Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Přesné výpočty s reálnými čísly

2021

Ondřej Slavík

Vedoucí práce:
doc. RNDr. Michal Krupka, Ph.D.

Studijní obor:
Informatika, prezenční forma

Bibliografické údaje

Autor:	Ondřej Slavík
Název práce:	Přesné výpočty s reálnými čísly
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2021
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	doc. RNDr. Michal Krupka, Ph.D.
Počet stran:	85
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Ondřej Slavík
Title:	Precise computation of real numbers
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2021
Study field:	Computer Science, full-time form
Supervisor:	doc. RNDr. Michal Krupka, Ph.D.
Page count:	85
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Fenomén vyčíslitelnosti reálných čísel provází každého informatika, který se někdy snažil používat počítač k počítání. Jakmile se totiž musíme spolehnout na výpočty s čísly uloženými jako hodnoty, narážíme na limity přesnosti a rozsahu takto reprezentovaných čísel. Řešením není zpřesňování pomocí vyšší dotace paměťového prostoru (např. binary32 $\rightarrow$ binary64) a související změna architektury systému, nýbrž fundamentální změna v přístupu k vyčíselní reálných čísel. Tato práce dává návod, jak takovýto přístup přijmout, a přináší knihovnu, která umožňuje základní výpočty a vyčíslení reálných čísel.

Synopsis

The Real Numbers' computation phenomenon meets every single computer scientist trying to use computer to compute. When one needs to trust to calculations of numbers saved with their values, the limitations of these numbers' precisivity and range appear. The solution is not to assign more memory (e.g. binary32 $\rightarrow$ binary64) and linked swap of system's architecture but fundamental switch in approach to real numbers' computations. The Work gives a guideline how to admit the access and brings library providing basic real numbers' enumeration and calculation.

Klíčová slova: reálná čísla, funkce, Lisp, líné vyhodnocování, libovolná přesnost

Keywords: real numbers, functions, Lisp, lazy evaluation, arbitrary precision

Mockrát děkuji doc. RNDr. Michalu Krupkovi, Ph.D. za vedení práce a rodině za podporu.

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

0 Úvod 1

I Teorie 2

1 Čísla 2
- 1.1 Přirozená čísla . . . 3
- 1.2 Vyšší obory čísel . . . 5
- 1.3 Operace s čísly . . . 6
- 1.4 Funkce čísel . . . 7

2 Čísla v počítači 11
- 2.1 Čísla uložená jako hodnoty . . . 11
  - 2.1.1 Vážený poziční kód . . . 11
  - 2.1.2 Záporná čísla . . . 11
  - 2.1.3 Plovoucí řádová tečka . . . 12
- 2.2 Přesná reprezentace čísel jako hodnot . . . 12
  - 2.2.1 Přirozená čísla . . . 12
  - 2.2.2 Celá čísla . . . 13
  - 2.2.3 Racionální čísla . . . 14
- 2.3 Přesná reprezentace čísel jako struktur . . . 14
  - 2.3.1 Přirozená čísla . . . 15
  - 2.3.2 Celá čísla . . . 15
  - 2.3.3 Racionální čísla . . . 15
- 2.4 Reálná čísla . . . 16
  - 2.4.1 Představa . . . 16
  - 2.4.2 Existující nástroje . . . 17

II Implementace 22

3 Tnumy 22
- 3.1 Vztah čísel a tnumů . . . 22
- 3.2 Ludolfovo číslo . . . 24
- 3.3 Přenásobování numem . . . 25

4 Operace tnumů 27
- 4.1 Aditivní operace . . . 27
- 4.2 Multiplikativní operace . . . 28
- 4.3 Mocninné operace . . . 34

5 Funkce tnumů **35**
5.1 Aproximace funkcí . . . 35
5.2 Exponenciála . . . 36
5.2.1 Exponenciála čísla . . . 37
5.2.2 Exponenciála tnumu . . . 39
5.3 Goniometrické . . . 43
5.3.1 Sinus . . . 43
5.3.2 Kosinus . . . 45
5.3.3 Další goniometrické funkce . . . 46
5.4 Logaritmus . . . 47

III Rozhraní **49**

6 Uživatelské funkce **49**
6.1 Vymezení . . . 49
6.2 Instalace . . . 51
6.3 Převody a konstanty . . . 51
6.4 Operace . . . 53
6.5 Funkce . . . 54
6.6 Rychlost . . . 55
6.7 Vnější volání . . . 56

7 Perspektiva **57**
7.1 Kalkulačka . . . 57
7.1.1 Vytvoření tnumu . . . 57
7.1.2 Vypsání tnumu . . . 59
7.2 Optimalizace . . . 61
7.2.1 Paralelizace . . . 61
7.2.2 Databáze . . . 62
7.3 Peripetie . . . 63
7.3.1 Ideové . . . 63
7.3.2 Matematické . . . 64
7.3.3 Implementační . . . 64
7.3.4 Mezery . . . 64
7.4 Svébytnost . . . 65
7.4.1 Kontext knihovny . . . 65
7.4.2 Kontext práce . . . 66
7.4.3 K zapamatování . . . 66
7.4.4 Adakvátnost . . . 66

Závěr **67**

Conclusions **68**

Seznam literatury **69**

A Některé důkazy **72**
A.1 Fakt 24 – O částečném součtu geometrické řady 72
A.2 Fakt 25 – O geometrické řadě 72
A.3 Fakt 26 – O zbytku geometrické řady 72
A.4 Fakt 50 – O exponenciále jako Maclaurinově řadě 72
A.5 Fakt 58 – O sinu jako Maclaurinově řadě 73
A.6 Fakt 61 – O kosinu jako Maclaurinově řadě 73

B Obsah přiloženého CD/DVD **74**

Seznam tabulek

1 Symboly operací s čísly 7
2 Nekonečná vstupně/výstupní tabulka funkce *sinus* 7
3 Vnitřní a vnější funkce 50
4 Nízkoúrovňové a vysokoúrovňové funkce 50
5 Doba výpočtů daných výrazů 55
6 Funkce nabízené knihovnou `tnums` 56

Seznam obrázků

1 Seřazení bitů v datovém typu binary32 [19] 12
2 Přirozená čísla v jazyce C 13
3 Celá čísla v jazyce C 13
4 Racionální čísla v jazyce C 14
5 Používání knihovny `mpmath` 18
6 Používání knihovny `JScience` 19
7 Používání knihovny `GMP` 20
8 Implicitní používání knihovny `CLN` 20
9 Používání knihovny `computable-reals` 21
10 Obraz přesnosti po průchodu exponenciálou 39
11 Vzor přesnosti před průchodem exponenciálou 40
12 Zobrazení neznámé w 41
13 Načtení knihovny `tnum` do `SBCL` 51

Seznam definic

1 Definice (Číslo – naivní [1]) 2
4 Definice (Induktivní množina) 4
5 Definice (Přirozená čísla) 4
8 Definice (Uspořádaná n-tice [4]) 7
9 Definice (Kartézský součin [12] [13]) 8
11 Definice (Relace [12]) 8
12 Definice (Zobrazení [12]) 8
13 Definice (Reálná posloupnost [12]) 8
14 Definice (Nekonečná číselná řada [15]) 8
15 Definice (Posloupnost částečných součtů [15]) 8
16 Definice (Konvergence řady [15]) 9
17 Definice (Divergence řady [15]) 9
18 Definice (Zbytek po n-tém členu řady [15]) 9
19 Definice (Reálná funkce reálné proměnné [14]) 9
21 Definice (Mocninná řada [15]) 10
22 Definice (Taylorova řada [15]) 10
23 Definice (Geometrická řada [15]) 10
29 Definice (Tnum) 22
32 Definice (Ludolfovo číslo [30]) 24
38 Definice (Nenulový tnum) 28
39 Definice (Bezpečné epsilon) 29
56 Definice (Precizní iterátor exponenciály) 42
70 Definice (Uživatelská funkce) 49
71 Definice (Zlatý řez [37]) 54

Seznam faktů

24 Fakt (Částečný součet geometrické řady [15]) 10
25 Fakt (Geometrická řada) 10
26 Fakt (Zbytek geometrické řady) 10
33 Fakt (Ludolfovo číslo jako řada [32]) 24
37 Fakt (Rozdíl tnumů) 28
43 Fakt (Součin tnumů) 33
44 Fakt (Podíl tnumů) 33
46 Fakt (Odmocnina jako mocnina [7]) 34
48 Fakt (Taylorova věta [35]) 36
50 Fakt (Exponenciála jako Maclaurinova řada [15]) 37
51 Fakt (Omezení Taylorova zbytku exponenciály) 37
58 Fakt (Sinus jako Maclaurinova řada [15]) 43
61 Fakt (Kosinus jako Maclaurinova řada [15]) 45
63 Fakt (Tangens jako poměr sinu a kosinu [7]) 46
65 Fakt (Kosekans jako obrácená hodnota sinu [7]) 46
66 Fakt (Sekans jako obrácená hodnota kosinu [7]) 46
67 Fakt (Kotangens jako obrácená hodnota tangentu [7]) 47
68 Fakt (Logaritmus jako řada [36]) 47
72 Fakt (Obecný logaritmus jako podíl přirozených [7]) 54

Seznam vět a lemmat

30 Lemma (O numu jako tnumu) 23
31 Lemma (O převodu tnumu na num) 23
34 Věta (O přenásobení tnumu racionální konstantou) 25
36 Věta (O součtu tnumů) 27
40 Lemma (O nenulovém tnumu nenulového čísla) 29
41 Věta (O převráceném tnumu) 29
42 Věta (O součinu dvou tnumů) 31
45 Lemma (O mocnině tnumu) 34
55 Lemma (O přesnosti závislé proměnné) 42
60 Lemma (O sinu tnumu) 44
73 Lemma (O sčítání tnumu s numem) 58

Seznam zdrojových kódů

1 Lispový kód (num-to-tnum) . . . 23
2 Lispový kód (rat-expt) . . . 23
3 Lispový kód (tnum-to-num) . . . 24
4 Lispový kód (tnum-pi) . . . 25
5 Lispový kód (tnum*num) . . . 26
6 Lispový kód (-tnum) . . . 26
7 Lispový kód (tnum+) . . . 28
8 Lispový kód (tnum-) . . . 28
9 Lispový kód (get-nonzero-num+eps) . . . 29
10 Lispový kód (/tnum) . . . 30
11 Lispový kód (create-list-for-multiplication) . . . 33
12 Lispový kód (tnum*) . . . 33
13 Lispový kód (tnum/) . . . 34
14 Lispový kód (factorial) . . . 35
15 Lispový kód (num-exp) . . . 38
16 Lispový kód (tnum-e) . . . 38
17 Lispový kód (tnum-exp) . . . 43
18 Lispový kód (num-sin) . . . 44
19 Lispový kód (tnum-sin) . . . 44
20 Lispový kód (num-cos) . . . 45
21 Lispový kód (tnum-cos) . . . 46
22 Lispový kód (tnum-tan) . . . 46
23 Lispový kód (tnum-csc) . . . 46
24 Lispový kód (tnum-sec) . . . 47
25 Lispový kód (tnum-ctan) . . . 47
26 Lispový kód (num-ln) . . . 47
27 Lispový kód (tnum-ln) . . . 48
28 Lispový kód (tnum-expt) . . . 48
29 Lispový kód (tnum-root) . . . 48
30 Lispový kód (tnum-to-string) . . . 52
31 Lispový kód (tnum-1+) . . . 53
32 Lispový kód (tnum-1-) . . . 53
33 Lispový kód (tnum-sqrt) . . . 53
34 Lispový kód (tnum-phi) . . . 54
35 Lispový kód (tnum-log) . . . 54
36 Lispový kód (tsin) . . . 57
37 Lispový kód (t+) . . . 58
38 Lispový kód (t*) . . . 59
39 Lispový kód (calc-sin) . . . 59
40 Lispový kód (global-sin) . . . 60

Seznam testů

1 Lispový test (`num-to-tnum`) 52
2 Lispový test (`tnum-to-num`) 52
3 Lispový test (Typ výstupu je číslo) 52
4 Lispový test (`tnum-string` a `tnum-pi`) 53
5 Lispový test (`tnum-string` a `tnum-e`) 53
6 Lispový test (`tnum-phi`) 54
7 Lispový test (`tnum-log`) 55
8 Lispový test (`tnum-sin`) 55
9 Lispový test (`tsin`) 57
10 Lispový test (`t+`) 58
11 Lispový test (`t*`) 59
12 Lispový test (`calc-sin`) 60
13 Lispový test (`global-sin`) 60

Seznam zbytku

2 Příklad (Nejnižší přirozená čísla coby množiny) 3
3 Poznámka (Provázání hodnot a podmnožin) 3
6 Poznámka (Značení přirozených čísel) 5
7 Poznámka (Nula je přirozená) 5
10 Poznámka (Množinovost kartézského součinu) 8
20 Příklad (Sinus jako zobrazení) 9
27 Příklad (Převod z váženého pozičního kódu) 11
28 Příklad (Plovoucí číslo – jednoduchá přesnost) 12
35 Důsledek (Opačný tnum) 26
47 Připomenutí (Taylorova a Maclaurinova řada) 35
49 Poznámka (Značení exponenciály) 36
52 Důsledek (O tnumu exponenciály numu) 37
53 Poznámka (Eulerovo číslo jako exponenciála) 38
54 Poznámka (Obecnější přesnost vzoru) 42
57 Důsledek (O exponenciále tnumu) 42
59 Důsledek (Sinus numu) 43
62 Důsledek (Kosinus numu) 45
64 Úmluva (O vypuštění některých důsledků) 46
69 Poznámka (Dokončení systému) 48

0 Úvod

Funkcionální programování umožňuje důsledně popsat matematický svět, což v této práci bude ukázáno na příkladu výpočtů reálných čísel. Jako funkcionální jazyk byl zvolen Common Lisp (Lisp) pro jeho syntaktickou jednoduchost, dynamičnost a lenost. Každý Lispový kód následuje vždy za matematickým výrazem a ve stejném znění ho převádí. Lisp představuje nástroj, pomocí něhož realizujeme matematické výpočty, hlavním těžištěm poznání je ale vybudovaná matematická teorie, která není specifická pro konkrétní programovací jazyk.

V první části je teoreticky popsáno, co čísla jsou a jak se rozvrstvují podle vlastností na obory. Také je zde ukázáno, jak se s čísly operuje a je představen pojem zobrazení a jeho dva důležité typy – funkce a posloupnost. Je zde zavedena stěžejní představa, co znamená přesná reprezentace čísla pro jeho různé podoby a představeno několik existujících knihoven.

Ve druhé části práce je popsána konstrukce knihovny `tnums` přesně vyčíslující reálná čísla. Je využita existence racionálních čísel v Lispu a přidáno Ludolfovo číslo. Zavedená čísla jsou poté kombinována operacemi a měněna svými funkcemi, v návaznosti na to je zavedeno Eulerovo číslo.

V závěrečné části je uvedeno praktické užití řešení a vymezen pojem uživatelské funkce. Je zde ukázáno, jak lze takové funkce přidávat, a je zde demonstrováno, že některé vznikly mimoděk už při programování. Také doplníme poslední konstantu, a to Zlatý řez. V poslední kapitole je představena možnost napsání přesné kalkulačky, nápady na urychlení práce knihovny `tnums` a nakonec její nedostatky.

Značení

$\square$	konec důkazu
$\blacksquare$	konec poznámky
modré pozadí	obrázek
nachové pozadí	tabulka
(a, b)	otevřený interval mezi a a b
$[a, b]$	uzavřený interval mezi a a b
$\rightarrow$	logická implikace nebo „do“
$\leftrightarrow$	logická ekvivalence
$\Leftrightarrow$	„právě tehdy, když“
$\cup$	množinové sjednocení
$\cap$	množinový průnik
$\|a\|$	absolutní hodnota čísla a
$:=$	přiřazení
$\frac{d}{dx}f(x)$	derivace funkce f v bodě x, je-li jasná proměnná, pak jen $f^{'}(x)$.

Část I
Teorie

V teoretické části je představena axiomatická teorie množin a naznačeno, jak se z ní dají vytvořit čísla. Je ukázáno, že čísla jsou množinami. Poté stručně představím matematické operace a matematické funkce. Dále vyplyne, že jakékoli číslo lze vyjádřit jako funkci a že funkce je také množina. Závěr teoretické části je zaměřen na problém uložení čísla v paměti počítače, která je fyzicky konečná. Nejprve je rozebrán v čistě teoretické rovině a poté je diskutováno řešení v podobě reálně existujících knihoven.

1 Čísla

Číslo je matematický objekt, který na intuitivní úrovni všichni chápeme. To znamená, že každý dokáže o libovolném objektu říci, zda se jedná o číslo, nebo nikoli, a všichni se na tom shodneme. Je číslem 3? Je číslem 2.999...? Je číslem e?

Vágní pokus o definici by mohl vypadat tak jako na české wikipedii:

Definice 1 (Číslo – naivní [1])
Číslo je abstraktní entita užívaná pro vyjádření množství nebo pořadí.

Uvedená definice přiřazuje číslům dvě funkce – kardinální a ordinální. Jinak o povaze čísel neříká mnoho. Víme, že je to abstraktní entita – to znamená, že ji nelze zapsat samu o sobě, ale je třeba použít nějaký symbol. Symbolem reprezentujícím číslo tři je 3, $\frac{6}{2}$ nebo třeba i 2.999.... Symbolický zápis čísla zřejmě není jednoznačný. 3 není to samé jako 2.999..., ale číslo 3 je to samé jako číslo 2.999.... Z toho plyne, že symbol s referencí na nějakou entitu a tato entita samotná se od sebe liší. Alenka takhle zjistila rozdíl mezi tím, jak se říká názvu písně, jaké je její jméno, jak se píseň jmenuje a co píseň opravdu je [2]. Známe-li rozdíl mezi koncepty čísla a symbolu, který ho repreznetuje, je možné využít pro reprezentaci čísel i jiné symboly než číslice jako jsou například písmena či složené matematické výrazy – například $e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = \sum_{n\in\mathbb{N}}\frac{1}{n!}$.

Číslo je tedy charakterizováno nikoli svým zápisem, ale svým obsahem. Tato vlastnost je společná s jinými matematickými objekty – s množinami. Množina je také dána svým obsahem, nikoli svým zápisem (této vlastnosti se říká princip extenzionality). Například $\{0,1,2,3\} = \{n | n \in \mathbb{N} \land n \leq 3\}$ je stejná množina zapsaná dvěma různými způsoby. Symbol „=“ je tedy ekvivalence na hodnotách.

1.1 Přirozená čísla

Přirozená čísla jsou čísla, jejichž *kardinalita* určuje počet nedělitelných částí celku. Historicky asi vznikla nejdříve, proto se nazývají přirozená (anglicky *natural* – přírodní). Jedná se o čísla nula, jedna, dva, atd. Běžnými symboly těchto čísel jsou 0, 1, 2, atd. Vlastností přirozených čísel je, že *každé* číslo n má následníka. Toho značíme $s(n)$ nebo $n+1$ [3].

Hodnota přirozeného čísla je počet entit v nějakém souboru – například počet krav ve stádu, počet prstů na ruce, nebo počet třešniček na dortu. *Následník* takového čísla značí, kolik entit bude v souboru, pokud se do stáda narodí další kráva, na ruku se přišije jeden prst nebo je do krému přidána další třešnička.

I přirozená čísla jsou *množiny*. Konkrétně v tomto textu je zavedu v Zermelově-Fraenkelově (ZF) axiomatici teorii množin. Je snadné nahlédnout, že stačí zkonstruovat číslo nula a *zobrazení* s přiřazující každému číslu následníka. Poté budeme mít celou nekonečnou množinu přirozených čísel zkonstruovanou. Protože se jedná o výklad teorie množin, celý zbytek této podkapitoly je pouze velmi stručný výtah z [4].

Definice přirozených čísel je induktivní a stojí na jednoduché myšlence Johna von Neumanna, že „přirozené číslo je množina všech menších přirozených čísel“. Číslo nula je zde prázdná množina značená 0, $\emptyset$ nebo $\{\}$. Následník čísla je sjednocení čísla s množinou obsahující toto číslo, čili $s(n) = n \cup \{n\}$.

Několik nejmenších přirozených čísel vypadá tedy následovně.

Příklad 2 (Nejnižší přirozená čísla coby množiny)

$$0 = \emptyset \tag{1}$$

$$1 = s(0) = 0 \cup \{0\} = \emptyset \cup \{\emptyset\} = \{\emptyset\} \tag{2}$$

$$2 = s(1) = 1 \cup \{1\} = \{\emptyset\} \cup \{\{\emptyset\}\} = \{\emptyset, \{\emptyset\}\} \tag{3}$$

$$3 = s(2) = 2 \cup \{2\} = \{\emptyset, \{\emptyset\}\} \cup \{\{\emptyset, \{\emptyset\}\}\} = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \tag{4}$$

$$\begin{aligned} 4 = s(3) = 3 \cup \{3\} = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \cup \{\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\} = \\ = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\} \end{aligned} \tag{5}$$

Poznámka 3 (Provázání hodnot a podmnožin)

Všimněme si, že kardinalita v tomto pojetí znamená počet podmnožin, neboli číslo n má n podmnožin. Tím se provázaly dva zdánlivě cizí pojmy a sice *podmnožinovost* a *hodnota* čísla. V tomto kontextu pak není překvapivá podobnost srovnávacích symbolů $\subseteq$ a $\leq$. ■

Zermelova-Fraenkelova teorie stojí na následujících pěti axiomech a jednom axiomovém schématu (někdy uváděno jako 6 axiomů).

- Axiom extenzionality: $(\forall u)((u \in x \leftrightarrow u \in y) \rightarrow x = y)$
- Axiom fundovanosti: $(\forall a)(a \neq \emptyset \rightarrow (\exists x)(x \in a \wedge x \cap a = \emptyset))$
- Axiom sumy: $(\forall a)(\forall b)(\exists z)(\forall x)(x \in z \leftrightarrow (x = a \vee x = b))$
- Axiom potence: $(\forall a)(\exists z)(\forall x)(x \in z \leftrightarrow x \subset a)$
- Axiom nekonečna: $(\exists z)(\emptyset \in z \wedge (\forall x)(x \in z \rightarrow x \cup \{x\} \in z))$
- Schéma axiomů nahrazení: Je-li $\Psi(u, v)$ formule, která neobsahuje volné proměnné w a z, potom formule

$$(\forall u)(\forall v)(\forall w)((\Psi(u,v) \wedge \Psi(u,w)) \rightarrow v = w) \rightarrow \\ \rightarrow (\forall a)(\exists z)(\forall v)(v \in z \leftrightarrow (\exists u)(u \in a \wedge \Psi(u,v)))$$

  je axiom nahrazení.

Z těchto axiomů je možné odvodit i (slabší) tvrzení, která se někdy přijímají jako axiomy a sice

- Axiom dvojice: $(\forall a)(\forall b)(\exists z)(\forall x)(x \in z \leftrightarrow (x = a \vee x = b))$
- Schéma axiomů vydělení: Je-li $\varphi(x)$ formule, která neobsahuje volnou proměnnou z, potom formule $(\forall a)(\exists z)(\forall x)(x \in z \leftrightarrow (x \in a \wedge \varphi(x)))$ je axiom vydělení.

Z axiomů ZF je pro naše zkoumání důležitý například axiom nekonečna – existuje alespoň jedna (nekonečná) množina. Za pomoci dalších axiomů poté konstruujeme další prvky, jako třeba prázdnou množinu – tu dostaneme pomocí axiomu vydělení s jakoukoli množinou a a formulí $\varphi(x) = x \neq x$ – prázdná množina je tedy $\emptyset = \{x | x \in a \wedge x \neq x\}$. Dále získáváme, že sjednocení množin je také množina, podobně jejich průnik.

Definice 4 (Induktivní množina)
Množina A je induktivní, pokud platí $(\emptyset \in A) \wedge (\forall a \in A)((a \cup \{a\}) \in A)$.

Definice 5 (Přirozená čísla)

$$\mathbb{N} = \bigcap \{A | A \textit{ je induktivní}\} \tag{6}$$

Množina přirozených čísel je nejmenší induktivní množina a je podmnožinou každé induktivní množiny. $\mathbb{N}$ je nejmenší nekonečný *kardinál* a také jediný *spočetný* kardinál.

Poznámka 6 (Značení přirozených čísel)

Množina z v axiomu nekonečna je stejná množina jako $\mathbb{N}$. V teorii množin se značí ω, při zkoumání kardinality pak $\aleph_0$. Pro označení přirozených čísel bez nuly zaveďme horní index $+$ ($\mathbb{N}^+ = \mathbb{N} - \{0\}$). ■

Poznámka 7 (Nula je přirozená)

Z definice přirozených čísel podle ZF přímo vyplývá, že součástí přirozených čísel je i číslo nula. Dle mého i stádo, ve kterém není žádná kráva, je stále stádem; ruka bez prstů je stále rukou; dort bez třešniček je i tak dort. ■

1.2 Vyšší obory čísel

V minulé podkapitole jsem rigorózně zavedl přirozená čísla. S dalšími obory už nebudu postupovat takto exaktně, u jedné množiny to – myslím – stačilo.

Vyšším oborem čísel jsou čísla celá, značená $\mathbb{Z}$ a jsou to všechna čísla, která mohou vzniknout libovolným odčítáním přirozených čísel. Jejich výčet je diskrétní: $\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$[5]. Zde „$-$“ je znaménko odčítání (snížení hodnoty prvního argumentu o hodnotu druhého). Značení: místo $0 - 2$ se píše úžeji -2.

Další obor vyjadřuje poměr velikosti nějakého celku vůči jinému. Zde si opět vypomůžu obory, které už máme zadefinované a mohu představit čísla racionální, pro která se vžil zápis $\mathbb{Q}$ a takovým číslem je číslo x, pokud jde zapsat jako y/z, kde $y \in \mathbb{Z}$ a $z \in (\mathbb{Z} \setminus \{0\})$ a „/“ je symbol operace dělení [6]. Taková čísla jsou například $\mathbb{Q} = \{0, 1, 1/2, -1, 1/3, -1/2, 2, 1/4, \ldots\}$, a jsou také spočetná.

