sh
./run.sh out.bin
```

Spustí se okno emulátoru a program se začne vykonávat. Je-li v programu příkaz `read`, tedy je-li vyvzadován vstup od uživatele, tak ten se zadává v binární reprezentaci bez prefixu, tedy pouze `0` a `1`, jakákoliv jiná klávesa ukončí vstup. Volání výpisu (`write`) tatkéž vypisuje v binární podobě.

4.3 Jiný OS a emulátor QEMU

Pokud je program spouštěn na jiném OS než linux, pak bude pravděpodobně nutné spustit překlad a program jiným způsobem.

Jednoduchou cestou lze postupovat jako v případě linuxu, akorát drobně upravit skripty, aby na daném OS fungovaly. Oba skripty byly psány tak, aby byly co nejjednodušší, takže každý z těch skriptů obsahuje pouze jeden příkaz, který lze tedy zadat přímo do konzole nebo ho taktéž upravit.

Příkazy, které by se musely upravit jsou:

```
java -jar compiler.jar $1 $2
qemu-system-i386 -drive file=${1:-out.bin},format=raw
```

4.4 Spuštění přímo na HW

Tento krok nemohu doporučit, protože program bude spuštěn v tzv. reálném módu a může teoreticky poškodit nějaké SW vybavení počítače. Překlad programu se provede stejně jako je uvedeno výše, postup se liší až v případě spuštění.

Program se nakopíruje na začátek paměti bootovatelného média (např. flash nebo HDD), toho lze v linuxu dosáhnout např. příkazem `dd`. V BIOSu se případně nastaví bootování z toho media a následně počítač při bootování program spustí. Výhoda tohoto postupu je v tom, že program nebude emulován a bude se vykonávat rychleji. Nevýhodou je složitější postup spuštění a hlavně nebezpečí poškození SW daného počítače. Tento postup by měl provést pouze pokročilý uživatel se znalostí překládaného programu.

5 Struktura projektu

Projekt kromě tohoto souboru obsahuje pět složek, bashovsé skripty `*.sh`, testovací překládané soubory `*.pl0`, přeložený překladač `compiler.jar`, ANTLR jazykový processor `antlr.jar`, a markdownový soubor `README.md`.

5.1 Podsložky projektu

Projekt obsahuje pět podsložek. První složka je `bin`, kde jsou uloženy přeložené programy. Druhá složka `compiler` obsahuje zdrojové soubory překladače v jazyce Java. Třetí podsložkou je `kernel`, kde je zdrojový soubor zaváděného jádra v jazyce symbolických adres, který program spouští. Dále následuje složka `pictures` se snímky běhu testovacích programů a nakonec složka `shouldfail` obsahuje zdrojové soubory, které jsou sice syntakticky správně, ale nejsou správně sémanticky a nemělo by možné je přeložit, resp. překlad se ukončí chybou.

5.2 Bash skripty

V rootu projektu se nachází pět bashovsých skriptů. Dva z nich `compile.sh` a `run.sh` jsou již popsány v sekci 4. Dalším je skript `build.sh`. Ten se musí spustit v případě, že se mění zdrojové soubory překladače (`*.java`) nebo jádra (`loader.asm`). Dalším skriptem je `debug.sh`, který vypíše v jazyce symbolických adres, co vlastně program dělá. Posledním je `clean.sh`, který maže soubory které se generují při překladu.

5.3 Překládané programy

Projekt dodává pět předpřipravených testovacích zdrojových souborů s koncovkou `pl0`. Výstup těch nejzajímavějších je znázorněn na obrázcích 3 a 4 v příloze B, další obrázky jsou ve složce `pictures`.

5.4 Překladač

Spustitelný soubor překladače je soubor `compiler.jar`. K tomu aby fungoval potřebuje mít ve stejné pracovní složce i soubor `antlr.jar`

5.5 Vedení projektu

Projekt je verzován programem `git` a veden na webové stránce Githubu na adrese `https://github.com/cervenkam/fjp`.

6 Příklad překladu

Mějme minimalistický program uvedený v 1. Program pouze alokuje na zásobníku jednu proměnnou, přiřadí jí hodnotu 3 a tuto hodnotu také vypíše.

```
var x;
begin
        x:=3;
        write x;
end.
```

Obrázek 1: Překládaný program

Přeložením získáme strojový kód, který když převedeme do jazyka symbolických adres, bude vypadat jak je uvedeno v 2.

