



FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

SEMESTRÁLNÍ PRÁCE Z PŘEDMĚTU KIV/PC

Interpret podmnožiny jazyka Lisp

Patrik Harag

harag@students.zcu.cz

(A15B0034P)

20. prosince 2016

Obsah

1	Zadání	1
2	Analýza	2
2.1	Charakteristika jazyka Lisp	2
2.2	Návrh parseru	2
2.3	Návrh interpretu	3
3	Implementace	4
3.1	Postup interpretace	4
3.1.1	Načítání ze souboru	4
3.2	Datové typy	4
3.2.1	Logické hodnoty	5
3.2.2	Celá čísla	5
3.2.3	Symboly	5
3.2.4	Seznamy	5
3.2.5	Funkce	5
3.2.6	Speciální formy	6
3.2.7	Výjimky	6
3.3	Globální paměť	7
4	Uživatelská příručka	8
4.1	Sestavení	8
4.2	Spuštění	8
4.3	Seznam funkcí	9
5	Závěr	12
5.1	Zhodnocení	12
5.2	Možná vylepšení	12

Kapitola 1

Zadání

Naprogramujte v ANSI C přenositelnou konzolovou aplikaci, která bude fungovat jako jednoduchý interpret podmnožiny funkcionálního programovacího jazyka Lisp. Vstupem aplikace bude seznam příkazů v jazyce Lisp. Výstupem je pak odpovídající seznam výsledků vyhodnocení každého výrazu.

Program se bude spouštět příkazem `lisp.exe [<vstupní-soubor>]`. Symbol `<vstupní-soubor>` zastupuje nepovinný parametr – název vstupního souboru se seznamem výrazů v jazyce Lisp. Není-li první parametr uveden, program bude fungovat v interaktivním módu, kdy se příkazy budou provádět přímo zadáním z konzole do interpretu.

Vámi vyvinutý program tedy bude vykonávat následující činnosti:

1. Při spuštění bez parametru bude čekat na vstup od uživatele. Zadaný výraz vyhodnotí a bude vyžadovat další vstup, dokud nebude uveden výraz (`quit`).
2. Při spuštění s parametrem načte zadaný vstupní soubor, každý výraz v něm uvedený vypíše na obrazovku, okamžitě vyhodnotí a výsledek vypíše na obrazovku. Po zpracování posledního výrazu dojde k ukončení programu. Proto nemusí být jako poslední výraz uveden výraz (`quit`). Na jedné řádce v souboru může být uvedeno více samostatných výrazů a program je musí být schopen správně zpracovat.

Váš program může být během testování spuštěn například takto:

```
lisp.exe test.lisp
```

Celé zadání je dostupné na adrese:

<http://www.kiv.zcu.cz/studies/predmety/pc/doc/work/sw2016-01.pdf>

Kapitola 2

Analýza

2.1 Charakteristika jazyka Lisp

Program v jazyce Lisp¹ je složen z takzvaných *s-výrazů*² neboli seznamů ve tvaru

$$(<funkce><argument-1><argument-2>\dots),$$

kde na prvním místě je název funkce a na dalších její argumenty. Prvkem s-výrazu může být opět s-výraz. Všechny funkce a syntaktické konstrukty jazyka mají identický zápis. Zdrojový kód v jazyce Lisp má stejnou strukturu jako jeho *abstraktní syntaktický strom* (AST) – tato vlastnost se nazývá *homoikonicita*.

Programovací jazyk Lisp je vybaven takzvanou smyčkou *REPL* (*Read-Evaluate-Print-Loop*), interaktivním prostředím, v němž je možné zapisovat jednotlivé výrazy, které jsou ihned vykonány a jejichž výsledek je vypsán na standardní výstup. Naším úkolem bude podobnou smyčku vytvořit.