Stále v našem číselném systému nemáme například délku úhlopříčky jednotkového čtverce – takovéto číslo značíme $\sqrt{2}$. Nebo třeba poměr obvodu kružnice k jejímu průměru – toto značíme π. Když do našeho systému doplníme všechna tato čísla, získáváme konečně tzv. číselnou osu (přímku) reprezentující čísla reálná, značená $\mathbb{R}$. Ta čísla, která jsme přidali a tudíž byla iracionální (nebyla racionální), označíme $\mathbb{I}$, $\mathbb{I} = \mathbb{R} \setminus \mathbb{Q}$ [7]. Ke každým dvěma reálným číslům r a ε lze najít racionální číslo q tak, aby

$$|r - q| \leq \varepsilon \text{ [8]}. \tag{7}$$

V reálných číslech matematikové objevili ještě jemnější struktury, než jen dělení na obory, které jsem teď představil. První jmenované iracionální číslo ($\sqrt{2}$) je tzv. *algebraické*, protože může vypadnout jako řešení z nějaké algebraické rovnice, např $x^2 = 2$ nebo $(-x)^2 = 2$, zatímco druhé jmenované – tzv. *Ludolfovo číslo* (π) takovouto vlastností nedisponuje, je *nealgebraické*, též *transcendentní* [4]. Dále v reálných číslech existují tzv. *rekurzivní čísla* (*computable reals* – vizte kapitolu 2.4.2.5), která se dají vyčíslit v konečném čase. Pro reálné číslo r a dané ε existuje vyčíslitelná funkce (pro konečný vstup někdy skončí), jejímž výsledkem je racionální číslo q tak, že platí nerovnice 7. Ostatní čísla jsou nerekurzivní a protože Turingových strojů je spočetně mnoho, je nerekurzivních čísel nespočetně mnoho [9]. Poznamenejme už nyní, že přechod od racionálních čísel k rekurzivním je velmi složitý a tato práce je hlavně o tomto přechodu. V jednom oboru jsou i složité věci vcelku jednoduché, zatímco v tom druhém jsou i jednoduché věci vceklu složité.

Pokud nebudeme uvažovat pouze jednu číselnou osu, ale číslo bude mít více složek, přesněji 2^n složek, pak připouštíme existenci ještě vyšších číselných oborů – komplexních čísel ($\mathbb{C}, n = 1$), kvaternionů ($\mathbb{H}, n = 2$), oktonionů ($\mathbb{O}, n = 3$) – těmto čtyřem (společně s reálnými čísly) oborům říkáme *normované algebry s dělením* (*normed division algebra*) [10]. Ještě vyšší obory (sedeniony – $\mathbb{S}, n = 4$ atd.) už jsou úplně mimo ambice tohoto úvodu.

V této práci již nadále „číslem“ myslím číslo *rekurzivní*. Jiným názvem této práce by tedy mohlo být „Přesné výpočty s rekurzivními čísly“. V jistém smyslu tedy nesplním zadání, ale dokazatelně nejde počítat se *všemi* reálnými čísly, nýbrž s jejich vlastní podmnožinou v podobě rekurzivních čísel.

1.3 Operace s čísly

Už při vymezování číselných oborů jsem zmínil 3 operace, které čísla většinou „zmenšují“, bylo to *odčítání* (rozdíl), *dělení* (poměr) a *odmocňování* (odmocnina). K nim patří ještě operace opačné, které čísla veskrze „zvětšují“ a ty po řadě nazýváme *sčítání* (součet), *násobení* (součin) a *umocňování* (mocnina).

Binární operace se značí znaménkem mezi operandy. Například součet čísel x a y značíme $x + y$, součin $x * y$ atd. Základní značení operací je uvedeno v tabulce 1. Binárním operacím, které mají jako první operand neutrální prvek budeme říkat unární. Je to číslo opačné (k číslu x je opak číslo $0 - x$ a značíme ho $-x$) a převrácené číslo (k číslu $x \neq 0$ je jeho převrácením číslo $1/x$ a značíme ho $/x$ nebo x^{-1}).

Operacím $\langle +, - \rangle$ říkáme aditivní, $\langle +, / \rangle$ multiplikativní a $\langle \hat{}\,, \sqrt{} \rangle$ mocninné. Zajímavé jsou vztahy mezi operacemi. Že se dvojice jmenují spolu naznačuje jejich příbuznost. Tyto vztahy bych nazval „horizontální“. Mnohem zajímavější jsou ale vztahy „vertikální“. Platí

$$x * n = \underbrace{x + x + \ldots + x}_{n\text{-krát}} \text{ a také } x^n = \underbrace{x * x * \ldots * x}_{n\text{-krát}}. \tag{8}$$

Máme tedy návod, jak teoreticky vytvořit více operací. Další operací je

$$x \# n = \underbrace{x\hat{}x\hat{}\ldots\hat{}x}_{n\text{-krát}} \tag{9}$$

a nazýváme ji *tetrace* (*tetration*) [11].

Pro operace vyšší arity je pak zvykem používat prefixovou notaci jednoho velkého znaménka – součet $a_0 + a_1 + a_2$ je pak zapsán jako $+(a_0, a_1, a_2)$ nebo též $\sum_{i=0}^{2}(a_i)$. Součin $a_0 * a_1 * a_2$ pak $\ast(a_0, a_1, a_2)$ nebo $\prod_{i=0}^{2}(a_i)$.

Tabulka 1: Symboly operací s čísly

Operace	Značení
sčítání	$x + y$
odčítání	$x - y$
násobení	$x * y$, $x \cdot y$ nebo xy
dělení	x/y nebo $\frac{x}{y}$
umocňování	x^y nebo $x\hat{\ }y$
odmocňování	$\sqrt[y]{x}$

Tabulka zobrazuje zápis binárních operací s čísly x a y.

1.4 Funkce čísel

Další zajímavá manipulace s čísly jsou tzv. funkce, které si (ne)lze představit jako nekonečnou tabulku o dvou sloupcích – výstupní a výstupní hodnoty (jako příklad pro funkci *sinus* jsem vytvořil tabulku 2). Příkladem takových funkcí jsou funkce *goniometrické*, funkce *exponenciální* či *logaritmus*.

Tabulka 2: Nekonečná vstupně/výstupní tabulka funkce *sinus*

Vstup	Výstup
0	0
1	$0.8414709848078965\ldots$
2	$0.9092974268256817\ldots$
3	$0.1411200080598672\ldots$
4	$-0.7568024953079282\ldots$
$\vdots$	

Tabulka zobrazuje v prvním sloupci vstup do funkce sinus a ve druhém její výstup. Tabulka je nekonečná ve vertikálním směru, v horizontálním jsou pouze dva sloupce – pro argument a vrácenou hodnotu.

Funkce jsou velmi užitečným a potřebným nástrojem takřka ve všech odvětvích matematiky a jsou často zdrojem *iracionálních* (*transcendentních*) čísel – například takový atan(1) dá za výsledek čtvrtinu *Ludolfova čísla*.

Definice 8 (Uspořádaná n-tice [4])

Jsou-li dány množiny $a_1, a_2, \ldots a_m$, uspořádanou n-tici množin $a_1, a_2, \ldots, a_n$ pro $n < m$ definujeme tak, že pro $n = 1$ položíme

$$\langle a_1 \rangle = a_1 \tag{10}$$

a je-li již definována uspořádaná n-tice $\langle a_1, a_2, \ldots, a_n \rangle$, položíme

$$\langle a_1, a_2, \ldots a_{n+1} \rangle = \langle \langle a_1, a_2 \ldots a_n \rangle, a_{n+1} \rangle. \tag{11}$$

Definice 9 (Kartézský součin [12] [13])

Nechť $A_0, A_1 \dots A_n$ *jsou množiny. Symbolem* $\bigtimes_{i=0}^{n} A_i$ *či* $A_0 \times A_1 \times \dots \times A_n$ *označujeme množinu všech uspořádaných* $(n+1)$*-tic tvaru* $\langle a_0, a_1, \dots a_n \rangle$*, přičemž* $(a_0 \in A_0) \wedge (a_1 \in A_1) \wedge \dots \wedge (a_n \in A_n)$*, neboli*

$$A_0 \times A_1 \times \dots \times A_n = \{\langle a_0, a_1, \dots, a_n \rangle | (\forall i \in \mathbb{N}, i \leq n)(a_i \in A_i)\} \tag{12}$$

a tuto množinu nazýváme kartézským součinem množin $A_0, A_1, \dots A_n$.

Poznámka 10 (Množinovost kartézského součinu)

Že je kartézský součin (KS) množina jsem zavedl definitoricky, jako to udělaly autorky v [12], ovšem možná to nemusí být úplně jasné. V [4] se kartézský součin zavádí jako třída (soubor množin, který množina být nemusí – například třída všech množin množinou není) a poté se pomocí axiomu potence jeho množinovost dokazuje. Tato práce na tomto faktu nestojí a proto jsem si dovolil přijmout množinovost KS jako fakt, ač je – podobně jako všech 13 axiomů reálných čísel včetně axiomu o supremu (poté tedy věty o supremu) [14] – dokazatelná z ZF teorie množin. ■

Definice 11 (Relace [12])

Relace mezi množinami A *a* B *je libovolná podmnožina* $\mathcal{R}$ *kartézského součinu* $A \times B$*. Je-li* $A = B$*, mluvíme o relaci na* A*. Náleží-li dvojice* $\langle a, b \rangle$ *relaci* $\mathcal{R}$*, t.j.* $\langle a, b \rangle \in \mathcal{R}$*, říkáme, že* a *a* b *jsou v relaci* $\mathcal{R}$*, a zapisujeme též* $a\mathcal{R}b$.

Definice 12 (Zobrazení [12])

Relaci $f \subseteq A \times B$ *nazveme zobrazením množiny* A *do množiny* B*, jestliže platí, že ke každému prvku* $x \in A$ *existuje právě jeden prvek* $y \in B$ *takový, že* $\langle x, y \rangle \in f$.

Je-li relace $f \subseteq A \times B$ zobrazení, pak skutečnost, že $\langle x, y \rangle \in f$ zapisujemme ve tvaru $y = f(x)$. Rovněž používáme zápis $f : A \to B$, což znamená, že f je zobrazení A do B. Dále x nazýváme *nezávisle proměnnou* a y *závisle proměnnou* [14].

Definice 13 (Reálná posloupnost [12])

Zobrazení množiny $\mathbb{N}$ *do množiny* $\mathbb{R}$ *nazýváme reálná posloupnost.*

Místo obecného značení $a : \mathbb{N} \to \mathbb{R}$ pro zobrazení resp. značení $a(n)$ pro obraz bodu n se vžilo pro posloupnost značení $\{a_n\}_{n \in \mathbb{N}}$ nebo $\{a_n\}_{n=0}^{\infty}$. Obraz bodu n se značí a_n a říkáme mu také n-tý člen posloupnosti a [12].

Definice 14 (Nekonečná číselná řada [15])

Nechť $\{a\}_{n \in \mathbb{N}}$ *je reálná posloupnost. Symbol* $\sum_{n \in \mathbb{N}} a_n$ *nebo* $a_0 + a_1 + a_2 + \dots$ *nazýváme nekonečnou číselnou řadou.*

Definice 15 (Posloupnost částečných součtů [15])

Posloupnost $\{s_n^a\}$*, kde* $s_n^a = \sum_{i=0}^{n} a_i$ *nazýváme posloupnost částečných součtů řady* $\sum_{n\in\mathbb{N}} a_n$*.*

Definice 16 (Konvergence řady [15])

Existuje-li vlastní limita $\lim_{n\to\infty} s_n^a = s$*, řekneme, že řada* $\sum_{n\in\mathbb{N}} a_n$ *konverguje a má součet* s*.*

Definice 17 (Divergence řady [15])

Neexistuje-li vlastní limita $\lim_{n\to\infty} s_n^a$*, řekneme, že řada* $\sum_{n\in\mathbb{N}}$ *diverguje. Pokud limita* $\lim_{n\to\infty} s_n^a$ *neexistuje, říkáme, že řada osciluje. Pokud je* $\lim_{n\to\infty} s_n^a = \infty$*, pak říkáme, že řada diverguje k* ∞*. Pokud je* $\lim_{n\to\infty} s_n^a = -\infty$*, pak říkáme, že řada diverguje k* $-\infty$*.*

Definice 18 (Zbytek po n-tém členu řady [15])

Nechť $\sum_{n\in\mathbb{N}} a_n$ *je konvergentní řada. Její součet* s *lze psát ve tvaru* $s = s_n^a + R_n^a$*, kde* $s_n^a = \sum_{i=0}^{n} a_i$ *je* n*-tý částečný součet řady* $\sum_{n\in\mathbb{N}} a_n$ *a* $R_n^a = \sum_{i=n+1}^{\infty} a_i$ *se nazývá zbytek po* n*-tém členu a znamená velikost chyby, které se doupouštíme, když místo celé řady poščítáme pouze prvních* n *členů posloupnosti* $\{a_n\}_{n\in\mathbb{N}}$*.*

Definice 19 (Reálná funkce reálné proměnné [14])

Buď $M \subseteq R$*. Zobrazení* $f : M \to \mathbb{R}$ *nazýváme reálnou funkcí reálné proměnné nebo stručně funkcí jedné proměnné.*

Množina M *se nazývá definiční obor funkce* f *a značí se* $D(f)$*, množina* $H(f) = \{f(x)|x \in M\}$ *se nazývá obor hodnot funkce* f*.*

Příklad 20 (Sinus jako zobrazení)

Funkce $y = sin(x)$ je definována pro všechna $x \in \mathbb{R}$ a její obor hodnot je interval $[-1, 1]$, jedná se tedy o reálnou funkci reálné proměnné. Pokud budeme hledět jen na obrazy $sin(x), x \in \mathbb{N}$ (jako v tabulce 2), jedná se o reálnou posloupnost.

Povšimněme si, že zobrazení $f : A \to B$ je definováno tak, že A i B jsou množiny. Jak ale víme z poznámky 10, je množinou i kartézský součin množin. Tedy lze definovat též zobrazení $\mathbb{R} \times \mathbb{I} \to \mathbb{Q} \times \mathbb{Z} \times \mathbb{N}$ atp., aniž bychom museli definici zobrazení jakkoli upravovat.

Při bližším zkoumáním by se daly najít jisté podobnosti mezi operacemi a funkcemi. Pokud se odprostíme od infixové notace a místo $a + b$ napíšeme $+(\langle a, b\rangle)$, lze i operace vyjádřit jako funkce. Matematickou operaci tedy chápeme jako speciální případ funkce, tedy že n-ární reálná operace je funkce $\bigtimes_{i=1}^{n} \mathbb{R} \to \mathbb{R}$.

Ba co více. Pokud vezmeme funkci $\emptyset \to \mathbb{R}$, zjistíme, že se jedná o konstantu, protože podle definice zobrazení pokud jsou v relaci $\langle\emptyset, x\rangle$ a $\langle\emptyset, y\rangle$, pak $x = y$. Čili taková funkce se vždy zobrazuje na stejné číslo a proto se jedná konstantu. Takže funkcemi můžeme vymodelovat jakákoli čísla i manipulaci s nimi.

Definice 21 (Mocninná řada [15])

Buď $\{a_n\}_{n\in\mathbb{N}}$ posloupnost reálných čísel, x_0 libovolné reálné číslo. Mocninnou řadou se středem v bodě x_0 a koeficienty a_n rozumíme řadu ve tvaru

$$a_0 + a_1(x - x_0) + a_2(x - x_0)^2 + \ldots + a_n(x - x_0)^n + \ldots = \sum_{n\in\mathbb{N}} a_n(x - x_0)^n. \quad (13)$$

Definice 22 (Taylorova řada [15])

Nechť funkce f má v bodě x_0 derivace všech řádů. Mocninnou řadu ve tvaru $\sum_{n\in\mathbb{N}} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n$ nazýváme Taylorovou řadou funkce f v bodě x_0.

Je-li $x_0 = 0$, mluvíme o Maclaurinově řadě funkce f a je ve tvaru $\sum_{n\in\mathbb{N}} \frac{f^{(n)}(0)}{n!}x^n$.

Zbytku Taylorovy řady říkáme Taylorův zbytek a značíme ho $R_n^{f,a}(x)$.

Definice 23 (Geometrická řada [15])

Řadu nazýváme geometrickou, pokud je ve tvaru

$$a + a * q + a * q^2 + a * q^3 + \ldots = \sum_{i\in\mathbb{N}} aq^i \quad (14)$$

Fakt 24 (Částečný součet geometrické řady [15])

Pro geometrickou řadu ve tvaru $\sum_{i\in\mathbb{N}} aq^i$ a $|q| < 1$ platí

$$s_n^a = \sum_{i=0}^{n} a * q^i = a\frac{1 - q^{n+1}}{1 - q} \quad (15)$$

Pro důkaz vizte podkapitolu A.1 v příloze A.

Fakt 25 (Geometrická řada)

Pro geometrickou řadu ve tvaru $\sum_{i\in\mathbb{N}} aq^i$ a $|q| < 1$ platí

$$\sum_{i\in\mathbb{N}} aq^i = \frac{a}{1 - q} \quad (16)$$

Pro důkaz vizte podkapitolu A.2 v příloze A.

Fakt 26 (Zbytek geometrické řady)

Pro n-tý zbytek geometrické řady $\sum_{i\in\mathbb{N}} aq^i$ platí

$$R_n = \frac{aq^{n+1}}{1 - q} \quad (17)$$

Pro důkaz vizte podkapitolu A.3 v příloze A.

2 Čísla v počítači

Standardně jsou čísla v počítači uložena jako sled bitů v nějaké reprezentaci vyjadřující hodnotu. Tímto způsobem lze reprezentovat číslo jen s danou přeností a rozsahem, což je však v řadě případů dostačující. Čísla lze reprezentovat i naprosto přesně – pomocí struktur je reprezentujícími a s nimi manipulujícími. I tyto samozřejmě musí být uloženy v paměti, v tomto případě se však nejedná o uložení hodnoty čísla, nýbrž se ukládá abstrakce, která číslo na požádání umí vygenerovat. Tento přístup nazýváme líným. Nejprve bude popsáno ukládání čísel jako hodnot a na jeho přesnost. Poté navrhneme struktury pro přesná čísla a nakonec budou ukázány existující knihovny.

2.1 Čísla uložená jako hodnoty

Paměť počítače většinou pracuje s tzv. *bity*, paměťovými buňkami nabývajícími dvou rozlišitelných hodnot [16]. Ke kódování čísel se tedy používá výhradně dvojková soustava. Celá následující podkapitola pojednávající o uložení čísel jejich hodnotami je převzata z [17].

2.1.1 Vážený poziční kód

Číslo v soustavě o základu b lze dešifrovat následovně:

$$(\dots a_2 a_1 a_0 . a_{-1} a_{-2} \dots)_b = \dots + a_2 b^2 + a_1 b + a_0 + \frac{a_{-1}}{b} + \frac{a_{-2}}{b^2} + \dots \tag{18}$$

kde čísla a_n nazýváme číslice a symbol „.“ řádovou tečkou – speciálně pak tečkou *desetinnou* ($b = 10$) nebo *dvojkovou* ($b = 2$).

Příklad 27 (Převod z váženého pozičního kódu)

$$(101010.101)_2 = (42.625)_{10} \tag{19}$$

2.1.2 Záporná čísla

V případě zápisu čísla váženým pozičním kódem lze strukturu pokrývající i záporná čísla přímočaře vytvořit přidáním příznaku (rozšířit jeho reprezentaci o jeden bit), zda se jedná o číslo kladné či nikoliv. V této reprezentaci lze vyjádřit zápornou i kladnou nulu. Tomutu kódování se říká kód *velikostí a znaménkem (signed-magnitude)*.

Jinou možností je záporná čísla vnímat jako opak čísel kladných i na úrovni reprezentace, čili záporné číslo opačné ke kladnému v binární reprezentaci vypadá jako negace bitů daného kladného čísla. Opět zde vyvstává problém se zápornou nulou. Tomutu kódování se říká *jedničkový doplněk (ones' complement)*.

Když od všech záporných čísel v jedničkovém doplňku odečteme číslo jedna, rozprostřeme čísla efektivně, tj. odpadne problém s dvojí reprezentací nuly. Tomutu kódování se říká *dvojkový doplněk (two's complement)*.

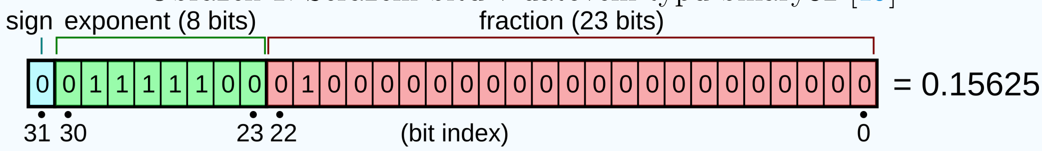
2.1.3 Plovoucí řádová tečka

U váženého pozičního kódu známe přesně pozici řádové tečky – je mezi číslicemi a_0 a a_{-1}. Alternativním přístupem je kódování s *plovoucí* řádovou tečkou.

Plovoucí číslo je dáno uspořádanou dvojicí $\langle e, f\rangle = f * b^{e-q}$, kde e nazýváme *exponent* a f *zlomkovou částí (fraction)*. Čísla b a q jsou konstanty dané použitým typem, b se nazývá *základ* a q *přesah.*

Příklad 28 (Plovoucí číslo – jednoduchá přesnost)
Číslo s jednoduchou přesností (*single* dle IEEE 754-1985, *binary32* dle IEEE 754-2008) je uloženo v paměti jako 32b struktura. První bit je příznak znaménka, dalších 8 bitů je pro exponent a dalších 23 bitů pro fraction. Konstanty jsou ohodnoceny následovně: $q = 127$, $b = 2$. V *C*, *C++*, *C#* nebo *Javě* se tento číselný datový typ nazývá *float*, v *Haskellu Float* a v *Lispu* pak *single-float* [18].

Obrázek 1: Seřazení bitů v datovém typu binary32 [19]



32-bitový datový typ pro čísla s plovoucí řádovou tečkou je dle standardu uložen v konfiguraci $\langle s, e, f\rangle = \langle 1, 8, 23\rangle$.

2.2 Přesná reprezentace čísel jako hodnot

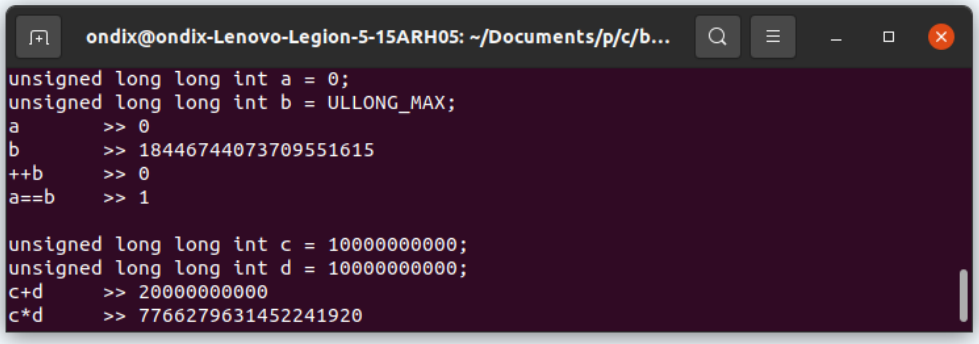
Nyní se podívejme, jaká čísla uložená jako hodnoty považujeme za přesná. Projdeme opět všechny obory jako v první kapitole a poprvé propojíme matematickou teorii s informatickou realitou. Jako modelový jazyk nám poslouží C.

Všechny ukázky v této kapitole představují reálné chování představených datových typů na konkrétní AMD64 architektuře. Nejde teď tedy o žádnou teorii a jen ukazuji reálné limity. Na 128-bitové architektuře mohou být tyto limity jiné a typy použitelnější. Nicméně tato práce směřuje k přesným výpočtům rekurzivních čísel bez ohledu na architekturu.

2.2.1 Přirozená čísla

Přirozená čísla jsou uzavřena na operace sčítání a násobení. V počítači se jako hodnoty ukládají pomocí váženého pozičního kódu a tento má horní limit, jak velké číslo lze na dané architektuře uložit. Takto uložená přirozená čísla ovšem nejsou na operace uzavřena, protože může dojít k přetečení – číslo ukládané se liší od čísla uloženého. Příklad tohoto chování vydíme na Obrázku 2.

Obrázek 2: Přirozená čísla v jazyce C



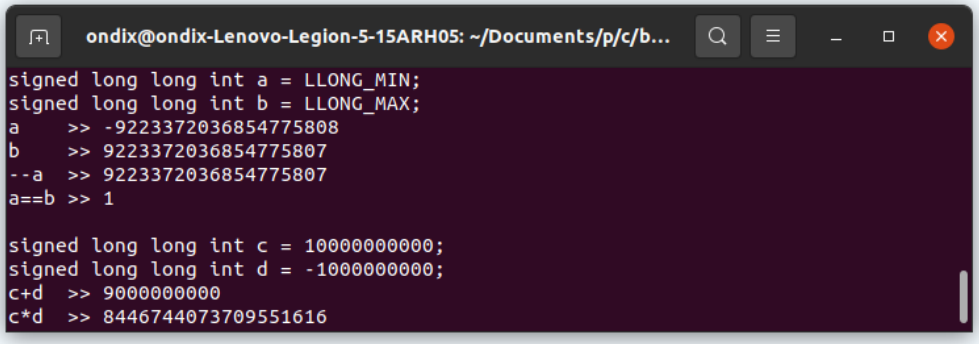
Největším datovým typem, který pracuje s přirozenými čísly je v jazyce C typ `unsigned long long int` a na 64-bitové architektuře umí uchovat čísla v intervalu $[0, 2^{64}-1]$. Na příkladu vidíme přetečení u inkrementace i násobení.

Pokud ale ukládáme přirozené číslo v intervalu, ve kterém jej zvládne uložit datový typ jako hodnotu, můžeme ho považovat za přesné.

2.2.2 Celá čísla

Celá čísla jsou uzavřena na sčítání, odčítání a násobení. V paměti počítače se pak musí používat kódování záporných čísel jako hodnot, často jde o dvojkový komplement. Opět zde vyvstává problém s limity jakéhokoli hodnotového datového typu, totiž že vyjadřitelná čísla jsou ohraničená a hrozí přetečení (a i podtečení). Implementace celého čísla jako hodnoty tedy není na operace uzavřená (Obr. 3).

Obrázek 3: Celá čísla v jazyce C



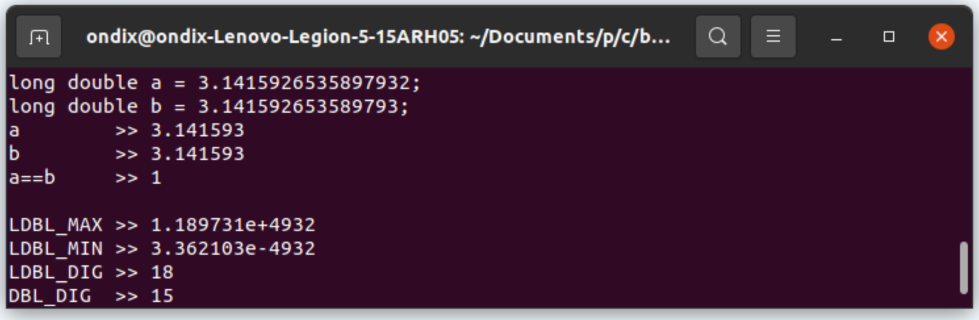
Nejširší datový typ jazyka C, který umí uložit celá čísla je `signed long long int`. Ten na 64-bitové architektuře zvládne uložit čísla z intervalu $[-2^{63}, 2^{63}-1]$. Na příkladu vidíme, že není uzavřen vůči operacím a že dochází k podtékání.

