Herocomp - Překladač jazyka Heroc

Tomáš Mikula

Srpen 2017

Obsah

1 Úvod

Herocomp je překladač pro jazyk Heroc, který je specifikovaný následujícím dokumentem (http://vychodil.inf.upol.cz/kmi/prkl/heroc-2016.pdf).

Překladač je implementován v jazyku Python 3.6. Parser byl vytvořen za pomoci nástroje ANTLR 4 s podporou exportu do jazyka Python 3.6.

Jak již dokumentace jazyka uvádí cílová platforma je assembler pro platformu x86–64 v notaci GAS. Který je poté sestavitelný programem `gcc` (`yasm` nebyl v mém případě testován).

2 Specifikace jazyka

2.1 Jazyk Heroc

Jak již s výše uvedené specifikace vyplývá jazyk Heroc je podmnožinou jazyka C s drobnými rozdíly. Jediným datovým typem je datový typ `long`. Operátor reference lze použít s libovolnou proměnnou nebo funkcí a výsledná hodnota je ukazatel na proměnnou nebo funkci reprezentovaný hodnotou `long`. Inicializace pole je možná za pomoci výrazu `{x,y,z}`, který vrací ukazatel na výsledné pole.

2.2 Má implementace jazyka Heroc

Má implementace jazyka Heroc nepodporuje líné vyhodnocování. Globální pole jsou deklarovány po vzoru `gcc` za pomoci `.comm identifikátor, délkapole*8, 32`. Do primitivní globální proměnné je stále možné přiřadit alokované pole.

3 Požadavky na překlad a spuštění překladu

3.1 Požadavky na spuštění překladače

Překladač byl testován s následujícími požadavky:

- Python 3.6
- Knihovna `antlr4-python3-runtime==4.6` a její potřebné knihovny
- OS (překlad): MacOS Sierra 10.12.5
- OS (sestavení): Debian 9.0 (Docker)
- gcc version 6.3.0 20170516 (Debian 6.3.0-18)
- Cílová platforma x86–64

Ke zprovoznění překladače doporučuji použít virtuální prostředí a balíčkovač `pip`. Veškeré další příkazy se spouští z vytvořeného prostředí.

3.2 Spuštění překladu

K přeložení zdrojového souboru `in.heroc` do výstupního souboru `out.s` je možné použít následující příkaz:

```
python3.6 main.py in.heroc > out.s
```

Překladač zapisuje na standardní výstup a standardní chybový výstup. Je tedy možné použít příkaz ve tvaru:

```
python3.6 main.py in.heroc > out.s 2> errorout
```

Poté je možné přeložený soubor složit za pomoci `gcc`:

```
gcc -m64 -o output out.s herocio.c
```

Jak je z příkazu patrné je nutné přiložit soubor `herocio.c`, který obsahuje základní sadu funkcí jazyka Heroc.

4 Překladač

4.1 Souborová struktura překladače

Základní popis souborové struktury přiloženého překladače.

4.1.1 Složka `documentation`

Složka obsahující plnou specifikaci jazyka Heroc a tento dokument.

4.1.2 Složka `heroc-examples`

Složka obsahující příklady od doc. Vychodila. Některé příklady byly opraveny, jedná se především o problém indexování pole (`b + 1` vs. `b + 1 * sizeof(long)`). Složka obsahuje následující typy souborů:

- Soubory `exampleXX.heroc` obsahující zdrojové kódy příkladů v jazyce Heroc.
- Soubory `exampleXX.s` obsahující assembler kód programu `exampleXX.heroc`.
- Soubory `exampleXX.txt` obsahující očekávaný výstup programu `exampleXX.heroc`.
- Soubory `exampleXX.out.txt` obsahující výstup programu `exampleXX.heroc` přeloženého překladačem Herocomp.

Překladač byl během vývoje testován na různých modifikacích výchozích příkladů, ty jsem si však bohužel průběžně neukládal, tudiž je nemohu k překladači přiložit.