2.2 Návrh parseru

Díky výše uvedeným vlastnostem je tvorba parseru, oproti konvenčním jazykům, relativně jednoduchá. V první fázi bude vstup rozdělen na jednotlivé tokeny. Token může být závorka nebo libovolné slovo, které odděluje bílé znaky nebo závorky. Z tokenů se následně sestaví abstraktní syntaktický strom, neboli soustava s-výrazů.

Tvorbu parseru však zkomplikuje speciální zápis funkce *QUOTE*, znak „‘“, který se přepisuje jako:

$$'<výraz> \rightarrow (QUOTE <výraz>)$$

Jako vhodné řešení se nabízí provést tuto transformaci až po sestavení abstraktního syntaktického stromu s tím, že parser „nebude vědět“ nic o tomto speciálním zápisu a zachová si tak svoji jednoduchost. Bude se tak v podstatě jednat o makro.

¹Většinou bývá názvem Lisp označována celá rodina jazyků, přičemž jednotlivé dialekty se od sebe mohou výrazně lišit. Zadání neuvádí žádný konkrétní dialekt nebo verzi, pouze přikládá odkaz na online interpret jazyka *Common Lisp* s názvem *JSCL*.

²Termín *s-výraz* (*s-expression*) bude používán pouze ve smyslu způsobu zápisu syntaxe, nikoli ve smyslu způsobu reprezentace dat ve stromové struktuře.

2.3 Návrh interpretu

Datové typy Jazyk bude podporovat několik datových typů – minimálně logický typ, celé číslo, symbol a seznam. Hodnoty budou ukládány do generické struktury, která bude obsahovat číslo určující typ hodnoty a ukazatel na hodnotu.

Paměť Uživatelské proměnné je nutné mezi jednotlivými vyhodnocovanými výrazy někde ukládat. Stejně tak je nutné někde uložit seznam funkcí. Je zde možnost tyto struktury spojit do jedné.

Výjimky Při implementaci interpretu bude nutné od začátku počítat s nutností nějakým způsobem zpracovávat výjimky. K výjimce může dojít i uvnitř vnořených forem, je tedy potřeba implementovat jejich propagaci.

Vlastnosti funkcí V některých dialektech jazyka Lisp je funkce takzvaný *first-class citizen*. To znamená, že je s funkcí nakládáno, jako s jakýmkoliv jiným objektem, tedy může být předána jako argument funkce, uložena do proměnné nebo vrácena jinou funkcí. Jiné implementace zacházejí s funkcemi odlišně, než s jinými hodnotami. V naší implementaci bude funkce *first-class citizen*, protože to považují za důležitou vlastnost, která do jazyka přinese jednotnost a jednoduchost.

Kapitola 3

Implementace

3.1 Postup interpretace

Významnou roli hraje soubor *repl.c*, který spojuje ostatní moduly a řídí tak proces interpretace.

Vstupní řetězec, načtený z konzole nebo ze souboru, je nejprve rozdělen na jednotlivé tokeny (zajišťuje *lexer.c*), z nich je následně sestaven AST a aplikováno makro *QUOTE* (*reader.c*). Kompletní AST je poté předán k vyhodnocení (*interpreter.c*) a výsledek je vypsán do konzole (*console.c*).

3.1.1 Načítání ze souboru

Při načítání ze souboru se znaky ukládají do vyrovnávací paměti, dokud není načten celý výraz. Následně je načtený výraz vypsán do konzole, standardním způsobem vyhodnocen a výsledek vypsán.

Interpret by si sice dokázal poradit i s celým zdrojovým kódem najednou (REPL standardně podporuje i více výrazů na jednom řádku), nicméně už bychom z něj nedostali vstupní řetězce, které potřebujeme jednotlivě vypsát do konzole.

3.2 Datové typy

Interpret podporuje následující datové typy:

- Logická hodnota
- Celé číslo
- Symbol
- Seznam
- Funkce
- Speciální forma
- Výjimka

Hodnoty všech výše uvedených datových typů jsou ukládány do jedné generické struktury s názvem *Value*.

```
typedef struct _Value {
    int type; /* konstanta určující typ hodnoty */
    void *pointer; /* pointer na hodnotu */
} Value;
```

Funkce pro práci s hodnotami, příslušné struktury a konstanty jsou uloženy v souborech *types.h* a *types.c*.

