Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Interpret Scheme pro Z80

2015

Jan Janeček

Vedoucí práce:
Doc. RNDr. Vilém Vychodil, Ph.D.

Studijní obor:
Informatika, prezenční forma

Bibliografické údaje

Autor:	Jan Janeček
Název práce:	Interpret Scheme pro Z80
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2015
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	Doc. RNDr. Vilém Vychodil, Ph.D.
Počet stran:	31
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Jan Janeček
Title:	Scheme interpreter for Z80
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2015
Study field:	Computer Science, full-time form
Supervisor:	Doc. RNDr. Vilém Vychodil, Ph.D.
Page count:	31
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Předmětem práce je implementace interpretu fragmentu jazyka Scheme pro procesor Z80. Text popisuje implementaci elementů, některých procedur a optimalizace koncové rekurze. Interpret je implementován v assembleru a je navržen tak, aby byl snadno přenositelný. Podporuje vysokoúrovňové konstrukty Scheme jako jsou makra a procedury jako elementy prvního řádu.

Synopsis

The aim of this thesis is implementing an interpreter of a subset of Scheme language for the Z80 processor. This text describes implementation of elements, some of the procedures and a tail call optimization. The interpreter is implemented in assembly language, and is designed to be easily portable. The interpreter supports high-level Scheme constructs such as macros and procedures as first-class citizens.

Klíčová slova: Scheme, interpret, Z80, assembler

Keywords: Scheme, interpreter, Z80, assembler

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam zdrojových kódů

1 Úvod

Cílem práce bylo implementovat interpret jazyka Scheme pro procesor Zilog Z80. Procesor Z80 je odvozený z procesoru Intel 8080 a má osmibitové registry, které lze používat ve dvojicích jako šestnáctibitové a může tak adresovat až 64KB paměti. V 80. letech byl procesor často používán v osobních počítačích jako například ZX Spectrum. Zpravidla byl pro tyto počítače dostupný interpret BASICu.

BASIC je velice nízkoúrovňový a dalo by se říct nepřátelský k funkcionálnímu programování. Jeho procedury nepodporují návratové hodnoty. Práce si klade za cíl ukázat, že i Scheme je na této platformě možné implementovat dostatečně dobře pro běh menších programů.

Pro MZ-800 existuje velice omezená Lispová implementace Tiny Lisp. Pro ZX Spectrum existuje implementace SpecLisp, ta však nepodporuje například uživatelská makra a anonymní procedury. K těmto interpretům navíc není dostupný zdrojový kód, což komplikuje případné přenesení na jinou platformu.

Primárním cílem mé implementace interpretu byl počítač Sharp MZ-800, ale interpret je snadno přenositelný na jiné platformy s procesorem Z80, viz. 4.

1.1 Omezení

Předmětem práce bylo implementovat fragment jazyka Scheme. Nejedná se tedy o implementaci R5RS – taková implementace by byla příliš velká a nevhodná pro Z80, vzhledem k jeho paměťovým a výkonostním omezením. Proto bych chtěl předem upozornit na omezení a odlišnosti od běžného Scheme, které budou pro případného uživatele nejviditelnější:

1. Operace s integery přetékají.
2. Zabudované procedury nejsou navázány na své názvy. Symbol musí být (podobně jako v Lispu) na prvním místě seznamu, aby se choval jako zabudovaná procedura. Pokud chce uživatel zabudované procedury mapovat nebo je z jiného důvodu předávat jako element, je možné je „obalit“ uživatelskou procedurou.
3. Procedury (uživatelské i zabudované) podporují maximálně 255 argumentů.
4. Symboly a reprezentace pokračování mohou mít maximálně 252 bytů.
5. Při příliš hluboké rekurzi může přetéct zásobník. Narozdíl od většiny chyb (jako například nedostatek paměti), se z této interpret neumí vrátit do REPLu, protože přepsání paměti zásobníkem vede k nepředvídatelnému chování.
6. V názvech zabudovaných procedur a maker jsou všechna písmena velká (protože MZ-800 píše implicitně velká písmena a navíc jsou na této platformě malá písmena kódovaná jinak, než ve standardním ASCII).

7. Na MZ-800 je omezení délky vstupního výrazu 80 znaků, kvůli omezení procedury MONITORu, která je použita pro vstup, viz 3.3.

8. Implementovanými elementy jsou jen prázdný seznam, páry, symboly, procedury, makra, pokračování, pravdivostní hodnoty, celá čísla a čísla s plovoucí desetinnou čárkou. Implementován je také vektor bytů, ale je odlišný od vektoru v běžném Scheme, má pouze speciální použití, viz 3.1.8.

9. Seznam všech implementovaných procedur a maker je v kapitole 3.6.

2 Použití

Interpret je možné sestavit ze zdrojových kódů v adresáři `scheme/`, ale pro jednodušší použití jsou ve stejném adresáři také předkompilované binární soubory pro MZ-800 i MSX.

2.1 Sestavení interpretu

Závislosti:

- z80asm
- GNU make
- GNU bash
- echo
- cat
- Perl (jen u verze pro MZ-800)
- kompilátor C

Verze interpretu pro MZ-800 se sestaví příkazem `make`. Verze pro MSX se sestaví příkazem `make openmsx`.

