VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

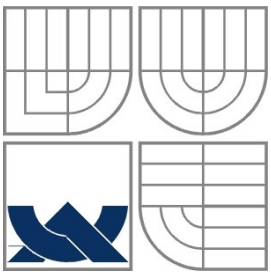
ZOBRAZOVAČ GRAFŮ A VYHODNOCOVAČ MATEMATICKÝCH VÝRAZŮ

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

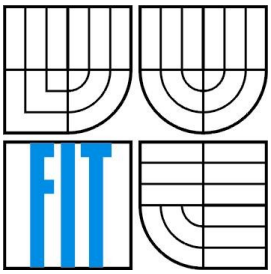
AUTOR PRÁCE
AUTHOR

IVO SKALICKÝ

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ZOBRAZOVAČ GRAFŮ A VYHODNOCOVAČ MATEMATICKÝCH VÝRAZŮ

GRAPHDRAWER AND MATHEMATHIC EQUATION EVALUATOR OF EXPRESSIONS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

IVO SKALICKÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ROMAN LUKÁŠ, Ph.D.

BRNO 2007

Abstrakt

Práce se zabývá využitím precedenční syntaktické analýzy pro převod matematického výrazu na dynamický objektový model za účelem vyhodnocování a dalšího zpracovávání výrazů. Cílem bylo vytvořit program, který spojuje funkci vědeckého kalkulátoru a zobrazovače grafů. Program umí na základě textového uživatelského vstupu vyčíslovat matematické výrazy v oboru reálných čísel, počítat lomené výrazy, numericky integrovat a analyticky derivovat výraz podle zadané proměnné. Vedle toho umožňuje také vykreslení průběhu libovolné explicitně, implicitně nebo parametricky zadané křivky do 2D grafu. Grafický výstup je pak možno exportovat s volitelným rozlišením do několika základních rastrových i vektorových formátů. Implementačním jazykem byla z důvodu přenositelnosti zvolena Java.

Klíčová slova

Precedenční syntaktická analýza, bezkontextová gramatika, kalkulačka, vyhodnocování výrazů, explicitní křivka, implicitní křivka, parametrická křivka, analytická derivace, graf funkce

Abstract

This work deals with utilization of precedence syntax analysis for conversion of mathematical expression into dynamic object model in order of evaluate and go on processing expressions. Objective was to create a program, which connects function of scientific calculator and graph drawer. Program can evaluate mathematical expressions in scope of real numbers, calculate fractional expressions, numerically integrate and analytically derive in accordance with a given parameter. Besides it provides function for depiction of any given curve explicitly, implicitly or parametricly into 2D graph. Graphic output can be exported in selectable resolution into several basic raster and vector formats. On account of portability Java was chosen as a language of implementation.

Keywords

Precedence syntax analysis, context-free grammar, calculator, expression evaluation, explicit curve, implicit curve, parametric curve, analytic derivation, function graph

Citace

Ivo Skalický: Zobrazovač grafů a vyhodnocovač matematických výrazů, bakalářská práce, Brno, FIT VUT v Brně, 2007

Zobrazovač grafů a vyhodnocovač matematických výrazů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Romana Lukáše Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Ivo Skalický
14.5.2007

Poděkování

Poděkování za odborné vedení, rady a konzultace patří vedoucímu práce Ing. Romanu Lukášovi Ph.D.
Za testování, návrhy, připomínky a oponentské posudky na mou předchozí práci Středoškolské odborné činnosti, ze které tato bakalářská práce vychází, si poděkování zaslouží: Jiří Tužil, David Sehnal, Ing. Jan Plíšek, Mgr. Alena Kvasničková, Jaroslav Mrázek, Petr Dittrich, Jaroslav Procházka, Jan Bydžovský, Dr. Miroslav Kureš, Mgr. Milada Klímová, Ing. Robert Kohout, Jan Pavlík, RNDr. Pavel Chmelař , Tomáš Macek.
Za jazykovou korekturu děkuji Mgr. Ivaně Skalické.

Obsah

Obsah.....1
1 Úvod.....3
2 Základní definice.....4
2.1 Základní pojmy.....4
2.1.1 Definice abecedy.....4
2.1.2 Definice řetězce.....4
2.1.3 Definice jazyka.....4
2.1.4 Definice regulárního výrazu.....4
2.1.5 Definice konečného automatu.....5
2.1.6 Definice bezkontextové gramatiky.....5
2.1.7 Definice nejpravější derivace.....5
2.2 Fáze překladu a interpretace.....5
2.2.1 Lexikální analýza.....5
2.2.2 Syntaktická analýza.....5
2.2.3 Sémantická analýza.....6
2.2.4 Interpretace.....6
3 Zpracování výrazů.....7
3.1 Navržený jazyk.....7
3.1.1 Číselné literály.....7
3.1.2 Identifikátory.....8
3.1.3 Vestavěné funkce a operátory.....8
3.2 Lexikální analýza.....8
3.2.1 Tokeny.....9
3.3 Syntaktická a sémantická analýza.....10
3.3.1 Precedenční syntaktická analýza.....10
3.3.2 Tabulka symbolů.....13
3.4 Objektová reprezentace výrazu.....13
3.4.1 Objektový model.....13
3.5 Vyhodnocování výrazu.....14
3.5.1 Nastavení vyhodnocování.....15
3.5.2 Vyhodnocování lomených výrazů.....16
3.5.3 Analytická derivace.....17
3.5.4 Optimalizace.....18
3.5.5 Chybová hlášení.....18
3.6 Znovupoužitelnost vyhodnocovače.....19

4 Vykreslování grafů..... 20
4.1 Objektový model..... 20
4.2 Explicitní křivky..... 22
4.3 Parametrické křivky..... 24
4.4 Implicitní křivky..... 24
4.5 Znovupoužitelnost vykreslovače grafů..... 27
5 Možnosti dalšího vývoje..... 29
6 Závěr..... 30
Literatura..... 31
Seznam příloh..... 32
Seznam obrázků..... 33
Seznam tabulek..... 34
Příloha A – Uživatelský manuál..... 35

1 Úvod

Tato bakalářská práce navazuje na mou práci Středoškolské odborné činnosti ročníku 2002/2003. Protože práce prošla soutěží úspěšně až do celostátního kola, získal jsem díky kritice a přípomínkám porotců v jednotlivých kolech dobrou představu o tom, co by měl program pro vyhodnocování výrazů a kreslení grafů umět a splňovat. Původní program je od roku 2003 volně přístupný ke stažení na internetu a dosáhl několika desetitísíců stažení, což mělo za následek také řadu reakcí uživatelů. Všechny tyto poznatky a zkušenosti jsem se rozhodl využít v této práci a přetvořit program zcela znovu v jiném programovacím jazyce při zhodnocení znalostí o lexikální a syntaktické analýze, které jsem nabyl v předmětech Formální jazyky a překladače a Výstavba překladačů v rámci výuky na VUT v Brně.

Hlavním cílem bylo vytvořit aplikaci uživatelsky přehlednou a snadno ovladatelnou. Pro zadávání výrazů bylo proto třeba navrhnout jazyk s ohledem na zvyklosti uživatelů z obvyklých matematických programů jako je například Microsoft® Excel, Matlab, Maple, ale i z předchozích verzí programu SMath.

Řešení využívá ke zpracování textové interpretace matematického výrazu precedenční syntaktické analýzy ve směru zdola nahoru. Protože navržený jazyk obsahuje i funkce s neurčitým počtem parametrů či vnořené definice pole, dokazuje implementace vhodnost použití precedenční analýzy i pro složitější matematické výrazy. V průběhu syntaktické analýzy jsou vytvářeny objekty reprezentující jednotlivé části výrazu a tyto objekty jsou pak na sebe vzájemně dynamicky vázány. Takovýto objektový strom pak umožňuje výraz nejen klasicky vyhodnotit, ale i provádět nad ním další operace jako například provedení analytické derivace či zjednodušení výrazu.

Technická zpráva je vedle úvodu a závěru dělena do čtyř věcných kapitol – kapitoly 2 - 5.

Druhá kapitola technické zprávy definuje některé základní pojmy, které jsou dále v textu zprávy používány a jsou nutné pro pochopení samotné činnosti programu.

Ve třetí kapitole je probráno zpracování výrazu po jednotlivých fázích od lexikální přes syntaktickou analýzu až po samotné vyhodnocení a zpracování výrazu jeho objektovou reprezentací. Na začátku kapitoly je také definován navržený jazyk a jsou uvedeny platné konstrukce jazyka.

Čtvrtá kapitola se zabývá programovou realizací zobrazení grafů funkcí. Vysvětluje metody vykreslování jednotlivých typů funkcí zadaných křivek, ale i datovou strukturu tyto křivky uchovávající.

Pátá kapitola popisuje možnosti dalšího vývoje aplikace.

2 Základní definice

Než se dostaneme k vysvětlení činnosti programu, je třeba nejprve definovat některé základní pojmy v technické zprávě použité. Dále je také nutné definovat jednotlivé fáze překladu a interpretace matematického výrazu.

2.1 Základní pojmy

Definice ve všech těchto podkapitolách jsou převzaty ze zdroje [1].

2.1.1 Definice abecedy

Abeceda je konečná, neprázdná množina elementů, které nazýváme symboly.

2.1.2 Definice řetězce

Nechť Σ je abeceda.

1. ε je řetězec nad abecedou Σ
2. pokud x je řetězec nad Σ a $a \in \Sigma$, potom xa je řetězec nad abecedou Σ.

2.1.3 Definice jazyka

Nechť Σ^* značí množinu všech řetězců nad Σ. Každá podmnožina $L \subseteq \Sigma^*$ je jazyk nad Σ.

2.1.4 Definice regulárního výrazu

Nechť Σ je abeceda. Regulární výrazy nad abecedou Σ a jazyky, které značí, jsou definovány následovně:

- ∅ je regulární výraz značící prázdnou množinu (prázdný jazyk)
- ε je regulární výraz značící jazyk {ε}
- a, kde $a \in \Sigma$, je regulární jazyk značící jazyk $\{a\}$
- Nechť r a s jsou regulární výrazy značící po řadě jazyky L_r a L_s, potom:
  - $(r.s)$ je regulární výraz značící jazyk $L = L_r L_s$
  - $(r + s)$ je regulární výraz značící jazyk $L = L_r \cup L_s$
  - (r^*) je regulární výraz značící jazyk $L = L_r^*$

Informace v tomto textu definované regulárními výrazy jsou zapsány podle Unixové normy POSIX. Tečka pro konkatenaci se tedy implicitně vypouští, místo plus pro sjednocení se píše symbol svislítko „|“ a množiny přípustných znaků jsou udávány v hranatých závorkách.

2.1.5 Definice konečného automatu

Konečný automat je pětice $M = (Q, \Sigma, R, s, F)$, kde:

- Q je konečná množina stavů
- Σ je vstupní abeceda
- R je konečná množina pravidel tvaru: pa $\rightarrow$ q, kde $p, q \in Q, a \in \Sigma \cup \{\varepsilon\}$
- s $\in$ Q je počáteční stav
- F $\subseteq$ Q je množina koncových stavů

2.1.6 Definice bezkontextové gramatiky

Bezkontextová gramatika je čtveřice $G = (N, T, P, S)$, kde:

- N je abeceda neterminálů
- T je abeceda terminálů, přičemž $N \cap T = \varnothing$
- P je konečná množina pravidel tvaru $A \rightarrow x$, kde $A \in N, x \in (N \cup T)^*$
- $S \in N$ je počáteční neterminál

2.1.7 Definice nejpravější derivace

Nechť $G = (N, T, P, S)$ je bezkontextová gramatika, $u \in (N \cup T)^*$ a $v \in T^*$ a $p = A \rightarrow x \in P$ je pravidlo. Pak říkáme, že uAv přímo derivuje v nejpravější derivaci uxv podle pravidla p, a zapisujeme $uAv \Rightarrow_{rm} uxv[p]$ nebo také zkráceně $uAv \Rightarrow_{rm} uxv$.

Během nejpravější derivace je přepsán nejpravější nonterminál.

2.2 Fáze překladu a interpretace

2.2.1 Lexikální analýza

Lexikální analýza je jedinou fází překladu, která má přímý přístup k uživatelem zadanému zdrojovému kódu. Jejím úkolem je na žádost vyšších vrstev překladu rozpoznat v textu *lexém* (logicky oddělená lexikální jednotka) a ten pak reprezentovat datovou instancí obecně zvanou *token*.

Programová část, která se stará o lexikální analýzu, se obvykle nazývá *scanner*.