3.2.1 Logické hodnoty

Logické hodnoty jsou vnitřně reprezentovány stejně jako celá čísla. Toto řešení bylo použito zejména kvůli konzistenci.

3.2.2 Celá čísla

Celá čísla jsou reprezentovány datovým typem *int* jazyka C. Do struktury *Value* se tedy uloží pointer na *int*.

3.2.3 Symboly

Symboly slouží primárně k pojmenování hodnot prostřednictvím funkce *SET*. Vnitřně jsou reprezentovány textovým řetězcem.

3.2.4 Seznamy

V seznamu je možné uchovávat hodnoty libovolného typu. Seznam je implementován polem:

```
typedef struct _ValueList {
    int length;
    Value **values;
} ValueList;
```

Do struktury *Value* se uloží pointer na strukturu *ValueList*.

3.2.5 Funkce

Funkce jsou ukládány jako struktury, které obsahují pointer na zdrojovou funkci.

Pointer na funkci nemůže být použit (uložen do struktury *Value*) přímo, protože v ANSI C není možné kombinovat datové pointery s pointery ukazujícími na funkce. Jinými slovy pointer na funkci nelze přetypovat na *void **.

Zdrojové funkce se nestarají o uvolňování paměti předaných parametrů. Zároveň také musí vracet vždy novou hodnotu, nikdy tedy nesmí vrátit hodnotu předanou parametrem. Tyto podmínky jsou dány systémem, jakým se v programu pracuje s dynamickou pamětí.

Typy pro práci s funkcemi:

```
/* pointer na funkci */
typedef Value * (*Function) (int argc, Value **argv);

/* struktura pro zabalení pointeru na funkci */
typedef struct _FunctionWrapper {
    Function function;
} FunctionWrapper;
```

Příklad implementace funkce *LENGTH* (*built-in_list.c*):

```
Value * func_list_length(int argc, Value **argv) {
    if (argc < 1)
        return value_exception(MSG_TOO_FEW_ARGUMENTS);

    if (argv[0]->type != TYPE_LIST)
        return value_exception(MSG_WRONG_TYPE);

    return value_integer(as_list(argv[0])->length);
}
```

3.2.6 Speciální formy

Speciální formy jsou prostředkem, kterým se zavádí jazykové konstrukce (smčky, podmíněné příkazy, částečně vyhodnocované formy atd.).

Podobají se funkcím, ale jsou vyhodnocovány odlišným způsobem. Na rozdíl od běžných funkcí nejsou jejich argumenty před předáním vyhodnoceny. Speciální formy tedy při vyhodnocení dostávají, místo seznamu hodnot, seznam s-výrazů.

Stejně jako u „klasických“ funkcí jsou pointery speciálních forem zabalovány do struktur. Funkce implementující speciální formy se také nestarají o uvolňování předaných parametrů.

Typy pro práci se speciálními formami:

```
/* pointer na speciální formu */
typedef Value * (*SpecialForm) (int count, SExpression **exprs);

/* struktura pro zabalení pointeru na funkci */
typedef struct _SpecialFormWrapper {
    SpecialForm function;
} SpecialFormWrapper;
```

3.2.7 Výjimky

Výjimky jsou implementovány stejně jako jiné běžné hodnoty. Obsahem výjimky je řetězec obsahující chybovou zprávu.

Pokud některá funkce vrátí výjimku, interpret ji zachytí, případně přeruší další vyhodnocování a výjimku propaguje do vyšší úrovně.

