Implementace překladače jazyka heroc

Martin Jašek

5. srpna 2016

Abstrakt

Tento text slouží jako zevrubná dokumentace k mé implementaci překladače programovacího jazyka heroc.

Obsah

1 Úvod

Překladač jazyka heroc je jednoduchá implementace překladače jazyka na bázi jazyka C. Kód v jazyce heroc překládá do GNU assembleru („GAS“).

2 Popis jazyka a překladače

2.1 Specifikace jazyka heroc

Jazyk nemá oficiální specifikaci, pro popis jazyka slouží adresář s ukázkovými `*.heroc` soubory.

Z požadavků však vyplývá, že jazyk heroc je podmnožina jazyka C. Jazyk heroc však má podporovat pouze datový typ `long`, standardní řídící konstrukce (kromě příkazu `switch`) a operátory, procedury a pole. Také jazyk musel obstojně zvládat práci s ukazateli.

2.2 Mé vlastnosti navíc

Má implementace oproti doporučení obsahuje několik vlastností navíc. Ze základních k nim patří podpora jednořádkových komentářů a znakových konstant. Z pokročilejších pak možnost inicializovat globální proměnné hodnotou libovolného (ne nutně konstantního) výrazu nebo podpora jednoduchých lambda-výrazů.

Ukázky jejich použití jsou v adresáři s příklady.

2.3 Požadavky na sestavení

Překladač byl naprogramován v jazyce C (standart C11) za použití standartních nástrojů lex (flex) a yacc (bison). Požadavky pro sestavení (systém, na kterém byl překaldač vyvíjen a je odladěn) jsou následující:

- platforma GNU/Linux, architektura x86_64
- GCC verze 4.9.2
- flex verze 2.5.39
- bison verze 3.0
- make verze 4.0

Kromě starších verzí bisonu (verzi 2.5 už použít nelze) by měl jít překladač sestavit i s jinými verzemi nástrojů. Při sestavování na jiné platformě je nutné počítat s tím, že generovaný assembler je vždy generován pro procesor x86_64.

2.4 Sestavení a spuštění

Sestavení překladače se provádí standartně pomocí make:

```
$ make
```

Sestavit je možné jen testovací programy pomocí cíle `tests`, nebo naopak jen samotný překladač (bez testů, cíl `compiler`):

```
$ make tests
$ make compiler
```

V Makefile je také možné upravit chování překladače (aktivovat ladící logy nebo změnit výstupní jazyk).

Make vytvoří adresář `bin` a v něm binární soubor překladače `compiler`. Po spuštění příkazem:

```
$ ./bin/compiler
```

očekává na standartním vstupu kód v jazyce heroc a na standartní výstup vypíše vygenerovaný assembler. Případná informativní, chybová nebo varovná hlášení jsou vypisována na standardní chybový výstup. Překladač je také možné spustit s explicitním uvedením vstupního a výstupního souboru:

```
$ ./bin/compiler input.heroc output.s
```

2.5 Adresářová struktura

Překladač je tvořen následujícími adresáři:

- `src` – zdrojové kódy (hlavičkové a `*.C` soubory, soubor lexéru a gramatiky)
- `test` – zdrojové kódy testovacích programů
- `examples` – ukázkové `*.heroc` soubory
- `lib` – dvojice souborů s knihovnou herocio (`herocio.c` a `herocio.scm`)
- `test-scripts` – sada shellových skriptů především pro (dávkové) testování

Při sestavení dále vzniknou následující adresáře:

- `gen` – vygenerované soubory lexéru a syntaxéru
- `obj` – zkompilované objektové soubory
- `bin` – adresář s binárním souborem překladače
- `test-bin` – adresář s binárními soubory testovacích programů
- `tmp` – pomocný adresář pro pozdější použití (pro samotný překlad nepotřebný)

3 Komponenty překladače

3.1 Tokeny

Základním stavebním kamenem mé implementace jsou tokeny. Kromě toho, že slouží pro předávání informací mezi lexérem a sytaxérem a dohromady tvoří ast (abstraktní syntaktický strom), jsou použity také k uchovávání metainformací generovaných při sémantické analýze.