Pakliže ukládáme celé číslo jako hodnotu, která se vejde do datového typu bez přetečení nebo podtečení, lze takto vyjádřené číslo považovat za přesné.

2.2.3 Racionální čísla

Racionální číslo se v jazyce C ukládá jako číslo s plovoucí řádovou tečkou. S největší přesností se ukládá datový typ `long double`. Není specifikováno, jak má být přesný, pouze že má být minimálně tak přesný jako `double` [20]. Čísla, která jsou různá, ale kvůli zaokrouhlení se vyjádří jako stejná hodnota plovoucího čísla, jsou v tomto směru potom nerozeznatelná. Každý plovoucí typ pak má garantovanou přesnost, na kterou by se různá čísla neměla reprezentovat stejně.

Obrázek 4: Racionální čísla v jazyce C



Nejrozsáhlejším datovým typem čísla s plovoucí řádovou tečkou je v jazyce C `long double`. Na příkladu vidíme, že jeho přesnost by měla být 18 desetinných míst, ale už na patnácti místech jsou dvě různá zapisovaná čísla zapsána stejně. Někde se tedy stala chyba a přesnost `long double` je stejná jako `double`, ale knihovna `float.h` to nereflektuje.

Racionální čísla vyjádřená jako plovoucí můžeme považovat za přesná, pokud se nedostaneme mimo interval a přesnost stanovenou typem. Na Obrázku 4 vidíme, že rozsah datového typu `long double` je téměř 10^4 řádů a přesnost je maximálně 15 desetinných míst. Čísla s plovoucí řádovou tečkou jsou nejlepší přiblížení k racionálním číslům, které lze vyjádřit jako hodnoty.

2.3 Přesná reprezentace čísel jako struktur

Již víme, že některá čísla lze přesně uložit jejich hodnotou. Narazili jsme ale na limity standardních datových typů, například přetékání nebo nedostatečná přesnost. Nyní se podíváme na filozofii reprezentace čísla nějakým referenčním typem, nikoli hodnotovým. Vynořuje se zde paralela s popisem čísel z první kapitoly, že stejné číslo lze vyjádřit několika symboly, i když jeho hodnota je pouze jedna. Pokusíme se tedy nyní podívat na tento druhý přístup a zjistíme, že takto reprezentovaná čísla umíme velmi dobře v počítači vyjadřovat. Struktury si budeme pouze představovat a nebudu tedy psát žádný konkrétní kód, jen se pokusíme vymyslet, jak by konkrétní struktury mohly pracovat. Stačí totiž jediná struktura, na které se shodneme, že může představovat číslo a pak už bude jasné, že může existovat i nějaká její implementace.

2.3.1 Přirozená čísla

Jak bylo řečeno v minulé podkapitole, přirozená čísla jsou uzavřena na sčítání a násobení, `unsigned inty` ovšem nikoli. Potřebovali bychom vymyslet strukturu, která zajistí, že dokáže reprezentovat jakékoli přirozené číslo. Jako hodnoty umíme na n bytech uložit čísla 0 až $2^n - 1$. Nápad na přirozené číslo je rozumně upravovat toto n a pak velikost čísla bude omezena jen velikostí paměti.

Struktura představující reálné číslo musí zajišťovat, že při operacích jeho hodnota nepřeteče. Pokud tedy dostane požadavek na násobení nebo sčítání, musí mít připravený další prostor a tam posunout bity, které by normálně přetekly. To by šlo zajistit tak, že ve svém pomyslném paměťovém prostoru bude vždy po provedení operace procházet svých levých 2^{n-1} bitů a pokud tam najde alespoň jednu jedničku, požádá o alokaci dalšího prostoru o velikost 2^n, tedy zvedne n o jedna.

Pak už bude na operačním systému, kolik paměti bude moci struktuře přiřadit a tato paměť je vždy prakticky omezená, ale teoreticky je neomezená. Naši strukturu si tedy lze představit jako pouhý chytrý řadič bloků paměti za sebe. Pak lze *naprosto přesně* zapsat jakékoli přirozené číslo. Paměťově toto není moc efektivní, ale je to funkční představa a nám teď stačí jen princip, jak by něco takového fungovat mohlo a optimalizaci necháváme až na konkrétní knihovny.

2.3.2 Celá čísla

U celých čísel je nápad velmi podobný jako u přirozených čísel. Kromě přetečení lze hrozí také podtečení, protože celá čísla musí být uzavřena i na odčítání. Vytvořme naši strukturu pro celé číslo jako dvojici přirozeného čísla a příznaku kladnosti.

Metody struktury pak budou jen nastavovat příznak kladnosti – u násobení jako exklusivní disjunkci, odčítání je pak sčítání s negací příznaku a u sčítání se příznak nastaví podle většího čísla. Takto vytvořené struktury pak *naprosto přesně* reprezentují jakékoli celé číslo.

2.3.3 Racionální čísla

Racionální čísla jsme definovali jako poměr celého a nenulového celého čísla. Racionální čísla (bez nuly) jsou uzavřená i na dělení. Zaveďme teď racionální číslo jako dvojici celých čísel s invariantem, že druhé číslo nesmí být nikdy nula a že obě čísla jsou nesoudělná.

Násobení bude fungovat jako násobení na složkách, dělení pak bude jen prohození obou složek druhého argumentu a následné násobení. Odčítání je opět sčítání s opačným číslem a sčítání musí najít společný jmenovatel a poté specificky přenásobovat jednotlivé operandy, ale složité to není.

Metody musí stále kontrolovat, zda nedochází k dělení nulou. Také po konci výpočtu musí zkrátit obě složky čísla. Predikát rovnosti dvou racionálních čísel kvůli podmínce nesoudělnosti je pak jednoduchý a je to rovnost na složkách.

Tato struktura je *naprosto přesnou* reprezentací racionálních čísel. Jazyk Lisp se vydal cestou implementace racionálních čísel jako takto přesných a nabízí typ `ratio`.

2.4 Reálná čísla

Předchozí číselné obory lze tedy s dostatkem paměti naprosto přesně reprezentovat. To není málo. Zbývájí už „jen“ reálná čísla. Protože máme přesná racionální čísla, do reálných chybí už jen čísla iracionální. Těch je ale bohužel nepočitatelně mnoho, a tak nepůjdou vytvořit z přirozených čísel nabalováním struktur jako předchozí obory (důkaz, že toto není jednoduché je existence práce, kterou právě čtete).

Nic jiného ale v paměti, která pracuje s diskrétními hodnotami neumíme vytvořit. Přepneme teď přístup. Dosud jsme mluvili o *naprosto přesných* číslech, teď ovšem přijmeme názor, že struktura vyjadřující přesné číslo může být i taková, která počítá jen přibližná čísla, ale nastavitelně vzdálená od přesné hodnoty, kterou ovšem neznáme. Takových čísel je počitatelně mnoho a říkáme jim čísla rekurzivní. Ve skutečnosti je právě hranice mezi racionálními čísly a rekurzivními čísly ta velká bariéra, která odděluje svět, kde vše funguje relativně jednoduše a ten, kde je všechno o řád těžší.

Na začátku této podkapitoly ještě zůstaneme v myšlenkové rovině a budeme si abstraktní struktury představovat, ve druhé polovině potom ukážu, že některé struktury pro přesnou reprezentaci rekurzivních čísel existují.

2.4.1 Představa

Než struktury definovat imperativně pomocí definic, jak by co měla implementovat se spíše pokusím o definici struktury deklarativně, čili pomocí jejích vlastností, které by měla splňovat.

Abstraktní struktura reprezentující reálné číslo by měla umožnit

- jeho vyčíslení – když struktura existuje, pak po zavolání vhodného nástroje je výsledkem hodnota, kterou tato struktura představuje;
- přesnost – i když má číslo nekonečný rozvoj nebo je velmi velké, abstrakce umožňuje jeho vyčíslení na danou přesnost v konečném čase;
- podporovat matematické operace – ve smyslu kapitoly 1.3;
- podporovat matematické funkce – ve smyslu kapitoly 1.4;

- být vracena jako výsledek funkce – mít jasně popsanou strukturu, aby se dala rozšiřovat funkčnost;

- být použita jako argument nějaké funkce – typicky funkce pro vyčíslení.

První dvě podmínky jsou přímo esenciální – zajišťují, že abstrakce mohou reprezentovat reálná čísla na libovolnou (kladnou) přenost. Kdykoli budu potřebovat číslo dané nějakou abstrakcí, zavolám příslušnou funkci ještě s nějakým parametrem ε reprezentujícím přesnost, na kterou toto číslo potřebuji. Obecně totiž nemusí být výsledkem vyčíslovací funkce přesná hodnota hledaného čísla, avšak díky těmto podmínkám budu výsledku vzdálen maximálně o zadanou hodnotu odchylky. To je tedy význam onoho *přesného* v názvu této práce – výsledkem enumerace nikdy nemusí být naprosto přesné číslo, ale číslo, které je od výsledku vzdálené o jasně definovanou hodnotu a tento výpočet skončí. Když budu uvažovat nějakou strukturu s_x přesně vyjadřující nějaké číslo x, zavolám vyčíslovací funkci *enum* a jako argumenty použiji tuto strukturu a libovolné kladné ε, pak bych měl dostat výsledek $enum(s_x, \varepsilon)$ splňující nerovnost

$$|enum(s_x, \varepsilon) - x| \leq \varepsilon. \tag{20}$$

Druhé dvě podmínky vštěpují dané struktuře reprezentující reálná čísla vlastnosti reálných čísel a sice že s nimi jde sčítat, násobit atp. a že mohou být argumentem nějaké matematické funkce. Jistě nejde o to tyto struktury vkládat do stejných operací jako si představujeme s normálními čísly, například ve výrazu $3 + 4$ nechceme operandy nahrazovat abstraktními strukturami a očekávat správný výsledek, nýbrž chceme existenci ekvivalentní operace pro tyto struktury. To nutně neznamená, že musí být skutečně někde implementována, ale potřebujeme docílit stavu, kdy existence této není dokazatelně vyloučena.

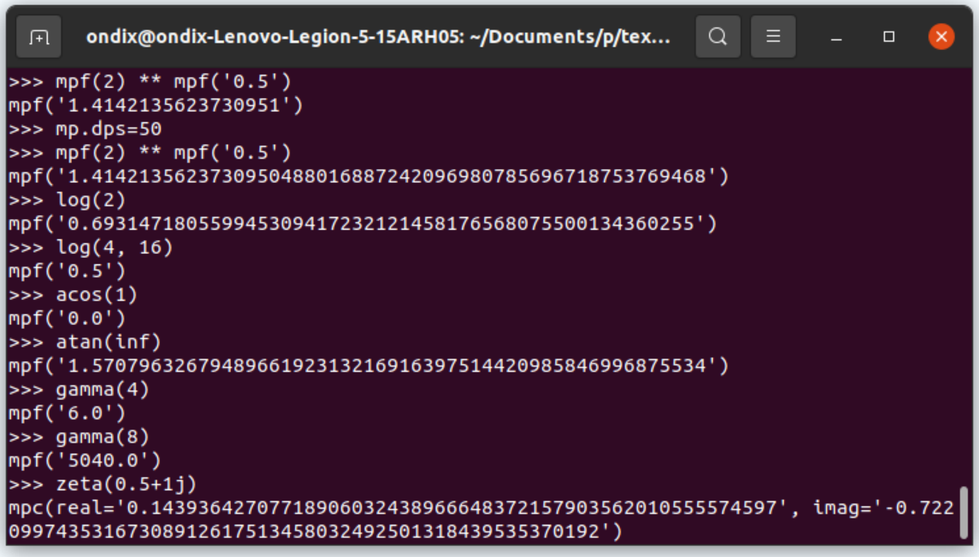
Třetí a poslední dvě podmínky jsou spíše návod pro praktické použití těchto hypotetických struktur, aby se s nimi dalo smysluplně pracovat. To vlastně znamená, že musí být elementy prvního řádu (*first-class citizen*), čili implementovány jako niterná součást použitého jazyka a ne jako svébytná konstrukce, která sice funguje, ale je nekompatibilní s jazykem svého vzniku, takže je vlastně nepoužitelná, uživatelsky nerozšiřitelná.

2.4.2 Existující nástroje

Nyní se podíváme, co na poli vyčíslování reálných čísel existuje v současné době.

2.4.2.1 Mpmath [21] `mpmath` je velmi rozsáhlá knihovna pro Python. Je publikována pod licensí BSD a je dosažitelná i pomocí `pip`u. Kromě základní funkčnosti pro výpočet funkcí a operací s čísly jsou implementovány i funkce pro výpočet funkcí intervalů, určitých integrálů, podpora tvorby grafů a mnoho dalšího. Dokumentace je velice hezky zpracovaná se spoustou příkladů. Přesnost se nastavuje proměnnou `mp.dps`. Jde o počet vypisovaných míst. Knihovna oplývá tolika možnostmi, že dokonce existuje stránka pro výpočet Ludolfova čísla sty

Obrázek 5: Používání knihovny `mpmath`



Vidíme, že struktura `mpf` podporuje matematické operace, na příkladu je odmocnina ze dvou. Ta je ve dvou provedeních, s defaultní přesností a s nastavenou na 50. Dále vidíme matematické funkce, jmenovitě logaritmus, cyklometrické, gamma a Riemannovu zeta funkci. Na jejím vstupu i výstupu vidíme komplexní číslo.

možnými one-linery. Knihovna používá tři vlastní datové typy a sice `mpf` pro reálná čísla, `mpc` pro komplexní čísla a `matrix` pro matice.

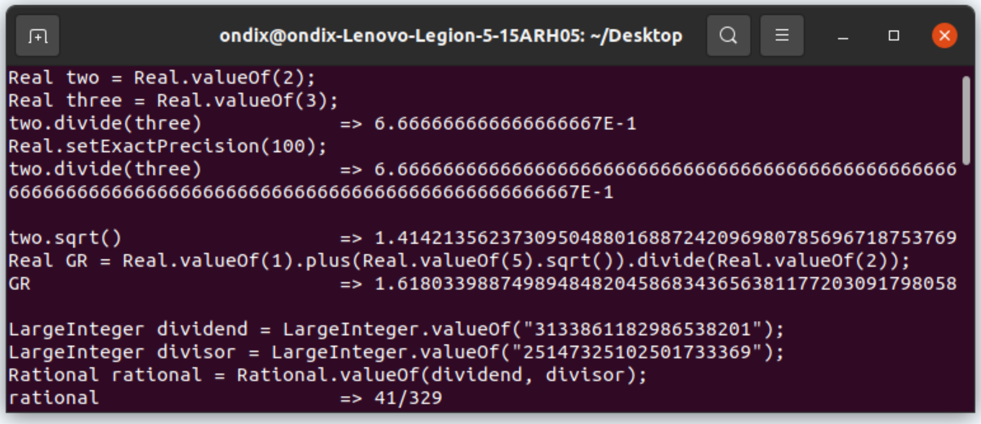
2.4.2.2 JScience [22]

`JScience` je knihovna pro jazyk Java. Její část pro práci s jednotkami se dostala do knihovny `javax`. Knihovna je široce rozkročena. Přináší podporu pro porovnávání a počítání jednotek z fyziky, geografie nebo ekonomie. Z matematiky je zde podpora pro jednoduchou symbolickou analýzu a strukturální algebru. Nás nejvíce zajímá část `org.jscience.mathematics.number`. Zde jsou zajímavé 3 datové typy a to `Real` umožnující základní výpočty s nastavitelnou přesností, `LargeInteger` ukládající velká celá čísla a `Rational` ukládající dvojice `LargeInteger`ů a umožňující jejich implicitní usměrňování a základní matematické operace.

2.4.2.3 GNU Multiple Precision Arithmetic Library [23] (GMP)

`GMP` je knihovna pro jazyk C. Datový typ `mpz_t` představuje celé číslo, u kterého nehrozí přetečení nebo podtečení. Zlomky velkých čísel představuje `mpq_t` a operace podporují usměrňování. Přesný ekvivalent čísla s plovoucí řádovou tečkou představuje `mpf_t`. Minimální počet bytů, ve kterém je uložen v paměti se nastavuje funkcí `mpf_set_default_prec`. Všechny typy implementují základní matema-

Obrázek 6: Používání knihovny `JScience`



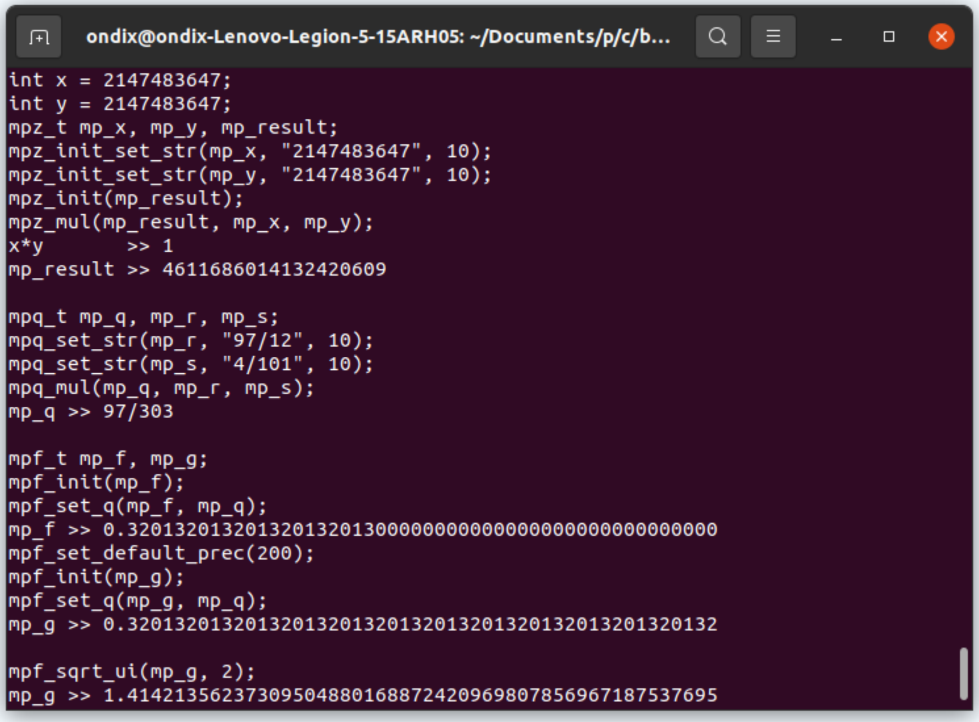
Vidíme používání třídy `Real`. Operace jsou použitelné jako metody, nikoli operátorem. Vidíme nastavování přesnosti, výpočet druhé odmocniny, zlatého řezu a zlomkové struktury s implicitním krácením.

tické operace. Protože je GMP součástí projektu GNU a protože je to knihovna pro C, je velmi mnoho nadstavbových knihoven, které její funkcionalitu využívají a rozšiřují. Z C-čkových jmenujme například `GNU MPFR`, která na je na GMP přímo založena [24] a přináší matematické funkce floatů, nebo `MPIR` – paralelní projekt odtrhnuvší se od vývoje GMP a jdoucí svojí cestou, přesto snažící se implementovat rozhraní GMP, aby byly zastupitelné [25]. Dále existují wrappery pro kompatibilitu s jinými jazyky a tudíž je GMP velmi rozšířená, ač se to nemusí uživateli zdát. Například pro platformu .NET je to knihovna `Math.GMP.Native`, pro Python `gmpy`, pro R `gmp` a takových příkladů najdeme hodně.

2.4.2.4 Class Library for Numbers [26] (CLN) `CLN` je knihovna pro jazyk C++. Je také zdarma šířená pod licensí GPL. Implementuje jak čísla s plovoucí řádovou tečkou, tak racionální čísla jako zlomky, navíc komplexní čísla. Za přesné se považují racionální čísla a komplexní čísla s přesnou imaginární i reálnou částí. Plovoucí čísla pak typově přesně kopírují Lispovské a dají se tedy použít k Lispovským implementacím. Tím pádem se zkratka CLN dá chápat i jako „Common Lisp Numbers“.

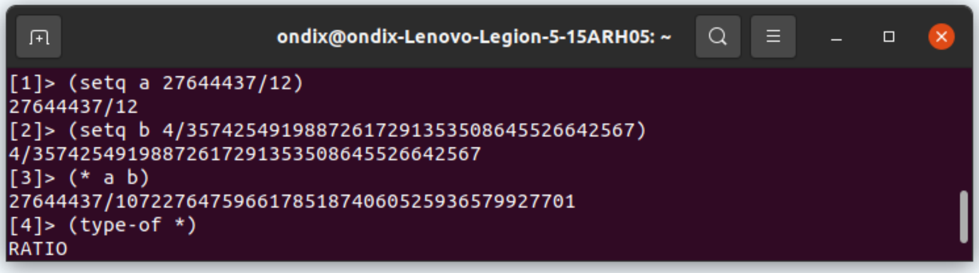
2.4.2.5 Computable-reals [27] `computable-reals` je Lispovská knihovna. Je volně ke stažení a dokonce k dostání pomocí `quicklispu`. Podporuje základní funkcionalitu. Její funkce poznáme tak, že končí koncovkou `-r`. Vracené výsledky nejsou čísla, ale vlastního typu `C-REAL`. Defaultně se tisknou výsledky na 20 míst, ale nastavením proměnné `*print-prec*` se tento počet dá měnit [28]. Kromě odmocniny jsou zde třeba základní násobky čísla π, logaritmy, mocniny, základní goniometrické funkce, arcus tangens. Knihovna se používá jako kalkulačka s na-

Obrázek 7: Používání knihovny `GMP`



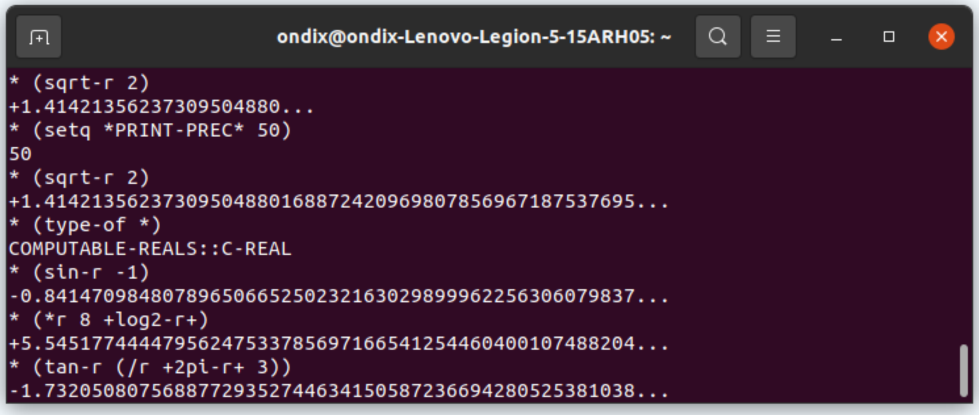
Na příkladu vidíme nejprve porovnání násobení klasických intů versus struktur typu `mpz_t`. U intů dojde k přetečení, u mpz nikoli. Dále vidíme práci se zlomky, jejich inicializaci ze stringu a násobení. Nakonec vidíme základní operace s typem `mpf_t`, protějškem čísel s plovoucí řádovou tečkou.

Obrázek 8: Implicitní používání knihovny `CLN`



Knihovnu CLN používá CLisp, interpret jazyka Lisp. Využívá implementace čísel. Příklad ukazuje, že se zlomky zkracují, aniž by bylo třeba k tomu voláním vybízet, ale děje se automaticky. Že je CLisp založen na CLN usuzuji podle osoby Bruna Heible-a, který figuruje jako autor jak u CLN, tak u CLisp-u.

Obrázek 9: Používání knihovny `computable-reals`



Na příkladu vidíme výpis odmocniny ze dvou na různé přesnosti, operace násobení a několik matematických funkcí.

stavitelnou přesností.

2.4.2.6 Cíl práce Tolik tedy k strukturám již nyní implementujícím čísla a jejich operace, případně funkce. V některých implementacích se jedná o třídy, v jiných jde o strukturované datové typy. V následujícím textu se pokusíme navázat tam, kde končí naprostá přesnost nad racionálními čísly jazyka Lisp a naprogramujeme knihovnu přinášející některá iracionální čísla.

V této práci nám jde o přesnost a nikoli o rychlost a proto jako nativní typy budu používat právě zlomky, ačkoli pro rychlé operace s desetinnými čísly se používají plovoucí čísla, které mají často i hardwarovou podporu v jednotce FPU. Ostatně proto jsou v Lispu i plovoucí typy [29].

K programování jsem jako textový editor použil *Visual Studio Code* s rozšířením *Rainbow Brackets* a jako překladač *SBCL*.

Část II
Implementace

V implementační části propojíme teoreticky orientované výsledky s aplikovaným aspektem a dáme vzniknout knihovně `tnums`. Jak jsem napsal výše, všechna čísla i manipulaci s nimi lze vyjádřit jako funkce. Nejpřímější aplikace tohoto poznání tedy vede k užití funkcionálního paradigmatu. Jako jeho zástupce byl vedoucím práce zvolen Lisp. Nejprve se podíváme na jednoduché převody mezi reprezentacemi v Lispu a tnumy, poté se podíváme na matematické operace tnumů a nakonec také na matematické funkce tnumů.

3 Tnumy

Svoje struktury jsem zvolil jako funkce dvou proměnných – reprezentovaného čísla a přesnosti. Podíváme se, jak se dají matematicky definovat, po Lispovsku vymodelovat a na závěr dokážu několik tvrzení, aby čtenář pochopil, jak se tnumy pracuje. Nebude chybět prvních několik kódů, i když nejvíce programování bude spíše ke konci této části.

3.1 Vztah čísel a tnumů

Funkce, kterými budu modelovat rekurzivní čísla a implementovat práci s nimi pomocí jazyka Lisp, budu nazývat *True Numbers*, zkráceně *tnums*. Vystihuje to jejich podstatu a cíl - budou ve výsledku opravdovější a přesnější, než ostatní čísla, která by měla mít nekonečný rozvoj, ale jsou uložena jako hodnoty – často jako nějaká plovoucí čísla (jakási Nil Numbers). Vzniknuvší knihovna se pak jmenuje `tnums`.

Definice 29 (Tnum)
Funkci $\mathcal{T}(x, \varepsilon)$ nazýváme tnum. Po částečném dosazení za x dostáváme funkci jedné proměnné, kterou budeme značit $\mathcal{T}^x(\varepsilon)$.