Verze pro MSX lze také sestavit pouze příkazem `z80asm scm-openmsx.asm`. U verze pro MZ-800 je třeba sestavit hlavičku souboru, doporučuji tedy použít příkaz `make`. O samotné sestavení hlavičky se stará skript `mkheader.pl`, původně napsaný doc. Vychodilem.

2.2 Emulátory

2.2.1 mz800em

Ke spuštění verze pro MZ-800 lze použít emulátor mz800em (ke stažení na http://sourceforge.net/projects/mz800em/). Na stránkách jsou dostupné jen zdrojové kódy.

Verze pro GNU/Linux (`gmz800em`) závisí na GTK verze 1, která dnes není v distribucích běžná. Pro usnadnění použití na GNU/Linuxu je tedy v adresáři `scheme/` sestavený spustitelný soubor emulátoru pro architekturu x86, který je staticky slinkovaný se všemi závislostmi kromě GNU C knihovny. Je tedy potřeba jen 32-bitová (i386) knihovna GLIBC verze 2.15 nebo vyšší.

Tento emulátor také potřebuje proprietární obsah ROM počítače MZ-800. Konkrétně jde o soubory `mz700.rom`, `mz700fon.dat` a `mz800.rom`. Je třeba je umístit do `/usr/local/lib/`.

Emulátor je třeba spustit v adresáři `scheme/`, kde se nachází sestavený soubor `scm.mzf`. Po spušťení emulátoru se klávesou `M` spustí MONITOR. V MONITORu se příkazem `L SCM` interpret načte do paměti a spustí. Po vypsání zprávy `READY!` je možné zadávat výrazy.

Je nutné mít nastavené americké rozložení klávesnice kvůli způsobu, jakým emulátor zpracovává vstup. Klávesy jsou pak v emulátoru namapované přibližně na stejných místech, jako na počítači MZ-800. Znaky `@` a `` ` `` jsou namapovány na klávesu F7.

2.2.2 openmsx

Emulátor openmsx je ke stažení na stránkách http://openmsx.sourceforge.net/, a to jak binární soubory pro několik platforem, tak zdrojové kódy. V distribucích GNU/Linuxu je často dostupný v repozitářích.

Verze interpretu pro MSX se spustí příkazem `openmsx -cart openmsx.bin`. Po vypsání zprávy `READY!` je možné zadávat výrazy. Malá písmena ve vstupním výrazu jsou automaticky převedena na velká.

3 Implementace

Implementačním jazykem je assembler pro Z80 (instrukční sada je dobře zdokumentovaná [1][2]). Alternativou se srovnatelným výkonem by mohl být jazyk C. Pro Z80 existuje kompilátor SDCC, ale kód, který generuje, je znatelně delší a méně efektivní v porovnání s ručně psaným assemblerem.

Na uvedeném příkladě funkce plus je na první pohled vidět neefektivní volací konvenci, díky které je potřeba mnohem více instrukcí a práce se zásobníkem, než kdyby se hodnoty jednoduše předaly v registrech.

```
1  short plus(short a, short b)
2  {
3          return a + b;
4  }
```

Zdrojový kód 1: Funkce plus v C

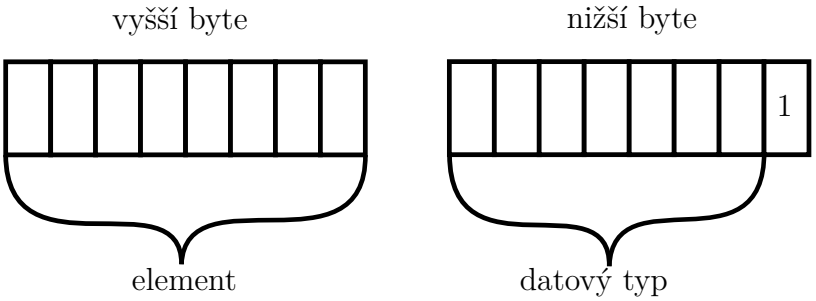
```
1  _plus:
2          push    ix
3          ld      ix,#0
4          add     ix,sp
5          ld      a,4 (ix)
6          add     a,6 (ix)
7          ld      l,a
8          ld      a,5 (ix)
9          adc     a,7 (ix)
10         ld      h,a
11         pop     ix
12         ret
```

Zdrojový kód 2: Výstup kompilace funkce plus pomocí SDCC

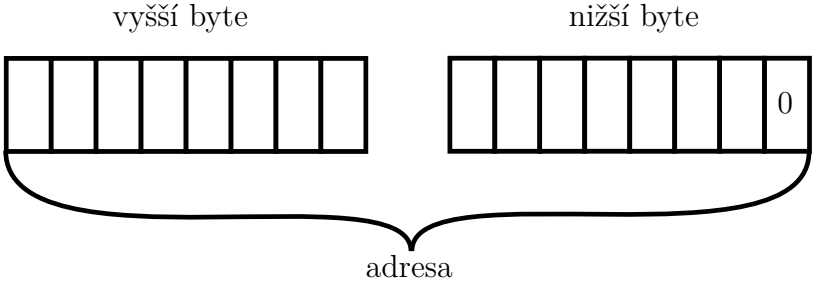
3.1 Reprezentace elementů

Elementy jsou reprezentovány šestnáctibitovým číslem, které je buď zakódováním celého elementu, nebo je ukazatelem na místo v paměti, kde je reprezentace elementu alokovaná.