2.2.2 Syntaktická analýza

Syntaktická analýza na základě zadaných pravidel řídí činnost *scanneru*. Podle *tokenů* od *scanneru* získaných a pevně daných pravidel pak sestavuje (nebo alespoň kontroluje, zda je možno sestavit) derivační strom. Podle toho může rozhodnout, zda vstupní řetězec tokenů reprezentuje syntakticky správě zapsaný program.

Programová část provádějící syntaktickou analýzu se obvykle nazývá *parser*.

2.2.3 Sémantická analýza

Sémantický analyzátor kontroluje sémantické aspekty programu, především kontroluje datové typy a provádí potřebné datové konverze.

V případě synstaxí řízeného překladu se o sémantickou analýzu stará již přímo *parser*, který v průběhu své činnosti provádí i sémantické akce a generuje *abstraktní syntaktický strom*.

2.2.4 Interpretace

Interpretace je samotné vykonání instrukcí získaných v přechozích fázích překladu. Vykonáván je buď vnitřní kód (např. tříadresný kód), nebo je možné vykonávat i přímo *abstraktní syntaktický strom*.

V průběhu interpretace je také třeba kontrolovat běhové chyby, jakými může být například dělení nulou nebo použití neinicializované proměnné.

3 Zpracování výrazů

Tato kapitola pojednává o samotném vyhodnocování výrazů. Naleznete zde také popis implementace jednolivých fází zpracování a vyhodnocení výrazu. Než se však dostaneme k těmto informacím, je nejprve třeba vymyslet a popsat jazyk vhodný k zadávání výrazů.

3.1 Navržený jazyk

Protože jedním z hlavních cílů bylo vytvoření uživatelsky přívětivé aplikace, byla volba a definice jazyka jedním z nejdůležitějších úkonů, kterým bylo třeba věnovat pozornost. Jazyk vychází z předchozí verze programu SMath a vznikl spojením jazyků z matematických programů Matlab, Maple, Microsoft® Excel, ale i programovacího jazyka C.

Aby se zmírnily rozdíly mezi jednotlivými jazyky (například konstanta π musí být někde zadána jako PI, jinde jako Pi), jsou všechny vestavěné funkce, operátory a konstanty necitlivé na velikost písmen (*case insensitive*). Oproti tomu uživatelské identifikátory jsou na velikost písmen citlivé (*case sensitive*), tato vlastnost byla zvolena zcela záměrně pro použití při výpočtech například fyzikálních příkladů. Zde je třeba rozlišovat například malé *p* (tlak) a velké *P* (příkon).

Další vlastností, která by měla usnadnit uživatelům práci s programem, je zavedení aliasů pro některé vestavěné funkce. V naší zemi je zvykem značit funkci *tangens* značkou *tg* velké množství ostatních zemí však značí *tangens* jako *tan*. Podobně je tomu například i u funkcí *asin* / *arcsin*, *sign* / *signum* apod. Uživatel tedy může použít různého zápisu pro vyjádření stejné funkce.

Jako oddělovač příkazů je použit konec řádků (\n).

3.1.1 Číselné literály

Číselný literál (dále číslo) je základním stavebním kamenem matematického výrazu. Výraz je vždy tvořen z naprosté většiny čísly. Protože samotný vyhodnocovač výrazů má být použitelný pro různé obory, bylo třeba umožnit zadávání čísel i v soustavách o jiném základu než 10 (tedy dekadicky). Pro programátory nebo HW designery je velmi užitečná binární a hexadecimální číselná soustava. Přehled zápisu čísel a podporovaných soustav naleznete v tabulce 1.

Název soustavy	**n_{base}**	**Regulární výraz popisující literál**	**Příklady**
binární	2	[01]+[bB]	010b, 100b
oktálová	8	0[0-7]+	0644, 0123
dekadická	10	([+-])?[0-9]+(\.([0-9]+([eE][+-]?[0-9]+)?)?, [eE][+-]?[0-9]+)?	35, -1.2e-3, 1e6
hexadecimální	16	0x[0-9A-Fa-f]+	0xFF, 0x12

Tabulka 1: Přehled podporovaných číselných soustav a jejich zápisu

3.1.2 Identifikátory

Jak již bylo zmíněno, uživatelské identifikátory jsou v programu jediným prvkem, který je citlivý na velikost písmen. Identifikátor je definován následujícím RV: [A-Za-z]+[A-Za-z0-9_]*.

3.1.3 Vestavěné funkce a operátory

Jazyk definuje následující:

- *binární operátory:* +, -, *, /, ^, >, <, >=, <=, =, <, >, !=, <>, and, or, xor, mod, div
- *sufixové unární operátory:* %, !
- *prefixové unární operátory:* not, -, +
- *operátor přiřazení:* :=
- *výrazové závorky:* (,)
- *dereferenční závorky:* [,]
- *závorky pro definici pole:* {, }
- *vestavěné funkce:* sin, cos, tan, cot, cotg, cotan, sec, cosec, csc, asin, acos, atan, atg, acot, acotg, acotan, asec, acsc, acosec, arcsin, arccos, arctan, arctg, arccot, arccotg, arccotan, arcsec, arccsc, arccosec, sinh, cosh, tanh, tgh, coth, cotgh, cotanh, sech, cosech, acsch, asinh, acosh, atanh, atgh, acoth, acotgh, acotanh, asech, acosech, acsch, arcsinh, arccosh, arctanh, arctgh, arccoth, arccotgh, arccotanh, arcsech, arccosech, arccsch, sqr, sqrt, round, floor, ceil, abs, log, log10, ln, logn, exp, random, rand, int, frac, root, sum, avg, gcd, lcm, combin, fact, factorial, pow, power, rad, deg, grad, sign, max, min, size, count, avg, average, sign, signum, diff, integ, integral

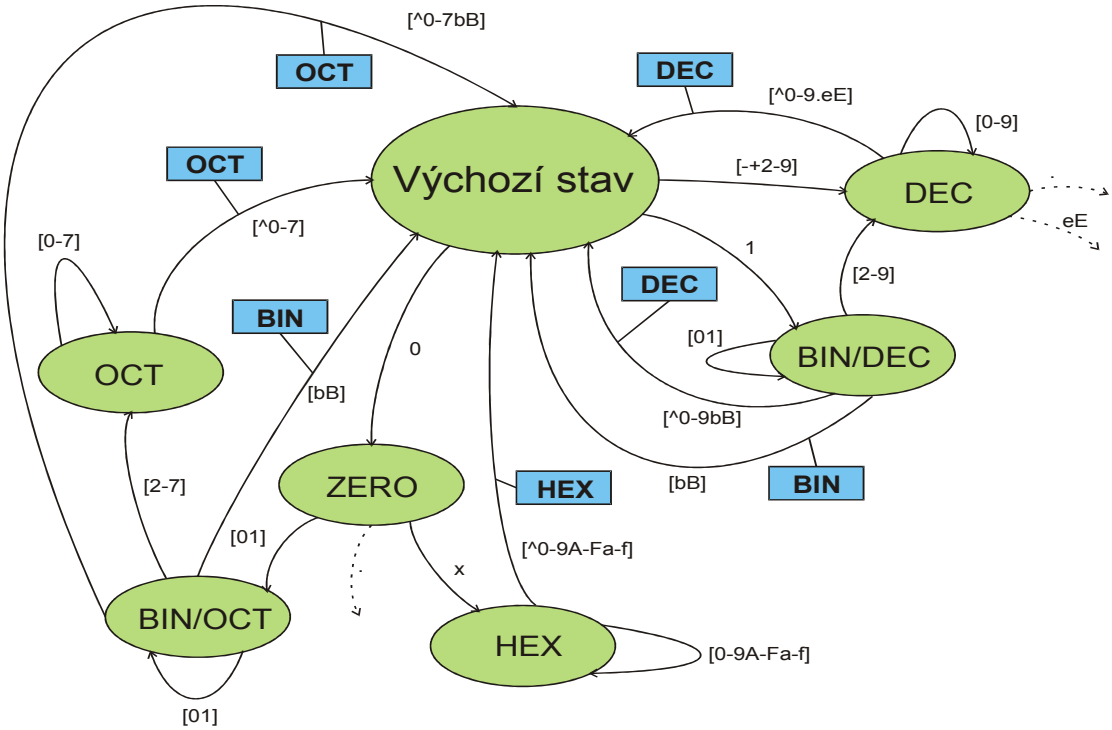
Podporovány jsou řádkové komentáře uvozené dvěma lomítky (//) a komentáře blokové uzavřené sekvencí (/*) a (*/).
Pro více informací a popis významu uvedených prvků jazyka nahlédněte do přílohy A. (Uživatelský manuál).

3.2 Lexikální analýza

Lexikální analyzátor (*scanner*) je implementován třídou `Scanner` v balíku `cz.itpro.libs.math.compiler`. V konstruktoru třídy je *scanneru* předána instance objektu `Reader`. Díky tomu může *scanner* zpracovávat vstup z předem uloženého řetězce, souboru nebo i ze streamovaných dat.

Hlavní částí scanneru je metoda `getToken()`, která po zavolání čte po znacích bufferovaný vstup a na základě rozhodnutí vestavěného konečného automatu vstup zpracovává. Princip, jakým scanner funguje, ilustruje obrázek č. 1. Na obrázku je část konečného automatu, která se stará

o rozpoznání číselného literálu v různých číselných soustavách. Přerušovanou čárou jsou naznačeny přechody do nezobrazených částí automatu, které mají na starost zpracování desetinných čísel v desítkové soustavě, a čísla zadaná v semilogaritmickém tvaru. Pokud je s hranou grafu spjat modrý obdelník, dojde při průchodu touto hranou k vytvoření *tokenu* odpovídajícímu přečtené číselné hodnotě.



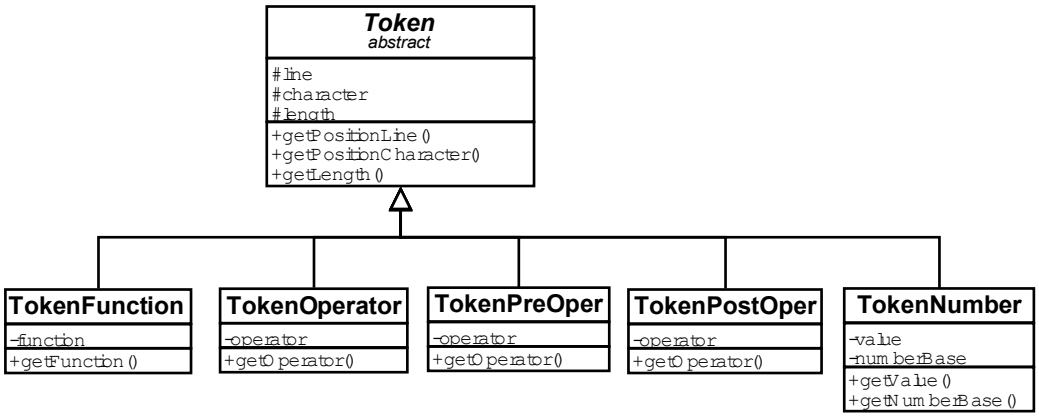
Obrázek 1: Část konečného automatu zpracovávající číselné literály

Implementace automatu je však ještě vůči klasickému pojetí konečného automatu odlišná jednou zásadní vlastností. *Scanner* si totiž pamatuje i typ *tokenu*, který vrátil při posledním volání své metody `getToken()`. Tato funkce je nezbytná pro potřeby rozpoznání unárního plus (+) a mínus (-) od jejich identických variant binárních. Pokud se *scanner* nachází ve výchozím stavu a narazí na symbol plus nebo mínus, nejprve zkontroluje, zda předchozím vráceným *tokenem* byl číselný literál, identifikátor, konstanta, libovolná pravá závorka (*)*, *]*, *}*). Pokud ano, je vrácen binární operátor plus nebo mínus. V opačném případě se pravděpodobně jedná o operátor unární. Toto rozšíření umožňuje použití precendenční syntaktické analýzy pro zpracovávání výrazů s unárním mínus. Samotná precedenční analýza by si totiž s jedním typem mínus nedokázala poradit.

3.2.1 Tokeny

Díky objektové orientaci zvoleného implementačního jazyka se jako ideální jeví reprezentovat tokeny jako instance odpovídajících tříd. Za tímto účelem byla definována abstraktní třída `Token` v balíku `cz.itpro.libs.math.compiler.tokens`. Tato třída má atributy určující pozici tokenu v textu a délku jeho textové reprezentace. Toho se využije při zvýrazňování syntaxe, ale také při

značení syntaktických, sémantických i běhových chyb ve vstupu. Třídy, které dědí od abstraktní třídy `Token`, pak obsahují doplňující atributy tak, jak je například vidět na obrázku č. 2.