Proměnnou x myslíme číslo, které tnum reprezentuje a proměnná ε představuje přesnost, na kterou chceme číslo x vyčíslit. Přesnost předpokládáme v rozmezí $0 < \varepsilon < 1$. Pro hodnotu tnumu pak platí

$$|\mathcal{T}^x(\varepsilon) - x| \leq \varepsilon. \tag{21}$$

Místo tohoto vztahu lze psát $\mathcal{T}^x(\varepsilon) \in [x - \varepsilon, x + \varepsilon]$ nebo $(\mathcal{T}^x(\varepsilon) \leq x + \varepsilon) \wedge$ $\wedge(\mathcal{T}^x(\varepsilon) \geq x - \varepsilon)$.

Kvůli možnosti částečného dosazení (kterému říkáme *currying*) lze nejprve k číslu x vytvořit částečně dosazený tnum (toto bude výpočetně rychlé) a až poté tento tnum nechat vyčíslit (zavolat s přesností) a toto může být na dlouho. Ekvivalencí částečně dosazených tnumů rozumíme ekvivalenci na číslech, čili

$(\forall x \in \mathbb{R})(\forall y \in \mathbb{R})((x = y) \rightarrow (\mathcal{T}^x = \mathcal{T}^y))$. Symbolem $\mathfrak{T}$ označíme množinu všech tnumů.

Tnumy přesných čísel lze vyčíslit s dokonalou přesností. Využiji co nejvíce z přesnosti, kterou nabízí Lisp a ten přesně reprezentuje všechna racionální čísla. To je ve shodě s představou o vyčíslení rekurzivního čísla. Pomocí $\mathcal{T}^r(\varepsilon)$ tedy získáváme q z nerovnice 7. Číslu, jak ho chápe Lisp říkám nadále num (number).

Lemma 30 (O numu jako tnumu)

Pro všechna $x \in \mathbb{R}$ a všechna $\varepsilon \in (0,1)$ platí: číslo $\mathcal{T}^x(\varepsilon)$ lze nahradit x.

Důkaz

Z nerovnosti 21 získáváme $|\mathcal{T}^x(\varepsilon) - x| \leq \varepsilon$. Po dosazení $\mathcal{T}^x(\varepsilon) := x$ pak $|x - x| = 0 \leq \varepsilon$, což platí pro všechna uvažovaná x i ε. □

Nejpřesnější reprezentace čísla reprezentovaného tnumem reprezentujícím číslo je toto číslo samotné. Proto je tedy vhodné co nejvíce takových čísel přenechat na reprezentaci Lispu a počítat jen s těmi, které nezvládne. Protože Lisp pracuje i se zlomky (typ `ratio`), nejvyšší obor čísel, který umí vracet s nulovou odchylkou jsou racionální čísla.

```
(defun num-to-tnum (num)
  (let ((rat_num (rationalize num)))
    (lambda (eps) (declare (ignore eps))
      rat_num)))
```

Lispový kód 1 (`num-to-tnum`)**:** *Funkce převádějící číslo z interní reprezentace Lispu na tnum*

Převod opačným směrem je také jednoduchý. Pokud chci vyčíslit číslo x s přesností ε, stačí zavolat $\mathcal{T}^x(\varepsilon)$. Přesnost musí být z $(0,1)$, takže jiné číslo by byl vstup nevalidní, budeme ho interpretovat jako $10^{-|\varepsilon|}$.

Lemma 31 (O převodu tnumu na num)

Pokud existuje funkce $\mathcal{T}^x$, pak po zavolání s argumentem ε vrací hodnotu $\mathcal{T}^x(\varepsilon)$ splňující $(|\mathcal{T}^x(\varepsilon) - x| \leq \varepsilon)$.

Důkaz

Plyne přímo z definice 29. □

Protože se někdy při vyhodnocení funkce `expt` výsledek reprezentuje jako plovoucí číslo, přidáme si ještě vlastní funkci pro racionální umocňování.

```
(defun rat-expt (num exp)
  (rationalize (expt num exp)))
```

Lispový kód 2 (`rat-expt`)**:** *Funkce pro racionální umocňování*

```lisp
(defun tnum-to-num (tnum eps)
  (when (or (>= 0 eps) (<= 1 eps))
    (setf eps (rat-expt 10 (- (abs eps)))))
  (funcall tnum (rationalize eps)))
```

Lispový kód 3 (`tnum-to-num`)**:** *Funkce převádějící tnum na číslo*

Zatímco tedy pro převod z čísla na tnum jsme toto mohli udělat pro všechna čísla, opačným směrem toto funguje pouze za předpokladu, že daný tnum existuje. V našem systému teď máme jen tnumy pro racionální čísla a umíme je převádět tam a zpět. V dalším textu tedy půjde hlavně o to zaplnit tuto mezeru a přinést existenci co nejvíce tnumů.

3.2 Ludolfovo číslo

Prvním iracionálním číslem, které do knihovny přidáme je číslo Ludolfovo.

Definice 32 (Ludolfovo číslo [30])
Ludolfovým číslem myslíme poměr obvodu kružnice k jejímu průměru.

Ludolfovo číslo je asi nejslavnější transcendentní konstanta a proto není divu, že pro její vyčíslení existuje bezpočet vzorců. Asi nejpřímější je Leibnizův vzorec, který vypočítává čtvrtinu Ludolfova čísla a plyne z Taylorovy řady funkce arctan v bodě 1. Pokud Ludolfovo číslo značím π, pak ho lze zapsat jako $\pi = 4\sum_{n\in\mathbb{N}} \frac{(-1)^n}{2n+1}$ [31], tato řada ale konverguje velmi pomalu. Já proto použiji aproximaci jinou. Tento vzorec se jmenuje BBP podle svých tvůrců (Bailey, Borwein, Plouffe) a je zapsán ve formě řady.

Fakt 33 (Ludolfovo číslo jako řada [32])
Nechť π značí Ludolfovo číslo. Pak jej lze zapsat jako

$$\pi = \sum_{i\in\mathbb{N}} \frac{1}{16^i}\left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6}\right). \tag{22}$$

Mám tedy řadu, která generuje konstantu, kterou chci přidat do `tnums`. Výraz $\left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6}\right)$ je pro $i > 0$ menší než jedna, proto se každý nenultý člen může zhora omezit $\frac{1}{16^i}$ a to je geometrická posloupnost, jejíž zbytek je dle faktu 26 roven $\frac{1}{16^{i+1}} * \frac{16}{15}$, což je $\frac{1}{16^i * 15}$. Platí tedy

$$\left|\pi - \sum_{i=0}^{n} \frac{1}{16^i}\left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6}\right)\right| \leq \frac{1}{16^n * 15}. \tag{23}$$

Kód vypadá trochu složitěji, ale není to nic jiného, než co bylo právě popsáno. Nižší čitelnost je zde vykoupena vyšší efektivitou a protože je vyčíslování π jedna z nejdůležitějších funkcionalit, rozhodl jsem se ji zavést takto efektivně, ač na úkor čitelnosti.

```
(defun tnum-pi ()
  (lambda (eps)
    (let* ((n 0) (/16pown 0) (result 0) (above 1))
      (loop
        until (<= above eps)
        do (progn
          (setf /16pown (rat-expt 16 (- n)))
          (incf result
            (* /16pown
              (- (/ 4 (+ (* 8 n) 1))
                (/ 2 (+ (* 8 n) 4))
                (/ 1 (+ (* 8 n) 5))
                (/ 1 (+ (* 8 n) 6)))))
          (setf above (/ /16pown 15))
          (incf n))
        finally (return result)))))
```

Lispový kód 4 (`tnum-pi`): *Funkce na vytvoření $\mathcal{T}^{\pi}$*

3.3 Přenásobování numem

Posledním dílkem, který přidám v této kapitole je přenásobování tnumu konstantou. Když už máme všechna racionální čísla a Ludolfovo číslo, zvládneme pak i například 2π nebo $\frac{\pi}{-2}$.

Věta 34 (O přenásobení tnumu racionální konstantou)
Pro racionální konstantu c, reálné x a jeho tnum platí

$$\mathcal{T}^{c*x}(\varepsilon) = \begin{cases} c * \mathcal{T}^{x}\left(\frac{\varepsilon}{|c|}\right) & \textit{pro } c \neq 0 \\ \mathcal{T}^{0} & \textit{jinak} \end{cases} \tag{24}$$

Důkaz
Pokud přenásobíme tnum nulou, je výsledkem nula, protože je to agresivní prvek vůči násobení. Znění věty pro nenulovou konstantu dokážeme tak, že z předpokladu $|\mathcal{T}^{x}(\varepsilon) - x| \leq \varepsilon$ odvodíme $|c * \mathcal{T}^{x}(\frac{\varepsilon}{|c|}) - c * x| \leq \varepsilon$. Protože pracujeme s nerovnicemi, budeme v důkazu postupovat dvěmi větvemi – pro c kladné a záporné.

Z definice tnumu předpokládáme

$$|\mathcal{T}^{x}(\varepsilon) - x| \leq \varepsilon, \tag{25}$$

po přenásobení kladným $c > 0$ dostáváme

$$c * |\mathcal{T}^{x}(\varepsilon) - x| \leq c * \varepsilon, \tag{26}$$

protože je ale c kladné, můžu jím absolutní hodnotu roznásobit

$$|c * \mathcal{T}^x(\varepsilon) - c * x| \leq c * \varepsilon, \tag{27}$$

a protože na pravé straně potřebuji přesnost ε, v argumentu ji podělím c a pak

$$\left|c * \mathcal{T}^x\left(\frac{\varepsilon}{c}\right) - c * x\right| \leq \varepsilon. \tag{28}$$

Pro zápornou konstantu je běh důkazu podobný a protože jako argument tnumů bereme kladné číslo, přibývá v děliteli v argumentu tnumu ještě absolutní hodnota. Dohromady pak získáváme

$$\left|c * \mathcal{T}^x\left(\frac{\varepsilon}{|c|}\right) - c * x\right| \leq \varepsilon, \tag{29}$$

což jsme chtěli ukázat. □

```
(defun tnum*num (tnum num)
  (let ((rat_num (rationalize num)))
    (lambda (eps)
      (if (zerop num)
        (num-to-tnum 0)
        (* (tnum-to-num tnum (/ eps (abs rat_num))) rat_num)))))
```

Lispový kód 5 (`tnum*num`)**:** *Funkce přenásobující tnum racionální konstantou*

Důsledek 35 (Opačný tnum)

$$\mathcal{T}^{-x}(\varepsilon) = -\mathcal{T}^x(\varepsilon) \tag{30}$$

Důkaz

Protože $-x = (-1)x$ a $|-1| = 1$, pak podle přechozí věty dostáváme $\mathcal{T}^{-x}(\varepsilon) = \mathcal{T}^{(-1)x}(\varepsilon) = (-1)\mathcal{T}^x(\frac{\varepsilon}{|-1|}) = (-1)\mathcal{T}^x(\frac{\varepsilon}{1}) = (-1)\mathcal{T}^x(\varepsilon) = -\mathcal{T}^x(\varepsilon)$. □

```
(defun -tnum (tnum)
  (tnum*num tnum -1))
```

Lispový kód 6 (`-tnum`)**:** *Funkce pro opačný tnum*

4 Operace tnumů

Matematické operace se neimplementují všechny stejně složitě. Zatímco aditivní vyřešíme relativně rychle, multiplikativní budou o úroveň těžší, tak mocninné v této kapitole ani nezvládneme a budeme na ně muset počkat až na konec další kapitoly. Začneme tedy operacemi sčítání a odčítání, pak se přesuneme k násobení a dělení.

4.1 Aditivní operace

Součet je operace neomezeného počtu argumentů, pro žádný vrací nulu (neutrální prvek aditivní grupy [13]), pro jeden vrací tento a pro více se pak jedná o postupné zvětšování výsledku o hodnoty argumentů. Rozdíl potom vyžaduje alespoň jeden argument, v případě zadání pouze tohoto se vrací tnum k němu opačný, v případě více pak součet prvního a opačného tnumu součtu ostatních.

Věta 36 (O součtu tnumů)
Mějme čísla $x_0, x_1, \dots, x_n$ a jejich tnumy. Pak platí

$$\mathcal{T}^{\sum_{i=0}^{n} x_i}(\varepsilon) = \sum_{i=0}^{n} \mathcal{T}^{x_i}\left(\frac{\varepsilon}{n+1}\right) \tag{31}$$

Důkaz
Předpokládejme $\mathcal{T}^{x_i}(\varepsilon) + \varepsilon \geq x_i$ a $\mathcal{T}^{x_i}(\varepsilon) - \varepsilon \leq x_i$ pro $i \in \{0, 1, \dots, n\}$ a ukažme $\sum_{i=0}^{n} \mathcal{T}^{x_i}\left(\frac{\varepsilon}{n+1}\right) + \varepsilon \geq \sum_{i=0}^{n} x_i$ a $\sum_{i=0}^{n} \mathcal{T}^{x_i}\left(\frac{\varepsilon}{n+1}\right) - \varepsilon \leq \sum_{i=0}^{n} x_i$.
Z definice tnumu předpokládáme

$$\begin{gathered}
\mathcal{T}^{x_0}\left(\frac{\varepsilon}{n+1}\right) + \frac{\varepsilon}{n+1} \geq x_0 \wedge \mathcal{T}^{x_0}\left(\frac{\varepsilon}{n+1}\right) - \frac{\varepsilon}{n+1} \leq x_0, \\
\mathcal{T}^{x_1}\left(\frac{\varepsilon}{n+1}\right) + \frac{\varepsilon}{n+1} \geq x_1 \wedge \mathcal{T}^{x_1}\left(\frac{\varepsilon}{n+1}\right) - \frac{\varepsilon}{n+1} \leq x_1, \\
\vdots \\
\mathcal{T}^{x_n}\left(\frac{\varepsilon}{n+1}\right) + \frac{\varepsilon}{n+1} \geq x_n \wedge \mathcal{T}^{x_n}\left(\frac{\varepsilon}{n+1}\right) - \frac{\varepsilon}{n+1} \leq x_n.
\end{gathered} \tag{32}$$

Sečtěme teď všechny výrazy a získáváme

$$\sum_{i=0}^{n} \mathcal{T}^{x_i}\left(\frac{\varepsilon}{n+1}\right) + \sum_{i=0}^{n} \frac{\varepsilon}{n+1} \geq \sum_{i=0}^{n} x_i \wedge \sum_{i=0}^{n} \mathcal{T}^{x_i}\left(\frac{\varepsilon}{n+1}\right) - \sum_{i=0}^{n} \frac{\varepsilon}{n+1} \leq \sum_{i=0}^{n} x_i, \tag{33}$$

dále $\sum_{i=0}^{n} \frac{\varepsilon}{n+1} = \varepsilon$, takže

$$\sum_{i=0}^{n} \mathcal{T}^{x_i}\left(\frac{\varepsilon}{n+1}\right) + \varepsilon \geq \sum_{i=0}^{n} x_i \wedge \sum_{i=0}^{n} \mathcal{T}^{x_i}\left(\frac{\varepsilon}{n+1}\right) - \varepsilon \leq \sum_{i=0}^{n} x_i, \tag{34}$$

výraz $\sum_{i=0}^{n} \mathcal{T}^{x_i}\left(\frac{\varepsilon}{n+1}\right)$ je tedy ekvivalentní s $\mathcal{T}^{\sum_{i=0}^{n} x_i}(\varepsilon)$. □

V našem chápání nevracíme přímo čísla, ale tnumy, takže pro prázdný argument místo nuly její tnum – $\mathcal{T}^0$.

```
(defun tnum+ (&rest tnums)
  (if (null tnums)
    (num-to-tnum 0)
    (lambda (eps)
      (let ((new-eps (/ eps (list-length tnums))))
        (apply '+
          (mapcar (lambda (tnum) (tnum-to-num tnum new-eps))
            tnums))))))
```

Lispový kód 7 (`tnum+`): *Funkce na součet tnumů*

Odčítání využívá právě dokázané věty a také důsledku 35.

Fakt 37 (Rozdíl tnumů)
Mějme čísla $x_{-1}, x_0, \ldots, x_n$, a jejich tnumy. Pak platí

$$\mathcal{T}^{x_{-1}-x_0-\ldots-x_n}(\varepsilon) = \mathcal{T}^{x_{-1}-\sum_{i=0}^{n} x_n}(\varepsilon) = \mathcal{T}^{x_{-1}+\left(-\sum_{i=0}^{n} x_n\right)}(\varepsilon) \tag{35}$$

Kód odpovídá Lispovskému `-`, tedy pro jeden argument vrací opačný tnum onoho a pro více argumentů vrací jejich rozdíl.

```
(defun tnum- (tnum1 &rest tnums)
  (if (null tnums)
    (-tnum tnum1)
    (tnum+ tnum1 (-tnum (apply 'tnum+ tnums)))))
```

Lispový kód 8 (`tnum-`): *Funkce pro rozdíl tnumů, případně opačný tnum*

4.2 Multiplikativní operace

Jako první multiplikativní operaci představím převrácení hodnoty tnumu. Je to podobná operace jako opačný tnum, jen jde o jinou inverzi.

Protože převrácení pracuje pouze s nenulovými čísly, přidáme do našeho aparátu ještě práci s tnumy nenulových čísel.

Definice 38 (Nenulový tnum)
Tnum $\mathcal{T}$, který nikdy nenabývá nulové hodnoty, neboli $(\forall \varepsilon \in (0,1))(\mathcal{T}(\varepsilon) \neq 0)$ *budeme nazývat nenulový tnum a budeme ho značit $\mathcal{T}_{\emptyset}$.*

Čísla, která nenabývají nuly tedy budeme moci reprezentovat nenulovými tnumy.

Definice 39 (Bezpečné epsilon)

Pro libovolné ε *a tnum* $\mathcal{T}^x$, *kde* $x \neq 0$ *uvažujeme „funkci"* $\varepsilon_\emptyset : \mathfrak{T} \times (0,1) \to (0,1)$ *tak, aby*

1. $0 < \varepsilon_\emptyset(\mathcal{T}^x, \varepsilon) \leq \varepsilon$,
2. $\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon)) \neq 0$ *a*
3. $|\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))| > \varepsilon_\emptyset(\mathcal{T}^x, \varepsilon)$.

a její hodnotu nazýváme bezpečným epsilonem.

Lemma 40 (O nenulovém tnumu nenulového čísla)

Tnum nenulového čísla lze vyčíslit nenulově, čili $(\forall x \neq 0)((\exists \mathcal{T}^x) \to (\exists \mathcal{T}^x_\emptyset))$.

Důkaz

Vezměme za $\mathcal{T}^x_\emptyset$ tnum, který místo ε dosadí $\varepsilon_\emptyset$, neboli $\mathcal{T}^x_\emptyset(\varepsilon) := \mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))$. Pak díky podmínce 1 v definici 39 se přesnost nemůže zhoršit a tudíž vyčíslení proběhne v pořádku. Dále díky bodu 2 ve stejné definici bude vyčíslení nenulové, takže se jedná o nenulový tnum. □

Funkce pro vracení bezpečného epsilonu musí kvůli kontrole nenulovosti vypočítat i num zadaného tnumu a musí také vracet nové epsilon. Aby se tnum nevyčísloval vícekrát, když už jeho hodnotu známe, vrací funkce i tento num.

```
(defun get-nonzero-num+eps (tnum eps)
  (let ((num (tnum-to-num tnum eps)))
    (if (and (zerop num) (<= (abs num) eps))
      (get-nonzero-num+eps tnum (/ eps 10))
      (values num eps))))
```

Lispový kód 9 (`get-nonzero-num+eps`)**:** *Funkce pro nalezení přesnosti, při které nebude po aplikaci tnumu nulový výsledek a následné vrácení výsledku i epsilonu*

Věta 41 (O převráceném tnumu)

Mějme $x \neq 0$ *a jeho tnum. Pak platí*

$$\mathcal{T}^{x^{-1}}(\varepsilon) = [\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, (\varepsilon * |\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))| * (|\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))| - \varepsilon_\emptyset(\mathcal{T}^x, \varepsilon)))))]^{-1} \tag{36}$$

Důkaz

Podle rovnice 21 platí

$$|\mathcal{T}^x(\varepsilon) - x| \leq \varepsilon, \tag{37}$$

což lze díky lemmatu 40 a předpokladu nenulovosti x přepsat na

$$|\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon)) - x| \leq \varepsilon, \tag{38}$$

díky absolutní hodnotě pak platí

$$|x - \mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))| \leq \varepsilon. \tag{39}$$

Nerovnici vydělíme kladným číslem $|\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon)) * x|$

$$\frac{|x - \mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))|}{|\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon)) * x|} \leq \frac{\varepsilon}{|\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon)) * x|} \tag{40}$$

a protože $|a| * |b| = |a * b|$, po dvojí aplikaci platí

$$\left|\frac{x - \mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))}{\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon)) * x}\right| \leq \frac{\varepsilon}{|\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))| * |x|} \tag{41}$$

a po roztržení levého výrazu na rozdílné jmenovatele dostáváme

$$\left|\frac{1}{\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))} - \frac{1}{x}\right| \leq \frac{\varepsilon}{|\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))| * |x|}. \tag{42}$$

Dále díky předpokladu 3 z definice 39 $|x| \geq |\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))| - \varepsilon_\emptyset(\mathcal{T}^x, \varepsilon)$ a proto

$$\left|\frac{1}{\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))} - \frac{1}{x}\right| \leq \frac{\varepsilon}{|\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))| * (|\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))| - \varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))}, \tag{43}$$

takže po úpravě přesnosti dostáváme

$$\left|\frac{1}{\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, (\varepsilon * |\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))| * (|\mathcal{T}^x(\varepsilon_\emptyset(\mathcal{T}^x, \varepsilon))| - \varepsilon_\emptyset(\mathcal{T}^x, \varepsilon)))))} - \frac{1}{x}\right| \leq \varepsilon. \tag{44}$$

□

```
(defun /tnum (tnum)
  (lambda (eps)
    (multiple-value-bind (num eps0)
      (get-nonzero-num+eps tnum eps)
      (let* ((absnum (abs num))
          (neweps (* eps absnum (- absnum eps0))))
        (/ (if (>= neweps eps) num
          (get-nonzero-num+eps tnum neweps)))))))
```

Lispový kód 10 (`/tnum`)**:** *Funkce pro převracení hodnoty tnumu. Pokud je nové epsilon větší nebo stejné, vrací se již vypočtený tnum.*

Násobení bere libovolně mnoho argumentů. Pro žádný vrátí jedničku (jednotka v multiplikativní grupě [13]), pro jeden vrátí tento a pro více pak jejich součin.

Věta 42 (O součinu dvou tnumů)
Pro nenulová čísla x, y a jejich tnumy platí

$$\mathcal{T}^{x*y}(\varepsilon) = \mathcal{T}^{x}\left(\frac{\varepsilon}{2(|\mathcal{T}^{y}(\varepsilon)| + \varepsilon + 1)}\right) * \mathcal{T}^{y}\left(\frac{\varepsilon}{2(|\mathcal{T}^{x}(\varepsilon)| + \varepsilon + 1)}\right) \tag{45}$$

Důkaz
Nejprve si dokážeme dvě nerovnice, které posléze použijeme v těle důkazu. První nerovnicí je

$$\frac{|y|}{|\mathcal{T}^{y}(\varepsilon)| + \varepsilon + 1} \leq 1. \tag{46}$$

To je ekvivalentní s

$$|y| \leq |\mathcal{T}^{y}(\varepsilon)| + \varepsilon + 1 \tag{47}$$

a protože $\mathcal{T}^{y}(\varepsilon) + \varepsilon \geq y$, platí

$$|y| \leq |y| + 1, \tag{48}$$

což je jistě pravda.

Druhá nerovnice je

$$\frac{|\mathcal{T}^{x}(\frac{\varepsilon}{2(|\mathcal{T}^{y}(\varepsilon)|+\varepsilon+1|)})|}{|\mathcal{T}^{x}(\varepsilon)| + \varepsilon + 1} \leq 1. \tag{49}$$

Ekvivalentní zápis je

$$|\mathcal{T}^{x}(\frac{\varepsilon}{2(|\mathcal{T}^{y}(\varepsilon)| + \varepsilon + 1|)})| \leq |\mathcal{T}^{x}(\varepsilon)| + \varepsilon + 1 \tag{50}$$

a protože $\mathcal{T}^{x}(\varepsilon) + \varepsilon \geq x$, platí

$$|\mathcal{T}^{x}(\frac{\varepsilon}{2(|\mathcal{T}^{y}(\varepsilon)| + \varepsilon + 1|)})| \leq |x| + 1 \tag{51}$$

a stejným postupem rozepíšeme levou stranu, takže dostáváme

$$|x| + |\frac{\varepsilon}{2(|\mathcal{T}^{x}(\varepsilon)| + \varepsilon + 1)}| \leq |x| + 1 \tag{52}$$

a po odečtení $|x|$ zbyde

$$|\frac{\varepsilon}{2(|\mathcal{T}^{x}(\varepsilon)| + \varepsilon + 1)}| \leq 1 \tag{53}$$

což ale platí, protože

$$|\mathcal{T}^{x}(\varepsilon) + \varepsilon \geq 0. \tag{54}$$

Nyní konečně přejděme k důkazu věty. Aby věta platila, musíme dokázat

$$\left|\mathcal{T}^x\left(\frac{\varepsilon}{2(|\mathcal{T}^y(\varepsilon)|+\varepsilon+1)}\right)*\mathcal{T}^y\left(\frac{\varepsilon}{2(|\mathcal{T}^x(\varepsilon)|+\varepsilon+1)}\right)-xy\right|\le\varepsilon. \tag{55}$$

Rozepíšeme proto levou stranu

$$\left|\mathcal{T}^x\left(\frac{\varepsilon}{2(|\mathcal{T}^y(\varepsilon)|+\varepsilon+1)}\right)*\mathcal{T}^y\left(\frac{\varepsilon}{2(|\mathcal{T}^x(\varepsilon)|+\varepsilon+1)}\right)-xy\right|= \tag{56}$$

přičtením a odečtením členu $\mathcal{T}^x\left(\frac{\varepsilon}{2(|\mathcal{T}^y(\varepsilon)|+\varepsilon+1)}\right)y$ dostáváme

$$\begin{aligned}=\Big|\mathcal{T}^x\left(\frac{\varepsilon}{2(|\mathcal{T}^y(\varepsilon)|+\varepsilon+1)}\right)*\mathcal{T}^y\left(\frac{\varepsilon}{2(|\mathcal{T}^x(\varepsilon)|+\varepsilon+1)}\right)-\\ -\mathcal{T}^x\left(\frac{\varepsilon}{2(|\mathcal{T}^y(\varepsilon)|+\varepsilon+1)}\right)y+\mathcal{T}^x\left(\frac{\varepsilon}{2(|\mathcal{T}^y(\varepsilon)|+\varepsilon+1)}\right)y-xy\Big|=\end{aligned} \tag{57}$$

a po vytknutí $|\mathcal{T}^x\left(\frac{\varepsilon}{2(|\mathcal{T}^y(\varepsilon)|+\varepsilon+1)}\right)|$ z prvních dvou členů a $|y|$ z druhých dvou máme

$$\begin{aligned}=|\mathcal{T}^x\left(\frac{\varepsilon}{2(|\mathcal{T}^y(\varepsilon)|+\varepsilon+1)}\right)||\mathcal{T}^y\left(\frac{\varepsilon}{2(|\mathcal{T}^x(\varepsilon)|+\varepsilon+1)}\right)-y|+\\ +|y||\mathcal{T}^x\left(\frac{\varepsilon}{2(|\mathcal{T}^y(\varepsilon)|+\varepsilon+1)}\right)-x|\le\end{aligned} \tag{58}$$

a po dvojím použití pravidla $|\mathcal{T}^x(\varepsilon)-x|\le\varepsilon$ získáváme

$$\begin{aligned}\le|\mathcal{T}^x\left(\frac{\varepsilon}{2(|\mathcal{T}^y(\varepsilon)|+\varepsilon+1)}\right)||\frac{\varepsilon}{2}||\frac{1}{(|\mathcal{T}^x(\varepsilon)|+\varepsilon+1)}|+\\ +|y||\frac{\varepsilon}{2}||\frac{1}{(|\mathcal{T}^y(\varepsilon)|+\varepsilon+1)}|=\end{aligned} \tag{59}$$

a zjednodušíme-li pomocí zlomku, dostáváme

$$\begin{aligned}=|\frac{\varepsilon}{2}||\frac{|\mathcal{T}^x\left(\frac{\varepsilon}{2(|\mathcal{T}^y(\varepsilon)|+\varepsilon+1)}\right)|}{(|\mathcal{T}^x(\varepsilon)|+\varepsilon+1)}|+\\ +|\frac{\varepsilon}{2}||\frac{|y|}{(|\mathcal{T}^y(\varepsilon)|+\varepsilon+1)}|\le\end{aligned} \tag{60}$$

a díky dokázaným nerovnostem 46 a 49 pak platí

$$\le\frac{\varepsilon}{2}+\frac{\varepsilon}{2}=\varepsilon. \tag{61}$$

□

Právě dokázaná věta mluví o součinu dvou tnumů. Zobecnění na konečný počet tnumů už uvedu bez důkazu.