3.3 Globální paměť

Díky jednotnému typovému systému mohou být všechny funkce, speciální formy i uživatelské proměnné uloženy v jedné datové struktuře. Tuto strukturu budeme nazývat, jak je zvykem, halda. Halda umožňuje vytvořit proměnné i konstanty. Pokud se uživatel pokusí konstantu předefinovat, bude mu v tom zabráněno.

Pro jednoduchost jsou logické literály *T* a *NIL* ve skutečnosti uloženy na haldě jako konstanty, ale pro uživatele v tom není žádný rozdíl.

Halda je definována v souboru *heap.h* a implementovaná v *heap.c*.

Kapitola 4

Uživatelská příručka

4.1 Sestavení

Pro snadné sestavení je připraven *Makefile*, který funguje na operačním systému Linux a v případě správné konfigurace i na Windows. Sestavení se provede po zavolání příkazu `make` v kořenovém adresáři. Předpokladem je nainstalovaný *GCC*, popřípadě *MinGW* (Windows) a nástroj *make*.

Pro operační systém Windows je připraven ještě poněkud typičtější *Makefile.win*. Překlad je obvykle možné provést příkazem `make -f Makefile.win`. Předpokladem je nainstalovaný překladač *Microsoft C/C++* a nástroj *make*.

Příkazy pro přeložení se mohou lišit podle použitého překladače a nastavení.

4.2 Spuštění

Program je možné spustit dvěma způsoby:

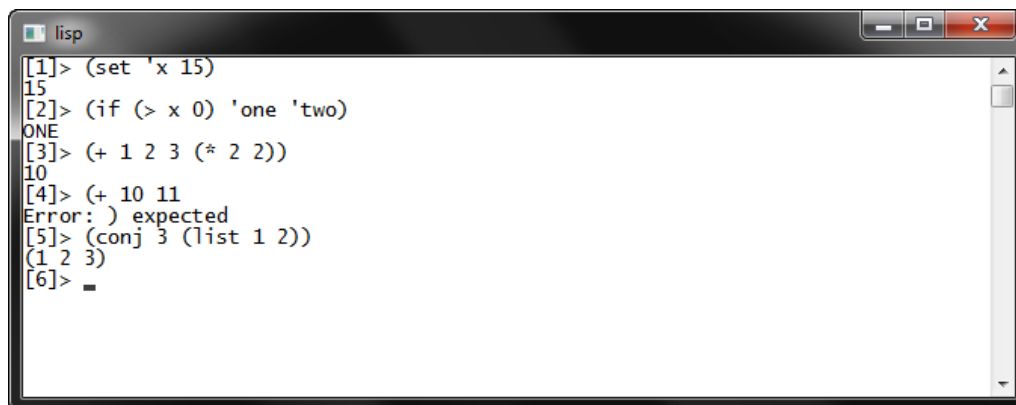
- `lisp.exe`

Program bude fungovat v interaktivním módu, kdy se příkazy budou provádět přímo zadáním z konzole do interpretu. Příklad použití ukazuje Obrázek 4.1

- `lisp.exe <vstupní-soubor>`

Při spuštění s parametrem načte zadaný vstupní soubor, každý výraz v něm uvedený vypíše na obrazovku, okamžitě vyhodnotí a výsledek vypíše na obrazovku. Po zpracování posledního výrazu dojde k ukončení programu.

Na jedné řádce v souboru může být uvedeno více samostatných výrazů. V takovém případě se vyhodnotí a vypíše všechny.



```
lisp
[1]> (set 'x 15)
15
[2]> (if (> x 0) 'one 'two)
ONE
[3]> (+ 1 2 3 (* 2 2))
10
[4]> (+ 10 11
Error: ) expected
[5]> (conj 3 (list 1 2))
(1 2 3)
[6]> ■
```

Obrázek 4.1: Ukázka práce v interaktivním módu (REPL)

4.3 Seznam funkcí

Tabulka 4.1: Aritmetické funkce.