Tokenů je využito několik druhů, všechny (bez ohledu na to, kde a jak jsou využívány) jsou deklarovány na jednom místě, v deklarační části syntaxéru. Každý token má prefix názvu udávající jeho druh. V programu tak lze nalést tokeny s prefixy:

- `JLT` (just lexical token), tokeny generované lexérem a zpracovávané syntaxérem. Dále již nejsou využívány, patří sem tedy pouze tokeny ryze lexikální, tedy závorky, oddělovače (a další speciální znaky čárka, dvojtečka a otazník) a tzv. multi-used tokeny. To jsou tokeny, reprezentující tytéž lexémy avšak gramatika je interpretuje různě. To jsou tokeny plus, mínus, hvězdička, inkrementace a dekrementace.
- `ATT` (atomic token), atomické tokeny. Sem patří číslo a textový řetězec (od syntaxéru dále použit pouze pro název identifikátoru).
- `STK` (special token), pro lexér tokeny klíčových slov, dále pak uzly odpovídajících řídících struktur (např. if, for, lambda, continue, return).
- `JST` (just semantic token), tokeny, které jsou naopak od JLT používány až od syntaxéru. Jedná se o tokeny, které zaštiťují složitejší syntaktické konstrukce (jako proměnná, pole, deklarace procedury, volání procedury a podob.).
- `CNT` (container token), tokeny, jejiž potomky bývají seznamy dalších tokenů. Jedná se o tokeny: „příkazy“, „čísla“ (nepoužívá se), „výrazy“ a „seznam parametrů procedury“.
- `META` (metatokeny), tokeny, které obsahují dodatečné informace ohledně ast (např. odkaz na deklaraci proměnné, cyklus k ukončení).
- `OPT` (operator token), token popisující operátor/operaci.

3.2 Databáze tokenů

Každý token, pokud nereprezentuje přímo konkrétní lexém (u těch je to zjevné) má přiřazenu textovou reprezentaci. Tabulka (resp. funkce pro převod) všech tokenů na jejich textové reprezentace (a u některých i zpět) se nachází v souboru `tokens.c`.

3.3 Lexér

Lexikální analyzátor neboli lexér je generován pomocí programu flex. Z důvodu úspory kódu lexéru jsou lexémy rozpoznávány po skupinách a následně (je-li to nutné) pomocí databáze tokenů převáděny na konkrétní tokeny. Mezi tyto skupiny patří: klíčová slova, čísla, identifikátory, přiřazovací operátory, operátory, závorky, ostatní symboly. Dodatečný C kód je vytlačen do souborů `lexer-headers.h` a `lexer-headers.c`.

V případě lexikální chyby je vypsána varovný hláška a lexér se daný chybný token pokusí přeskočit.

3.4 Konstruktor ast

Soubor `ast.c` obsahuje konstruktory pro uzly ast (abstraktního syntaktického stromu). Každý vygenerovaný uzel má kromě typu svého tokenu a jeho hodnoty (kterou může být číslo, řetězec, operátor nebo seznam dalších uzlů) také jednoznačný identifikátor (a vzhledem k tomu, že sám může být součástí nějakého seznamu uzlů, tak ukazatel na následníka).

Jednoznačný identifikátor uzlů slouží jednak k přehlednější práci se stromem a navíc také pro generování labelů u assembleru. Je implementován pomocí statické proměnné.

Seznam uzlů je standartní jednosměrný lineární seznam zakončený ukazatelem na `NULL` následníka. Může být prázdný.

3.5 Syntaxér

Syntaxér, neboli syntaktický analyzátor, definuje samotnou gramatiku jazyka. Sémantické akce každého pravidla bývají obvykle jen volání některého z konstruktorů ast. Kde to má smysl tak je také změněna hodnota naposledy zpracovaného uzlu na právě zpracovaný, takže při syntaktické chybě uživatel dostane alespoň skromnou informaci o tom, kde mohlo dojít k chybě. Podobně jako lexér má syntaxér dodatečný C kód přesunut do souborů `syntaxer-headers.h` a `syntaxer-headers.c`.

3.6 Zobrazování ast

Pro vyobrazení struktury ast byl vytvořen soubor `ast-displayer.c`, obsahující funkce pro vypsání stromu na zadaný proud. Rekurzívně prochází strom a na řádky vypisuje jednotlivé potomky, u každého uzlu vypíše jeho adresu, jednoznačný identifikátor a textovou reprezentaci tokenu.