Fakt 43 (Součin tnumů)

Pro nenulová čísla $\{x_i\}_{i=0}^{n}$ a jejich tnumy platí

$$\mathcal{T}^{\prod_{i=0}^{n} x_i}(\varepsilon) = \prod_{i=0}^{n} \mathcal{T}^{x_i}\left(\frac{\varepsilon}{(n+1) * \prod_{j=0,i\neq j}^{n}(|\mathcal{T}^{x_j}(\varepsilon)| + \varepsilon + 1)}\right). \tag{62}$$

Věta mluví o nenulových číslech. Nesnižujeme ale obecnost, protože nula je agresivní prvek a výsledkem násobení čehokoli s nulou je nula, takže se ostatní numy ani nemusejí počítat a výsledek se může vrátit.

Samotná implementace pak využívá pomocnou mapovací funkci, která vypadá trochu složitěji, ale velmi zlepšila porozumění funkci `tnum*`, o kterou nám teď jde především. Nejprve tedy pomocná

```
(defun create-list-for-multiplication (tnums eps)
  (let ((result nil)
        (nums
          (mapcar (lambda (tnum) (tnum-to-num tnum eps)) tnums)))
    (dotimes (i (list-length tnums) result)
      (let ((actual-eps (/ eps (list-length tnums))))
        (dotimes (j (list-length tnums))
          (unless (= i j)
            (setf actual-eps (/ actual-eps
              (+ (nth j nums) eps 1)))))
        (setf result (cons
          (tnum-to-num (nth i tnums) actual-eps) result))))))
```

Lispový kód 11 (`create-list-for-multiplication`)**:** *Pomocná funkce pro násobení*

a konečně už slíbená hlavní funkce. Místo jedničky vrací odpovídající $\mathcal{T}^1$.

```
(defun tnum* (&rest tnums)
  (if (null tnums)
    (num-to-tnum 1)
    (lambda (eps)
      (apply '* (create-list-for-multiplication tnums eps)))))
```

Lispový kód 12 (`tnum*`)**:** *Funkce pro násobení tnumů*

Druhou obecnou multiplikativní funkcí je dělení.

Fakt 44 (Podíl tnumů)

Mějme čísla $x_{-1}, x_0, \ldots, x_n$ a jejich tnumy. Pak platí

$$\mathcal{T}^{x_{-1}/x_0/\ldots/x_n}(\varepsilon) = \mathcal{T}^{x_{-1}/\prod_{i=0}^{n} x_n}(\varepsilon) = \mathcal{T}^{x_{-1}*(\prod_{i=0}^{n} x_n)^{-1}}(\varepsilon) \tag{63}$$

Jedná se o tnumovský protějšek funkce `/`. Pro jeden argument vrací jeho převrácení, pro více pak jejich postupný podíl.

```lisp
(defun tnum/ (tnum1 &rest tnums)
  (if (null tnums)
    (/tnum tnum1)
    (tnum* tnum1 (/tnum (apply 'tnum* tnums)))))
```

Lispový kód 13 (`tnum/`)**:** *Funkce pro dělení tnumů*

Pro tnumy $a, b \in \mathfrak{T}$ existuje tnum $a + b$ a $a * b$. Tyto operace jsou tedy uzavřené. Tnum i v tomto smyslu dobře reprezentuje rekurzivní číslo, rekurzivní čísla totiž tvoří číselné těleso [33].

4.3 Mocninné operace

Jak už sem psal v úvodu k této kapitole, na mocnění a odmocňování ještě nemáme v našem systému dostatečný aparát. K implementaci mocninných operací totiž potřebujeme funkce přirozeného logaritmu a exponenciály, které přidáme až v další kapitole.

Lemma 45 (O mocnině tnumu)
Mějme $a > 0$ a $b \in \mathbb{R}$, pak jejich mocninu a^b lze vyjádřit jako $e^{(b*\ln(a))}$.

Důkaz
Kladné číslo a lze vyjádřit jako $e^{(\ln(a))}$, a^b je pak $e^{(\ln(a))^b}$, což je pak $e^{(b*\ln(a))}$.
$\square$

Odmocninu potom přijmeme jako mocninu obrácené hodnoty.

Fakt 46 (Odmocnina jako mocnina [7])
Mějme $a, b \in \mathbb{R}^+$. Pak platí

$$\sqrt[a]{b} = b^{(a^{-1})} \tag{64}$$

Operace dokončíme na konci následující kapitoly.

5 Funkce tnumů

Knihovna `tnums` v této chvíli umí přidávat racionální čísla, Ludolfovo číslo a provádět mezi nimi multiplikativní a aditivní operace. Bylo by vhodné teď přidat další rekurzivní čísla. Jejich dobrým zdrojem, jak jsem napsal již v podkapitole 1.4, jsou matematické funkce. V této kapitole se podíváme na funkci exponenciální, šest funkcí goniometrických a přirozený logaritmus. Na konci potom dodělám matematické operace a tím bude knihovna v použitelné verzi hotová.

5.1 Aproximace funkcí

Představa funkce tnumu je, že bude opět vracet tnum. Chci totiž opět libovolnou přesnost a také umožnit zřetězování funkcí. Hledám pak formu aproximace, která bude umožňovat libovolně škálovat, jak blízko ke kýženému číslu se výpočet ukončí. Dobrým nástrojem k tomu jsou Taylorovy polynomy. Ty se snaží hledat hodnotu $T(x)$ tak, aby byla co nejblíže hledané hodnotě $f(x)$ tak, že z nějakého bodu, kterému budeme říkat počátek, se co nejlépe snaží nepodobit průběh funkce, kterou aproximují. Pro funkci f budu Taylorův polynom stupně n se středem v a značit $T_n^{f,a}$. Pro práci s Taylorovými polynomy potřebujeme ještě naprogramovat faktoriál přirozeného čísla. To bývá typická úloha na rekurzi – té se ale vyhýbáme, protože pro velké vstupy může přetékat zásobník. Iterativní verze by tímto neduhem neměla trpět.

```
(defun factorial (n)
  (let ((result 1))
    (loop for i from n downto 1
      do (setf result (* result i)))
    result))
```

Lispový kód 14 (`factorial`): *Funkce pro výpočet faktoriálu přirozeného čísla*

Podívejme se teď na to, jak se prakticky dá počítat aproximace funkce v bodě x. Nejjednodušší je vzít funkci $T_0^{f,a}(x) = f(a)$. Je to jednoduchá aproximace, která na velmi blízkém okolí bodu a může fungovat i velmi uspokojivě. Lepší nápadem je vzít přímku, která se bude dotýkat grafu funkce f v bodě a. Předpis takovéto bude $T_1^{f,a}(x) = f(a) + f'(a)(x - a)$. To už je lepší aproximace, protože nebere v úvahu jen hodnotu funkce f v bodě a ale i její první derivaci, takže víme více o směru, kam se možná bude pohybovat. Ještě lepším nápadem pak je vzít parabolu přimknutou k grafu funkce f jako $T_2^{f,a}(x) = f(a) + f'(a)(x - a) + \frac{f''(a)}{2}(x - a)^2$. Teď už zohledňujeme funkční hodnotu, směr křivky i konvexnost. Ještě lepším nápadem je použít $T_3^{f,a}(x) : y = f(a) + f'(a)(x - a) + \frac{f''(a)}{2}(x - a)^2 + \frac{f'''(a)}{6}(x - a)^3 \ldots$[34].

Připomenutí 47 (Taylorova a Maclaurinova řada)

Kdybychom takto postupovali donekonečna (v limitním smyslu), dostali bychom Taylorovu řadu z definice 22. Pro $a = 0$ pak Taylorovu řadu nazýváme řadou Maclaurinovou.

Lze odvodit, že pokud Taylorovy zbytky konvergují k nule, lze Taylorovou řadou $T_\infty^{f,a}$ nahradit funkci f [15]. Nám ale nestačí pouhá konvergence zbytků, ale chtěli bychom jejich velikost nějak omezovat. Nejprve si zkusme nějakou formou zbytky vyjádřit.

Fakt 48 (Taylorova věta [35])

Nechť f má spojité derivace až do řádu $n+1$ na nějakém intervalu obsahujícím a. Pak pro každé x z tohoto intervalu máme Taylorův vzorec

$$f(x) = T_n^{f,a}(x) + R_n^{f,a}(x), \text{ kde} \tag{65}$$

$$T_n^{f,a}(x) = \sum_{i=0}^{n} \frac{f(i)(a)}{i!}(x-a)^i, \tag{66}$$

$$R_n^{f,a}(x) = \int_a^x \frac{(x-t)^n}{n!} f^{(n+1)}(t)dt. \tag{67}$$

Navíc existuje číslo ξ, z intervalu s krajními body x a a takové, že

$$R_n^{f,a}(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-a)^{n+1}. \tag{68}$$

Pro důkaz vizte kapitolu 7.5 v [35].

Součet v rovnici 66 nazýváme Taylorův polynom funkce f stupně n v bodě a, $R_n^{f,a}(x)$ nazýváme n-tým Taylorovým zbytkem. Vyjádření 67 pak říkáme *integrální tvar* zbytku a 68 je Lagrangeův tvar zbytku [34].

Když už máme vyjádřeny zbytky, můžeme se pokusit je zhora omezovat, stejně jako tomu bylo u geometrické řady. Ve skutečnosti nám na celou kapitolu vystačí pouze tyto dva mechanismy, tedy *Taylorův zbytek* a *Zbytek geometrické řady.*

5.2 Exponenciála

Exponenciála je funkce s předpisem $\exp(x) = e^x$ kde e je tzv. *Eulerovo číslo* definované $e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n$ [12]. To je transcendentní konstanta a je to též základ přirozeného logaritmu. Exponenciále se proto také dá říkat přirozená mocnina. Ještě podotkněme, že $\frac{d}{dx}\exp(x) = \exp(x)$.

Poznámka 49 (Značení exponenciály)

Mimo informatickou oblast jsem si nikde nevšiml, že by se exponenciála čísla x značila jinak než e^x, mé značení $\exp(x)$ tedy možná působí neadekvátně. V dalším textu ale používám i pouze funkci (exp), nikoli její hodnotu ($\exp(x)$) a zápis e^x umožňuje jen toto druhé použití. Proto se omlouvám matematickému čtenáři za neintuitivní značení, ale je zde důvodné. Navíc lépe vyjadřuje, že je exponenciála funkcí. ■

5.2.1 Exponenciála čísla

Fakt 50 (Exponenciála jako Maclaurinova řada [15])

Funkci $\exp(x)$ *lze vyjádřit jako Maclaurinovu řadu ve tvaru*

$$\exp(x) = \sum_{i\in\mathbb{N}} \frac{x^i}{i!} = \frac{1}{1} + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \tag{69}$$

Pro důkaz vizte podkapitolu A.4 v příloze A.

Podívejme se nyní na zbytek této řady. Když rozepíšeme Lagrangeův tvar, získáváme pro nějaké $\xi \in (0, x)$

$$R_n^{exp,0}(x) = \frac{e^\xi}{(n+1)!} x^{n+1}. \tag{70}$$

Podotkněme, že exponenciála je rostoucí a že $e < 2.72$ a proto

$$(\forall \xi \in (0,x))(\exp(\xi) < \exp(x) < 2.72^x). \tag{71}$$

Když vše poskládáme dohromady, získáváme aproximaci exponenciály na shora omezenou přesnost, pro $x \in \mathbb{R}$ platí

Fakt 51 (Omezení Taylorova zbytku exponenciály)

$$|R_n^{exp,0}(x)| = \left|\exp(x) - \sum_{i=0}^{n} \frac{x^i}{i!}\right| \le \left|\frac{2.72^x}{(n+1)!} x^{n+1}\right|. \tag{72}$$

Důsledek 52 (O tnumu exponenciály numu)

Pro všechna ε *existuje* $n \in \mathbb{N}^+$ *tak, aby* $\left|\frac{2.72^x}{(n+1)!} x^{n+1}\right| \le \varepsilon$ *a pak*

$$\mathcal{T}^{\exp(x)}(\varepsilon) = \sum_{i=0}^{n} \frac{x^i}{i!}. \tag{73}$$

Důkaz

Existence čísla n je zřejmá z definice limity posloupnosti a z toho, že limita podílu polynomu a faktoriálu je rovna nule.

Dále protože $\exp(x) = T_n^{exp,0}(x) + R_n^{exp,0}(x)$, lze psát

$$T_n^{exp,0}(x) \in [\exp(x) - |R_n^{exp,0}(x)|, \exp(x) + |R_n^{exp,0}(x)|], \tag{74}$$

přičemž dle předchozího faktu platí

$$T_n^{exp,0}(x) \in [\exp(x) - \left|\frac{2.72^x}{(n+1)!}x^{n+1}\right|, \exp(x) + \left|\frac{2.72^x}{(n+1)!}x^{n+1}\right|] \tag{75}$$

a z předpokladu pak

$$T_n^{exp,0}(x) \in [\exp(x) - \varepsilon, \exp(x) + \varepsilon] \tag{76}$$

a tedy

$$\mathcal{T}^{\exp(x)}(\varepsilon) = T_n^{exp,0}(x). \tag{77}$$

□

Při implementaci stačí jen iterovat přes n, dokud nebude právě odvozené omezení zbytku menší než kýžená přesnost. Stejný přístup jsme viděli již u Ludolfova čísla.

```
(defun num-exp (num eps)
  (let ((above (rat-expt 272/100 num)) (n 0)
      (nfact 1) (xpown 1) (result 1))
    (loop
     until (<= (/ (* above xpown) nfact) eps)
     do (progn
       (incf n)
       (setf nfact (factorial n)
         xpown (expt num n))
       (incf result (/ xpown nfact)))
     finally (return result))))
```

Lispový kód 15 (`num-exp`): *Funkce pro výpočet exponenciály čísla na danou přesnost*

Poznámka 53 (Eulerovo číslo jako exponenciála)
Protože triviálně platí $e = e^1$, můžu do knihovny přidat i samotné Eulerovo číslo jako jednoduchou uživatelskou funkci.

```
(defun tnum-e ()
  (lambda (eps)
    (num-exp 1 eps)))
```

Lispový kód 16 (`tnum-e`): *Funkce pro* $\mathcal{T}^e$ ■

Už tedy umíme exponenciálu čísla na danou přesnost. Teď jsme tedy ve stádiu, kdy lze pro $q \in \mathbb{Q}$ napsat

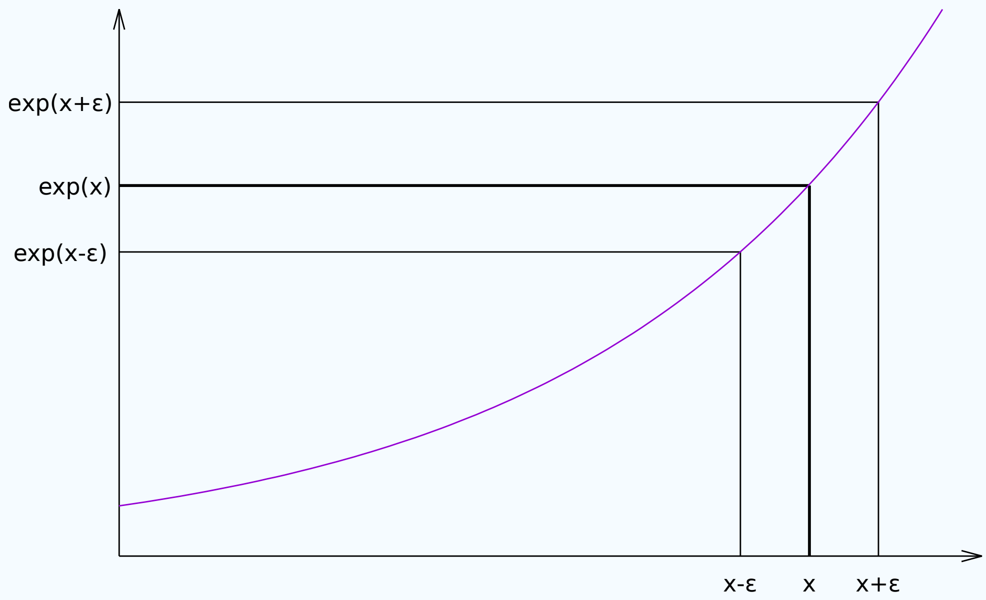
$$\mathcal{T}^{\exp(q)} = (\texttt{num-exp } q). \tag{78}$$

To není malý výsledek, bohužel nás ale sotva uspokojí. Nyní ještě musíme rozšířit funkcionalitu na všechna reálná čísla, která mají tnum. Opět jde o linku rozdílu mezi racionálními a rekurzivními čísly, která prochází celou prací.

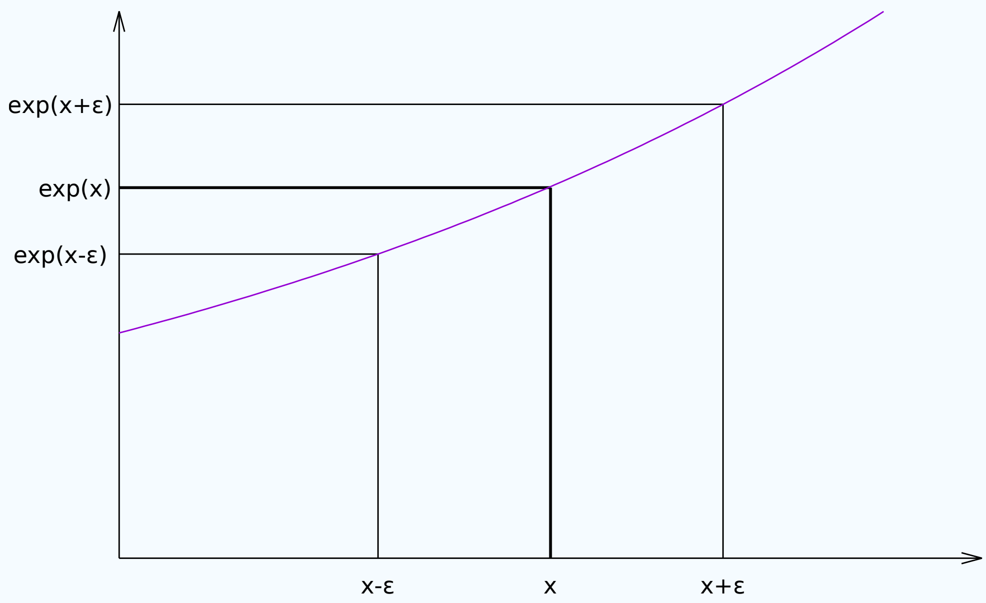
5.2.2 Exponenciála tnumu

Víme, že $\mathcal{T}^x(\varepsilon) \in [x - \varepsilon, x + \varepsilon]$, podívejme se, jak se chová přesnost čísla po projití exponenciální funkcí.

Obrázek 10: Obraz přesnosti po průchodu exponenciálou



Horní interval je vyšší než spodní.



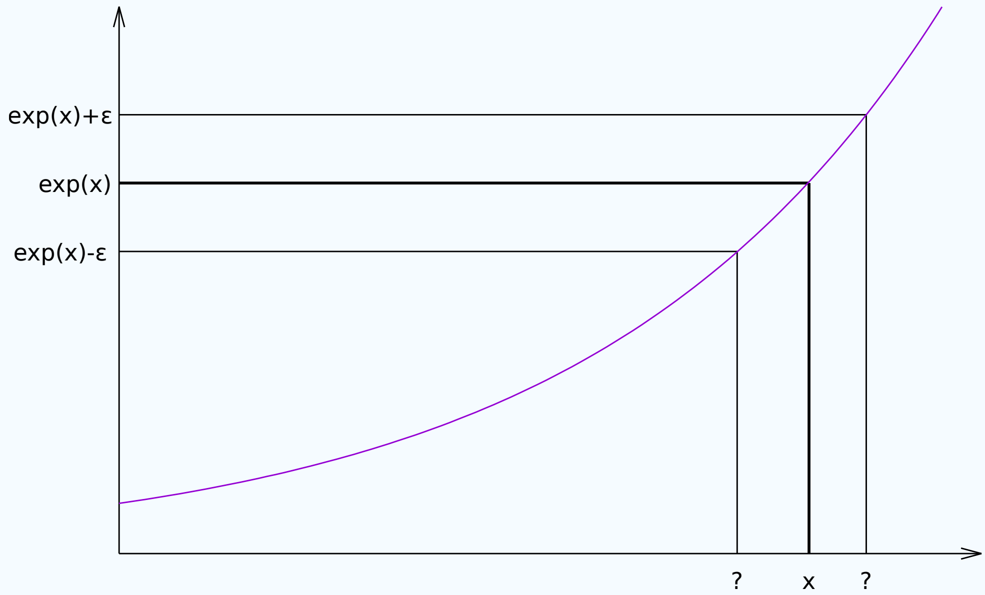
Interval $[x-\varepsilon, x+\varepsilon]$ se nezobrazí na $[e^x-\varepsilon, e^x+\varepsilon]$. Tím pádem $\mathcal{T}^{\exp(x)}(\varepsilon) \neq \mathcal{T}^{\exp(\mathcal{T}^x(\varepsilon))}(\varepsilon)$.

Vidíme, že $|(x-\varepsilon)-x| = |(x+\varepsilon)-x|$, ale $|\exp(x-\varepsilon)-\exp(x)| \neq |\exp(x+$

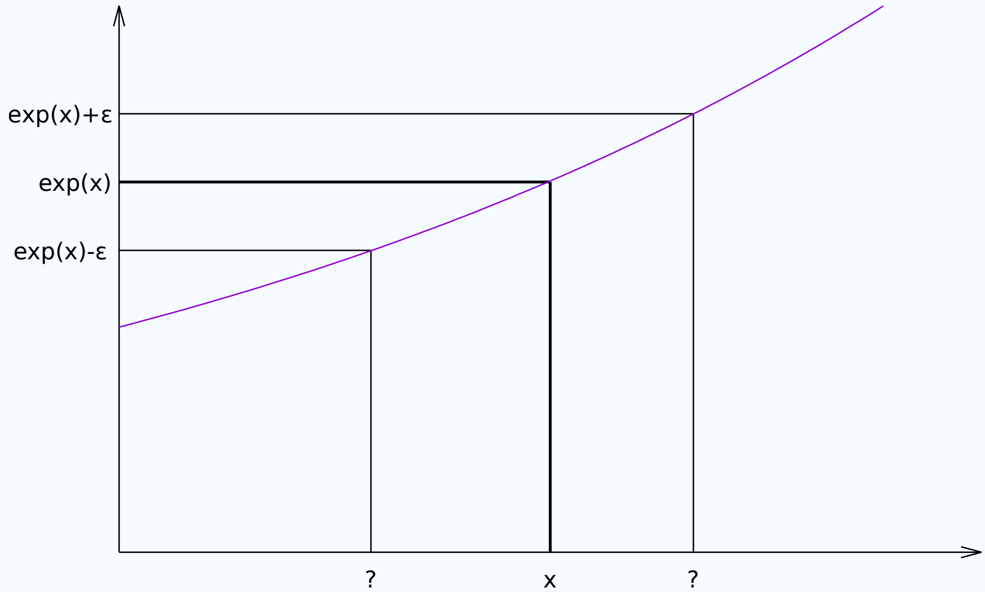
$\varepsilon) - \exp(x)|$, tedy že přesnost se průchodem nelineární funkcí deformuje a proto se interval $[\exp(x-\varepsilon), \exp(x+\varepsilon)]$ neshoduje s intervalem $[\exp(x)-\varepsilon, \exp(x)+\varepsilon]$. Důsledkem pak je, že nelze rozšířit funkce racionálních čísel na tnumy ve smyslu $\mathcal{T}^{\exp(x)}(\varepsilon) := \mathcal{T}^{\exp(\mathcal{T}^{x}(\varepsilon))}(\varepsilon)$, ale budeme to muset udělat šetrněji.

Pátráme po metodě, která nám řekne, jak přesné má být číslo na vstupu do funkce, aby jeho obraz byl v zadané přesnosti.

Obrázek 11: Vzor přesnosti před průchodem exponenciálou



Levý interval je širší než pravý.



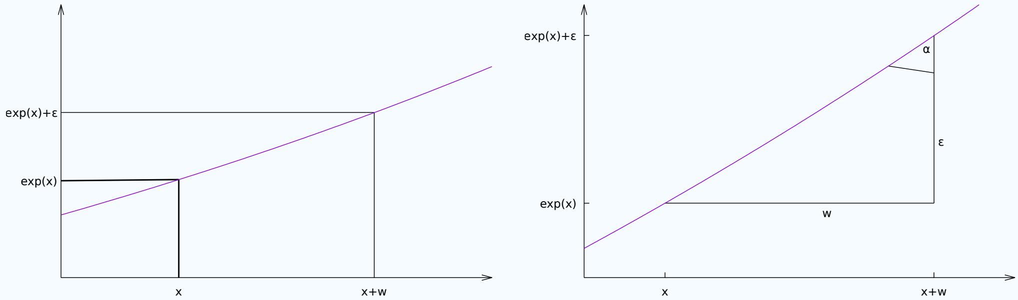
Hledáme, jaké okolí bodu x se zobrazí na ε-okolí bodu e^x.