Protože je paměť alokovaná po slovech, všechny adresy elementů jsou sudé, tedy v binární soustavě končí nulou. Díky tomu můžeme jednobytové elementy uložit přímo do vyššího bytu, přičemž nejnižší bit nižšího bytu je 1 a zbytek nižšího bytu je použit k identifikaci datového typu.



Vícebytové elementy je potřeba alokovat v paměti. Bylo by paměťově neefektivní uchovávat informaci o datovém typu zvlášť u každého elementu a proto jsou elementy uspořádány do bloků stejného typu. Informace o datovém typu je zapsaná jen v prvním bytu každého bloku. Jeden blok má 256 bytů a adresa začátku bloku je vždy násobkem 256. Takové zarovnání je výhodné zejména proto, že k adrese můžeme snadno získat datový typ tak, že vynulujeme nižší byte a přečteme byte na takto vzniklé adrese.



3.1.1 Páry

Pár je dvojice elementů, tedy zabírá 2 slova a musí tak být vždy alokovaný v paměti.

3.1.2 Prázdný seznam

Slovo, které reprezentuje prázdný seznam je rovno nule. To umožňuje snadné detekování konce seznamu. Přestože je i nejnižší bit 0, nehrozí záměna s ukazatelem, protože je celý nižší byte 0 a muselo by se tedy jednat o ukazatel na identifikátor datového typu, který se ale nikdy nevyskytuje ve stejném kontextu jako elementy.

3.1.3 Symboly

Jednobytové symboly jsou uloženy přímo ve vyšším bytu, jak je popsáno výše.

Vícebytové symboly jsou alokované v paměti. První byte na dané adrese je délka a následuje sekvence znaků. Na konci jsou jedna nabo dvě nuly (podle zarovnání na slova).

Při vytváření nového symbolu se vždy prohledá paměť a pokud je v ní stejný symbol nalezen, není třeba nic alokovat. To šetří paměť a zároveň usnadňuje

porovnávání symbolů. Díky tomu, že před symbolem je vždy zapsaná jeho délka a na konci je zarovnán nulami, nemusí se při prohledávání paměti ztrácet čas s náhodnými shodami a prefixy.

Maximální délka symbolu je 252 znaků, protože délka bloku paměti je 256 bytů, jedno slovo je alokováno pro identifikaci datového typu bloku a jedno pro délku symbolu.

3.1.4 Celá čísla

Integery jsou ve dvojkově doplňkovém kódu. Jednobytový integer (0 – 255) je uložen přímo ve vyšším bytu, dvoubytové (−32768 – 32767) jsou alokované v paměti.

3.1.5 Floaty

Pro čísla s plovoucí desetinnou čárkou je použita knihovna Math48[1]. Jsou 48-bitová a tedy vždy alokovaná v paměti.

3.1.6 Uživatelské procedury a makra

Procedura je alokovaná v paměti a zabírá 4 slova. První byte je počet argumentů. Druhý byte obsahuje příznaky: nejnižší bit určuje, jestli procedura přijímá variabilní počet argumentů a druhý nejnižší bit určuje, jestli se jedná o makro. Makra jsou tedy implementována jako procedury, které nevyhodnotí své argumenty a svůj výsledek ještě jednou vyhodnotí. Druhé slovo je seznam symbolů, na které je potřeba navázat argumenty. Třetí slovo je prostředí vzniku procedury. Čtvrté slovo je samotné tělo procedury.

3.1.7 Pokračování

Pokračování je implementovño jako uložený zásobník. Při vytváření pokračování je tedy veškerý kontext uložen na zásobník a při zavolání pokračování je tento kontext po obnovení zásobníku obnoven. Kvůli způsobu alokování paměti je velikost ukládaného zásobníku omezena na 129 slov (252 bytů), protože blok paměti má 256 bytů, jedno slovo je alokované pro identifikaci typu a jedno slovo pro velikost uloženého zásobníku.

3.1.8 Vektor

Vektor bytů je implementován pouze pro účely snadného implementování krátkých procedur přímo pomocí opkódů Z80. Jediná zabudovaná procedura, která s tímto typem přímo pracuje je `ADDRESS`. Maximální velikost je 252 bytů, ze stejného důvodu jako u pokračování.

[1] Knihovna je dostupná na http://www.fam-bundgaard.dk/DOWNLOAD/DOWNLOAD.HTM.

3.1.9 Bool

Na reprezentování pravdivostní hodnoty zjevně stačí jeden byte. Vyšší byte je 1, pokud se jedná o pravdu a 0, pokud se jedná o nepravdu. Pokud je vyšší byte jiné číslo, je element nedefinovaná hodnota.

3.1.10 Generovaný symbol

Generovaný symbol se také nealokuje v paměti, takže kvůli omezení jednoho bytu může být použito maximálně 256 takovýchto symbolů.

3.1.11 Reprezentace prostředí

Prostředí nemá vlastní datový typ. Je reprezentováno párem, kde první prvek je předek daného prostředí a druhý prvek je seznam vazeb. Seznam vazeb je seznamem párů, které reprezentují jednotlivé vazby tak, že první prvek je symbol a druhý prvek je element, který je na něj navázán.