Obrázek 2: Diagram tříd reprezentující tokeny

Od třídy `Token` dědí i řada dalších tříd, které na obrázku č. 2 nejsou uvedeny. Pro vytvoření instance tokenu tedy existují tyto třídy:

`TokenAssign` (přiřazení), `TokenComma` (oddělovač čárka), `TokenConstant` (vestavěná konstanta), `TokenEOF` (konec vstupu), `TokenError` (chybný symbol), `TokenFunction` (vestavěná funkce), `TokenIdentifier` (identifikátor), `TokenLBracket` (levá hranatá závorka), `TokenLParenthesis` (levá kulatá závorka), `TokenLVinculum` (levá složená závorka), `TokenNumber` (číselný literál), `TokenOperator` (binární operátor), `TokenPreOper` (prefixový unární operátor), `TokenPostOper` (sufixový unární operátor), `TokenRBracket` (pravá hranatá závorka), `TokenRParenthesis` (pravá kulatá závorka), `TokenRVinculum` (pravá složená závorka), `TokenTerminator` (ukončovač příkazu).

3.3 Syntaktická a sémantická analýza

Funkci syntaktického a sémantického analyzátoru plní v implementaci třída `ExpressionParser` z balíku `cz.itpro.libs.math.compiler`. Analýza se provádí ve směru zdola nahoru. Jednotlivé *terminály* a *neterminály* jsou tedy postupně slučovány, až je dosaženo výchozího *neterminálu*. Pro překlad je použita precedenční syntaktická analýza.

3.3.1 Precedenční syntaktická analýza

3.3.1.1 Algoritmus

Algoritmus, jakým funguje precedenční syntaktická analýza, popisuje následující zápis v algoritmickém pseudojazyce [1]:

```
vlož na zásobník $
opakuj:
  nechť  a = aktuální symbol na vstupu
         b = terminál na zásobníku nejblíže vrcholu
  když znak na pozici tabulka[b, a] je:
```

- `=`, tak: push(**a**) & přečti další symbol **a** ze vstupu
- `<`, tak: zaměň **b** za **b**`<` na zásobníku & push(**a**) & přečti další symbol **a** ze vstupu
- `>`, tak: když `<`*y* je na vrcholu zásobníku a zároveň existuje pravidlo *r*: *A* → *y* ∈ *P*, tak zaměň `<`*y* za *A*, jinak **chyba**
- **E**, tak **chyba**

```
dokud se a nerovná $ nebo b se nerovná $
```

3.3.1.2 Precedenční tabulka

Precedenční tabulka obsahuje řídící symboly podle kterých se syntaktický analyzátor rozhodne, zda má provést *shift* nebo *redukci*, případně umožňuje zjistit, že došlo k syntaktické chybě. Pokud bychom chtěli mít vlastní řádek a sloupec pro každý typ tokenu a jeho atributů, byla by tabulka zbytečně veliká a obsahovala by velké množství rendundantních údajů. Například pro operátory krát a děleno jsou stejné řídící symboly, mohou být tyto dva operátory z pohledu precedeční tabulky sloučeny.

Toto sloučení a výslednou precedeční tabulku znázorňuje tabulka č. 2. Vysvětlivky k jednotlivým číslům sloupců a řádků uvádí tento přehled:

1. levá kulatá závorka
2. pravá kulatá závorka
3. levá hranatá závorka
4. pravá hranatá závorka
5. levá složená závorka
6. pravá složená závorka
7. oddělovač čárka
8. prefixové unární operátory
9. sufixové unární operátory
10. binární plus a mínus
11. krát, děleno
12. mocnina
13. operátory porovnávání
14. logické spojky
15. identifikátory
16. literály
17. vestavěné funkce
18. přiřazení
19. konec vstupu

	1.(	2.)	3. [	4.]	5. {	6. }	7. ,	8. Not,+-	9. !%	10. +-	11. */	12. ^	13. CMP	14. LOG	15. ID	16. LIT	17. FNS	18. :=	19. $
1. (	<	=	E	E	<	E	<	<	<	<	<	<	<	<	<	<	<	E	E
2.)	E	>	E	>	E	>	>	E	>	>	>	>	>	>	E	E	E	E	>
3. [	<	E	E	=	E	E	<	<	<	<	<	<	<	<	<	<	<	E	E
4.]	E	>	>	>	E	>	>	E	E	>	>	>	>	>	E	E	E	>	>
5. {	<	E	E	E	<	=	<	<	<	<	<	<	<	<	<	<	<	E	E
6. }	E	>	E	>	E	>	>	E	E	>	>	>	>	>	E	E	E	E	>
7. ,	<	>	E	>	<	>	>	<	E	<	<	<	<	<	<	<	<	E	>
8. Not,+-	<	>	E	>	E	>	>	<	>	>	>	>	>	>	<	<	<	E	>
9. !%	E	>	E	>	E	>	>	E	>	>	>	>	>	>	E	E	E	E	>
10. +-	<	>	E	>	E	>	>	<	<	>	<	<	>	>	<	<	<	E	>
11. */	<	>	E	>	E	>	>	<	<	>	>	<	>	>	<	<	<	E	>
12. ^	<	>	E	>	E	>	>	<	<	>	>	<	>	>	<	<	<	E	>
13. CMP	<	>	E	>	E	>	>	<	<	<	<	<	>	>	<	<	<	E	>
14. LOG	<	>	E	>	E	>	>	<	<	<	<	<	<	>	<	<	<	E	>
15. ID	=	>	=	>	E	>	>	E	>	>	>	>	>	>	E	E	E	=	>
16. LIT	E	>	E	>	E	>	>	E	>	>	>	>	>	>	=	E	E	E	>
17. FNS	=	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
18. :=	<	>	E	>	<	E	>	<	<	<	<	<	<	<	<	<	<	<	>
19. $	<	E	<	E	<	E	E	<	<	<	<	<	<	<	<	<	<	<	E

Tabulka 2: Použitá precedenční tabulka

3.3.1.3 Redukční pravidla

Vždy, když algorimus precedeční analýzy narazí na symbol >, je třeba provést redukci části řetězce uzavřeného mezi dvojici symbolů < a > podle redukčního pravidla, pokud takové existuje.

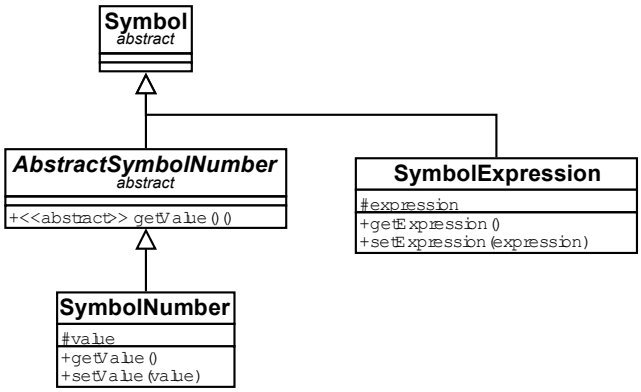
Navržená gramatika obsahuje tato pravidla (terminály jsou značeny modře, neterminály červeně):

1. $E \rightarrow$ num
2. $E \rightarrow$ const
3. $E \rightarrow$ id
4. $E \rightarrow$ preoper E
5. $E \rightarrow$ num id
6. $E \rightarrow$ num const
7. $E \rightarrow E$ postoper
8. $E \rightarrow$ { }
9. $E \rightarrow$ (E)
10. $E \rightarrow$ { E }
11. $E \rightarrow$ { E_list }
12. $E\_list \rightarrow E\_list$, E
13. $E\_list \rightarrow E$, E
14. $E \rightarrow$ id := E
15. $E \rightarrow E$ oper E
16. $E \rightarrow$ fnc ( )
17. $E \rightarrow$ fnc (E_list)
18. $E \rightarrow$ fnc (E)
19. $E \rightarrow$ id [E]
20. $E \rightarrow E$ [E]

Vždy při použití libovolného syntaktického pravidla je také provedena část sémantické kontroly. Je například zkontrolováno, zda má funkce přípustný počet parametrů.

3.3.2 Tabulka symbolů

Pro uchovávání hodnot potřebných pro výpočet je třeba uschovávat hodnoty jednotlivých uživatelských identifikátorů v tabulce symbolů. Tato tabulka je implementována pomocí tabulky s rozptýleným indexem (*hashtable*). Jako klíč slouží název identifikátoru, hodnotou je odkaz na instanci třídy Symbol. Hodnotou nemůže být přímo instance, protože navržený jazyk není typový a ani nevyžaduje dopředné deklarace. Typ hodnoty v identifikátoru se může za běhu libovolně měnit. Obrázek č.3 znázorňuje třídy použitelné pro uchování hodnoty symbolu.



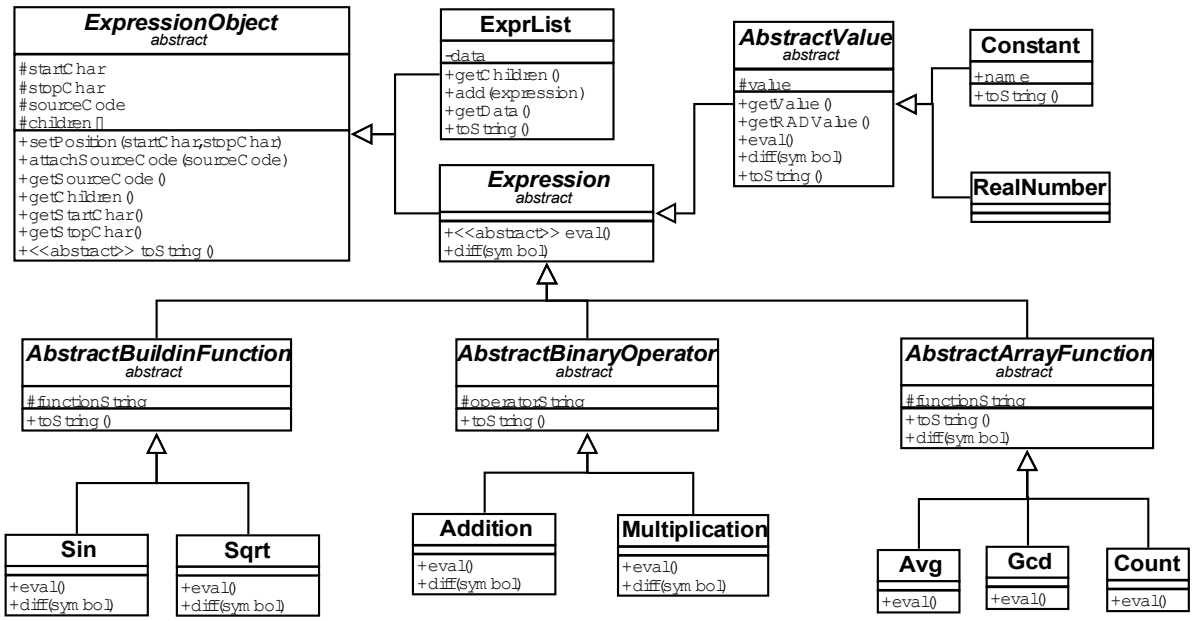
Obrázek 3: Diagram tříd pro uchovávání hodnot symbolů

3.4 Objektová reprezentace výrazu

Výstupem z *parseru* (syntaktický a sémantický analyzátor) je objektová reprezentace výrazu, který byl na vstupu *scanneru*. V průběhu aplikace každého redukčního pravidla je vygenerována část objektového stromu. Při tomto kroku je také upravována informace o odpovídající pozici právě vytvořeného objektu v textovém zadání výrazu. Aktualizaci této informace je nutné provádět z důvodu chybových hlášení, ke kterým může dojít v případě běhových chyb (např. dělení nulou).

3.4.1 Objektový model

Pro potřeby reprezentace výrazu objektem bylo třeba vytvořit model tříd, které budou schopny uchovávat informace o jednotlivých částech výrazu. Všechny tyto třídy jsou uloženy v balíku `cz.itpro.libs.math.objects`. Část schematu vytvořeného modelu znázorňuje obrázek č. 4.