3.7 Výstup ast do ...

Pro lepší ladění byl zaveden koncept „výstupu ast do ...“. Na počátku, ve fázi vývoje syntaxéru bylo nutné sledovat pouze hrubě vygenerovaný ast. Proto vznikla komponenta výstupu ast v základním formátu (`ast-basic-exporter.c`). Spolu s ním byla také snaha vyobrazovat ast v syntaxi jazyka Scheme a vznikl proto `ast-scheme-exporter.c`. Pro účely testování generátoru zásobníkového kódu poté vznikla komponenta `ast-stackode-exporter.c`, která vypisuje stackode program syntaxí jazyka Scheme. Na závěr byl pochopitelně zkonstruován také `ast-gas-exporter.c`, tedy komponenta, která ast převádí na assembler.

Jaký „výstupní jazyk“ má být při překladu použit se specifikuje přilinkováním adekvátního objektového souboru a specifikuje se to makrem v Makefile.

3.8 Sémantér

Sémantér, neboli sémantický analyzátor, má za úkol projít hrubý ast a doplnit (nebo modifikovat) informace v něm, aby následný generátor kódu měl co nejméně práce. Sémantér má tedy na starosti:

- přidává do ast umělé deklarace předdefinovaných procedur (`print_long`, `print_char` a `print_nl`)
- přidává explicitní volání funkce `main`
- u každé proměnné doplní odkaz na její definici
- při každé definici proměnné dodá odkaz na předchozí definici a adresu proměnné na zásobníku

- tam, kde je použita hodnota proměnné namísto její reference, ji obaluje dereferencí
- inkrementační a dekrementační operátory a operátor indexace expanduje na odpovídající výrazy
- u klíčových slov `break` a `continue` přidává metainformaci s odkazem na cyklus, ke kterému se vztahují
- u volání procedur (pokud to jde) kontroluje cíl volání a aritu
- u deklarace procedury předpočítává adresy parametrů
- řeší alokaci polí na zásobníku

3.9 Generátor zásobníkového kódu

Generátor zásobníkového kódu, neboli stackode, převádí ast na zásobníkový assembler. Pro účely testování je možné jej převést do textové podoby v syntaxi jazyka Scheme. Zásobníkový kód obsahuje následující instrukce:

- `SKI_COMMENT` *komentář*, vloží komentář
- `SKI_LABEL` *label*, vloží label
- `SKI_CALL`, provede zavolání funkce na vrcholu zásobníku
- `SKI_RETURN`, provede návrat z volání, na vrcholu zásobníku očekává návratovou hodnotu
- `SKI_CLEANUP_AFTER_CALL` *počet*, odklidí ze zásobníku argumenty volání
- `SKI_JUMP_ON_ZERO` *label*, skočí, pokud je na vrcholu zásobníku 0
- `SKI_JUMP_ON_NON_ZERO` *label*, skočí, pokud není na vrcholu zásobníku 0
- `SKI_JUMP_TO` *label*, skočí na uvedený label
- `SKI_LOAD`, provede načtení hodnoty z adresy na vrcholu zásobníku
- `SKI_ASSIGN`, uloží hodnotu pod vrcholem zásobníku na adresu na vrcholu zásobníku
- `SKI_DECLARE_ATOMIC`, vloží na zásobník jednu buňku s nedefinovanou hodnotou
- `SKI_DECLARE_ARRAY` *počet*, vloží na zásbník zadaný počet buněk s nedefinovanými hodnotami
- `SKI_PUSH_CELL_SIZE`, vloží na zásobník velikost jedné buňky
- `SKI_PUSH_CONSTANT` *číslo*, vloží na zásobník číselnou konstantu
- `SKI_PUSH_LABEL_ADRESS` *label*, vloží na zásobník adresu labelu
- `SKI_PUSH_RELATIVE_ADRESS` *adresa*, vloží na zásobník relativní (vzhledem k rámci) adresu