První prvek vazby může být i prázdný seznam. Takové vazby jsou vytvářeny jen při inicializaci interpretu, a to procedurou `alloc_sym` k alokování symbolů, které jsou názvy zabudovaných procedur. Účelem je zabránit garbage collectoru tyto symboly smazat.

3.2 Správa paměti a garbage collector

3.2.1 Alokace paměti pro elementy

Jak už jsem zmínil v kapitole 3.1, paměť pro elementy je alokovaná po slovech a elementy jsou alokovány podle typu v blocích o velikosti 256 bytů. Paměť pro elementy je vymezena hodnotami `MEM_START` a `MEM_END` v platformově závislé části kódu.

Pro účely správy paměti s elementy je v paměti bitmapa. Každý její bit reprezentuje jedno slovo v paměti pro elementy. Z toho důvodu je velikost mapy šestnáctkrát menší, než velikost paměti pro elementy. Jednotlivý bit je 1, pokud je slovo v paměti alokované a 0, pokud je slovo volné.

3.2.2 Garbage collector

Garbage collector funguje na principu mark and sweep. V první fázi vynuluje bitmapu paměti. V druhé fázi prochází všechny dosažitelné elementy a místo, kde jsou alokované označí v bitmapě jedničkami.

Garbage collector je spouštěn ve dvou případech. Je spouštěn na konci každého read-eval-print cyklu – to by ale nestačilo, kvůli dlouze běžícím programům, které se do REPLu vůbec nevracejí. Proto je spuštěn také procedurou `alloc_words` v případě, že v paměti není dost místa pro alokaci požadovaného počtu slov požadovaného typu. Z tohoto důvodu garbage collector kromě

procházení prostředí prochází také všechna slova na zásobníku a chová se konzervativně, tj. vše co může být odkaz do paměti s elementy je v bitmapě alokováno.

Pokud se ani po skončení garbage collectoru nepodaří proceduře `alloc_words` alokovat paměť, je vypsána chyba a uživatel je vrácen do REPLu.

3.2.3 Proměnné interpretu

Nezávisle na paměti pro elementy používá interpret proměnné, které dohromady zabírají 68 bytů a jejich umístění v paměti je určené hodnotou `VARS` v platformově závislé části kódu.

3.3 Reader

Reader zavolá `getline`, naparsuje z bufferu jeden element a následující znaky ignoruje. Páry a seznamy jsou parsovány a konstruovány rekurzivním voláním `parse_list`. Výrazy reprezentující ostatní elementy jsou analyzovány konečným deterministickým automatem převzatým z parseru doc. Vychodila, viz 3.4. Podle koncového stavu automatu se pak výraz parsuje a alokuje se element příslušného typu.

Na MZ-800 je délka vstupního výrazu omezena na 80 znaků, protože procedura `getline` je implementována pomocí procedury MONITORu (což je soubor rutin obsažených na ROM) na přečtení řádku ze vstupu, která dokáže přečíst jen 80 znaků.

3.4 Parser

Upravil jsem parser výrazů doc. Vychodila tak, aby místo zvýrazňování syntaxe generoval reprezentaci elementů v paměti. Jedinným argumentem je adresa začátku paměti s elementy (`MEM_START`), na standardním vstupu přečte výraz, obsah paměti vygeneruje do souboru `mem` a na výstup vypíše adresu vygenerovaného elementu v paměti.

Pro MZ-800 jsem použití parseru zautomatizoval. Pokud je soubor `expr.scm` neprázdný, `make` sestaví parser, vygeneruje k interpretu paměť a pak s touto pamětí sestaví interpret. Při inicializaci takto sestaveného interpretu se vyhodnotí (jen první) výraz z `expr.scm`.

3.5 Eval

Eval nejdříve zjistí datový typ vyhodnocovaného elementu. Podle něj provede skok do příslušné podprocedury, nebo element přímo vrátí, pokud se vyhodnocuje sám na sebe.

```
1  eval:
2          ld a, l                 ;spodni byte elementu
3          cp 0                    ;
4          ret z                   ;pokud je 0, jedna se o prazdny
               seznam (ukazatel se spodnim bytem 0 by nedaval smysl,
               protoze by ukazoval na oznaceni datoveho typu)
5          bit 0, l                ;je nejnizsi bit 0?
6          jp z, eval_pointer      ;pokud ano, jedna se o ukazatel
7          cp 111B
8          jp z, eval_symbol
9          cp 1111B
10         jp z, eval_symbol
11         cp 11111B
12         jp z, eval_get_stack_elem
13         ret                     ;zbytek se vyhodnoti sam na sebe
14
15 eval_pointer:
16         call get_hl_pointer_type
17         cp 0                    ;zjistime o jaky datovy typ se jedna
18         jp z, eval_pair         ;0 - par
19         cp 2
20         jp z, eval_symbol       ;2 - symbol
21         ret                     ;zbytek se vyhodnoti sam na sebe
```

Zdrojový kód 3: Eval - detekce datového typu

Procedura `eval_get_stack_elem` vyhodnocuje speciální element reprezentující offset na zásobníku. Byla součástí nedokončené optimalizace popsané v kapitole 3.8. Nyní je použita pouze interně v implementaci volání uživatelských procedur.