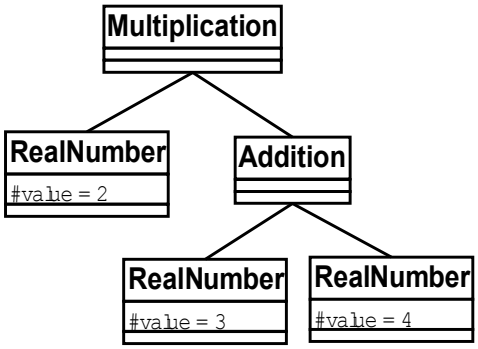
Obrázek 4: Diagram tříd určených k reprezentaci matematického výrazu

Po rozsahové stránce je výše uvedený diagram značně omezený. Pro každou funkci a operaci, kterou lze ve výrazu provést, totiž existuje vlastní třída. V diagramu jsou uvedeny tedy pouze všechny třídy, které jsou v hierarchickém stromu chápány jako kořen, většina listových tříd byla vynechána.

3.5 Vyhodnocování výrazu

K vyhodnocení výrazu slouží metoda `eval()`, kterou má povinně implementovanou každá instance potomka třídy `Expression`. Volání této metody vrací opět instanci třídy `Expression`.

Jak tedy vyhodnocení skutečně funguje? Vezměme si například objektový strom, který *parser* vytvoří na vstup: `2*(3+4)`. Takový strom je znázorněn na obrázku č. 5.

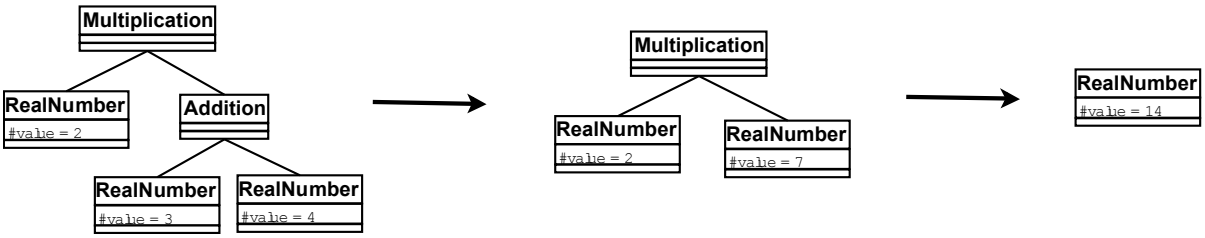


Obrázek 5: Příklad objektového stromu výrazu

K obrázku si nyní musíme uvědomit, že se již nejedná o vazby mezi třídami, ale že zobrazené objekty jsou již instance těchto tříd. Jednotlivé vazby mezi objekty jsou definovány pomocí pole `children`, které obsahuje všechny přímo podřízené instance daného objektu. Pole `children` dědí všechny objekty od třídy `ExpressionObject`.

Představme si nyní, že tedy máme celý výraz uložen v datovém typu třídy `Expression`. Protože tato třída má abstraktní metodu `eval()`, můžeme tuto metodu zavolat. Kořenová instance objektového stromu výrazu, instance třídy `Multiplication` ve své přetížené (*override*) metodě `eval()` zavolá nejprve metodu `eval()` u všech svých přímo podřízených objektů, tedy u objektu typu `RealNumber` s hodnotou 2 a `Additon`. Instance třídy `RealNumber` nemá co vyhodnocovat, a tak vrátí sebe sama. Volání `eval()` na objekt typu `Addition` nejprve opět zavolá metodu `eval()` u všech svých přímo podřízených objektů. Jak již bylo zmíněno, vyhodnocení instance třídy `RealNumber` je vždy instance sama o sobě. Objekt typu `Addition` tedy nyní má vyhodnoceny oba přímo podřízené objekty. Zkontroluje zda jsou výsledky typu `AbstractValue`, tedy jestli je možné přímo získat jejich číselnou hodnotu. Pokud ano, jako v našem případě, zjistí si tyto hodnoty a sečte je. Jako výsledek na volání metody `eval()` vrací naše instance třídy `Addition`, instanci třídy `RealNumber` s hodnotou výsledného součtu, tedy s atributem `value` rovným 7. Nyní se můžeme vrátit o úroveň výše do běžícího volání metody `eval()` patřící objektu typu `Multiplication`. Ta již nyní má vyhodnoceny všechny své přímo podřízené objekty a může provést stejně jako objekt `Addition` svůj výpočet a vrátit instanci třídy `RealNumber` s celkovým výsledkem, číslem 14. Postup výpočtu ilustruje obrázek č. 6.

Co se ale stane, když přímo podřízený objekt jako výsledek volání metody `eval()` nevrátí instanci třídy `AbstractValue`? Taková situace může za jistých okolností, které jsou popsány v další kapitole, nastat. Vyhodnocení se pak zachová tím způsobem, že jako výsledek vrátí novou instaci své třídy a jako přímo podřízené objekty nastaví alespoň částečné vyhodnocení původních synovských objektů. To může sloužit například ke zjednodušování výrazů nebo k výpočtu lomených výrazů, jak se dozvíme dále.



Obrázek 6: Postup vyhodnocení objektového stromu

3.5.1 Nastavení vyhodnocování

Samotné vyhodnocování má také určité možnosti nastavení. Tato potřeba již zcela intuitivně vyplývá ze samotného použití goniometrických, cyklometrických a dalších funkcí, které mají jako vstupní nebo výstupní parametr úhlovou jednotku. Program tedy podporuje a uživatel může mít nastavenou jednu z jednotek stupně (*DEG*), radiány (*RAD*) a grady (*GRAD*). Ke správné interpretaci úhlových hodnot slouží metoda třídy `AbstractValue getRADValue()`. Tato metoda vrátí hodnotu vždy v radiánech bez ohledu na momentálně nastavenou úhlovou jednotku.

Nastavení je realizováno třídou `MathOptions`. Instance této třídy uchovává nejen nastavení úhlové jednotky, ale i formátovací pravidla pro výsledná čísla, nastavení speciálních možností vyhodnocování, či soustavu, ve které mají být zobrazována výsledná čísla. V průběhu vyhodnocovaní lze aktuální nastavení získat pomocí volání statické metody `MathOptions.getOptions()`, která vrací instanci třídy `MathOptions` udávající platná nastavení.

3.5.2 Vyhodnocování lomených výrazů

Jedna z možností, kterou lze nastavit v `MathOptions`, je mód vyhodnocování `FRACTION`. Tento mód umožňuje programu počítat ve zlomcích.

Pokud je tento mód aktivován, mění se chování některých objektů. Nejvýrazněji se tato změna projeví u instancí třídy `Division`. Pokud v průběhu volání metody `eval()` zjistí tato funkce, že výsledkem vyhodnocení přímo podřízených objektů jsou celá čísla (instance třídy `AbstractValue`, které mají desetinnou část rovnu nule), nevrátí jako svůj výsledek podíl těchto dvou čísel, ale vrací novou instanci dělení (třída `Division`) s tím, že přímo podřízené objekty jsou instance třídy `RealNumber`, jejichž hodnota value je rovna výsledku vyhodnocení původních synovských objektů.

Zároveň ještě třída Division provádí krácení vzniklých zlomků za využití *Euklidova algoritmu* [2] pro nalezení největšího společného dělitele dvou čísel. Tento algoritmus vypadá následovně:

```
Mějme dána dvě přirozená čísla, uložená v proměnných u a v.
Dokud v není nulové, opakuj:
  Do r ulož zbytek po dělení čísla u číslem v
  Do u ulož v
  Do v ulož r
Konec algoritmu, v u je uložen největší společný dělitel původních
čísel.
```

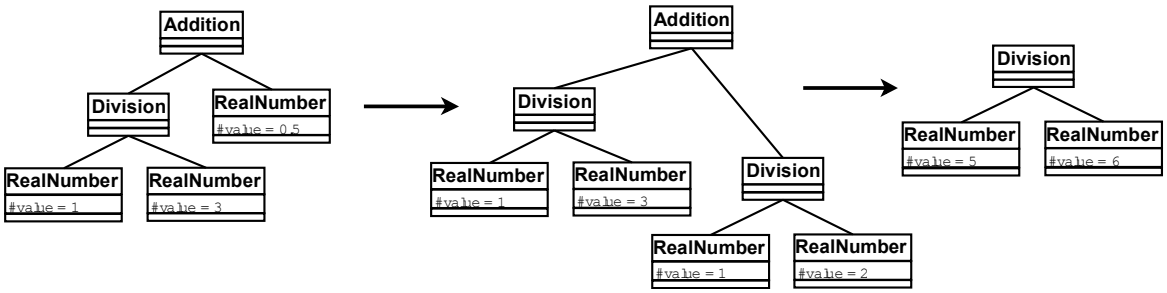
Další třídou, která má v módu vyhodnocování `FRACTION` značně odlišné chování, je třída `RealNumber`. Ta, pokud ve svém atributu `value` nalezne desetinné číslo, pokusí se toto převést na zlomek a na volání metody `eval()` nevrací sebe sama, ale instanci třídy `Division` se vzniklým zlomkem.

Aby ale výpočet nebyl zastaven po vytvoření prvního zlomku, musí mít všechny základní operace a funkce definováno, jak se zlomky počítat. V příslušných třídách je tedy uvedeno, jak se zlomky sčítají, odčítají, násobí, dělí, umocňují a odmocňují. Pro výpočty ze zlomky jsou použity následující vztahy:

$$a=\frac{a}{1} \quad , \quad \frac{a}{b}+\frac{c}{d}=\frac{ad+cb}{bd} \quad , \quad \frac{a}{b}-\frac{c}{d}=\frac{ad-cb}{bd} \quad , \quad \frac{a}{b}\cdot\frac{c}{d}=\frac{ac}{bd} \quad , \quad \frac{\frac{a}{b}}{\frac{c}{d}}=\frac{ad}{bc} \quad ,$$

$$\left(\frac{a}{b}\right)^n=\frac{a^n}{b^n} \quad , \quad \sqrt[n]{\frac{a}{b}}=\frac{\sqrt[n]{a}}{\sqrt[n]{b}}$$

Ilustraci výpočtu s aktivovaným módem `FRACTION` uvádí obrázek č. 7, který demonstruje vyhodnocení výrazu: `1/3+0.5`.



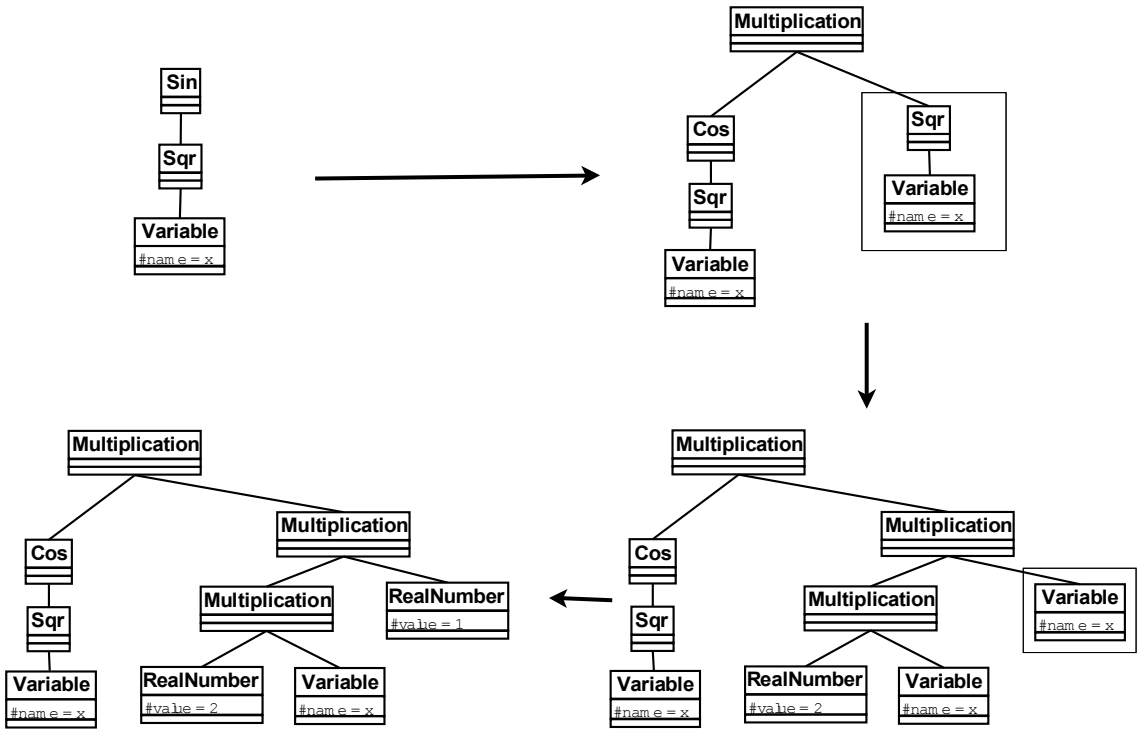
Obrázek 7: Příklad vyhodnocení lomeného výrazu

3.5.3 Analytická derivace

Jak je patrno z obrázku č. 4 v kapitole 3.4.1, navržený model tříd počítá i s možností analytické derivace vstupního výrazu. K tomu slouží metoda `diff()`, kterou povinně implementuje každý potomek třídy `Expression`. Vyhodnotit je možné jakýkoli výraz, derivovat však všechno nelze, anebo by výsledek derivace nebyl příliš smysluplný (např. derivace faktoriálu, či derivace funkce zaokrouhlení). Ve třídě `Expression` je tedy přímo definovaná metoda `diff()` s tím, že vrací chybovou hlášku o tom, že výraz nelze diferencovat (více informací o chybových hlášení naleznete v dalších kapitolách). Třídy, kde derivace jejich odpovídající funkce nebo operace má nějaký smysl, tuto zděděnou metodu `diff()` přetěžují.