Když si to vyneseme do rovnice, bude vypadat

$$\exp(x) + \varepsilon = \exp(x + w), \tag{79}$$

kde hledaná neznámá je w.

Obrázek 12: Zobrazení neznámé w



Neznámou w lze zobrazit jako jednu z odvěsen, druhá je ε. Fialová čára není přepona, ale exponenciála.

Podívejme se nyní, jaký vztah je mezi w a ε. Pro úhel α při vrcholu W platí, že $\mathrm{ctan}(\alpha) = \frac{\varepsilon}{w}$. Tedy

$$w = \frac{\varepsilon}{\mathrm{ctan}(\alpha)} = \frac{\varepsilon}{\tan(\frac{\pi}{2} - \alpha)} = \frac{\varepsilon}{exp(x + w)}. \tag{80}$$

Poslední úprava je přepsání skutečnosti, že tangens je v tomto bodě roven derivaci a derivace exponenciály je exponenciála. Vychází rekurzivní vztah pro w, po jeho dosazení do vztahu 79 dostáváme

$$\exp(x) + \varepsilon = \exp\left(x + \frac{\varepsilon}{\exp\left(x + \frac{\varepsilon}{\exp(x+\ldots)}\right)}\right). \tag{81}$$

Podobným způsobem lze odvodit i vztah pro opačný kraj okolí a to

$$\exp(x) - \varepsilon = \exp\left(x - \frac{\varepsilon}{\exp\left(x - \frac{\varepsilon}{\exp(x-\ldots)}\right)}\right), \tag{82}$$

dohromady to pak po propojení s notací tnumů dává vztah

$$\mathcal{T}^{\exp(x)}(\varepsilon) \in \left[\exp\left(x - \frac{\varepsilon}{\exp\left(x - \frac{\varepsilon}{\exp(x-\ldots)}\right)}\right), \exp\left(x + \frac{\varepsilon}{\exp\left(x + \frac{\varepsilon}{\exp(x+\ldots)}\right)}\right)\right]. \tag{83}$$

Poznámka 54 (Obecnější přesnost vzoru)

Odložím si zde obecnější trvzení, které vychází z právě dokázaného vztahu pro exponenciálu, jeho platnost by nyní měla být jasná pro všechny neklesající funkce spojité v x.

Lemma 55 (O přesnosti závislé proměnné)

$$f(x) + \varepsilon = f(x+w), \text{ kde } w = f\left(x + \frac{\varepsilon}{f'(x+w)}\right). \tag{84}$$

Důkaz

Tělo důkazu je již v řádcích a obrázcích nad tímto lemmatem. Protože ε může být jakkoli malé, je úhel α roven derivaci v bodě $x+w$ a proto pak platí právě uvedený vztah. □

Rekurzivní vztah vede k nějakému iterativnímu výpočtu, na ten se podíváme v následujících řádcích ohledně exponenciály a logaritmu. ■

Z výše uvedených vztahů by mělo být jasné, že při implementaci budeme hledat pevný bod a proto si zavedeme tzv. *precizní iterátor*.

Definice 56 (Precizní iterátor exponenciály)

Definujme následující posloupnost:

$$[\mathcal{T}^x]_0^{\exp,\varepsilon} = \mathcal{T}^{\exp(\mathcal{T}^x(\varepsilon))}(\varepsilon), \tag{85}$$

$$[\mathcal{T}^x]_{n+1}^{\exp,\varepsilon} = \mathcal{T}^{\exp\left(\mathcal{T}^x\left(\frac{\varepsilon}{|[\mathcal{T}^x]_n^{\exp,\varepsilon}|+\varepsilon}\right)\right)}(\varepsilon) \tag{86}$$

a pokud existuje m tak, že

$$[\mathcal{T}^x]_m^{\exp,\varepsilon} = [\mathcal{T}^x]_{m+1}^{\exp,\varepsilon}, \text{ pak klademe } [\mathcal{T}^x]_\infty^{\exp,\varepsilon} := [\mathcal{T}^x]_m^{\exp,\varepsilon}. \tag{87}$$

Důsledek 57 (O exponenciále tnumu)

Pro tnum $\mathcal{T}^x$ lze najít tnum $\mathcal{T}^{exp(x)}$ a má tvar

$$\mathcal{T}^{exp(x)}(\varepsilon) = [\mathcal{T}^x]_\infty^{exp,\varepsilon}. \tag{88}$$

Důkaz

Vychází přímo ze vztahu 83. □

```lisp
(defun tnum-exp (tnum)
  (lambda (eps)
    (let* ((num (tnum-to-num tnum eps))
           (expnum (num-exp num eps))
           (new 1))
      (loop
        until (= num new)
        do (setf new (if (> expnum 1)
                (tnum-to-num tnum (/ eps (+ expnum eps)))
                num)
          num new
          expnum (num-exp num eps))
        finally (return expnum)))))
```

Lispový kód 17 (`tnum-exp`): *Funkce pro rozšíření funkcí z racionálních čísel na tnumy*

5.3 Goniometrické

Goniometrické funkce jsou opět reálné funkce reálné proměnné. Po zkušenosti s implementací exponenciály už nás kód nepřekvapí. První dvě naprogramujeme nízkoúrovňově, zbylé čtyři pak využijí již existujících. Poznamenejme, že $\frac{d}{dx}\sin(x) = \cos(x)$, $\frac{d}{dx}\cos(x) = -\sin(x)$ a že $H(\sin) = [-1, 1] = H(\cos)$.

5.3.1 Sinus

Fakt 58 (Sinus jako Maclaurinova řada [15])
Funkci $\sin(x)$ *lze vyjádřit jako Maclaurinovu řadu ve tvaru*

$$\sin(x) = \sum_{i \in \mathbb{N}} (-1)^i \frac{x^{2i+1}}{(2i+1)!} = \frac{x}{1} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \tag{89}$$

Pro důkaz vizte podkapitolu A.5 v příloze A.

Dále protože jsou funkční hodnoty všech možných derivací v intervalu $[-1, 1]$, lze Lagrangeův tvar zbytku vyjádřit bez znaménka a pak díky Taylorově větě platí

$$\left|\mathcal{R}_n^{sin,0}(x)\right| \leq \left|\frac{x^{2n+1+1}}{(2n+1+1)!}\right| = \left|\frac{x^{2n+2}}{(2n+2)!}\right|. \tag{90}$$

Důsledek 59 (Sinus numu)
Pro všechna ε *existuje* $n \in \mathbb{N}^+$ *tak, aby* $\left|\frac{x^{2n+2}}{(2n+3)!}\right| \leq \varepsilon$ *a pak*

$$\mathcal{T}^{\sin(x)}(\varepsilon) = \sum_{i=0}^{n} (-1)^i \frac{x^{2i+1}}{(2i+1)!} \tag{91}$$

Důkaz

Běží podobně jako u exponenciály. Jde opět o exponenciálu nad faktoriálem, proto je jasná limita i existence n. Dále z omezení $R_n^{\sin,0}$ lze odvodit $T_n^{\sin,0} \in [\sin(x) - |R_n^{\sin,0}|, \sin(x) + |R_n^{\sin,0}|]$ a pak $T_n^{\sin,0} \in [\sin(x) - |\frac{x^{2n+2}}{(2n+2)!}|, \sin(x) + |\frac{x^{2n+2}}{(2n+2)!}|]$ a tudíž $T_n^{\sin,0} \in [\sin(x) - \varepsilon, \sin(x) + \varepsilon]$, z čehož pak $\mathcal{T}^{\sin(x)}(\varepsilon) = T_n^{\sin,0}(x)$. $\square$

```
(defun num-sin (x eps)
  (let ((n 0) (result 0) (2n+1 1))
    (loop
      until (<= (abs (/ (rat-expt x (1+ 2n+1))
                        (factorial (1+ 2n+1))))
                eps)
      do (progn
           (incf result
             (/ (rat-expt x 2n+1)
                (factorial 2n+1)
                (expt -1 n)))
           (incf n)
           (setf 2n+1 (1+ (* 2 n))))
      finally (return result))))
```

Lispový kód 18 (`num-sin`)**:** *Funkce pro sinus čísla*

Protože derivace sinu je kosinus, který nabývá hodnot mezi -1 a 1, nebude nutné provádět korekce přesnosti podle funkční hodnoty derivace a tato skutečnost vede na jednoduchý vztah.

Lemma 60 (O sinu tnumu)

$$\mathcal{T}^{\sin(x)}(\varepsilon) = \mathcal{T}^{sin(\mathcal{T}^x(\varepsilon))}(\varepsilon) \tag{92}$$

Důkaz

Plyne nepřímo z lemmatu 55 a z omezení absolutních funkčních hodnot kosinu jedničkou. Lemma nelze použít doslovně, protože kosinus není neklesající funkce, ale z argumentu o omezení funkčních hodnot plyne, že nás stejně přesný tvar vzoru přesnosti a tudíž ani bod derivace nezajímá. $\square$

```
(defun tnum-sin (tnum)
  (lambda (eps)
    (num-sin (tnum-to-num tnum eps) eps)))
```

Lispový kód 19 (`tnum-sin`)**:** *Funkce pro sinus tnumu*

5.3.2 Kosinus

Fakt 61 (Kosinus jako Maclaurinova řada [15])
Funkci $\cos(x)$ *lze vyjádřit jako Maclaurinovu řadu ve tvaru*

$$\cos(x) = \sum_{i \in \mathbb{N}} (-1)^i \frac{x^{2i}}{(2i)!} = \frac{1}{1} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots \tag{93}$$

Pro důkaz vizte podkapitolu A.6 v příloze A.

Z Taylorovy věty získáváme omezení Taylorova zbytku

$$|R_n^{cos}(x)| \leq \left| \frac{x^{2n+1}}{(2n+1)!} \right| \tag{94}$$

a proto opět hledáme takové n, že když pro jakékoli ε je $|\frac{x^{2n+1}}{(2n+1)!}| \leq \varepsilon$, pak

Důsledek 62 (Kosinus numu)

$$\mathcal{T}^{\cos(x)}(\varepsilon) = \sum_{i=0}^{n} (-1)^i \frac{x^{2i}}{(2i)!}. \tag{95}$$

```
(defun num-cos (x eps)
  (let ((n 0) (result 0) (2n 0))
    (loop
      until (< (abs (/ (rat-expt x (1+ 2n))
                       (factorial (1+ 2n))))
                  eps)
      do (progn
           (incf result
             (/ (rat-expt x 2n)
                (factorial 2n)
                (expt -1 n)))
           (incf n)
           (setf 2n (* 2 n)))
      finally (return result))))
```

Lispový kód 20 (`num-cos`)**:** *Funkce pro výpočet kosinu čísla*

A nakonec právě naprogramovanou funkci využijeme ke kosinování jakékoli proměnné s tnumem. Postup je stejný jako u sinu a proto už nepíši příslušné lemma a nechávám ho ke zformulování čtenáři.

```
(defun tnum-cos (tnum)
  (lambda (eps)
    (num-cos (tnum-to-num tnum eps) eps)))
```

Lispový kód 21 (`tnum-cos`): *Funkce pro výpočet kosinu tnumu*

Zbylé goniometrické funkce už naprogramujeme uživatelsky.

5.3.3 Další goniometrické funkce

Další goniometrickou funkcí je tangens. Dá se vyjádřit pomocí sinu a kosinu, díky čemuž ho nemusím vyjadřovat jako řadu, i když pro všechny goniometrické funkce řady existují. Nejsou ale konvergentní na celé reálné ose, takže se pro naši knihovnu nehodí.

Fakt 63 (Tangens jako poměr sinu a kosinu [7])

$$\mathrm{tg}(x) = \frac{\sin(x)}{\cos(x)} \tag{96}$$

Úmluva 64 (O vypuštění některých důsledků)

Nyní by měl následovat důsledek že $\mathcal{T}^{\tan(x)} = \mathcal{T}^{\frac{\mathcal{T}^{\sin(x)}}{\mathcal{T}^{\cos(x)}}}$, což je ale myslím jasné a proto zde ani u dalších zřejmých přepsání vzorečků do jazyka tnumů tyto důsledky neuvádím. ■

```
(defun tnum-tan (tnum)
  (tnum/ (tnum-sin tnum) (tnum-cos tnum)))
```

Lispový kód 22 (`tnum-tan`): *Funkce pro výpočet tangentu tnumu*

Zbylé funkce jsou obrácenou hodnotou již napsaných.

Fakt 65 (Kosekans jako obrácená hodnota sinu [7])

$$\csc(x) = \sin^{-1}(x) \tag{97}$$

```
(defun tnum-csc (tnum)
  (/tnum (tnum-sin tnum)))
```

Lispový kód 23 (`tnum-csc`): *Funkce pro výpočet kosekantu tnumu*

Fakt 66 (Sekans jako obrácená hodnota kosinu [7])

$$\sec(x) = \cos^{-1}(x) \tag{98}$$

```
(defun tnum-sec (tnum)
  (/tnum (tnum-cos tnum)))
```

Lispový kód 24 (`tnum-sec`)**:** *Funkce pro výpočet sekantu tnumu*

Fakt 67 (Kotangens jako obrácená hodnota tangentu [7])

$$\mathrm{cotg}(x) = tan^{-1}(x) = \frac{\cos(x)}{\sin(x)} \tag{99}$$

```
(defun tnum-ctan (tnum)
  (tnum/ (tnum-cos tnum) (tnum-sin tnum)))
```

Lispový kód 25 (`tnum-ctan`)**:** *Funkce pro výpočet kotangentu tnumu*

5.4 Logaritmus

Logaritmus je inverzní funkce k exponenciále. Je opět vyjadřitelná řadou.

Fakt 68 (Logaritmus jako řada [36])
Pro $x \in \mathbb{R}^+$ platí

$$\ln(x) = 2\sum_{i\in\mathbb{N}} \frac{1}{2i+1}\left(\frac{x-1}{x+1}\right)^{2i+1} \tag{100}$$

Člen $\frac{1}{2i+1}(\frac{x-1}{x+1})^{2i+1}$ je menší než $(\frac{x-1}{x+1})^{2i+1}$ a tento je menší než $(\frac{x-1}{x+1})^{2i}$. Toto je geometrická posloupnost, jejíž n-tý zbytek je roven $\frac{(\frac{x-1}{x+1})^{2n+2}}{1-\frac{x-1}{x+1}}$ podle faktu 26.

```
(defun num-ln (x eps)
  (setf eps (/ eps 2))
  (let ((n 0) (result 0) (q (/ (1- x) (1+ x))))
    (loop
      until (<= (/ (expt q (* 2 (1+ n))) (- 1 q))
                eps)
      do (progn
           (incf result
             (/ (expt q (1+ (* 2 n))) (1+ (* 2 n))))
           (incf n))
      finally (return (* 2 result)))))
```

Lispový kód 26 (`num-ln`)**:** *Funkce pro logaritmus čísla*

Tím bychom měli přirozený logaritmus pro čísla. Podívejme se nyní, jak vypadá omezení nezávislé proměnné pro logaritmus. Po dosazení do vztahu 84 získáváme

$$\ln(x) + \varepsilon = \ln\left(x + \frac{\varepsilon}{\ln'(x+w)}\right) \tag{101}$$

a protože $\ln(x+w) = (x+w)^{-1}$, pak

$$\ln(x) + \varepsilon = \ln(x + \varepsilon(x+w)), \tag{102}$$

tedy násobíme přesnost hodnotou proměnné a protože přesnost nemůže být nulová, použijeme k vyčíslení $\mathcal{T}^x$ nenulový tnum a tím je problém vyřešen. Vezmeme totiž pesimistický odhad $\mathcal{T}^x(\varepsilon_\emptyset) - \varepsilon_\emptyset$ a přenásobíme jím epsilon. Díky třetí podmínce v definici 39 toto mohu udělat.

```
(defun tnum-ln (tnum)
  (lambda (eps)
    (multiple-value-bind (num eps0)
      (get-nonzero-num+eps tnum eps)
      (num-ln (tnum-to-num tnum eps) (* eps (- num eps0))))))
```

Lispový kód 27 (`tnum-ln`)**:** *Funkce pro logaritmus tnumu*

POZNÁMKA 69 (DOKONČENÍ SYSTÉMU)

Pomocí logaritmu v kombinaci s exponenciálou lze přinést i mocninné operace a ucelím tak základní funkcionalitu knihovny `tnums`, kterou jsem si předsevzal. Mocninu udělám jako v lemmatu 45.

```
(defun tnum-expt (tnum1 tnum2)
  (tnum-exp (tnum* tnum2 (tnum-ln tnum1))))
```

Lispový kód 28 (`tnum-expt`)**:** *Funkce pro umonování tnumů*

Odmocninu pak píši podle faktu 46. Oproti mocnině jsou prohozené argumenty, ale tnum-tou odmocninu tnumu chápu tak, že odmocnitel je jako první a odmocněnec jako druhý, říká se „třetí odmocnina z osmi“.

```
(defun tnum-root (tnum1 tnum2)
  (tnum-expt tnum2 (/tnum tnum1)))
```

Lispový kód 29 (`tnum-root`)**:** *Funkce pro odmocňování tnumu*

■

Takto je tedy dokončena základní funkcionalita knihovny `tnums` a v další části se podíváme na její používání, perspektivu a doprogramujeme nějaké uživatelské funkce.

Část III
Rozhraní

V uživatelské části pojednám o vlastním používání knihovny `tnums`. Nejprve zkusíme převádět čísla mezi interními reprezentacemi Lispu a tnumů. Poté se podíváme, jak se používají matematické operace a poté spočítáme nějaké matematické funkce. Také ještě několik funkcí doprogramujeme, ale nezávisle na vnitřní implementaci tnumů. Nakonec se podíváme na nedostatky knihovny a její výhled jak v oblasti rozšiřování funkcionality, tak na poli zvyšování efektivity.

6 Uživatelské funkce

Při tvorbě knihovny pracující s tnumy jsem už některé funkce, které bych prohlásil za uživatelské napsal. Uživatelskou funkcí myslím takovou funkci, která je nezávislá na vnitřní reprezentaci tnumů a jen rozšiřuje funkčnost a nevolá pomocné vnitřní funkce. Takovými byly třeba `tnum-`, `tnum-tan` nebo `tnum-root`. V této kapitole některé další uživatelské funkce dopíšeme, aby bylo vidět, jak se knihovna `tnums` používá. Poté, co si projdeme proces instalace se podíváme na jednoduché převody a vyčíslování konstant, poté přidáme operace a na závěr funkce. Ukážeme, že síla knihovny spočívá v jednoduchém vytváření nových tnumů a ve velmi silně oddělené vnitřní implementaci od vnějšího chování.

6.1 Vymezení

Definice 70 (Uživatelská funkce)
Uživatelskou funkcí myslíme takovou funkci, která nezná reálnou vnitřní implementaci tnumů a nepoužívá vnitřní pomocné funkce. Sama je veřejná.

Rozdělení funkcí na funkce na rozhraní a na pomocné vnitřní funkce zobrazuje tabulka 3. Uživatelská funkce rozšiřuje funkcionalitu, musí tedy být vnější.

Jiné rozdělení funkcí je na ty znalé vnitřní implementace tnumů a ty, které s tnumy nízkoúrovňově pracovat nemusí. Uživatelská funkce – která stavbu tnumu nezná – je tedy nezávislá na konkrétní implementaci a pokud se implementace změní (což já jsem udělal už dvakrát, více v sekci 7.3.1), tyto funkce zůstanou. Toto rozdělení zobrazuje tabulka 4.

Knihovna `tnums` obsahuje 29 funkcí. Podívejme se nyní, které z nich jsou uživatelské, tedy vnější funkce, které jsou implementačně nezávislé a nevolají žádnou pomocnou funkci: `tnum-`, `tnum/`, `tnum-csc`, `tnum-sec`, `tnum-tan`, `tnum-ctan`, `tnum-expt` a `tnum-root`. To je osm funkcí, což je celkem hezký počet na to, že jsme je uživatelsky naprogramovali v podstatě mimoděk. S přimhouřenýma očima lze za uživatelské prohlásit i `tnum-e` a `-tnum`, čímž jsme na více než třetině.

Tabulka 3: Vnitřní a vnější funkce

vnější		vnitřní
num-to-tnum	tnum-to-num	factorial
tnum-e	tnum-pi	rat-expt
tnum+	tnum*	num-cos
tnum-	tnum/	num-ln
-tnum	/tnum	get-nonzero-tnum+eps
tnum-exp	tnum-ln	create-list-for-multiplication
tnum-sin	tnum-csc	num-exp
tnum-cos	tnum-sec	num-sin
tnum-tan	tnum-ctan	tnum*num
tnum-expt	tnum-root	

Tabulka zobrazuje v prvním sloupci funkce pro používání uživatelem, ve druhém pak pomocné funkce, které by uživatelem být volány neměly.

Tabulka 4: Nízkoúrovňové a vysokoúrovňové funkce

implementačně závislé	implementačně nezávislé	
num-to-tnum	factorial	tnum-sec
tnum-to-num	rat-expt	tnum-tan
tnum-ln	tnum-e	-tnum
tnum-pi	num-ln	tnum-
tnum*num	tnum-ctan	tnum/
tnum+	tnum-expt	tnum-csc
/tnum	get-nonzero-tnum+eps	num-exp
tnum*	create-list-for-multiplication	num-sin
tnum-exp	tnum-root	num-cos
tnum-sin		
tnum-cos		

Tabulka zobrazuje v prvním sloupci funkce, které pracují s konkrétní implementací tnumů jako funkcí přesnosti a ve druhém ty, které jsou od tohoto faktu odstíněny.

V následujícím textu tento počet rozšíříme, aby bylo jasné, jaké možnosti uživatelské rozšiřitelnosti i takto funkcionálně skromná knihovna nabízí.

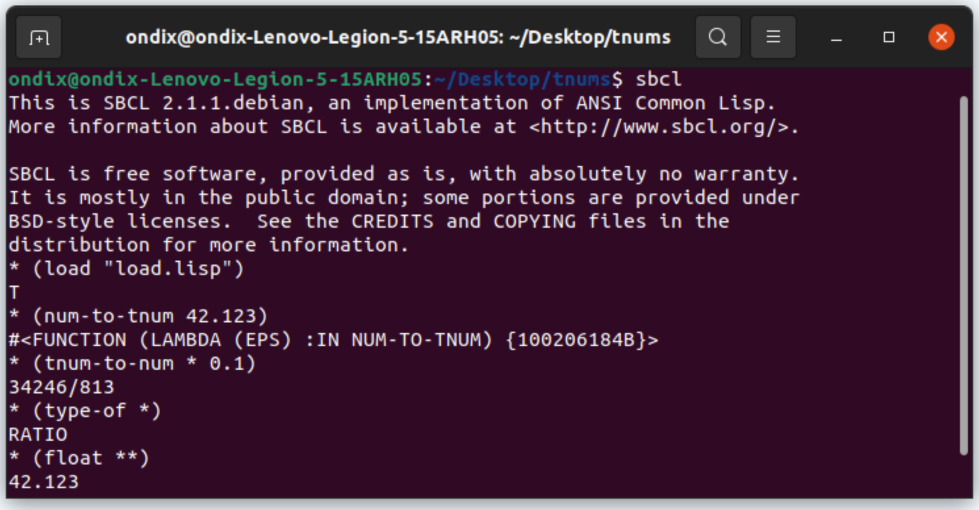
6.2 Instalace

Knihovna `tnums` je k dostání na githubu na odkaze ⟨https://github.com/slavon00/tnums⟩ nebo pro čtenáře tištěné verze na přiloženém fyzickém disku. Je to Lispová knihovna a od uživatele předpokládá základy práce s Lispem a REPLem.

Vše, co jsme doposud naprogramovali najdeme v souboru `src/tnums.lisp` a všechny funkce, které přidáme v této kapitole pak v `src/user-functions.lisp`. Testy, které budu ukazovat jsou v souboru `src/tests.lisp`. Vše je možné jednoduše načíst jen evaluací souboru `load.lisp`. Pro kompletní obsah adresáře vizte přílohu B.

Knihovna se v budoucnosti může měnit, takže tyto informace mohou zastarat. Pro zpětnou kompatibilitu ale knihovna na githubu bude vždy obsahovat soubor `README.md` nebo ekvivalentní, aby mohla instalace proběhnout bez problémů.

Obrázek 13: Načtení knihovny `tnum` do `SBCL`



Aby se nemusely ručně načítat všechny soubory, lze knihovnu `tnums` též načíst jen evaluací souboru `load.lisp`.

6.3 Převody a konstanty

Téměř všechny naprogramované funkce berou jako vstup tnumy. To jsou abstraktní struktury, které interně reprezentujeme jako funkce malých čísel. Pokud chceme vytvořit tnum z nějakého čísla, které již máme nějak uložené v Lispu, slouží k tomu funkce `num-to-tnum`.

```
* (num-to-tnum 42.123)
#<FUNCTION (LAMBDA (EPS) :IN NUM-TO-TNUM) 10020A056B>
```

Lispový test 1 (`num-to-tnum`)**:** *Představení funkce pro převod numu na tnum*

Pro převod opačným směrem máme inverzní funkci `tnum-to-num`, která kromě tnumu bere i druhý argument představující přesnost, se kterou chceme daný tnum vyčíslit.

```
* (tnum-to-num * 0.1)
34246/813
```

Lispový test 2 (`tnum-to-num`)**:** *Představení funkce na převod tnumu na num*

Jak vidno, num vrácený funkcí `tnum-to-num` je racionální číslo.

```
* (type-of *)
RATIO
* (float **)
42.123
```

Lispový test 3 (Typ výstupu je číslo)**:** *Ověření typu vraceného numu*

Protože knihovna vrací čísla jak je chápe Lisp, jsou výsledky vyčíslení plně kompatibilní s ostatními funkcemi pro čísla. Naprogramujme nyní funkci, která bude tnumy převádět na textové řetězce. Takto získáme dlouhé rozvoje v čitelné podobě, ne jen jako lidskému oku nic neříkající velké zlomky.

```
(defun tnum-to-string (tnum count)
  (let ((num (tnum-to-num tnum (1+ count))) (output ""))
    (when (< num 0) (setf output "-" num (- num)))
    (multiple-value-bind (digit rem)
        (floor num)
        (setf output (concatenate 'string output
                    (write-to-string digit) ".")
          num (* 10 rem)))
    (dotimes (i count (concatenate 'string output "..."))
      (multiple-value-bind (digit rem)
        (floor num)
        (setf output (concatenate 'string output
                    (write-to-string digit))
          num (* 10 rem))))))
```

Lispový kód 30 (`tnum-to-string`)**:** *Funkce na převod tnumu na textový řetězec*

Funkce bere jako vstup tnum a přirozené číslo značící počet desetinných míst. Otestujeme ji na výpisu Ludolfova čísla.

```
* (tnum-to-string (tnum-pi) 50)
"3.14159265358979323846264338327950288419716939937510..."
```

Lispový test 4 (`tnum-string` a `tnum-pi`)**:** *Vyčíslení Ludolfova čísla na 50 desetinných míst*

Vzhledem k rozšíření množiny přípustných hodnot funkce `tnum-to-num` je možné pohodlně přepínat mezi návratovou hodnotou jako číslem a textovým řetězcem. Ukážeme si to na vyčíslení Eulerova čísla.

```
* (tnum-to-num (tnum-e) 20)
611070150698522592097/224800145555521536000
* (tnum-to-string (tnum-e) 20)
"2.71828182845904523536..."
```

Lispový test 5 (`tnum-string` a `tnum-e`)**:** *Vyčíslení Eulerova čísla na 20 desetinných míst a jeho vrácení jako čísla a jako stringu*

6.4 Operace

Operace tnumu je funkce $\bigtimes_{i=0}^{n-1} \mathfrak{T} \to \mathfrak{T}$, kde $n \in \mathbb{N}$ nazýváme aritou. Operace `tnum+` a `tnum*` mohou mít libovolný počet argumentů, jsou tedy ($n \in \mathbb{N}$)-ární, operace `-tnum` a `/tnum` jsou striktně unární, operace `tnum-` a `tnum/` potřebují alespoň jeden argument, jsou tedy ($n \in \mathbb{N}^+$)-ární a operace `tnum-expt` a `tnum-root` jsou binární.