Procedura `eval_pair` jen ověří zda je element seznam a pokud ano, pokračuje skokem do podprocedury `eval_list`. Pokud ne, interpret vypíše chybu a vrátí uživatele do REPLu.

3.5.1 Vyhodnocování symbolů

K symbolu je potřeba najít jeho vazbu v prostředí. Pár reprezentující aktuální prostředí je vždy uložen v proměnné interpretu `env`. K samotnému procházení seznamu vazeb je implementována procedura `symbol_assoc`. Tato procedura vrací přímo pár reprezentující vazbu, aby bylo možné i změnit vazbu, čehož využívá procedura `eval_define`.

```
1  symbol_assoc:
2          xor a
3          cp l            ;overime, jestli jsme na konci seznamu
4          jp z, assoc_symbol_fail
5          ld c, (hl)
6          inc hl
7          ld b, (hl)      ;prvni element seznamu ulozime do bc
8          ld a, (bc)      ;prvni byte prvniho elementu paru
9          cp e            ;zjistime, jestli se rovna prvnimu bytu
               symbolu ktery hledame
10         jp nz, symbol_cmp_fail
11         inc bc
12         ld a, (bc)
13         cp d            ;totez overime pro druhy byte
14         jp z, eval_symbol_found
15 symbol_cmp_fail:
16         inc hl          ;presuneme se k druhemu elementu paru -
               pokracovani seznamu
17         car_hl          ;nacteme zbytek seznamu a pokracujeme
18         jp symbol_assoc
19 assoc_symbol_fail:
20         ld a, 1
21         ret
22 eval_symbol_found:
23         xor a
24         ld hl, 0xFFFF
25         add hl, bc
26         ret
```

Zdrojový kód 4: Procedura symbol_assoc

V případě nenalezení vazby v seznamu vazeb se hledá v prostředí předka. Pokud prostředí nemá předka a vazba nebyla nalezena, je vypsána chyba a uživatel je vrácen do REPLu.

3.5.2 Vyhodnocování seznamů

O vyhodnocování seznamů se stará podprocedura `eval_list`. Je zde výrazný rozdíl oproti běžnému Scheme – procedura nejprve zjistí, jestli není první prvek seznamu symbol, který je názvem zabudované procedury a případně provede skok do příslušné procedury. Díky způsobu alokování symbolů (popsán v 3.1.3) jsou tato porovnání jen porovnáním dvou šestnáctibitových čísel.

Pokud se tedy nejedná o volání zabudované procedury, pokračuje se částí procedury pojmenovanou `eval_list_normal`. Ta již dle očekávání vyhodnotí první prvek seznamu a podle výsledku se přesune k aplikaci uživatelské procedury (nebo makra), nebo aplikaci pokračování.

```
eval_list:
        push hl
        car_hl
        ld a, 111B
        cp l
        jp z, eval_list_char
        bit 0, l
        jp nz, eval_list_normal
        eq_hl quote eval_quote
        eq_hl quasiquote eval_quasiquote
        eq_hl if eval_if
        eq_hl car eval_car
        eq_hl cdr eval_cdr
...
```

Zdrojový kód 5: Začátek procedury eval_list

```
...
        cp '<'
        jp z, eval_lesser
eval_list_normal:
        push bc
        call eval
        call get_hl_pointer_type
        cp 4
        jp z, eval_proc
        cp 5
        jp nz, eval_wrong_type

eval_continuation:
...
```

Zdrojový kód 6: Konec procedury eval_list

3.5.3 Aplikace uživatelských procedur a maker

Aplikace uživatelských procedur i maker je implementována procedurou `eval_proc`. Jak je popsáno v 3.1.6, uživatelská makra jsou implementována jako speciální procedury. Procedura `eval_proc` tak musí na několika místech zjišťovat, zda se aplikuje procedura, nebo makro. Podle toho pak vyhodnotí argumenty, případně vyhodnotí výsledek znovu (po expanzi makra). Navíc je potřeba zjistit, zda se jedná o rekurzivní volání, kvůli optimalizaci koncové rekurze popsané v kapitole 3.7. Z těchto důvodů je vždy procedura při volání uložena do registru IY.

Z důvodu nepoužité optimalizace popsané v kapitole 3.8 jsou argumenty před vytvořením vazeb uloženy na zásobník.

3.5.4 Aplikace pokračování

Při volání pokračování je třeba obnovit uložený zásobník a z něj veškerý uložený kontext (prostředí a registry IX a IY). Argument pokračování je vrácen v registru HL, ve kterém očekává výsledek vyhodnocení procedura, která původně zavolala `CALL/CC`.

```
1  pop hl                   ;do hl obnovime pokracovani
2  ld b, 0                  ;do bc nacteme pocet bytu
3  ld c, (hl)
4  inc hl                   ;inkrementujeme hl, aby ukazoval na zasobnik
5  inc hl
6  ex de, hl                ;pokracovani ulozime do de
7  ld hl, (stack_bottom)    ;nacteme dno zasobniku
8  xor a
9  sbc hl, bc               ;ziskame vrchol zasobniku
10 ld sp, hl                ;ulozime do sp
11 ex de, hl                ;do hl nacteme zdroj (pokracovani) a do de
       cil (zasobnik)
12 ldir
13 pop ix                   ;obnovime ix
14 pop iy                   ;obnovime iy
15 pop hl
16 ld (env), hl             ;obnovime prostredi
17 exx
```

Zdrojový kód 7: Obnovení kontextu při volání pokračování

3.6 Zabudované procedury a makra

V této kapitole jsou uvedeny důležité implementační detaily zabdovaných procedur a maker.