Metodě `diff()` je jako parametr předán odkaz do tabulky symbolů, který vyjadřuje podle jaké proměnné má být výraz zderivován. Podobně jako při vyhodnocování výrazu je při derivování volána derivační metoda v objektovém stromu ve směru od kořenu k listovým prvkům. Výstup metody `diff()` se pak řídí podle derivačních pravidel [3]. Jako příklad si demostrujme zderivování výrazu zadaného jako: `sin(sqr(x))`. Vezměme tedy objekt `Sin` jako kořenový, který má jeden synovský objekt `Sqr`, a ten má také jeden synovský objekt `Variable`. Derivace funkce sinus je kosinus. Proto je vytvořena instance třídy `Cos` a jako synovský objekt je k ní připojen objekt `Sqr`, který byl přímo podřízen původnímu objektu. Podle vzorce pro derivaci složené funkce, pokud $f(x) = h(g(x))$, pak $f'(x) = h'(g(x)) \cdot g'(x)$, musíme ještě zderivovat funkci vnitřní. Proto je vytvořen nadřazený objekt `Multiplication`, kterému je jako první synovský objekt připojena vytvořená derivace. Jako druhý podřízený objekt je připojen výsledek volání metody `diff()` na vnitřní výraz.

Tímto způsobem se rekurzivně postupuje, dokud zbývá nějaký prvek ke zderivování. Popsaný příklad ilustruje obrázek č. 8.



Obrázek 8: Příklad výpočtu analytické derivace

Aby mohla být derivace vyvolána uživatelem, je v objektovém modelu definována třída `Diff`, která ve své metodě `eval()` volá metodu `diff()` svého synovského objektu a výsledek tohoto volání vrací jako svůj výsledek. Za účelem toho, aby se uživateli mohl vrátit také zderivovaný výraz a nejen hodnota derivace v bodě, existuje další možné nastavení vyhodnocování, které potlačuje vyhodnocení proměnných. Objekty třídy `Variable` tak nevrací po zavolání metody `eval()` hodnotu, kterou naleznou v tabulce symbolů, ale vrací pouze referenci samy na sebe.

3.5.4 Optimalizace

V předchozí kapitole o analytickém derivování si lze povšimnout, že při této operaci s výrazem vzniká velké množství podvýrazů, které jde zjednodušit (např. násobení jedničkou), nebo úplně vypustit (např. násobení nulou). U příslušných operací jsou tedy definovány podmínky, za kterých lze výstup optimalizovat. Zakódovány jsou tyto následující závislosti:

$$a \cdot 0 = 0 \quad , \quad a \cdot 1 = a \quad , \quad a + 0 = a \quad , \quad a - 0 = a \quad , \quad a^0 = 1 \quad , \quad a^1 = a \quad , \quad \frac{0}{a} = 0$$

3.5.5 Chybová hlášení

K chybě může dojít na mnoha místech. Chybu může zjistit už lexikální analyzátor přítomností neočekávaného znaku ve vstupu, různé typy chyb může objevit syntaktický a sémantický analyzátor, ale nejvíc chyb se může projevit až za běhu. Mezi nejobvyklejší běhové chyby patří dělení nulou,

použití neincializované proměnné nebo pro danou funkci nepřípustná hodnota parametru. Zahlásit pouze, že výraz je chybný, by pro uživatele ale nebylo příliš přívětivě. Implementace proto umožňuje identifikovat nejen typ chyby, ke kterému došlo, ale dokonce graficky znázornit pozici ve zdrojovém textu, kde se pravděpodobně nachází problém.

Zcela ideální pro vyhodnocování chyb se jeví systém vyjímek poskytovaný programovacím jazykem Java. V balíku `cz.itpro.libs.math.compiler.errors` je tedy definovaná hierarchie tříd, které jsou odvozeny od třídy `Exception`. Pokud dojde k nějaké neočekávané situaci, například funkce logaritmus dostane záporný argument, je vyvolána vyjímka `UndefinedResultException`, která nese jako svůj atribut referenci na potomka třídy `Expression`, ve kterém došlo ke zmíněné chybě. Při vyvolání vyjímky je tedy vždy nastavena reference na právě vykonávaný objekt.

Protože si vyjímka s sebou nese odkaz na objekt ve kterém došlo k chybě, lze snadno určit o jaký typ chyby se jednalo, a ve zdrojovém kódu je možno zvýraznit místo chyby.

3.6 Znovupoužitelnost vyhodnocovače

Kód byl navržen tak, aby byl bez problémů a úprav znovupoužitelný v dalších projektech. Pokud si tedy do jiného projektu naimportujeme balík `cz.itpro.libs.math`, lze vyhodnocovač výrazů snadno používat. Následují kód ukazuje jak:

```
String expression = "1+2*3";
// vytvoření tabulky symbolů
SymbolTable symbolTable = new SymbolTable();
// vytvoření objektu parseru a přiřazení tabulky symbolů
ExpressionParser parser = new ExpressionParser();
parser.setSymbolTable(symbolTable);
// vytvoření scanneru nad vstupním řetězcem
Scanner scanner = new Scanner(new StringReader(expression));
// převedení výrazu v řetězci na objektový model
Token inputToken = null;
Expression exprObj = parser.parse(scanner,inputToken);
// vyhodnocení a vypsání výsledku nebo chyby
try {
    System.out.println(exprObj.eval());
} catch (Exception e) {
    System.out.println("Nelze vyhodnotit!");
}
```

4 Vykreslování grafů

Vykreslování grafů průběhů křivek je asi nejzásadnější část programu. Aby bylo dosaženo možnosti vektorového exportu, je do programu začleněna knihovna FreeHEP VectorGraphics dostupná na http://java.freehep.org/vectorgraphics. Knihovna je dostupná pod licencí GPL[1]. Tato knihovna umožňuje kreslit na vektorové plátno, které je děděno od standardní třídy `Graphics`, což je ideální podmínkou pro bezproblémové začlenění knihovny do běžného vykreslovacího procesu grafiky v Javě. Celá koncepce vykreslování grafu je založena na třídě `Graph`, která se chová jako kolekce objektů třídy `GraphObject`. Každý objekt této třídy je pak schopen se sám vykreslit podle zadaných parametrů (více informací níže). O samotné vykreslení grafu na obrazovku se pak stará komponenta `GraphViewer`, která v sobě obsahuje odkaz na instanci třídy `Graph`, jež má být vykreslena.

Všechny třídy, které se starají o vykreslování a uchovávání dat grafu, jsou uloženy v balíku `cz.itpro.utils.graphs`.

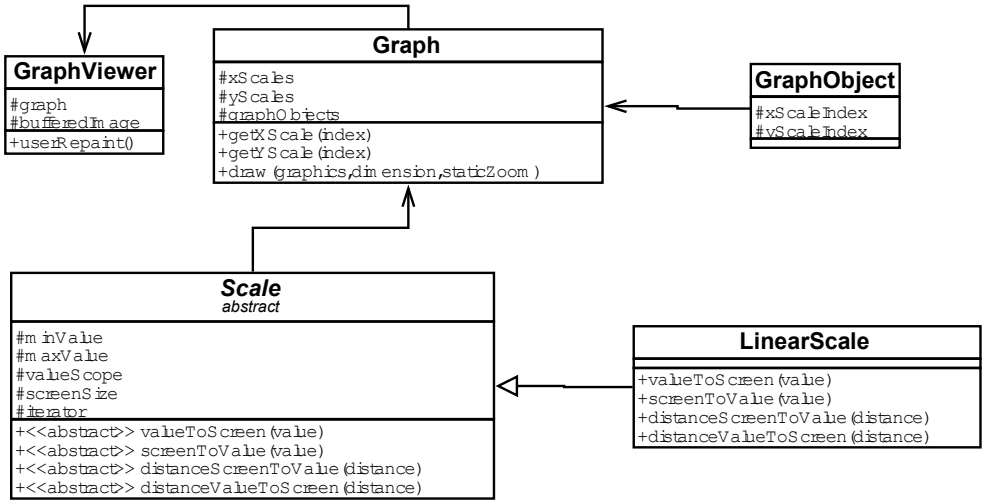
4.1 Objektový model

Jak již bylo zmíněno v úvodu této kapitoly, základem vykreslení grafu je třída `Graph`, která schraňuje vykreslované objekty v grafu v kolekci třídy `GraphObject`. Základní informací, kterou třída `Graph` uchovává, je kolekce měřítek pro vodorovný a svislý směr. V současné implementaci je vytvořena pouze třída pro lineární měřítko, koncept je však připraven tak, aby nebyl problém rozšířit program o podporu například logaritmického měřítka. Za tímto účelem byla vytvořena abstraktní třída `Scale`, která určuje metody, které se starají o přepočet souřadnic. Objekt reprezentující měřítko má za úkol pouze přepočítávat souřadnice zobrazení na obrazovce na souřadnice virtuální a nazpět. Každý objekt, který je v grafu vykreslen, má pomocí identifikačního čísla určeno, jaké měřítko má pro své vykreslování použít. To umožňuje mít v jednom grafu i objekty, které používají rozdílná měřítka.

Protože by bylo velmi neefektivní vykreslovat neustále celý graf znovu, zapouzdřuje v sobě komponenta `GraphViewer`, která je odvozena od kontejneru `JPanel`, bufferované kreslící plátno. Celý graf je tedy na vnitřní v paměti uložené plátno překreslen pouze na žádost, a to zavoláním metody `userRepaint()`. Pokud není explicitně zavolána tato metoda, vykresluje komponenta vždy jen zmíněný vypočítaný a uložený obraz. Třída `GraphViewer` plní navíc ještě funkci, která umožní rozpoznat objekty, které jsou ve vykresleném obrazu na dané souřadnici. Toho pak lze využít v uživatelském rozhraní k výběru objektu v grafu kliknutím myší (metoda zjišťování objektů na dané souřadnici je popsána níže).

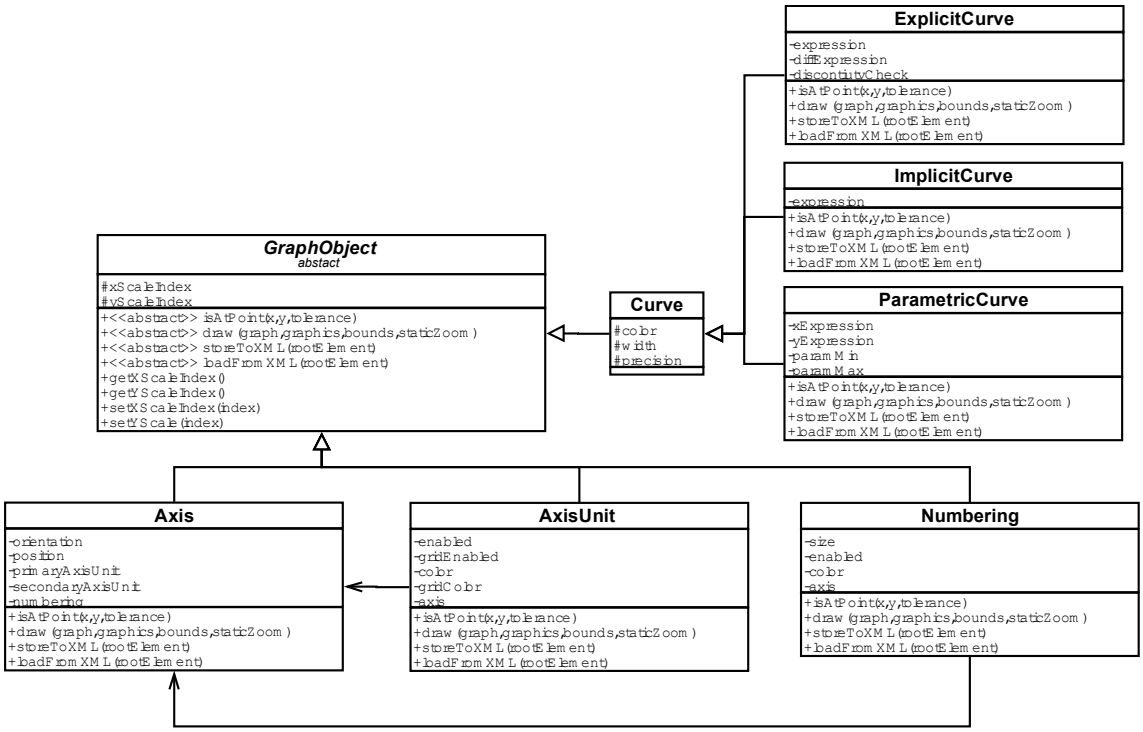
1 General Public Licence

Základní myšlenku objektového modelu ilustruje obrázek č. 9.