```
00000200 <.data+0x200>:
 200: 66 67 c8 04 00 01    addr32 enter 0x4,0x1
 206: 66 e9 00 00 00 00    jmp    0x20c
 20c: 66 89 e8             mov    eax,ebp
 20f: 66 2d 08 00 00 00    sub    eax,0x8
 215: 66 67 50             addr32 push eax
 218: 66 67 68 03 00 00 00 addr32 push 0x3
 21f: 66 67 58             addr32 pop eax
 222: 66 67 5b             addr32 pop ebx
 225: 66 67 36 89 03       mov    DWORD PTR ss:[ebx],eax
 22a: 66 89 e8             mov    eax,ebp
 22d: 66 2d 08 00 00 00    sub    eax,0x8
 233: 66 67 50             addr32 push eax
 236: 66 67 5b             addr32 pop ebx
 239: 66 67 36 8b 03       mov    eax,DWORD PTR ss:[ebx]
 23e: 9a 80 01 c0 07       call   0x7c0:0x180
 243: 66 67 c9             addr32 leave
 246: eb fe                jmp    0x246
```

Obrázek 2: Přeložený program, vygenerováno pomocí `debug.sh`

Proměnná se alokuje instrukcí `enter`, která pro 32bitovou proměnnou alokuje `0x4` bajty. Tato proměnná bude uložena na zásobníku na pozici `ebp-0x8`. Aby se proměnná nastavila na hodnotu 3, nejprve se načte její adresa a vloží se do zásobníku (první tři zelené instrukce), poté se do zásobníku vloží i přiřazovaná hodnota, tedy `0x3`, nakonec se hodnota i adresa ze zásobníku vyberou a hodnota se na adresu vloží.

V dalším příkazu výpisu se opět načte adresa proměnné `x` (první tři tmavě červené instrukce), poté se vrchol zásobníku dereferencuje a vloží do registru `eax` (následující dvě instrukce) a nakonec se zavolá výpis, o který se stará jádro a vypisuje hodnotu registru `eax`.

Nakonec se dealokuje proměnná `x` a program je „ukončen“ zacyklením na jedné instrukci.

7 Závěr

Co se týče negativních vlastností, překládaný jazyk nedisponuje formou přímého předávání parametrů. To je sice možné obejtít pomocí globálních a lokálních proměnných, ale dělá to kód nečitelným (viz program `hilbert.pl0`). Překladač taktéž neposkytuje závratné optimalizace. Většinu času pouze vkládá a vybírá ze zásobníku místo přímého výpočtu v registrech.

I přesto zde představený překladač dokáže přeložit i složitější program, například zde uvedený rekurzivní výpočet a vykreslení Hilbertovy křivky. Jazyk PL/0 byl rozšířen o několik syntaktických výrazů. Překladač byl otestován na několika programech s pozitivními výsledky a byl též testován na programy, které jsou sémanticky chybné, taktéž úspěšně. Program není závislý na žádném SW (kromě BIOSu) a je možné ho spustit na libovolném procesoru architekturou kompatibilní s i386.

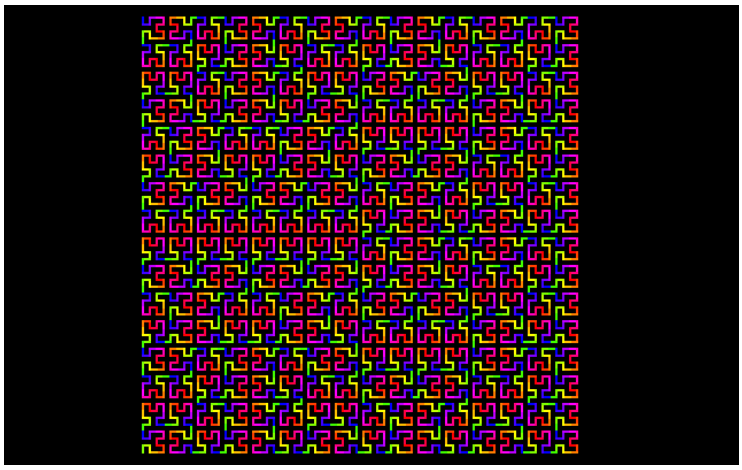
8 Reference

- Instrukce 80386: `https://pdos.csail.mit.edu/6.828/2007/readings/i386/c17.htm`
- Instrukce x86: `https://www.felixcloutier.com/x86/`
- Dokumentace ANTLR: `https://www.antlr.org/api/Java/index.html`
- Přerušení BIOSu: `http://www.ctyme.com/intr/int.htm`
- Jádro OS: `https://github.com/cervenkam/os`
- Github projektu: `https://github.com/cervenkam/fjp`

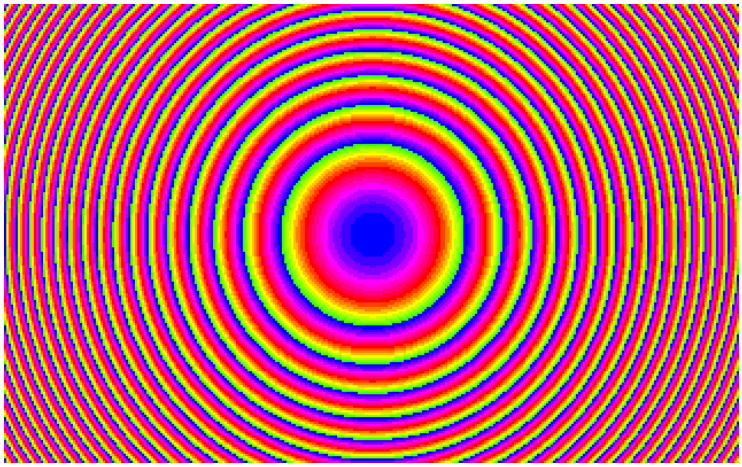
A Gramatika jazyka

```
program = block "." .
block = [ "const" ident "=" number {"," ident "=" number} ";"]
        [ "var" ident {"," ident} ";"]
        { "procedure" ident ";" block ";" } statement .
statement = [ ident ":=" logical
              | "call" ident
              | "read" ident
              | "write" expression
              | "begin" statement {";" statement } "end"
              | "if" condition "then" statement [ "else" statement ]
              | "goto" label
              | "execute" "0x" hexstring
              | "while" condition "do" statement ].
label = "label" ident.
condition = "odd" logical |
            logical ("="|"#"|"<"|"<="|">"|">=") logical .
logical = [ "not" ] expression
            | expression ("and"|"or"|"xor"|"left"|"right") logical.
expression = expression { ("+"|"-") term}
               | ["+"|"-"] term.
term = factor {("*"|"/") factor}.
factor = ident | number | "(" logical ")".
```

B Obrázky programů



Obrázek 3: Výstup programu `hilbert.pl0`.



Obrázek 4: Výstup programu `circle.pl0`.