Seznam zabudovaných procedur a maker:

- `QUOTE`
- `QUASIQUOTE`
- `UNQUOTE`
- `UNQUOTE-SPLICING`
- `IF`
- `CAR`
- `CDR`
- `CONS`
- `MAP`
- `LAMBDA`
- `DEFINE`
- `DEFMACRO`
- `SET!`
- `LENGTH`
- `AND`
- `OR`
- `EQUAL?`
- `GENSYM`
- `CALL/CC`
- `SET-CAR!`
- `SET-CDR!`
- `APPLY`
- `+`
- `-`

- `*`
- `/`
- `<`
- `>`
- `=`
- `PEEK`
- `POKE`
- `CALL`
- `INLINE`
- `ADDRESS`
- `EVAL`.

Většina z těchto procedur by se měla chovat dle očekávání (ale s maximálním počtem argumentů 255), pokud se chovají jinak, než ve standardním Scheme, rozdíl je vysvětlen v této kapitole.

3.6.1 APPLY a MAP

Hlavním rozdílem oproti standardnímu Scheme je, že těmto procedurám lze předat pouze uživatelské procedury (ale navíc i uživatelská makra). To souvisí s implementací. Tyto procedury nejdříve sestrojí seznam argumentů a pak skočí přímo do procedury `eval_proc` (která implementuje standardní volání uživatelské procedury) tak, aby nedošlo ke druhému vyhodnocení argumentů. V případě `MAP`u se toto opakuje a výsledky jsou ukládány do seznamu.

Pokud chce uživatel těmto procedurám předat některou zabudovanou proceduru, je možné ji jednoduše „obalit“ uživatelskou procedurou.

3.6.2 Operace s čísly

V případě, že jsou některé z procedur `+`, `-`, `*`, `/` předány argumenty, z nichž se některé vyhodnotí na integer a některé na float, jsou všechny operace na integerech předcházejících prvnímu floatu integerové. Všechny integery, které následují po prvním floatu jsou přetypovány na floaty. To opět souvisí s implementací.

Tyto procedury používají akumulátor uložený v alternativních registrech 'HL (respektive 'BCDEHL pro floaty) a ve standardních registrech postupně procházejí argumenty, které pomocí patřičné operace přidají k akumulátoru. Když jsou tedy argument i akumulátor integer, výsledný akumulátor bude integer. Pokud je argument float, dojde k přetypování akumulátoru na float. A pokud je akumulátor float, je argument vždy přetypován na float.

```
...
        ld de, map_fun_ret      ;nacteme navratovou adresu
        push de                 ;ulozime ji na zasobnik
        ld de, (env)
        push de                 ;ulozime prostredi
        push iy                 ;ulozime iy
        ld iy, 0                ;do iy nacteme proceduru
        add iy, bc
        push ix                 ;ulozime frame pointer
        ld c, (iy+0)            ;pocet argumentu nacteme do c
        jp eval_proc_pushargs   ;aplikujeme proceduru (bez
           vyhodnoceni argumentu)
map_fun_ret:
...
```

Zdrojový kód 8: Nastavení kontextu a skok do eval_proc v MAPu

```
math_op_ret:
        pop bc
do_math_op:
        djnz do_math_op_cont
        exx
        ret
do_math_op_cont:
        pop de
        pop hl
        push de
        push bc         ;ulozeni zbyvajiciho poctu argumentu
        ld de, math_op_ret
        push de         ;zavolani procedury v iy
        jp (iy)
```

Zdrojový kód 9: Cyklus procházející argumenty operací s čísly

Při procházení argumentů je adresa procedury provádějící danou operaci uložena v registru IY a při přetypování akumulátoru je přepsána adresou procedury, která pracuje s výsledným datovým typem.

Pro samotné matematické operace s floaty je použita knihovna Math48. Také pro matematické operace s integery jsou použity procedury, které komunita kolem Z80 poskytla online.

3.6.3 DEFMACRO

Makra jsou implementována jako procedury se speciálním příznakem. Proto `DEFMACRO` nejprve vyhodnotí zadaný lambda výraz na proceduru, pak u ní nastaví příznak a skočí do `eval_define`, kde se provede vytvoření vazby na symbol.

3.6.4 CALL/CC

Jak jsem popsal v 3.1.7, při vytváření pokračování se kontext uloží na zásobník, který se pak uloží do paměti. Kvůli způsobu alokace paměti je velikost ukládaného zásobníku omezena na 252 bytů a pokud by mělo pokračování víc, interpret vypíše chybu a uživatel je vrácen do REPLu.

3.6.5 EVAL

Procedura `eval` přijímá jen jeden argument. Tento argument je vyhodnocen vždy v globálním prostředí.

3.6.6 GENSYM

Generovaný symbol je popsaný v 3.1.10. Při zavolání procedury se inkrementuje počítadlo v proměnné interpretu a hodnota se použije jako id (vyšší byte) generovaného symbolu. Po 256 voláních se tedy symboly zčnou opakovat.