Obrázek 9: Základ konceptu objektového modelu pro vykreslování grafů

Samotné objekty, které se vykreslují do grafu, jsou odvozeny od třídy `GraphObject`. Tato třída povinně definuje například metodu, která zjistí, zda se daný objekt nachází na souřadnici zobrazené na obrazovce. Tuto metodu využívají metody třídy `GraphViewer` `getFirstObjectAt()` a `getObjectsAt()`, které zavolají zmíněnou metodu pro všechny objekty kolekce uvnitř třídy `Graph`. Hlavní metodu, kterou musí implementovat každá třída děděná od třídy `GraphObject`, je metoda `draw()`. Tato metoda slouží k vykreslení daného objektu do grafu. Důležité je však také vykreslovací pořadí objektů. Metodu `draw()` nelze v kolekci volat náhodně. Za tímto účelem je vytvořena třída `GraphObjectComparator`, která implementuje rozhraní `Comparator<GraphObject>` a umožňuje kolekci grafických objektů seřadit tak, aby například osa byla vykreslena dříve než křivka. Model tříd základních grafických objektů, které lze do grafu přidat, znázorňuje obrázek č. 10. Na něm si lze povšimnout vztahu mezi třídami `Axis`, `AxisUnit` a `Numbering`. Třídy `AxisUnit` a `Numbering` totiž nemohou existovat nezávisle (nelze je vykreslit bez existence rodičovského objektu typu `Axis`). Na obrázku je také vidět abstraktní třída pro vykreslované křivky Curve. Tato třída ukládá informaci o grafickém zobrazení křivky (barva, šířka čáry, přesnost). Jednotlivé detaily vykreslení pak definují až třídy od této třídy odvozené. Detailní informace o principu vykreslování jednotlivých typů křivek jsou popsány v dalších kapitolách.

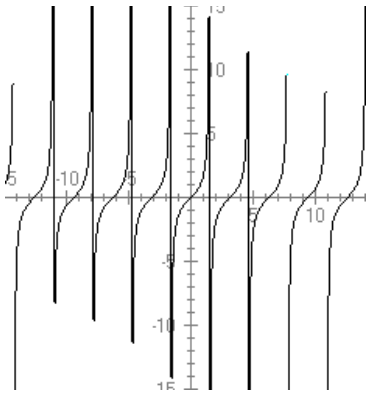


Obrázek 10: Diagram tříd objektů grafu

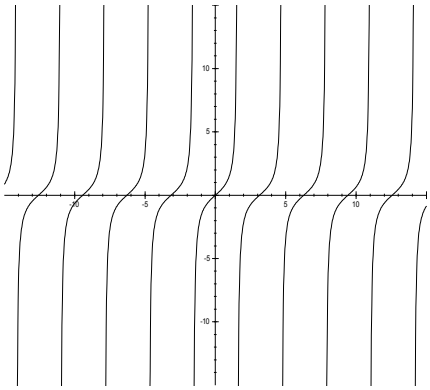
4.2 Explicitní křivky

Explicitní křivky jsou reprezentovány třídou `ExplicitCurve`. Nejjednodušším přístupem pro vykreslování explicitních křivek by bylo spočítat funkční hodnotu zadané funkce v každém bodu zobrazeném na obrazovce a tento bod spojit s předchozím spočítaným bodem. Tento přístup však ve skutečnosti není použitelný, protože neřeší nespojitosti funkcí a hodnoty sahající do nekonečna.

Nespojistost funkcí je při vykreslování průběhu funkce obecný problém. Nespojitost průběhu může být způsobena tím, že funkce není v nějaké části vykreslovaného intervalu definována, nebo je pro nějaký bod provedena nepovolená operace (např. dělení nulou). Pokud se jedná o odstranitelnou bodovou nespojitost, dá se problém vykreslení nespojistosti ještě ignorovat. V případě nespojitosti skokové by ale v grafu vznikaly zcela neodpodstatněné čáry, které by graf znehodnocovaly. Příklad takového znehodnocení grafu je patrný třeba na průběhu funkce *tangens*. Obrázek č. 11 zachycuje zobrazení průběhu funkce tangens v konkurenčním volně dostupném programu. Na obrázku č. 12 je zobrazení stejné křivky se stejnými vykreslovacími parametry v programu, který je předmětem této bakalářské práce. Dalším nedostatkem, kterého si lze na ukázce špatného vykreslení všimnout, je to, že křivka nejde až do nekonečna (potažmo na kraj obrazovky). Tento problém způsobuje nespojistost uložení čísel v počítači (přesně lze vyjádřit pouze takové desetinné číslo, jehož desetinnou část lze vyjádřit jako součet záporných mocnin čísla dvě).



Obrázek 11: Chybné zobrazení funkce tangens



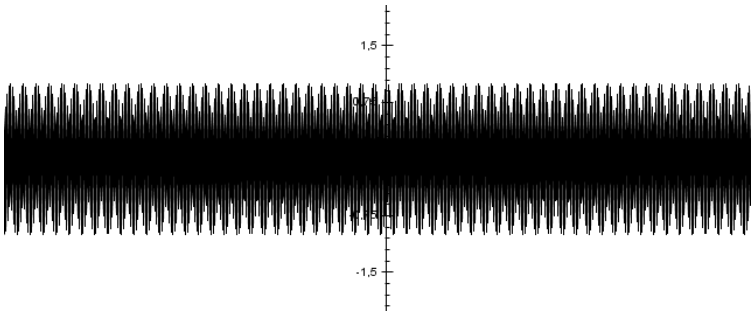
Obrázek 12: Správné zobrazení funkce tangens

Při vykreslování explicitně zadaných křivek v programu SMath v4.0 jsou řešeny oba tyto problémy. Při zadání výrazu je analyticky spočítána derivace zadané funkce. Samotné vykreslování je realizováno tak, že pro každý bod na ose X (s ohledem na nastavenou přesnost) je vypočtena funkční hodnota zadaného výrazu v tomto bodě. Zároveň je vypočtena i hodnota derivace v daném bodě dosazením do vypočtené analytické rovnice derivace. Hodnota derivace v bodě je porovnána s numericky vypočtenou derivací podle vzorce:

$$f'(x)=\frac{f(x)-f(x-\Delta x)}{\Delta x}$$, kde Δx je velikost vzorkovacího kroku.

Numericky získaná hodnota derivace v bodě je porovnána s analyticky získanou hodnotou. Pokud se tyto dvě hodnoty signifikantně liší, narazili jsme pravděpodobně na bod skokové nespojistosti a je třeba přerušit vykreslování křivky a pokračovat v něm na novém místě. Na stejném principu je řešeno i dotažení křivek do $\pm\infty$. Jakmile hodnota derivace překročí určitou hranici a poté je detekována nespojistost, program rozhodne dotáhnout křivku do nekonečna nebo mínus nekonečna, v závislosti na znaménku hodnoty derivace.

Problémem, který však v programu při vykreslování vyřešen není, je Shannonův vzorkovací teorém [5], který říká, že ideální frekvence pro vzorkování je rovna dvojnásobku maximální frekvence vyskytující se ve funkci *f*. Při kroku větším než polovina periody maximální frekvence vzniká alias. Zda nevznikl alias, si musí uživatel uvědomit sám na základě znalostí o zadané křivce. Obrázek č. 13 demonstruje situaci, na které vznikl alias při vykreslení funkce sinus tím, že na ploše grafu je vykreslen rozsah X v intervalu <-500π; 500π>. Pokud by mělo být vykreslení správné, měl by být v grafu vykreslen vybarvený černý obdelník.



Obrázek 13: Vznik aliasu u vykreslení funkce sinus

4.3 Parametrické křivky

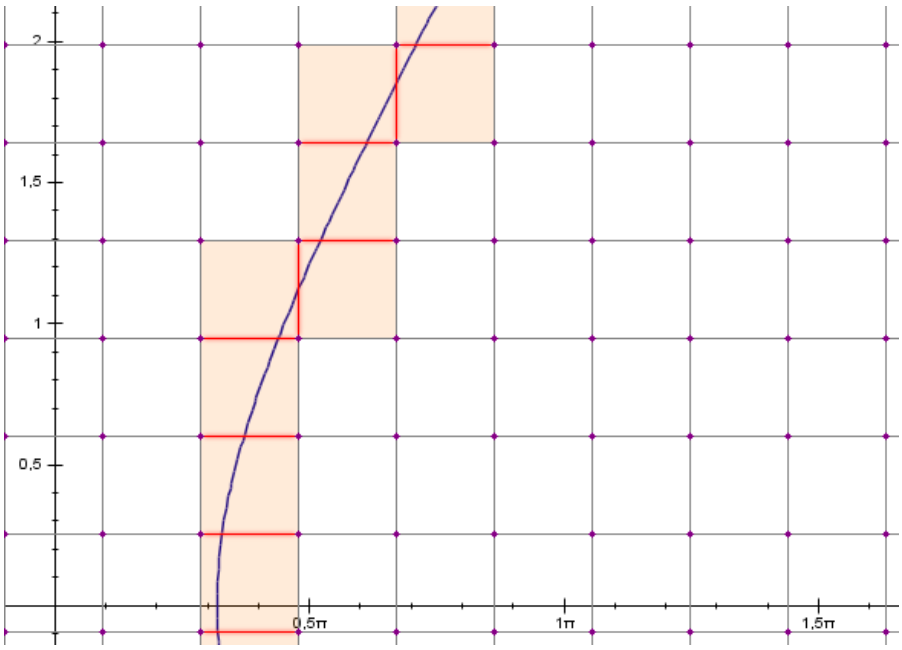
Parametrické křivky v programu reprezentuje třída `ParametricCurve`. Tato třída ve své metodě `draw()` iteruje přes rozsah parametru *p* od minimální do maximální hodnoty s krokem, jehož velikost je odvozena od zvolené přesnosti vykreslení křivky. Hodnota parametru je poté dosazena do rovnic určujících souřadnici X a Y. Výsledné hodnoty jsou přepočítány na zobrazení v grafu a vykresleny. Vykreslování parametrických křivek nevyžaduje žádné další zvláštní přístupy.

4.4 Implicitní křivky

Implicitní křivky jsou, co se týče vykreslování, snad nejzajímavější částí celého projektu. Křivky se vykreslují pomocí adaptivní interpolační mřížky, kterou se redukuje počet výpočtů, které jsou nutné k vykreslení celé křivky.

Naivním řešením, které by napadlo zřejmě každého by bylo projít celou vykreslovací matici a pro každý bod [*x*,*y*] spočítat hodnotu implicitní funkce *f*(*x*, *y*) v tomto bodě. Pokud by hodnota odpovídala požadované rovnosti, vykreslil by se bod, jinak by se pokračovalo dál. Tento přístup by sice byl funkční, ale velmi neefektivní. Pro vykreslení grafu o rozměru 640×480 pxl by bylo třeba vypočítat hodnotu funkce *f*(*x*, *y*) tři sta tisíckrát. Pro grafy většího rozměru by tato metoda byla i při výkonu dnešních počítačů zcela nepoužitelná – na vykreslení by se muselo čekat i několik sekund. Program proto používá speciální mřížku. Použití této mřížky snižuje počet potřebných vyhodnocení výrazu na pouhý zlomek ve srovnávání s naivním přístupem.