- `SKI_PUSH_ABSOLUTE_ADRESS` *adresa*, vloží na zásobník absolutní (od dna zásobníku) adresu
- `SKI_POP`, odebere a zahodí hodnotu na vrcholu zásobníku
- `SKI_DUPLICATE`, zduplikuje hodnotu na vrcholu zásobníku
- `SKI_UNARY_OPERATION`— *operace*, s hodnotou na vrcholu zásobníku provede operaci
- `SKI_BINARY_OPERATION` *operace*, s dvěmi hodnotami na vrcholu zásobníku provede operaci
- `SKI_INVOKE_EXTERNAL` *název*, *arita*, zavolá externí funkci s uvedeným počtem parametrů na zásobníku
- `SKI_END`, signalizuje (úspěšné) ukončení programu

3.10 Generátor assembleru

Generátor assembleru, neboli komponenta `ast-gas-exporter.c` překládá program ve stackode do GAS assembleru. Kód nebyl nijak optimalizován a je proto silně neoptimální. Do generovaného assembleru jsou vkládány komentáře – jednak popis stackode instrukce ale také text komentářů z instrukcí `SKI_COMMENT`. Assembler je generován přímo (jako text), bez jakékoliv další interní formy.

4 Testování a ladění

4.1 Ladící logy

Při sestavování překladače je možné zapnout funkci ladících logů pro jednotlivé komponenty (lexér, syntaxér, sémantér, stackode, gas). Logování se zapíná deklarací příslušného makra v Makefile překladače. Logy jsou vypisovány na standartní výstup.

4.2 Testovací `*.heroc soubory`

Základním kamenem testování překladače jsou testovací soubory. Všechny jsou umístěny v adresáři `examples`. Podle prefixu názvu souboru je lze rozdělit do následujích skupin:

- `vychodil*.heroc`, původní testovací soubory od doc. Vychodila
- `me*.heroc`, mé vlastní testovací soubory (většinou testující konkrétní implementovanou dílčí funkci)
- `x-*.heroc`, soubory protipříkladů, tedy soubory obsahující záměrně chyby
- `f-vychodil*.heroc`, modifikované zdrojové kódy doc. Vychodila tak, aby fungovaly v mé implementaci

4.3 Testovací programy a skripty, testování ve schemu

Prakticky každá komponenta překladače získala pro své ladění vlastní samostatně spustitelný program jehož drojový kód se nachází v adresáři `test-src`. Pro automatizaci testování, především pro spouštění testování nad více soubory v dávce, nebo pro rychlou detekci chyb a pádů vznikly v adresáři `test-scripts` různé testovací skripty.

Lze také automatizovaně spouštět přeložený kód. První variantou je spouštění přímo vygenerovaného scheme kódu. Tato technika však není doporučována vzhledem k tomu, že „běhová knihovna“ `eval-generated.scm` není důsledně odladěna a nejsou v ní zaneseny poslední změny v překladači.

Druhou (doporučenou) variantou je spouštění přeloženého stackode. Tato technika (implementována skriptem `run-generated-stackode.sh` za pomoci běhovné knihovny `stackode-evaluator.scm`) byla hlavním nástrojem pro vývoj a ladění překladače a může sloužit jako plnohodnotný výstup. Značnou nevýhodou, se kterou je často potřeba počítat je (oproti běhu kompilovaného assembleru) rapidní pokles rychlosti běhu.

Posledním nástrojem je tedy pochopitelně překlad do assembleru a jeho následný překlad a spouštění. Tuto funkcionalitu obstarává skript `run-generated-gas.sh`.

5 Další vývoj

Mezi další úkoly, které by bylo vhodné vyřešit patří:

- vyřešit konflikty v gramatice syntaxéru
- zrefactorovat ast za účelem kompletního zapouzdření uzlů (nejen konstruktory, ale i selektory)
- vylepšit reportování chyb
- přidat podporu pro více platforem (vč. generování assembleru v Intel syntaxi)
- doimplementovat optimalizace
- řádně otestovat

6 Závěr

Výsledkem je funkční překladač jednoduchého programovacího jazyka.

Nicméně oproti konkurenčnímu jazyku C má spíše nevýhody (značně méně funkcí, optimalita, absence vývojových a vývojářských nástrojů, ne 100% odladěnost), takže se nedá předpokládat jeho nějaké zásadní používání.