3.6.7 PEEK, POKE, CALL, INLINE a ADDRESS

Tyto procedury nejsou ve Scheme běžné, ale podobné procedury jsou běžné v interpretech BASICu a jsou užitečné při interakci s nativním kódem.

- `PEEK` vrátí hodnotu bytu na zadané adrese.
- `POKE` zapíše do paměti od adresy předané prvním argumentem posloupnost bytů a slov předaných dalšími argumenty.
- `CALL` zavolá nativní proceduru na zadané adrese.
- `INLINE` vytvoří vektor bytů a slov zadaných jako argumenty.
- `ADDRESS` vrátí adresu prvního bytu vektoru.

Všechny adresy jsou klasické elementy (integery), nelze je tedy zapisovat hexadecimálně.

```
1 (DEFINE CLS (INLINE 33 0 208 1 232 3 54 0 35 11 120 177 32 248 201))
2 (SET! CLS ((LAMBDA (X) (LAMBDA () (CALL (ADDRESS X)))) CLS))
```

Zdrojový kód 10: Implementace procedury, která na MZ-800 smaže obrazovku

3.6.8 Chybějící procedury a makra

Užitečné procedury a makra, jako například `LET` a `LIST`, které nejsou v interpretu implementované, si může uživatel definovat jako uživatelské procedury.

```
1 (DEFINE LIST (LAMBDA X X))
```

Zdrojový kód 11: Implementace procedury LIST

```
1 (DEFINE CAR (LAMBDA (X) (CAR X)))
2 (DEFINE CADR (LAMBDA (X) (CAR (CDR X))))
3 (DEFMACRO LET
4   (LAMBDA (A . B)
5     `((LAMBDA ,(MAP CAR A) ,@B) ,@(MAP CADR A))))
```

Zdrojový kód 12: Implementace makra LET

3.7 Optimalizace koncové rekurze

Interpret umí optimalizovat koncovou rekurzi uživatelských procedur, pokud jsou zavolány ve svém těle, tedy nejsou volány například ve vnořené uživatelské proceduře.

```
1 ...
2 eval_if_true:
3         pop hl          ;obnovime argumenty
4 eval_if_true2:
5         cdr_hl          ;presuneme se ke druhemu argumentu
6         car_hl
7         jp eval         ;vyhodnotime argument
```

Zdrojový kód 13: Koncové volání eval v implementaci makra IF

Využívá toho, že je koncová rekurze optimalizovaná přímo v zabudovaných makrech if, or a and, které tak při vyhodnocování posledního výrazu nepoužijí instrukce call a ret, ale přímo jp. Procedura `eval_proc` (volání uživatelské procedury) tak může přečtením hodnot ze zásobníku zjistit, jestli se nenachází v poslední iteraci cyklu `eval_begin_loop`, který je použitý pouze při vyhodnocování těla uživatelské procedury. Při volání uživatelské procedury se do registru IY načte ukazatel na danou proceduru, takže lze také snadno zjistit, jestli procedura, v jejíž těle se nacházíme je stejná jako procedura, kterou chceme zavolat. Pokud jsou obě tyto podmínky splněny, můžeme zásobník nastavit na úroveň původního volání a pokračovat.

3.8 Nepoužitá optimalizace procedur

V průběhu vývoje jsem implementoval netriviální optimalizaci aplikace procedur, která se ale nakonec ukázala být nepoužitelná. Myšlenka byla taková, že procedury většinou nepotřebují znát jména svých argumentů a nepotřebují ani vy-

```
1         pop bc          ;adresu aktualni procedury nacteme do bc
2         xor a
3         sbc hl, bc      ;zjistime jestli se jedna o rekurzivni
             volani
4         jp nz, eval_proc_norec
5         ld ix, 0
6         add ix, sp      ;presuneme se k navratove adrese
7         ld bc, eval_begin_loop_ret
8         ld l, (ix+4)    ;ziskame navratovou adresu
9         ld h, (ix+5)
10        xor a
11        sbc hl, bc      ;zjistime, jestli je to adresa vyhodnocovani
              tela lambdy
12        jp nz, eval_proc_norec
13        ld l, (ix+6)
14        ld h, (ix+7)    ;presuneme se k seznamu zbyvajicich vyrazu v
              tele lambdy
15        cp l            ;zjistime, jestli je seznam prazdny
16        jp nz, eval_proc_norec
```

Zdrojový kód 14: Detekce rekurzivního volání v eval_proc

tvářet vazby. Většinou stačí, když ví s kolikátým argumentem mají pracovat a můžou ho najít na zásobníku (se znalostí frame pointeru volané funkce).

Pro tyto účely jsem implementoval speciální element, který reprezentuje n-tý argument, a také procedury, které hledají volné a vázané výskyty symbolů při vyhodnocování lambda výrazu.

Problém ovšem nastal při implementaci maker. Když jsou v těle procedury makra, tak tato jednoduchá syntaktická analýza nelze použít. Navíc nemůžeme snadno zjistit přítomnost maker, protože vazby symbolů v prostředí vzniku procedury se mohou změnit.

Kvůli tomuto problému jsem nakonec optimalizaci zavrhnul. Většina kódu realizující optimalizaci je zakomentovaná, ale pozůstatky tohoto návrhu jsou vidět v implementaci `eval_lambda` a `eval_proc`.