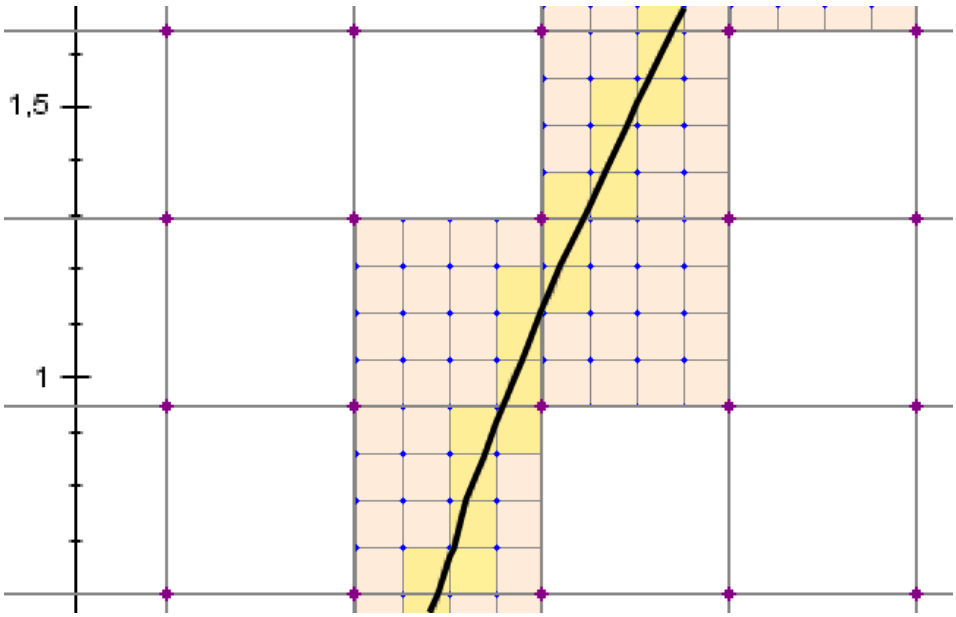
Nejprve je požadovaná rovnice převedena odečtením celé pravé strany do tvaru *f*(*x*, *y*) = 0. Celá plocha grafu je pak rozdělena na menší čtverce a ve všech vrcholech vzniklých čtverců jsou jejich souřadnice [*x*, *y*] dosazeny do rovnice a hodnota výrazu je v daném bodě vyčíslena a uložena. Tím máme k dispozici funkční hodnotu ve všech bodech, které jsou na obrázku č. 14 zvýrazněny fialovou tečkou.



Obrázek 14: Vykreslování implicitních křivek - krok 1

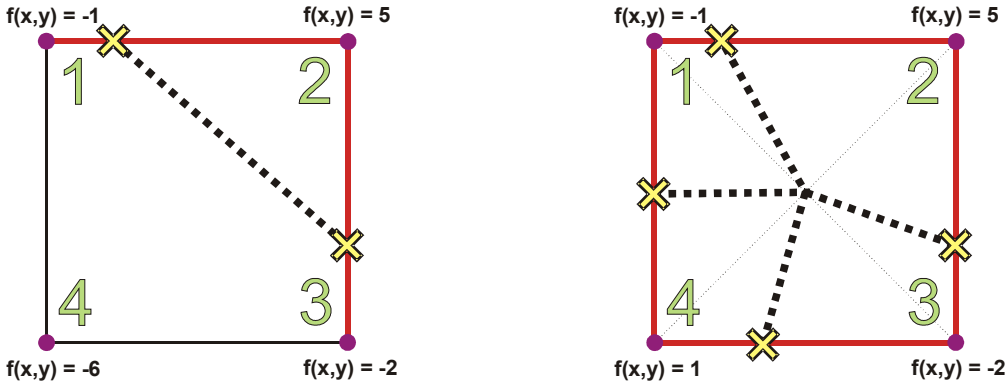
Po provedení výše zmíněného výpočtu jsou porovnány hodnoty ve všech sousedních bodech. Z teorie numerické matematiky víme [4], že je-li funkce *f(x)* spojitá na intervalu <*a*; *b*> a platí, že *f(a)·f(b)*<0, pak v intervalu <*a*; *b*> leží alespoň jeden kořen rovnice. Toho můžeme využít a pomocí funkce *signum* kontrolovat funkční hodnoty ve vodorovných a svislých úsečkách, které tvoří hrany čtverců. Pro každý čtvereček tvořící mřížku tak provedeme toto porovnání výsledků funkce *signum,* a pokud zjistíme, že se výsledek liší, můžeme prohlásit, že úsečka tvořená dvěma body mřížky protíná hledanou implicitní křivku. Tyto úsečky jsou v obrázku č. 14 červeně zvýrazněny.

Abychom dosáhli vyšší přesnosti, zmíněnou metodu opakujeme uvnitř čtverců, které jsme vyhodnotili jako čtverce, přes které hledaná implicitní křivka prochází. Obrázek č. 15 znázorňuje rekurzivní zavolání metody.



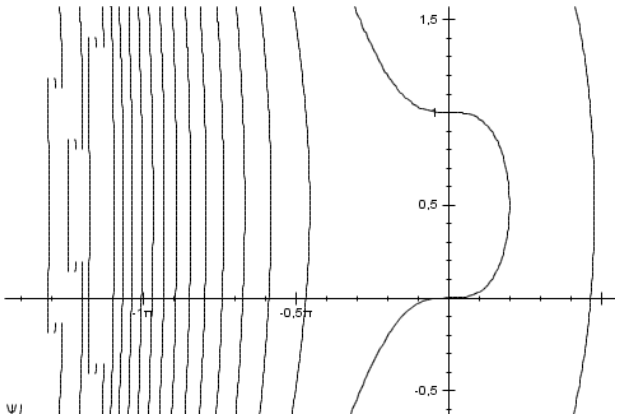
Obrázek 15: Vykreslování implicitních křivek - krok 2

Rekurzivní volání opakujeme až do zadané úrovně. Experimentálně jsem ověřil, že hloubka zanoření 3, se jeví jako ideální. V této části výpočtu víme, přes které čtverce křivka prochází. Nyní však potřebujeme zjistit přesnější souřadnice a přes čtverec nakreslit odpovídající část křivky. V ideálním případě bychom měli detekovat dvě hrany čtverce, přes které křivka prochází. Tato situace je znázorněna na obrázku č. 16 vlevo. Protože $y_1 = y_2$ a $signum(f(x_1,y_1)) \neq signum(f(x_2,y_2))$, víme, že hledaná implicitní křivka protíná úsečku mezi body 1-2. Protože víme, že funkční hodnota v bodě 1 je -1 a funkční hodnota v bodě 2 je 5, jsme schopni spočítat za předpokladu, že průběh křivky na tomto krátkém úseku lineárně aproximujeme, že průsečík se bude nacházet v jedné šestině vzdálenosti bodu 1 a 2 od bodu 1 (viz obrázek č. 16). Protože čtverec má čtyři strany, může se stát, že nalezneme průsečíky se dvěma hranami, ale i se čtyřmi. V takovém případě nastává problém, že není zcela jasné, jaké dvojice průsečíků se mají vzájemně pospojovat. Pokud je průsečík detekován na všech čtyřech hranách, vznikají celkem tři možnosti, jak tyto průsečíky pospojovat. Protože samotný čtvereček je v porovnání s velikostí plochy grafu velmi malý, můžeme si tento problém vyřešit tak, že každý z odhadnutých průsečíku spojíme se středem čtverce (viz obrázek č. 16 vpravo). Zkreslení, které tímto krokem vznikne, je dostatečně malé na to, abychom ho mohli zanedbat.



Obrázek 16: Vykreslení segmentu implicitní křivky

Samotné vykreslování má však jeden nedostatek. Pokud je vykreslovaná křivka taková, že se vedle sebe vyskytuje více čar a vzdálenost je mezi nimi příliš malá, může dojít k tomu, že již při prvním dělení na čtverečky je mylně označen čtvereček grafu jako takový, kterým křivka neprochází. K takovému rozhodnutí dojde velmi snadno, pokud má daná hrana čtverce s hledanou křivkou sudý počet průsečíků. Tento problém ilustruje obrázek č. 17, který zobrazuje implicitní křivku zadanou rovnicí $sin(x^3 + y^2 - y) = 0$. V levé části obrázku, kde by měl být hustý rastr tvořený oblouky křivky, jsou místo toho čtvercová bílá místa. Řešením tohoto problému může být pouze zmenšení počátečního rozměru čtverečku v mřížce.



Obrázek 17: Ilustrace chyby vzniklé při vykreslení implicitní křivky

4.5 Znovupoužitelnost vykreslovače grafů

Stejně jako kód vyhodnocovače výrazů, tak i samotné vykreslování grafů je navrženo tak, aby šlo snadno použít v dalších projektech. Pokud si do nového projektu naimportujeme balíky `cz.itpro.libs.graphs` a `cz.itpro.libs.math` můžeme použít následující kód pro vykreslení průběhu funkce $y = \sin(x)$ v novém okně aplikace.

```
// vytvoření rámce okna aplikace

JFrame frame = new JFrame();
```

```
// vytvoření tabulky symbolů
SymbolTable symbolTable = new SymbolTable();
// získání nezávislého symbolů x
SymbolTableItem x = symbolTable.getSymbol("x");
// vytvoření nového grafu
Graph graph = new Graph();
// vyvoření výrazu pro funkci y = sin(x)
Expression expression = new Sin(new Variable(x));
// vytvoření explicitní křivky
ExplicitCurve curve = new ExplicitCurve(expression,x);
// přidání křivky do grafu
graph.addGraphObject(curve);
// vyvoření vykreslovače grafu
GraphViewer graphViewer = new GraphViewer(graph);
// přidání zobrazovače do okna aplikace
frame.add(graphViewer);
```

5 Možnosti dalšího vývoje

Kocept programu je vyloženě postaven na požadavku snadné rozšiřitelnosti funkčnosti. Je vytvořena řada abstraktních tříd a dokonce jsou i předpřipraveny některé další funkce, které v této verzi programu ještě nebyly přes uživatelské rozhraní zpřístupněny. Takovými funkcemi jsou například vykreslování funkcí zadaných množinou souřadnic $[x, y]$ a výpočet aproximačních polynomů pro tyto body, nebo vykreslení grafického zobrazení výrazu včetně zlomků, odmocnin apod.

Co se týče dalších funkcí výpočtového jádra, bylo by vhodné rozšířit soubor vestavěných funkcí o funkce pro manipulaci s vektory a maticemi na základě použití stávajících polí, které program podporuje. Další zajímavou možností pro rozšíření programu by bylo vytvoření možnosti definovat vlastní uživatelské funkce pomocí interpretovaného programovacího jazyka. Část pro vykreslování grafů by bylo dobré rozšířit o možnost zakreslení samostatného bodu a možnost animace na základě nějakého parametru. Dále by pak bylo vhodné rozšířit rozhraní okna grafu o zobrazení na celou obrazovku[2] a o mód prezentace, který by šlo využít zejména pro výukové účely na středních školách. Jako další námět přidají v úvahu především 3D grafy.

Program budu zcela jistě i nadále rozvíjet. Konkrétní změny a vylepšení se budou odvíjet zejména od reakcí a námětů od uživatelů programu.

2 fullscreen

6 Závěr

Podle mého názoru se dokončením bakalářské práce podařilo vytvořit užitečný software, který bude moci používat široká veřejnost nejen ke studiu, ale i k běžně prováděným výpočtům na PC. Rozhraní programu se jeví jako přehledné a téměř intuitivně použivatelné. Program řeší některé nedostatky, kterými se řada programů s podobnou tématikou vůbec nezabývá. Jako příklad uvádím rozšířenou kontrolu nespojistosti křivek, vykreslování implicitních křivek, vykreslování derivace průběhu křivky apod. Skutečnou užitečnost a přínos programu však ukáže až čas a uživatelské ohlasy a připomínky. Věřím, že stávající uživatelé mého software SMath v3.x ocení příchod nové majoritní verze, která přináší řadu vylepšení a urychlení. Použitím Javy jako implementačního jazyka došlo také ke zpřístupnění programu pro uživatele operačních systémů Linux a Mac OS X. Požadavky zadání bakalářské práce se podařilo bez výhrad splnit.

Vývoj programu rozhodně není ukončen a na dalších verzích se bude pod hlavičkou ITPro CZ[3] nadále pracovat. Aktuální verze bude volně přístupná na internetu.