V lispu jsou dvě hezké funkce na inkrementaci a dekrementaci čísla. To samé nyní přidáme pro tnumy.

```
(defun tnum-1+ (tnum)
  (tnum+ (num-to-tnum 1) tnum))
```

Lispový kód 31 (`tnum-1+`)**:** *Funkce pro inkrementaci tnumu o jedničku*

Funkce pro dekrementaci se také dá napsat pohodlně uživatelsky.

```
(defun tnum-1- (tnum)
  (tnum- tnum (num-to-tnum 1)))
```

Lispový kód 32 (`tnum-1-`)**:** *Funkce pro dekrementaci tnumu o jedničku*

Také by šla napsat nejpoužívanější odmocnina a sice druhá.

```
(defun tnum-sqrt (tnum)
  (tnum-root (num-to-tnum 2) tnum))
```

Lispový kód 33 (`tnum-sqrt`)**:** *Funkce pro druhou odmocninu tnumu*

Výše naprogramované použijeme k zavedení další konstanty, zlatého řezu.

Definice 71 (Zlatý řez [37])
Zlatý řez představuje kladné řešení rovnice $x^2 - x - 1 = 0$, je tedy roven hodnotě

$$\varphi = \frac{1 + \sqrt{5}}{2} \tag{103}$$

Jedná se o další iracionální konstantu, tentokrát algebraickou, protože je řešením algebraické rovnice a její tnum je jen syntaktický přepis uvedené definice.

```
(defun tnum-phi ()
  (tnum/ (tnum-1+ (tnum-sqrt (num-to-tnum 5))) (num-to-tnum 2)))
```

Lispový kód 34 (`tnum-phi`): *Funkce vracející tnum představující zlatý řez*

A ještě test.

```
* (tnum-to-string (tnum-phi) 50)
"1.61803398874989484820458683436563811772030917980576..."
```

Lispový test 6 (`tnum-phi`): *Představení funkce pro zlatý řez*

6.5 Funkce

Funkce tnumů jsou všechny unární. Jedná se o přirozený logaritmus, goniometrické funkce a přirozenou exponenciálu. Omezení definičního oboru jsou stejná jako jsme zvyklí, naprogramované funkce tedy nejsou o nic „slabší“.

Exponenciálu jsme využili už při psaní obecné mocniny. V podobném duchu nyní zavedeme obecný logaritmus. Vyjdeme z faktu, že lze převádět mezi různými základy.

Fakt 72 (Obecný logaritmus jako podíl přirozených [7])
Pro $a > 1$ a $x \in \mathbb{R}^+$ platí

$$\log_a(x) = \frac{ln(x)}{ln(a)} \tag{104}$$

Nová funkce bude brát dva argumenty a proto nejde tak úplně o funkci tnumu, jak ji v této práci chápeme, ale spíše o matematickou operaci, nicméně na eleganci zápisu to nic neubírá.

```
(defun tnum-log (tnum1 tnum2)
  (tnum/ (tnum-ln tnum2) (tnum-ln tnum1)))
```

Lispový kód 35 (`tnum-log`): *Funkce pro výpočet obecného logaritmu*

Uživatelská funkce potom podobně jako u odmocniny prohazuje argumenty, protože mluvíme vždy o nějakém logaritmu něčeho, například „devítkový logaritmus dvou“. Ten je i předmětem následujícího testu.

```
* (tnum-to-string (tnum-log (num-to-tnum 9) (num-to-tnum 2)) 50)
"0.31546487678572871854976355717138042714979282006594..."
```

Lispový test 7 (`tnum-log`)**:** *Vyčíslení devítkového logaritmu dvou*

Gomiometrické funkce jsou z velké části napsány též uživatelsky, takže by mělo být jasné, jak se s nimi z tohoto pohledu pracuje. Ukážu tedy jen vyčíslení, aby bylo vidět, že funkce opravdu fungují. Následuje výpočet sinu jedničky.

```
* (coerce (tnum-to-num (tnum-sin (num-to-tnum 1)) -20)
    'long-float)
0.8414709848078965d0
```

Lispový test 8 (`tnum-sin`)**:** *Představení funkce na výpočet sinu tnumu*

Pokud má čtenář pocit, že takovéto číslo už někdy viděl, je tento pocit správný, protože přesně tímto způsobem jsem naplnil tabulku 2.

V tomto bodě bychom tedy měli mít jasnou představu, jak se funkce používají a že fungují. Dokonce jsme nějaké funkce přidali a to bez nutnosti znalosti vnitřního provedení tnumů. Také jsme viděli, že skrze rozhraní a hlavně funkci `tnum-to-num` lze získat přesné číslo a lze s ním dále nakládat v dalších aplikacích. Trochu blíže se na toto ještě zaměříme v další kapitole, nyní se podívejme ještě na rychlost, jakou knihovna pracuje a přehled všech funkcí rozhraní.

6.6 Rychlost

Tabulka 5 zobrazuje, jak dlouho trvalo vyhodnocení příkazů. Hodnoty budou vždy závislé na konkrétním stroji a jeho momentálním zatížení, obecnou představu o časové náročnosti výpočtů by ale měly poskytnout.

Tabulka 5: Doba výpočtů daných výrazů

Výraz	Doba vyhodnocení (s)
`((let ((tn (tnum/ (tnum-pi) (tnum-e) (tnum-phi)))`	-
`(tnum-to-string tn 50)`	1.367339
`(tnum-to-string (tnum-sin tn) 50)`	28.180623
`(tnum-to-string (tnum-csc tn) 50)`	3600.500439
`(tnum-to-string (tnum-ctan tn) 50))` 25431.702944 ?	

Tabulka v prvním sloupci zobrazuje výrazy, které byly vyhodnocovány a ve druhém čas, který toto vyhodnocení zabralo. Doba byla měřena makrem `time` a hodnoty jsou z řádku „(...) seconds of total run time“.

6.7 Vnější volání

Pro přehlednost, jaké rozhraní knihovna `tnums` nabízí následuje souhrnná tabulka zobrazující všechny funkce určené k volání uživatelem.

Tabulka 6: Funkce nabízené knihovnou `tnums`

Název	Argumenty	Význam
`tnum-to-num`	`tnum:tnum`, `eps:num`	převod `tnumu` na číslo s přesností `eps`
`tnum-to-string`	`tnum:tnum`, `count:num`	převod `tnumu` na textový řetězec o `count` desetinných místech
`num-to-tnum`	`num:num`	převod `numu` na tnum
`tnum-pi`	$\emptyset$	Ludolfovo číslo jako tnum
`tnum-e`	$\emptyset$	Eulerovo číslo jako tnum
`tnum-phi`	$\emptyset$	Zlatý řez jako tnum
`-tnum`	`tnum:tnum`	$-$`tnum`
`tnum+`	0+ `tnumů`	součet tnumů
`tnum-`	1+ `tnumů`	rozdíl tnumů
`/tnum`	`tnum:tnum`	1/`tnum`
`tnum*`	0+ `tnumů`	součin tnumů
`tnum/`	1+ `tnumů`	podíl tnumů
`tnum-expt`	`arg1:tnum`, `arg2:tnum`	$\mathtt{arg1}^{\mathtt{arg2}}$
`tnum-sqrt`	`arg1:tnum`, `arg2:tnum`	$\sqrt[\mathtt{arg1}]{\mathtt{arg2}}$
`tnum-log`	`arg1:tnum`, `arg2:tnum`	$\log_{\mathtt{arg1}}(\mathtt{arg2})$
`tnum-1+`	`tnum:tnum`	$\mathtt{tnum} + 1$
`tnum-1-`	`tnum:tnum`	$\mathtt{tnum} - 1$
`tnum-exp`	`tnum:tnum`	přirozená mnocnina `tnumu`
`tnum-ln`	`tnum:tnum`	přirozený logaritmus `tnumu`
`tnum-sqrt`	`tnum:tnum`	druhá odmocnina `tnumu`
`tnum-sin`	`tnum:tnum`	sinus `tnumu`
`tnum-cos`	`tnum:tnum`	kosinus `tnumu`
`tnum-tan`	`tnum:tnum`	tangens `tnumu`
`tnum-csc`	`tnum:tnum`	kotangens `tnumu`
`tnum-sec`	`tnum:tnum`	sekans `tnumu`
`tnum-ctan`	`tnum:tnum`	kosekans `tnumu`

Tabulka v prvním sloupci zobrazuje funkční symbol, v posledním význam funkce aplikované na argumenty z prostředního sloupce. Čtyři části rozdělené horizontálními čarami jsou po řadě funkce pro převody, konstanty, operace a matematické funkce. Součástí jsou i uživatelské funkce.

7 Perspektiva

V poslední kapitole probereme možné budoucí směřování knihovny `tnums` a její nedostatky. Nejprve zjistíme, proč není dobrý nápad vnímat `tnums` jako kalkulačku. Krátce se zastavíme u optimalizace implementace, kde popíšu dva hlavní směry, kterým bychom se při optimalizaci mohli vydat. Nakonec se ohlédneme na proces vzniku knihovny a odkryjeme nějaká děravá místa, která současná verze knihovny bohužel má.

7.1 Kalkulačka

Princip fungování knihovny `tnums` se dá popsat ve dvou krocích. Nejprve se vytvoří abstraktní datová struktura, která představuje číslo. Říkáme jí tnum. Tento proces je rychlý, protože se nic nevyhodnocuje a jen líně ukládá pro další práci. Vytvořený tnum se potom s danou přesností převede na číslo, kterému říkám num a toto je aproximací výsledku na danou přesnost.

Takto se ale normálně čísla nepočítají. Jsme zvyklí, že napíšeme například výraz `(+ 2 3)` a evaluátor rovnou vrátí výsledek `5`. Zkusme se tedy nyní zamyslet, jak napsat nějakou nadstavbu nad tnumy tak, aby se jednalo o „přesnou kalkulačku“. Obě fáze výpočtu rozebereme zvlášť.

7.1.1 Vytvoření tnumu

Všechny funkce jsou nyní napsány tak, že pracují s tnumy. Pokud na jejich vstup místo tnumu přijde číslo, bez ošetření toto volání skončí chybou. My ovšem máme nástroj na převod numu na tnum, takže by nemělo být těžké napsat funkci s povoleným vstupem jako tnum i číslo. Zkusme to například pro funkci sinus.

```
(defun tsin (arg)
  (when (realp arg)
    (setf arg (num-to-tnum arg)))
  (tnum-sin arg))
```

Lispový kód 36 (`tsin`): *Funkce vracející sinus tnumu, případně tnum sinu čísla*

Vše pak funguje, jak očekáváme.

```
* (float (tnum-to-num (tsin 1) 6))
0.84147096
```

Lispový test 9 (`tsin`): *Představení funkce na převod sinu čísla na sinus tnumu*

V podobném duchu by mohly probíhat i další úpravy. Pro sčítání odvodíme jedno lemma, které nám umožní efektivně rozšířit množinu argumentů.

Lemma 73 (O sčítání tnumu s numem)

Mějme reálnou proměnnou x a reálnou konstantu c. Pak se přesnost tnumu proměnné x přičtením konstanty nezmění, neboli

$$\mathcal{T}^{x+c}(\varepsilon) = \mathcal{T}^{x}(\varepsilon) + c \tag{105}$$

Důkaz

Levá strana znamená, že

$$\mathcal{T}^{x+c}(\varepsilon) \leq x + c + \varepsilon \wedge \mathcal{T}^{x+c}(\varepsilon) \geq x + c - \varepsilon, \tag{106}$$

po přerovnání získávám

$$\mathcal{T}^{x+c}(\varepsilon) \leq x + \varepsilon + c \wedge \mathcal{T}^{x+c}(\varepsilon) \geq x - \varepsilon + c. \tag{107}$$

Dále z definice tnumu máme, že

$$\mathcal{T}^{x}(\varepsilon) \leq x + \varepsilon \wedge \mathcal{T}^{x}(\varepsilon) \geq x - \varepsilon, \tag{108}$$

přidáním konstanty na všechny strany platnost nerovnice nezměnín, získávám

$$\mathcal{T}^{x}(\varepsilon) + c \leq x + \varepsilon + c \wedge \mathcal{T}^{x}(\varepsilon) + c \geq x - \varepsilon + c, \tag{109}$$

z čehož plyne platnost lemmatu. □

V kódu se pak nejprve posčítají dohromady zvlášť tnumy a zvlášť numy, ty se pak převedou na tnum a vrátí se součet těchto dvou tnumů.

```
(defun t+ (&rest args)
  (tnum+ (apply 'tnum+ (remove-if 'realp args))
    (num-to-tnum (apply '+ (remove-if-not 'realp args)))))
```

Lispový kód 37 (t+): *Funkce pro součet čísel a tnumů*

Sečtěme například $1 + \varphi + 2 + e + 3$.

```
* (float (tnum-to-num (t+ 1 (tnum-e) 2 (tnum-phi) 3) 6))
10.336316
```

Lispový test 10 (t+): *Představení funkce na součet čísel a tnumů*

V podobném smyslu budeme postupovat i u násobení. Snad už je jasný princip a takto by se napsaly funkce pro všechny funkce z tabulky 6. Byla by to sice práce, ale s použitím maker bychom mohli hodně kódu ušetřit. Násobení používá jednu z prvních funkcí, kterou jsme do našeho systému přidali.

```
(defun t* (&rest args)
  (tnum*num (apply 'tnum* (remove-if 'realp args))
    (apply '* (remove-if-not 'realp args))))
```

Lispový kód 38 (`t*`): *Funkce pro násobení tnumů a čísel*

A test.

```
* (float (tnum-to-num (t* 1/2 (tnum-pi) 4) 6))
6.2831855
```

Lispový test 11 (`t*`): *Představení funkce na součin čísel a tnumů*

Takto by bylo vyřešeno rozšíření funkcionality **`tnums`** z tnumů i na numy. Napsal jsem způsob, jak by se přepisovaly jednotlivé funkce rozhraní. Jde buď jen odchytávat čísla a měnit je pomocí `num-to-tnum` na tnumy a tyto potom dál zpracovávat jako normální tnumy (pro příklad vizte `tsin`). Nebo jde někde jít zkratkou a operaci nejprve provést s čísly a na výsledek pak aplikovat `num-to-tnum` a tento potom pustit do standardní funkce pro tnumy (pro příklad vizte `t+`). Největší zkratkou pak je úplně obejít standardní funkce pro zpracovávání tnumů a ještě více tak urychlit výpočet (pro příklad vizte `t*`). První přístup je ale obecný a jde nasadit na všechny případy, s těmi dalšími musíme mít při rozšiřování štěstí.

7.1.2 Vypsání tnumu

Při normální práci s tnumem jej nejprve vytvoříme a potom vyčíslíme (zavoláme s přesností). Na první část tohoto procesu jsem právě napsal návod, jak pracovat mimo tnumů i s numy. Nyní se podívejme na druhou část.

Při vyčíslování tnumu funkcí `tnum-to-num` vnitřně dochází k zavolání funkce reprezentující tnum s danou přesností a ten se sám vyhodnotí na výsledné číslo. Vše se děje hladce a syntakticky jednoduše. Když ale budeme chtít nad tnumy vytvořit kalkulačku, tento proces bude nabourán, protože uživatel nebude vědět, že tnumy chtějí zadat přesnost. Musíme ji tam tedy nějak dosadit. Podívejme se, jak by vypadal třeba sinus s fixní přesností na 20 desetinných míst.

```
(defun calc-sin (num)
  (coerce (tnum-to-num (tsin num) 20) 'long-float))
```

Lispový kód 39 (`calc-sin`): *Funkce pro výpočet sinu čísla*

Toto sice funguje,

```
* (calc-sin 2)
0.9092974268256817d0
```

Lispový test 12 (`calc-sin`)**:** *Představení funkce pro výpočet sinu čísla s pevnou přesností a převodem*

vypadá to ale velmi odpudivě. Ve funkci je napevno zadrátována přesnost i výsledný typ.

Vytvořme si tedy proměnné, které budou určovat chování systému a sice pro přesnost a pro funkci na převod mezivýsledku na výsledek.

```
(defvar *calc-eps* -6)
(defvar *calc-conversion* (lambda (num) (float num)))

(defun global-sin (num)
  (funcall *calc-conversion*
    (tnum-to-num (tsin num) *calc-eps*)))
```

Lispový kód 40 (`global-sin`)**:** *Funkce pro výpočet sinu čísla a definice chování kalkulačky*

Toto už se chová jako normální kalkulačka. Navíc umožňuji uživateli změnit si přesnost a interpretaci výsledku, což normální kalkulačky neumí.

```
* (global-sin 3)
0.14112002
* (setq *calc-eps* 20)
20
* (setq *calc-conversion*
        (lambda (num) (coerce num 'long-float)))
#<FUNCTION (LAMBDA (NUM)) 52A4FC5B>
* (global-sin 4)
-0.7568024953079282d0
```

Lispový test 13 (`global-sin`)**:** *Představení kalkulačky sinu s nastavitelnou přesností*

Vznikla tedy jednoduchá kalkulačka na sinus čísla vracející číslo. To ale není málo, protože tímto návodem lze napsat celou nadstavbu knihovny `tnums` a přinést tak naprosto přesnou kalkulačku.

Já jsem se ale tímto směrem nevydal a mám k tomu několik důvodů:

- Mým cílem bylo vymyslet, odvodit a naprogramovat systém, který s vědomím uživatele líně a přesto velmi precizně vytváří abstraktní struktury, o kterých uživatel ví, že jejich následné vyčíslování je nutné provést a to funkcí `tnum-to-num`, kterou zavolá s kýženým tnumem a libovolnou přesností a tento proces může trvat i velmi dlouho;

- Mým cílem nebylo vytvořit univerzální kalkulačku pro co nejširší použití, při kterém ostatní kalkulačky využívající hardwarovou optimalizaci a jednotku FPU v rychlosti jistě zvítězí;

- S výše uvedeným návodem již může vzniknout uživatelská kalkulačka, třeba grafická, napsaná objektově orientovaně nebo jiným, třeba mně neznámým přístupem. Sám sebe vidím v pozici autora základu a jeho vylepšování. Na dobře postavené knihovně pak mohou běžet aplikace a jejich autoři budou těžit z dobrého jádra a ne z existujícího konkurenčního produktu;

- Nerad používám globální proměnné a vedlejší efekt. Proto se mi nelíbí představa, že uživatel musí při přizpůsobování výsledků „setqovat“, a tak se přikláním spíše k uzpůsobování přímo argumentu funkce `tnum-to-num`.

7.2 Optimalizace

Namísto rozšiřování funkčnosti – což jsme vesměs dělali až doteď – se nyní podívejme, jak by se dala zrychlit práce knihovny. Mimo triviálního „koupit lepší stroj“ mě napadají dva směry, kudy by se optimalizace mohla ubírat. Jak jsme totiž viděli v tabulce 5, vyčíslení tnumu nemusí být dílem okamžiku, ale může trvat i velmi dlouho. První je využití paralelizace, druhý vložení databáze.

7.2.1 Paralelizace

Dosud napsaný kód je sekvenční. Při vyčíslování tnumu se postupně rozkrývá jeho struktura a hloubkově se prochází. Důležité je to slovo *postupně*. Když vyčíslujeme `tnum-pi`, po vypočtení prvního členu se jde na další, ten se přičte a takto se to opakuje až po ukončení cyklu. Rychlejší by ale bylo, kdyby jeden proces byl zodpovědný pouze za sčítání řady a jednotlivé členy sčítané posloupnosti delegoval na výpočet ostatním procesům, aby celý výpočet nemusel běžet jen v jednom procesu, na jednom jádře.

Dalším vhodným místem použití paralelizace je funkce `tnum+`. Ta sčítá namapovaný seznam tnumů. Mapování je postupné aplikování funkce jedné proměnné na prvky seznamu a tvorba seznamu nového. Jde tedy o proces, kde jednotlivé výsledky mohou přijít v různých časech a hlavní vlákno by se staralo jen o vytvoření výsledného seznamu a výpočet jednotlivých členů by mohl běžet pro každý člen zvlášť, ve vlastním procesu.

Další vhodným místem je funkce `create-list-for-multiplication`, která také vrací seznam. Jde o seznam čísel, která byla nezávisle na ostatních vypočtena funkcí `tnum-to-num`. Tyto výpočty mohou opět vykonávat paralelní procesy a je to vhodné místo pro nasazení paralelizace.

U funkcí platí to samé jako u π a sice, že jednotlivé členy se mohou počítat zvlášť a až poté sčítat. Kvůli komutativitě sčítání mohou dokonce přijít v různé časy a pokud budou všechny, bude výsledek v pořádku.

7.2.2 Databáze

Druhým podstatným vylepšením by bylo vytvoření nástroje pro přístup k již vypočteným výsledkům. Moje představa je taková, že když uživatel zadá příkaz k vypočtení nějakého tnumu, systém se podívá, zda už nemá záznam, že by někdo takovýto tnum s takovouto přesností počítal. Pokud ano, vrátí tento výsledek. Pokud ne, podívá se na nejbližší výsledek s menší přesností a do očekávané přesnosti jej dopočte.

Nemá smysl, aby se vždy vracel ten nejlepší výsledek. Například pokud chci znát desáté místo desetinného rozvoje Ludolfova čísla, nemusí mi databáze vracet výsledek volání `(tnum-to-num (tnum-pi) 1000000)`. Pokud se v databázi žádný výsledek pro daný tnum nenajde, je vše vypočítáno načisto. Po ukončení výpočtu se do databáze přidá nový záznam, aby ostatní uživatelé mohli využívat vypočtených výsledků. Takhle by časem vznikla databáze s výsledky a namísto optimalizace přímo v kódu bychom se jen zeptali na již někdy vypočtený výsledek. Pokud v knihovně `tnums` není chyba, měl by mít tento výsledek věčnou platnost.

Otázek s tímto zlepšením je několik:

- Jak ukládat tnumy?
- Jak navazovat na výpočet, pokud záznam uložený jako výsledek není přesně ten, který hledáme?
- Jak zabránit, aby nám někdo nahrával chybné výsledky?
- Je morální užívat výsledků, za které zaplatil strojovým časem někdo jiný?

Struktura tnumu je abstraktní a není jen tak uložitelná. Kvůli této nadstavbě by se tedy musela opět změnit reprezentace a byly by to třeba textové řetězce představující λ funkce. Zní to sice trochu bizarně, ale mohlo by to vést k cíli. Také je zde velký prostor pro ušetření výpočtů. Například pokud již bude v databázi záznam čísla, jehož chceme opačné číslo, může se vrátit opačný výsledek.

Také může přijít příklad, kdy ve struktuře bude přičítána nula nebo bude násobeno jedničkou. Bez těchto operací je však tnum stejný jako původní a není tedy důvod vést různé záznamy pro stejné tnumy, ačkoli mají jinou strukturu. Vynořuje se zde princip, který jsem již popsal v první kapitole. Tedy, že číslo je dáno svým obsahem, nikoli symbolem. Symboly $2.999\ldots$ a 3 jsou jiné, ale čísla, která představují jsou stejná. To samé platí pro tnumy $\mathcal{T}^{x}(\varepsilon)$ a $\mathcal{T}^{x*1}(\varepsilon)$, podobně pro `x` a `(tnum+ x (num-to-tnum 0))`. Pokud totiž $1+1=2$, pak i $\mathcal{T}^{1+1}=\mathcal{T}^{2}$, neboli ekvivalence čísel se přenáší i na jejich tnumy.

Další ekvivalence tnumů už může být mnohem skrytější. Číslo x^3 lze napsat jako $x*x*x$. Číslo $x*3$ lze napsat jako $x+x+x$. To už nejsou tak triviální vztahy a přitom jejich souvislost může vést k mnohem rychlejším výpočtům. A že platí rovnost $4*\sum_{i\in\mathbb{N}}\frac{(-1)^i}{2i+1}=\pi=\sum_{i\in\mathbb{N}}\frac{1}{16^i}(\frac{4}{8i+1}-\frac{2}{8i+4}-\frac{1}{8i+5}-\frac{1}{8i+6})$? To už je velmi složité.

7.3 Peripetie

V této podkapitole bych rád popsal hlavní překážky, které se během programování knihovny vyjevily a krátce okomentoval možné chyby a nedostatky.

7.3.1 Ideové

Celkem jednoduchá otázka, ale asi nejhlubší se týkala samotné reprezentace tnumů. Jak je patrné, nakonec jsou tnumy vytvořeny jako funkce přesnosti a to velmi dobře koresponduje s jejich posláním. Tento nápad ale nepřišel jako první a vpravdě k němu vedla dlouhá cesta. Je to v pořadí třetí reprezentace tnumů. První byla reprezentace pomocí řad a druhá pomocí posloupností.

Nápad využít řady byl jako první z jednoduchého důvodu. Bylo to nejjednodušší. Když jsem se začal zabývat návrhem tnumů, bylo jasné, že musím umět vyčíslovat Ludolfovo číslo. Liebnitzův vzorec je vyjádřen ve formě řady. Také funkce čísel jsou řady – buď Taylorovy nebo Fourierovy. Eulerovo číslo nabízí vyjádření řadou i posloupností, vše ale do sebe zapadalo a bylo přímočaré vzít existující vzorce a jen je přetavit do kódu. Takhle vznikla verze knihovny, která při vyčíslování tnumu vracela částečné součty. Při vnořování řad do řad se ale mnohé výpočty prováděly vícekrát a působilo to těžkopádně. Další problém přišel když jsem se snažil dokázat korektnost. To bylo v době, kdy v podstatě celá knihovna byla naprogramována. Nebyl jsem schopen najít jasné souvislosti mezi přesností a počty sčítaných členů. Celková neohrabanost narůstala s každou novou funkcí. Při hledání optimalizace výpočtů funkcí a převrácených hodnot jsem si uvědomil, že lepší než používat řady bude rychlejší přejít na posloupnosti.