4 Přenositelnost

Ke běhu interpretu je potřeba procesor Z80 a dostatečně velký souvislý blok paměti. Reader a printer pak potřebují implementaci platformově závislých procedur `putline` a `getline`. String, který je předáván v bufferu proceduře putline, je vždy zakončen bytem `0x0D`.

```
1  org     01200h
2
3  ld sp, $D000
4  jp init
5
6  MEM_START:      equ 0x3000
7  MEM_END:        equ 0xc000
8  MEM_MAP:        equ 0xc000      ;velikost ((MEM_END - MEM_START) /
       16)
9  VARS:           equ 0xc900
10 BUF:            equ 0xca00
11 ;GENERATED_ELEM: equ $0
12
13
14 ;platforme zavisla procedura pro precteni radku ze vstupu do bufferu
15 getline:
16         ld de, BUF      ;nacteme adresu bufferu
17         call $0003
18         ret
19
20 ;platforme zavisla procedura pro zapsani bufferu na obrazovku
21 putline:
22         ld de, BUF      ;nacteme adresu bufferu
23         call $0015      ;vytisknuti bufferu
24         call $0006      ;novy radek
25         ret
26
27 include "scm.asm"
28
29 ds      0x3000 - $, 0xff
30 include "mem.asm"
```

Zdrojový kód 15: Platformově závislý kód pro Sharp MZ-800

Při přenášení na jinou platformu je nutné vymezit paměť pro elementy pomocí `MEM_START` a `MEM_END`, nastavit začátek mapy paměti pomocí `MEM_MAP` (mapa je šestnáctkrát menší, než paměť pro elementy), `VARS` nastavit na místo, kam si interpret může uložit interní proměnné (potřeba 68 bytů) a `BUF` nastavit na adresu paměti použité jako buffer pro výstup z procedury `getline` a vstup procedury `putline` (velký podle potřeby procedur). Pokud uživatel nechce využívat generátor elementů, není třeba includovat `mem.asm` a stačí nastavit `GENERATED_ELEM` na 0.

Případně lze ještě před skokem do procedury init nastavit stack pointer a vyhradit víc místa pro zásobník, což umožní hlubší rekurzi.

V průběhu vývoje jsem takto snadno vytvořil verzi interpretu pro počítač MSX (sestavení `make openmsx`) kvůli snadnějšímu debugování ve svobodném emulátoru OpenMSX.

Závěr

Výsledkem práce je interpret Scheme pro počítače Sharp MZ-800 a MSX. Je implementován v assembleru a je snadno přenositelný na jiné platformy s procesorem Z80.

Výsledný interpret má 7,5 KB. To je více než Tiny Lisp (4 KB), oproti kterému má ale interpret více funkcionality. Navíc je to méně, než velikost SpecLispu (12 KB), oproti němuž má interpret také více funkcionality, například operace s floaty a makra. Právě kvůli makrům se ale komplikuje implementace některých optimalizací a vyhodnocování je tak pomalejší než u zmíněných alternativ.

Předmětem případného dalšího vývoje by mohlo být odstranění všech pozůstatků implementace výše zmíněné optimalizace, nebo naopak její dokončení, například detekcí uživatelských maker v těle procedury. Také by bylo možné zpřístupnit pár reprezentující prostředí uživateli a předávat ho jako druhý argument procedury `EVAL`.

Conclusions

The result of this work is a Scheme interpreter for Sharp MZ-800 and MSX computers. The interpreter is implemented in assembly language and can be easily ported to other Z80 platforms.

The size of this interpreter is 7.5 KB. It is bigger in size, but has more functionality than Tiny Lisp (4 KB). It is however smaller in size than SpecLisp (12 KB), while still having more functionality, for example floating point numbers and macros. Unfortunately, macros make implementation of certain optimizations much more complicated, and thus the mentioned alternatives have faster evaluation implementation.

The aim of futher development could be removing all remnants of mentioned optimization, or completing the implementation of this optimization, for example by detecting user macros in procedure body. It would also be possible to take the pair, by wich an enviroment is represented, make it avalible to the user and pass it as the second argument of the `EVAL` procedure.

A Obsah přiloženého CD/DVD

`scheme/`
Zdrojové kódy interpretu.

`scheme/parser/`
Zdrojové kódy parseru.

`scheme/scm.asm`
Zdrojový kód platformově nezávislé části interpretu.

`scheme/scm-sharp.asm`
Zdrojový kód platformově závislé části pro MZ-800.

`scheme/scm-openmsx.asm`
Zdrojový kód platformově závislé části pro MSX.

`scheme/scm.mzf`
Sestavený interpret pro MZ-800.

`scheme/openmsx.bin`
Sestavený interpret pro MSX.

`scheme/gmz800em`
Staticky linkovaný emulátor MZ-800 pro GNU/Linux.

`doc/scheme.pdf`
Text práce ve formátu PDF.

`doc/src/`
Zdrojové kódy textu práce.

Literatura

[1] Rodnay Zaks: *Programming the Z80*. Sybex, 1981

[2] Lance Leventhal: *Z80 Assembly Language Subroutines*. Osborne/McGraw-Hill Cop., 1983