4.1.3 Složka `herocomp`

Samotný překladač Herocomp v jazyce Python 3.6. Jeho struktuře se budu věnovat později.

4.1.4 Ostatní soubory

Projekt obsahuje následující soubory, ostatní soubory nejsou podstatné pro chod překladače:

- Soubor `Heroc.g4` obsahující gramatiku parseru pro jazyk Heroc pro nástroj ANTLR 4.
- Soubor `Lexer.g4` obsahující lexer jazyka Heroc pro nástroj ANTLR 4.
- Script `compile-all-examples.sh`, který přeloží veškeré příklady ve složce `heroc-examples`. Tento script je nutné pouštět jako první v pořadí.
- Script `run-all-examples.sh`, který složí a spustí veškeré příklady ve složce `heroc-examples`. Tento script je nutné pouštět jako druhý v pořadí.

4.2 ANTLR 4

Nástroj ANTLR 4 byl použit k vygenerování lexeru a parseru pro jazyk Heroc. Vygenerované Python kódy se nachází v kořenovém adresáři překladače `herocomp`. Jedná se o následující soubory:

- Soubor `Heroc.tokens`, ve kterém jsou uložené jednotlivé tokeny.
- Soubor `HerocLexer.py`, ve kterém je kód samotného lexeru.
- Soubor `HerocParser.py`, ve kterém je kód samotného parseru.
- Soubor `HerocVisitor.py`, ve kterém je originální neupravený visitor pro procházení vygenerovaného stromu. Slouží pouze jako šablona pro můj vlastni visitor, viz níže.

4.3 Abstraktní syntaktický strom

ANTLR 4 umí pro konkrétní zdrojový soubor vygenerovat Concrete Syntax Tree (CST), tento strom je nutné upravit do podoby/vytvořit Abstract Syntax Tree (AST). V tomto procesu nejvíce figurují následující soubory nacházející se ve složce `herocomp/tree`:

- Soubor `TreeVisitor.py`, ve kterém je mnou napsaný TreeVisitor založený na kostře ze souboru `HerocVisitor.py`
- Soubor `AST.py`, který tvoří samotný AST strom.

AST je stavěn postupným procházením CST stromu od kořene k listům. Původní CST se nemodifikuje. `TreeVisitor` pozná jaké uzly má do AST zahrnout a jaké může přeskočit. V tomto bodě se staví finální podoba AST stromu za pomoci třídy `Node` a jejich potomků. Tríd a `Node` a její potomci se nachází ve složce `herocomp/tree`, přesné struktuře se budu věnovat níže.

4.4 Trída `Node` a její potomci

Každá třída dědící od třídy `Node` obsahuje:

- Hodnotu `parent` pro uchování rodiče uzlu.
- Hodnotu `statements` pro uchování seznamu potomků uzlu.
- Hodnotu `depth` pro hezký výpis AST stromu.
- Hodnotu `number` pro uchování unikátního čísla uzlu, nutné především pro generování labelu pro cykly.
- Metodu `__str__()` (pouze její potomci) pro získání human-friendly printu daného uzlu.
- Metodu `get_code()` (pouze její potomci) pro získání ASM kódu daného uzlu.

4.5 Generování kódu

Generování kódu je zahájeno zavoláním metody `get_code()` v kořeni AST. Jednotlivé uzly volají své vlastní `get_code()` implementace a postupně vrací celkový ASM kód programu.

V následující sekci bych se rád věnoval základním konstrukcím, které generování kódu používá.

4.5.1 Tabulka proměnných

Na úrovni každého bloku kódu se nachází jednoduchá tabulka proměnných, které mají v aktuálním bloku platnost. Řádek tabulky se skládá s názvu proměnné a její adresy. Této tabulky se využívá k zajištění správné funkčnosti platnosti hodnot mezi bloky. Během předávání argumentů funkci jsou argumenty rovněž uloženy v této tabulce.