Název funkce	Popis
+	Sečte prvky a vrátí celočíselný výsledek.
-	Odečte prvky zleva doprava a vrátí celočíselný výsledek.
*	Vynásobí prvky a vrátí celočíselný výsledek.
/	Vydělí od sebe prvky zleva doprava a vrátí celočíselný výsledek. Prováděno bude vždy jen celočíselné dělení.

Tabulka 4.2: Porovnávací funkce.

Název funkce	Popis
=	Porovná sousední prvky na shodu a výsledkem je T nebo NIL. Výsledek je dán logickým součinem všech porovnání.
/=	Porovná sousední prvky na neshodu a výsledkem je T nebo NIL. Výsledek je dán logickým součinem všech porovnání.
<	Porovná sousední prvky operátorem < (menší než) a vrátí výsledek T nebo NIL. Výsledek je dán logickým součinem všech porovnání.
>	Porovná sousední prvky operátorem > (větší než) a vrátí výsledek T nebo NIL. Výsledek je dán logickým součinem všech porovnání.
<=	Porovná sousední prvky operátorem <= (menší nebo rovno než) a vrátí výsledek T nebo NIL. Výsledek je dán logickým součinem všech porovnání.
>=	Porovná sousední prvky operátorem >= (větší nebo rovno než) a vrátí výsledek T nebo NIL. Výsledek je dán logickým součinem všech porovnání.

Tabulka 4.3: Funkce pro práci se seznamy.

Název funkce	Popis
LIST	Argumenty funkce budou vráceny jako seznam.
LENGTH	Vrátí délku seznamu.
ELT	Ze seznamu daným prvním argumentem vrátí prvek na pozici dané druhým argumentem. Indexuje se od nuly.
CAR	Vrátí první prvek zadaného seznamu.
CDR	Vrátí zadaný seznam bez prvního prvku.
CONS	Připojí prvek daný prvním argumentem na začátek seznamu daným druhým argumentem.
CONJ	Připojí prvek daný prvním argumentem na konec seznamu daným druhým argumentem.
REDUCE	Redukuje seznam na jedinou hodnotu. Prvním argumentem je funkce, druhým výchozí hodnota a třetím seznam.

Tabulka 4.4: Ostatní funkce.

Název funkce	Popis
IF	Pokud bude první argument T, vyhodnotí a vrátí druhý argument. Jinak vyhodnotí a vrátí třetí argument nebo NIL, pokud není třetí argument zadán.
SET	Uloží do proměnné dané prvním argumentem hodnotu danou druhým argumentem. Funkce vrací ukládanou hodnotu. Uložená hodnota může být použita v dalších výrazech.
QUOTE, ’	Argument nebude vyhodnocen.
QUIT	Ukončí program.

Kapitola 5

Závěr

5.1 Zhodnocení

Byl vytvořen jednoduchý interpret podmnožiny jazyka Lisp. Kromě předepsaného seznamu funkcí byly implementovány ještě některé další. Ovšem ne všechny funkce přidané navíc přesně kopírují Common Lisp.

Při implementování funkcí jsem narazil na rozdílné chování celé řady funkcí na různých interpretech jazyka Common Lisp – *JSCL* a *GNU CLISP*. Jednalo se především o záležitosti typu: „Co se má stát, když nebudu zadávat některý parametr?“, „Co se má stát, když bude parametr jiného typu?“ nebo „Co se má stát, když použiji makro *QUOTE* (') vícekrát za sebou?“. V takovýchto případech jsem se obvykle přiklonil k variantě, která mi subjektivně připadala lepší.

Během celého vývoje jsem vytvářel jednotkové testy, což se mi velmi vyplatilo především v pozdějších fázích vývoje, kdy už by byl problém „ohlídat“ takové množství funkcí a vlastností. Zvláště v jazyce, jako je ANSI C. Testy zároveň popisují zamýšlené chování interpretu.

5.2 Možná vylepšení

- umožnit definici uživatelských funkcí
- umožnit zachycení výjimek
- další datové typy a funkce