Posloupnosti byly výhodné například na poli rychlosti operací. Tam, kde se u řad sčítaly dlouhé posloupnosti nul, posloupnosti rovnou vracely správné výsledky v konstantní rychlosti. Stále ale byl problém s korektností převodu přesnosti na přirozené číslo aby se mohl zavolat příslušný člen. To tedy vedlo na nápad použít přesnost jako niternou součást tnumů a vyjadřovat je jako funkce právě přesnosti.

Idea byla, že oproti posloupnostem by se neměla snížit rychlost, ale velmi zestručnit kód a celkový návrh zjednodušit. Místo přepočítávání počtu členů, případně pořadí členu na přesnost se teď děje obráceně. Využívají se k tomu jen dva mechanismy a sice Taylorův zbytek a zbytek geometrické řady. Po předchozích změnách interpretace jsem před samotným programováním a poté dokazováním začal postupovat opačně a velmi se to nakonec otisklo do vyznění této práce. Než opět zahazovat velké části kódu nejprve na papír zkusím, jestli nepůjde o slepou uličku. Takto vznikla i notace tnumu x jako $\mathcal{T}^x$. V původní představě to ale mělo být $\mathcal{T}(x)$, ale pak se začaly množit závorky při volání s argumenty a musel jsem v zájmu čitelnosti přejít na notaci s horním indexem. V názvu tnum je schován ještě malý easter egg: věděl, jsem, že budu hodně pracovat s Taylorovými rozvoji a že se snad někdy povede vyjádřit vztah jako třeba v rovnici 77, kde se potkávají dva fonty velkých T a celkově tento vztah působí velmi elegantně. Protože v Lispu T znamená True, bylo po krátkém zvážení jasné, že se knihovna musí jmenovat TrueNumbers.

Příští implementace možná bude muset zohledňovat zaměření knihovny na uložitelnost do databáze výsledků, jak bylo konstatováno výše. Teď je ale podle mě brzy přemýšlet o budoucích reprezentacích, přijdou samy jako přišly doteď.

7.3.2 Matematické

Samotná práce je hodně matematicky zaměřena. Ne snad složitostí ale celkově svým vyzněním, že informatický problém jako je efektivní uložení a manipulace s teoreticky nekonečnými čísly není tak o inženýrském přístupu kódování paměti, ale o výsledcích základního kurzu matematické analýzy.

Vše je psáno s důrazem na čitelnost a preciznost. To znamená, že u velkého množství použité matematiky je důkaz nebo se jedná o fakt – v tomto případě je buď na úrovni středoškolských znalostí nebo je u něj odkaz do literatury. Důkazy některých faktů, o které jsem nechtěl čtenáře připravit ale nejsou nutné k pochopení matematiky tnumů naleznete v příloze A.

Dále jsem řešil praktické nalezení řad funkcí a jejich důkazy. Dobře se chovají exponenciála, kosinus a sinus. U ostatních řad jsem řešil, že jejich konvergence je omezena pouze na nějaký interval. Když se počítá normálně s čísly, není toto takový problém a střed řady se posune k hodnotě hledaného obrazu. Protože ale tnumy nejsou čísla a jejich vyčíslování probíhá líně, není možné před výpočtem upravit střed řady. Konvergence by se musela kontrolovat během vyčíslování a interaktivně měnit střed a to celé hloubkově a to už mi přišlo obtížné. Naštěstí jsem našel řadu pro logaritmus, která konverguje pro všechna kladná čísla a goniometrické funkce jsem pak přepsal podle středoškolských vzorečků pomocí již existujících. Mimochodem řady dalších goniometrických funkcí jsou nesmírně zajímavé a používají posloupnosti jako Bernoulliho nebo Eulerova čísla.

7.3.3 Implementační

Implementace je relativně jednoduchá v tom smyslu, že velmi přesně kopíruje matematický jazyk, a tak není moc prostoru pro chyby. Také kód vypadá – až na tři výjimky v podobě funkcí `tnum-pi`, `create-list-for-multiplication` a `tnum-to-string` – velmi spořádaně a v základní verzi (po začátek poslední části) ho tvoří jen 174 řádků. To je také důvod, proč jsem si dovolil celý zdrojový kód vpravit i do této práce, ačkoli mi to zpočátku přišlo jako moc násilné. Nakonec mi ale nepřijde, že by kódy překážely a působily jako výplň, nýbrž podtrhují použitelnost odvozené matematiky a díky tomu přibližují abstraktní svět matematiky čísel aplikovanějšímu čtenáři v osobě informatika.

7.3.4 Mezery

Při psaní knihovny `tnums` i tohoto textu bylo mým cílem přinést produkt co nejvíce neprůstřelný a pokud se někde vyskytuje mezera, aby byla přesně lokalizovatelná.

7.3.4.1 Matematické Uvědomuji si, že můj jazyk a především zápis jsou na některých místech chaotické. Snad ale na druhé přečtení plně pochopíte nosnou myšlenku veškeré matematiky a proč byla přijímána.

Mimo kostrbatého vyjádření o žádných chybách nevím a pokud tam nějaké jsou, jsou nevědomé. Ikdyž jsem se snažil vše psát co nejvíce precizně, samozřejmě se mi tam něco mohlo vloudit a takováto chyba mě mrzí. Mým cílem v této diskuzi jistě není nějaké chyby zamlčovat. Naopak budu rád, pokud čtenář nalezne chybu nebo lépe i její řešení a kontaktuje mě na ondrej.slavik01@upol.cz.

7.3.4.2 Implementační Ohledně kódu celkem věřím, že je napsán kvalitně. Důkazem může být i že jsem se ho nebál otisknout do této práce a nepřidávám pouze odkaz na repozitář. Jeho vysoká stručnost a čitelnost je vykoupena spoustou práce na matematickém pozadí. V kódu by doufám žádná chyba právě pro jeho stručnost neměla být. Pokud se tak ale stalo, beru to jako menší chybu, než kdybych ji měl v matematice.

Pokud je chyba v implementaci, i když matematicky to bylo správně, pak je to hloupá chyba, protože kód je velmi podobný odvozeným tvrzením, v podstatě se jedná jen o syntaktický převod. Tam, kde to bylo důležité a byl na to prostor jsem si dovolil nějaké optimalizace, takže tam se kód od matematiky odchýlil, ale všechny funkce jsou pouze na několik řádků, a tak doufám, že jsem tam žádné chyby neudělal. I tak se samozřejmě mohlo stát.

7.3.4.3 Funkcionalita Co se týče funkčnosti, je velmi základní. V dalších verzích by měly přibýt cyklometrické funkce a některé konstanty. Možná ještě funkce více proměnných a tetrace. V tuto chvíli mi ale funkčnost přijde dostatečná.

7.3.4.4 Rychlost Co se týče rychlosti, v minulé podkapitole jsem navrhl nějaké směry, kudy by se mohla ubírat optimalizace. Přímo v existujícím kódu bez přidání paralelizace nebo databáze myslím nelze o mnoho rychlost výpočtů zlepšit. Systém jsem navrhl tak, aby byl syntakticky jednoduchý a přitom plně funkční. Možná nějaké dílčí zapamatovávání proměnných by ho mohlo zrychlit, ale bez fundamentálního přepracování jeho plynulost příliš nezlepším. Knihovna je tedy relativně pomalá, ale její poslání je přesnost a nikoli rychlost.

7.4 Svébytnost

7.4.1 Kontext knihovny

Knihovna `tnums` je unikátní ve své kompatibilitě se svým jazykem. Sice vytváří abstraktní struktury, výsledkem vyčíslení je ale vždy číslo. Narozdíl od ostatních představených knihoven tedy může zajišťovat přesné výpočty v části systému kritické na přesnost a s těmito výsledky lze přímo pracovat a nemusí se kopírovat nebo převádět na zlomky. Knihovna toho neumí tolik jako třeba `mpmath`, naopak umí více než `computable reals`. Při pohledu zvnějšku bere čísla a tři konstanty a vrací opět čísla. Uživatel je tedy odstíněn od implementace (jako funkce přesnosti) a v této bariéře je velká síla. Ostatní knihovny, které vracejí vlastní typy pak na mě působí, že jsou samy pro sebe a dají se použít jen jako kalkulačky.

7.4.2 Kontext práce

Když jsem pročítal bakalářské práce absolventů, přišla mi škoda, že si nemohu prohlédnout jejich kódy. Protože se odevzdávají na CD a do STAGu se nenahrávají. Pro čtenáře jsem tedy přidal odkaz ⟨`https://github.com/slavon00/tnums`⟩, na kterém je aktuální verze knihovny k dostání.

7.4.3 K zapamatování

Kdyby si čtenář měl odnést jen tři poučky z přečtení této práce, ať je to

1. že symbol je něco jiného než to, co představuje,
2. že věci nad racionálními čísly jsou diametrálně jednodušší než věci nad rekurzivními čísly
3. a že když se pochopí obě tyto věci, dá se i celkem přehledně s rekurzivními čísly pracovat.

7.4.4 Adakvátnost

Nyní v rychlosti zopakuji 6 podmínek, které jsem stanovil, že musí abstraktní struktury splňovat a připomenu, že je tnumy dodržují.

- Vyčíslování tnumů zajišťuje funkce `tnum-to-num`,
- přesnost zajišťuje samotná podstata tnumů jako funkcí přesnosti,
- matematické operace tnumy podporují,
- matematické funkce tnumy podporují,
- tnumy se dají vracet jako výsledky funkcí,
- tnumy se dají použít jako argumenty funkcí.

Tnumy tak splňují podmínky na abstraktní datové struktury. Tnumy jsou adekvátní pro realizaci přesných výpočtů s reálnými čísly.

Závěr

Popsal jsem, jak jsou vytvořena přirozená čísla pomocí teorie množin, také jak na tomto základě vznikají další číselné obory. Dále jsem popsal, jak se s **čísly** pracuje v paměti v počítače.

Také bylo zmíněno, že číselná osa je tvořena **reálnými čísly** a pokud použijeme více os, dostáváme strukturovaná čísla. Zamysleli jsme se, jestli jsou všechna reálná čísla rekurzivní a bohužel jsme dostali negativní odpověď.

Představil jsem, jak vypadají **výpočty s reálnými čísly**. Kromě matematických operací to byly matematické funkce. Zjistili jsme, že všechny tyto výpočty, včetně samotných reálných konstant lze reprezentovat jako funkce.

V textu jsme se věnovali i produktu celého tohoto snažení a sice programování Lispovské knihovny `tnums` implementující **přesné výpočty s reálnými čísly**. Také jsem přinesl několik příkladů, jak uživatelsky funkcionalitu rozšiřovat.

Conclusions

I have described how natural numbers are created using set theory, as well as on this basis, other numerical fields are created. I also described how to deal with **numbers** in memory of a computer.

It was also mentioned that the numerical axis is made up of **real numbers** and if we use more axes, we get structured numbers. We wondered if all real numbers are computable and unfortunately we got a negative answer.

I introduced what **real numbers' computation** looks like. In addition to mathematical operations there were mathematical functions. We found that all these calculations, including the real constants themselves, can be represented as functions.

In the text, we also focused on the product of all this effort, namely programming Lisp `tnums` library implementing **precise real numbers' computation**.

I also brought some examples of how user can extend functionality.

Seznam literatury

1. PŘISPĚVATELÉ WIKIPEDIE. *Číslo* [online]. Wikipedie: Otevřená encyklopedie, 2021 [cit. 2021-02-21]. Dostupné z: ⟨https://cs.wikipedia.org/w/index.php?title=%5C%C4%5C%8C%5C%C3%5C%ADslo&oldid=19341576⟩.
2. CARROLL, Lewis. *Alenka v kraji divů a za zrcadlem.* 1. vyd. Praha: Městská knihovna v Praze, 2018. ISBN 978-80-7602-231-7.
3. ROJAS, Raul. *A Tutorial Introduction to the Lambda Calculus* [online]. Berlin: Freie Universität, 2015 [cit. 2021-04-27]. Dostupné z: ⟨http://www.inf.fu-berlin.de/inst/ag-ki/rojas_home/documents/tutorials/lambda.pdf⟩.
4. BALCAR, Bohuslav; ŠTĚPÁNEK, Petr. *Teorie množin.* 2., opravené a rozšířené. Praha: Academia, nakladatelství AV ČR, 2001. ISBN 80-200-0470-X.
5. SLOANE, Neil James Alexander. *A001057: Canonical enumeration of integers: interleaved positive and negative integers with zero prepended* [online] [cit. 2021-04-27]. Dostupné z: ⟨https://oeis.org/A001057⟩.
6. SPIVAK, Michael. 3. vyd. Houston: Publish or Perish, 2006. ISBN 9780521867443.
7. MIKULČÁK, Jiří; CHARVÁT, Jura; MACHÁČEK, Martin; ZEMÁNEK, František. *Matematické, fyzikální a chemické tabulky a vzorce pro střední školy.* 1. vyd. Havlíčkův brod: Prometheus, 2012. ISBN 9788071962649.
8. HALAŠ, Radomír. *Teorie čísel.* 2., upravené. Olomouc: Univerzita Palackého, 2014. ISBN 978-80-244-4068-2.
9. PŘISPĚVATELÉ WIKIPEDIE. *Computable number* [online]. Wikipedie: Otevřená encyklopedie, 2020 [cit. 2021-03-03]. Dostupné z: ⟨https://en.wikipedia.org/w/index.php?title=Computable_number&oldid=997421913⟩.
10. BAEZ, John C. *Division Algebras and Quantum Theory* [online]. 2011 [cit. 2021-04-27]. Dostupné z: ⟨https://arxiv.org/pdf/1101.5690.pdf⟩.
11. RUBSTOV, Constantin A.; ROMERIO, Giovanni F. *Ackerman's Function and New Arithmetical Operations* [online]. 2004 [cit. 2021-04-27]. Dostupné z: ⟨http://www.rotarysaluzzo.it/Z_Vecchio_Sito/filePDF/Iperoperazioni%5C%20(1).pdf⟩.
12. PELANTOVÁ, Edita; VONDRÁČKOVÁ, Jana. *Matematická analýza I.* Praha: České vysoké učení technické, 2004. Dostupné také z: ⟨http://km.fjfi.cvut.cz/ma/data/uploads/skripta-matematicka-analyza-1.pdf⟩.
13. HORT, Daniel; RACHŮNEK, Jiří. *Algebra 1.* Olomouc: Univerzita Palackého, 2003.
14. DOŠLÁ, Zuzana; KUBEN, Jaromír. *Diferenciální počet funkcí jedné proměnné.* 1. vyd. Brno: Masarykova univerzita, 2004. ISBN 80-210-3121-2.

15. DOŠLÁ, Zuzana; NOVÁK, Vítězslav. *Nekonečné řady*. 1. vyd. Brno: Masarykova univerzita, 2002. ISBN 80-210-1949-2.
16. KEPRT, Aleš. *Operační systémy*. Olomouc: Univerzita Palackého, 2007. Dostupné také z: ⟨https://phoenix.inf.upol.cz/esf/ucebni/OpSys.pdf⟩.
17. KNUTH, Donald Ervin. *Umění programování. 2. díl: Seminumerické algoritmy*. 1. vyd. Brno: Computer Press, 2010. ISBN 978-80-251-2898-5.
18. PŘISPĚVATELÉ WIKIPEDIE. *Single-precision floating-point format* [online]. Wikipedie: Otevřená encyklopedie, 2021 [cit. 2021-02-20]. Dostupné z: ⟨https://en.wikipedia.org/w/index.php?title=Single-precision_floating-point_format&oldid=1005180810⟩.
19. STANNERED. *Binary representation of a 32-bit floating-point number* [online]. Wikimedia Commons, 2008 [cit. 2021-03-29]. Dostupné z: ⟨https://upload.wikimedia.org/wikipedia/commons/d/d2/Float_example.svg⟩. Dostupný pod licencí CC BY-SA 3.0.
20. PŘISPĚVATELÉ WIKIPEDIE. *Long double* [online]. Wikipedie: Otevřená encyklopedie, 2021 [cit. 2021-03-18]. Dostupné z: ⟨https://en.wikipedia.org/w/index.php?title=Long_double&oldid=1003717804⟩.
21. JOHANSSON, Fredrik a kol. *mpmath: a Python library for arbitrary-precision floating-point arithmetic* [online]. 2013 [cit. 2021-03-29]. Dostupné z: ⟨http://mpmath.org⟩.
22. DAUTELLE, Jean-Marie. *jscience: Tools & Libraries for the Advancement of Sciences* [online]. GitHub.io, 2017 [cit. 2021-03-29]. Dostupné z: ⟨https://github.com/javolution/jscience⟩.
23. GRANLUND, Torbjörn a kol. *GNU MP: The GNU Multiple Precision Arithmetic Library* [online]. 2020 [cit. 2021-03-29]. Dostupné z: ⟨https://gmplib.org/gmp-man-6.2.1.pdf⟩.
24. *GNU MPFR: The Multiple Precision Floating-Point Reliable Library* [online]. 2020 [cit. 2021-03-29]. Dostupné z: ⟨https://www.mpfr.org/mpfr-current/mpfr.pdf⟩.
25. GRANLUND, Torbjörn; HART, William; GLADMAN, Brian a kol. *MPIR: The Multiple Precision Integers and Rationals Library* [online]. 2017 [cit. 2021-03-29]. Dostupné z: ⟨http://mpir.org/mpir-3.0.0.pdf⟩.
26. HAIBLE, Bruno; KRECKEL, Richard B. *CLN: a Class Library for Numbers* [online]. 2019 [cit. 2021-03-29]. Dostupné z: ⟨https://www.ginac.de/CLN/cln.pdf⟩.
27. STOLL, Michael. *computable-reals: Arbitrary precision, automatic re-computing real numbers in Common Lisp* [online]. GitHub.io, 2021 [cit. 2021-03-29]. Dostupné z: ⟨https://github.com/stylewarning/computable-reals⟩.

28. THE COMMON LISP COOKBOOK PROJECT. *The Common Lisp Cookbook: Numbers* [online]. GotHub.io, 2020 [cit. 2021-03-29]. Dostupné z: ⟨https://lispcookbook.github.io/cl-cookbook/numbers.html⟩.
29. SEIBEL, Peter. *Practical Common Lisp*. 1. vyd. New York: Apress, 2005. ISBN 1590592395.
30. RICHESON, David. *Circular Reasoning: Who First Proved That C/d Is a Constant?* [Online]. Dickinson College, 2010 [cit. 2021-04-27]. Dostupné z: ⟨https://arxiv.org/pdf/1303.0904.pdf⟩.
31. JANSSON, Madeleine. *Approximation of* π [online]. Lund University, 2019 [cit. 2021-04-27]. Dostupné z: ⟨https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=8983341&fileOId=8983342⟩.
32. BAILEY, David; BORWEIN, Peter; PLOUFFE, Simon. *On the Rapid Computation of Various Polylogarithmic Constants* [online]. 1997 [cit. 2021-04-27]. Dostupné z: ⟨https://www.ams.org/journals/mcom/1997-66-218/S0025-5718-97-00856-9/S0025-5718-97-00856-9.pdf⟩.
33. RICE, Henry Gordon. Recursive Real Numbers. *Proceedings of the American Mathematical Society* [online]. 1954, roč. 5, č. 5, s. 784–791 [cit. 2021-07-22]. Dostupné z: ⟨https://www.ams.org/journals/proc/1954-005-05/S0002-9939-1954-0063328-5/home.html⟩.
34. HABALA, Petr. *Taylorův polynom* [online]. Praha: České vysoké učení technické [cit. 2021-03-29]. Dostupné z: ⟨https://math.fel.cvut.cz/mt/txtc/4/txc3ca4e.htm⟩.
35. APOSTOL, Tom M. *Calculus: Volume I*. 2. vyd. USA: John Wiley & Sons, 1967. ISBN 0-471-00005-1.
36. ABRAMOWITZ, Milton; STEGUN, Irene A. *Handbook of Mathematical Functions: With Formulas, Graphs and Mathematical Tables* [online]. 10, with corrections. Washington: U.S. Government Printing Office, 1964 [cit. 2021-04-28]. Dostupné z: ⟨http://www.math.ubc.ca/~cbm/aands/abramowitz_and_stegun.pdf⟩.
37. SPIRA, Michel. *On the Golden Ratio* [online]. Universidade Federal de Minas Gerais [cit. 2021-04-27]. Dostupné z: ⟨https://www.mathunion.org/fileadmin/ICMI/Conferences/ICME/ICME12/www.icme12.org/upload/submission/1948_F.pdf⟩.

A Některé důkazy

Zde uvedu některé důkazy, které jsou mimo těžiště práce a pro pochopení práce s tnumy nejsou zapotřebí. Nějaké zájemce by ale mohlo zajímat, jak se některé důkazy vedou a ikdyž se nehodí je napsat přímo v textu práce, zde mohou být.

A.1 Fakt 24 – O částečném součtu geometrické řady

Důkaz

Částečný součet je z definice $s_n^a = a + aq + \ldots + aq^n$. Užitím vztahu

$$(1-q)(1+q+q^2+\ldots+q^n) = (1-q)^{n+1} \tag{110}$$

dostaneme

$$s_n^a = a(1+q+q^2+\ldots+q^n) = a\frac{1-q^{n+1}}{1-q}. \tag{111}$$

□

A.2 Fakt 25 – O geometrické řadě

Důkaz

Po užití limity na obě strany ze lemmatu 24 dostáváme

$$\sum_{i\in\mathbb{N}} aq^i = \lim_{n\to\infty} s_n^a = \lim_{n\to\infty} a\frac{1-q^{n+1}}{1-q} = \frac{a}{1-q}. \tag{112}$$

□

A.3 Fakt 26 – O zbytku geometrické řady

Důkaz

$$R_n = \sum_{i\in\mathbb{N}} aq^i - s_n^a = \frac{a}{1-q} - a\frac{1-q^{n+1}}{1-q} = \frac{a-a+aq^{n+1}}{1-q} = \frac{aq^{n+1}}{1-q} \tag{113}$$

□

A.4 Fakt 50 – O exponenciále jako Maclaurinově řadě

Důkaz

$$\begin{aligned} e^x = \sum_{i\in\mathbb{N}} \frac{exp^{(i)}(0)}{i!}x^i = \left[\tfrac{d}{dx}exp(x) = exp(x)\right] = \frac{exp(0)}{1} + \frac{exp(0)}{1}x + \\ + \frac{exp(0)}{2}x^2 + \ldots = \left[exp(0) = 1\right] = \frac{1}{1} + \frac{x}{1} + \frac{x^2}{2} + \ldots = \sum_{i\in\mathbb{N}} \frac{x^i}{i!} \end{aligned} \tag{114}$$

□

A.5 Fakt 58 – O sinu jako Maclaurinově řadě

Důkaz

$$
\begin{aligned}
sin(x) = \sum_{i \in \mathbb{N}} \frac{sin^{(i)}(0)}{i!} x^i = \begin{bmatrix} \frac{d}{dx} sin(x) = cos(x) \\ \frac{d}{dx} cos(x) = -sin(x) \end{bmatrix} = \\
= \frac{sin(0)}{1} + \frac{cos(0)}{1} x - \frac{sin(0)}{2} x^2 - \frac{cos(0)}{6} x^3 + \frac{sin(0)}{24} x^4 + \frac{cos(0)}{120} x^5 \ldots = \\
= \begin{bmatrix} sin(0) = 0 \\ cos(0) = 1 \end{bmatrix} = \frac{x}{1} - \frac{x^3}{6} + \frac{x^5}{120} + \ldots = \sum_{i \in \mathbb{N}} (-1)^i \frac{x^{2i+1}}{(2i+1)!}
\end{aligned}
\tag{115}
$$

□

A.6 Fakt 61 – O kosinu jako Maclaurinově řadě

Důkaz

$$
\begin{aligned}
cos(x) = \sum_{i \in \mathbb{N}} \frac{cos^{(i)}(0)}{i!} x^i = \begin{bmatrix} \frac{d}{dx} cos(x) = -sin(x) \\ \frac{d}{dx} sin(x) = cos(x) \end{bmatrix} = \\
= \frac{cos(0)}{1} - \frac{sin(0)}{1} x - \frac{cos(0)}{2} x^2 + \frac{sin(0)}{6} x^3 + \frac{cos(0)}{24} x^4 \ldots = \\
= \begin{bmatrix} cos(0) = 1 \\ sin(0) = 0 \end{bmatrix} = \frac{1}{1} - \frac{x^2}{2} + \frac{x^4}{24} + \ldots = \sum_{i \in \mathbb{N}} (-1)^i \frac{x^{2i}}{(2i)!}
\end{aligned}
\tag{116}
$$

□

B Obsah přiloženého CD/DVD

Na samotném konci textu práce je uveden stručný popis obsahu přiloženého CD/DVD, tj. jeho závazné adresářové struktury, důležitých souborů apod.

`doc/`
: Adresář se soubory

  - `OndrejSlavikBP.pdf` – tento text ve formátu PDF a
  - `bak/` – adresář se všemi soubory pro vysázení tohoto textu, stačí dvakrát přeložit `PDFLaTex`em.

`load.lisp`
: Přeložením tohoto souboru ve vašem oblíbeném interpretu/kompilátoru Lispu získáte plnou funkcionalitu knihovny `tnums`, kterou jsme právě doprogramovali. Načítá soubory `src/tnums.lisp` a `src/user-functions.lisp`.

`src/`
: Adresář se soubory

  - `tnums.lisp` – základ knihovny z části 2 této práce,
  - `user-function.lisp` – rozšíření knihovny o vědomě uživatelské funkce ze šesté kapitoly a
  - `tests.lisp` – zakomentované výrazy, které zde byly popsány jako Lispový test i s výsledky, na které se vyhodnotí.

`README.md`
: Soubor představující knihovnu `tnums` a obsahující i několik příkladů výpočtů, které podporuje. Součástí je i kapitola o načtení knihovny evaluací souboru `load.lisp`, nejedná se tedy o instalaci v pravém slova smyslu.

`install/`
: Adresář se soubory

  - `sbcl-2.1.6-source.tar.bz2` – intalátor SBCL, konzolového kompilátoru ANSI Common Lispu,
  - `code-1.57.1-1623937013-amd64.deb` – instalátor VS code, rozšiřitelného textového editoru a
  - `2gua.rainbow-brackets-0.0.6.vsix` – instalátor rozšíření Rainbow Brackets pro přehledné obarvování závorek.

`LISENCE`
: Licenší soubor – knihovna je publikována pod GNU GPLv3.