Speciálním případem je tabulka globálních hodnot, která je uchovávána v uzlu `Program` a z důvodu implementace globálních polí uchovává i typ proměnné (variable nebo array). Nutnost této implemetance jsem objevil během příkladu, kdy je do normální globální proměnné uložena adresa lokálního pole. S takovou globální proměnnou je poté nutné pracovat jako s normálním lokálním polem. V opačném případě, ve kterém je globální proměnná definovaná jako pole, je nutné s takovou proměnnou pracovat jiným způsobem.

Tato odlišnost bude s největší pravděpodobností způsobena použitím konstruktu `.comm identifikátor, délkapole*8, 32`.

4.5.2 Konvence volání funkce

V mé implementaci je použita klasická x86-64 konvence volání funkce. Prvních šest argumentů je předáno pomocí registrů a zbytek je předán za pomoci zásobníku. Před voláním funkce jsou rovněž na zásobník uloženy registry, u kterých může nastat změna obsahu během volání funkce. Tyto registry jsou následně obnoveny.

Při volání funkce je rovněž ošetřen případ, kdy je v heroc kódu použit ukazatel na funkci. Pro tento účel je v uzlu `Program` uložena tabulka názvů funkcí. Díky této tabulce překladač pozná, zda je identifikátor na funkci platný, nebo zda má oznámit, že se překladač pokouší volat funkci, která neexistuje.

4.5.3 Binární operace

Binární operace a jejich argumenty jsou během překladu umisťovány na pomocný zásobník odkud jsou postupně vyhodnocovány. Tento mechanizmus spolu s návrhem gramatiky zaručuje správné vyhodnocení matematických a jiných výrazů.

Původně jsem v AST rozlišoval jednotlivé operace vlastním uzlem, to se z perspektivy generování kódu ukázalo jako zbytečné. Z tohoto důvodu jsem zachoval pouze dva typy uzlů, binární a unárni operace. Každý z těchto uzlů obsahuje pouze typ operace a její argumenty.

Jednotlivé implementace operací (jak binárních tak unárních), jsou umístěny v souboru `OperationType.py` v adresáři `herocomp/tree/nodes/operations/`.

4.5.4 Přiřazení

Implementace přiřazení se nachází v souboru `Assignment.py`, který se nachází ve složce `herocomp/tree/nodes/`. Je zde ošetřen případ přiřazení do běžné proměnné a do proměnné pole.

V případě složeného přiřazení a inkrementace `+=` (a jiné), se vytvoří v AST dočasný uzel a je simulována binární operace a poté primitivní přiřazení.

4.5.5 Pole a řetězce

Prvky pole jsou na zásobníku uloženy za sebou a v opačném pořadí. Řetězce jsou implementovány totožně. Po uložení prvků na zásobník je vypočítána adresa prvního prvku a uložena do proměnné. Proměnná pole tedy obsahuje adresu prvního prvku v poli/řetězci.

5 Závěr

Na základě zkušenosti získané psaním tohoto překladače bych volil vhodnější strukturu AST stromu s ohledem na generování kódu. I tak považuji získané zkušenosti za velmi přínosné. Díky použití ANTLR 4 je počáteční práce na gramatice jazyka a vytvoření parseru příjemnější. Rovněž jsem ocenil možnost generování parseru z gramatiky jazyka do několika cílových jazyků.

Volbu Pythonu 3.6 vidím do akademického projektu jako dobrou. Je viditelné, že překladač napsaný v Pytohnu je znatelně pomalejší, než překladače mých kolegů (kteří použili rovněž ANTLR 4). Na druhou stranu je příjemné psát překladač v jazyce jako je Python a s určitou zkušeností a lepší optimalizací by věci fungovaly rychleji a lépe.

Jsem rád, že mi tato zkušenost umožnila pochopit funkci překladače o něco lépe. Rovněž je tato zkušenost přínosná s ohledem na jazyk assembler a jazyk C.