3 www.itpro.cz

Literatura

[1] Meduna A., Lukáš R.: Přednášky k předmětu IFJ 2005, FIT VUT Brno

[2] *Wikipedie: Otevřená encyklopedie: Euklidův algoritmus* [online]. c2007 [citováno 27. 04. 2007].
Dostupný z WWW: <http://cs.wikipedia.org/wiki/Euklid%C5%AFv_algoritmus>

[3] *Wikipedie: Otevřená encyklopedie: Derivace* [online]. c2007 [citováno 27. 04. 2007].
Dostupný z WWW: <http://cs.wikipedia.org/wiki/Derivace>

[4] Fajmon B., Růžičková I.: Matematika 3, Skriptum FEKT VUT Brno, 2005

[5] *Wikipedie: Otevřená encyklopedie: Shannonův teorém* [online]. c2007 [citováno 02. 05. 2007].
Dostupný z WWW: <http://cs.wikipedia.org/wiki/Shannon%C5%AFv_teor%C3%A9m>

[6] Skalický I.: SMath, Maturitní zkouška, Praktická zkouška z odborných předmětů, písemná práce, SPŠE Pardubice, 2004

[7] Skalický I: Středoškolská odborná činnost: SMath, SOČ 2003

Seznam příloh

Příloha A Stručný uživatelský manuál k programu

Příloha B CD se zdrojovými texty, binárními soubory pro spuštění a úplnou programovou dokumentací

Seznam obrázků

Obrázek 1: Část konečného automatu zpracovávající číselné literály.....9
Obrázek 2: Diagram tříd reprezentující tokeny.....10
Obrázek 3: Diagram tříd pro uchovávání hodnot symbolů.....13
Obrázek 4: Diagram tříd určených k reprezentaci matematického výrazu.....14
Obrázek 5: Příklad objektového stromu výrazu.....14
Obrázek 6: Postup vyhodnocení objektového stromu.....15
Obrázek 7: Příklad vyhodnocení lomeného výrazu.....17
Obrázek 8: Příklad výpočtu analytické derivace.....18
Obrázek 9: Základ konceptu objektového modelu pro vykreslování grafů.....21
Obrázek 10: Diagram tříd objektů grafu.....22
Obrázek 11: Chybné zobrazení funkce tangens.....23
Obrázek 12: Správné zobrazení funkce tangens.....23
Obrázek 13: Vznik aliasu u vykreslení funkce sinus.....24
Obrázek 14: Vykreslování implicitních křivek - krok 1.....25
Obrázek 15: Vykreslování implicitních křivek - krok 2.....26
Obrázek 16: Vykreslení segmentu implicitní křivky.....27
Obrázek 17: Ilustrace chyby vzniklé při vykreslení implicitní křivky.....27

Seznam tabulek

Tabulka 1: Přehled podporovaných číselných soustav a jejich zápisu..... 7
Tabulka 2: Použitá precedenční tabulka..... 12

Příloha A – Uživatelský manuál

Zápis matematických výrazů

Číselné literály

Číselné literály mohou být zadávány v dekadické, hexadecimální, oktálové nebo binární soustavě. Desetinná a záporná čísla lze zadávat pouze v dekadické soustavě.

Soustava	**Příklady hodnot**
dekadická	*123, 10.5, 1e6, 1e-6, 1.5e-10, 1.23e+2*
hexadecimální	*0xFF, 0x123, 0xABCDE*
oktálová	*0777, 0644, 010*
binární	*1111b, 0101b,100b*

Pole

Pole je uspořádaná množina výrazů, přičemž pole je samo o sobě také chápáno jako výraz. To umožňuje použití i vícerozměrných zanořených polí. Pole lze definovat jako sérii výrazů oddělených čárkou, která je uzavřena ve složených závorkách. Pro adresování jednotlivých prvků pole se používají hranaté závorky. Prvky jsou počítány od nuly.

Zápis	**Vysvětlení**
{}	prázdné pole
{1,2,3}	pole se třemi prvky
{{1,2},{3,4}}	pole 2×2 prvky
pole[0]	první prvek jednorozměrného pole
pole[size(pole)-1]	poslední prvek jednorozměrného pole

Proměnné

Názvem proměnné může být jakýkoli řetězec, který začíná písmenem anglické abecedy a dále obsahuje písmena anglické abecedy, číslice a podtržítka. Název se navíc nesmí shodovat s rezervovaným slovem (názvy vestavěných funkcí a jejich aliasy, konstanty, operátory). Do proměnné se přiřazuje pomocí operátoru *:=*.

Zápis	**Vysvětlení**
cislo:=5	do proměnné *cislo* ulož číslo pět

cislo:=cislo+1	inkrementuj hodnotu v proměnné *cislo*
pole:={1,2,3}	do proměnné *pole* ulož pole se třemi prvky {1, 2, 3}

Aritmetické operátory

Textová reprezentace	Popis a příklad použití
+	**součet** → výraz + výraz *3 + 2 = 5*
-	**rozdíl** → výraz - výraz *3 - 2 = 1*
*	**součin** → výraz * výraz *2 * 3 = 6*
/	**podíl** → výraz / výraz *5 / 2 = 2.5*
^	**rovná se** → výraz ^ výraz *2 ^ 8 = 256*
mod	**zbytek po celočíselném dělení** → výraz mod výraz *10 mod 3 = 1*
div	**celočíselné dělení** → výraz div výraz *10 div 3 = 3*
!	**faktoriál** → výraz! *5! = 120*
%	**procenta** → výraz% *10% = 0.1*

Logické operátory

Jako „nepravda“ je chápán libovolný výraz, jehož hodnota odpovídá nule, ostatní výrazy jsou chápány jako „pravda“.

Textová reprezentace	Popis a příklad použití
AND	**logický součin** → výraz AND výraz *1 AND 1 = 1*
OR	**logický součet** → výraz OR výraz *1 OR 0 = 1*
XOR	**exkluzivní logický součet** → výraz XOR výraz *1 XOR 1 = 0*
NOT	**logická negace** → NOT výraz *NOT 0 = 1*

Porovnávací operátory

Výsledek operace porovnání je vždy vyhodnocen jako „1“, pokud je porovnání pravdivé, pokud není je výsledek vyhodnocen jako „0“.

Textová reprezentace	Popis a příklad použití
>	**je větší než** → výraz > výraz *5 > 3 = 1*
>=	**je větší nebo rovno** → výraz >= výraz *5 >= 5 = 1*
<	**je menší než** → výraz < výraz *5 < 3 = 0*
<=	**je menší nebo rovno** → výraz <= výraz *5 <= 5 = 1*
=	**rovná se** → výraz = výraz *5 = 5 = 1*
!=, <>	**nerovná se** → výraz != výraz *5 != 5 = 0*

Goniometrické funkce

Hodnota úhel je vždy chápána jako hodnota zadaná v aktuálně zvolené úhlové jednotce.

Textová reprezentace a aliasy funkce	Popis a příklad použití
sin	**sinus** → sin(úhel) *sin(90) = 1*
cos	**kosinus** → cos(úhel) *cos(90) = 0*
tan, tg	**tangens** → tan(úhel) *tan(45) = 1*
cot, cotan, cotg	**kotangens** → cot(úhel) *cot(45) = 1*
sec	**sekans** → sec(úhel) *sec(0) = 1*
csc, cosec	**kosekans** → csc(úhel) *csc(90) = 1*

Cyklometrické funkce

Vrácená hodnota je vždy převedena do akutálně zvolené úhlové jednotky.

Textová reprezentace a aliasy funkce	Popis a příklad použití
asin, arcsin	**inverzní sinus** → asin(výraz) sin(90) = 1
acos, arccos	**inverzní kosinus** → acos(výraz) cos(90) = 0
atan, atg, arctan, arctg	**inverzní tangens** → atan(výraz) tan(45) = 1
acot, acotan, acotg,	**inverzní kotangens** → acot(výraz)

arccot, arccotan, arccotg	cot(45) = 1
asec, arcsec	**inverzní sekans**→ asec(výraz) sec(0) = 1
acsc, acosec, arccsc, arccosec	**inverzní kosekans** → acsc(výraz) csc(90) = 1

Hyperbolické funkce

Hodnota úhel je vždy chápána jako hodnota zadaná v aktuálně zvolené úhlové jednotce.

Textová reprezentace a aliasy funkce	Popis a příklad použití
sinh	**hyperbolický sinus** → sinh(úhel) *sinh(0) = 0*
cosh	**hyperbolický kosinus** → cosh(úhel) *cosh(0) = 1*
tanh, tgh	**hyperbolický tangens** → tanh(úhel) *tanh(0) = 0*
coth, cotanh, cotgh	**hyperbolický kotangens** → coth(úhel) *coth(0) = ∞*
sech	**hyperbolický sekans** → sech(úhel) *sech(0) = 1*
csch, cosech	**hyperbolický kosekans** → csch(úhel) *csch(0) = ∞*

Hyperbolometrické funkce

Vrácená hodnota je vždy převedena do akutálně zvolené úhlové jednotky.

Textová reprezentace a aliasy funkce	Popis a příklad použití
asinh, arcsinh	**inverzní hyperbolický sinus** → asinh(výraz) *asinh(0) = 0*
acosh, arccosh	**inverzní hyperbolický kosinus** → acosh(výraz) *acosh(1) = 0*
atanh, atgh, arctanh,arctgh	**inverzní hyperbolický tangens** → atanh(výraz) *atanh(0) = 0*
acoth, acotanh, acotgh, arccoth, arccotanh, arccotgh	**inverzní hyperbolický kotangens** → acoth(výraz) *acoth(100) = 0.57*
asech, arcsech	**inverzní hyperbolický sekans** → asech(výraz) *asech(1) = 0*
acsch, acosech, arccsch, arccosech	**inverzní hyperbolický kosekans** → acsch(výraz) *acsch(-0.5) = -27.57*

Další aritmetické funkce

Textová reprezentace a aliasy funkce	Popis a příklad použití
ln	**přirozený logaritmus** → ln(výraz) *ln(e) = 1*
log, log10	**dekadický logaritmus** → log(výraz) *log(10) = 1*
exp	**mocnina Eulerova čísla** → exp(výraz) *exp(1) = 2.72*
sqr	**druhá mocnina čísla** → sqr(výraz) *sqr(3) = 9*
sqrt	**druhá odmocnina čísla** → sqrt(výraz) *sqrt(9) = 3*
pow, power	**mocnina čísla** → pow(základ, exponent) *pow(2,4) = 16*
root	**odmocnina čísla** → root(výraz, odmocnina) *root(27,3) = 3*
rand, random	**náhodné číslo** → rand(), rand(od), rand(od, do) *rand() = čísla 0-1, rand(10) = čísla 0-10, rand(10,20) =čísla 10-20*
round	**zaokrouhlit** → round(výraz) *round(2.5) = 3*
floor	**zaokrouhlit dolů** → floor(výraz) *floor(2.5) = 2*
ceil	**zaokrouhlit nahoru** → ceil(výraz) *ceil(-2.5) = -2*
combin	**kombinační číslo** → combin(výraz, výraz) *combin(2, 5) = 10*
fact, factorial	**faktoriál** → fact(výraz) *fact(5) = 129*
sign, signum	**znaménkové číslo** → sign(výraz) *sign(-27) = -1*

Funkce pro manipulaci s poli

Všem funkcím pro manipulaci s poli může být jako argument zadán buď výčet výrazů oddělený čárkami, nebo výraz, či proměnná typu pole.

Textová reprezentace a aliasy funkce	Popis a příklad použití
sum	**suma** → sum(pole) *sum(1,2,3) = 6*
avg	**aritmetický průměr** → avg(pole) *avg(1,2,3) = 2*
max	**maximální hodnota** → max(pole) *max(1,2,3) = 3*

min	**minimální hodnota** → min(pole) *min(1,2,3) = 1*
count	**počet atomických prvků pole** → count(pole) *count({1,2,{3,4}}) = 4*
size	**počet prvků pole** → size(pole) *size({1,2,{3,4}}) = 3*
gcd	**největší společný dělitel** → gcd(pole) *gcd(18,24) = 6*
lcm	**nejmenší společný násobek** → lcm(pole) *lcm(18,24) = 72*

Úhlové funkce

V příkladech uvažujme aktivní úhlovou jednotku stupně.

Textová reprezentace a aliasy funkce	Popis a příklad použití
deg	**úhel ve stupních** → deg(výraz) *deg(180) = 180*
rad	**úhel v radiánech** → rad(výraz) *rad(PI) = 180*
grad	**úhel v gradech** → grad(pole) *grad(200) = 180*

Derivační a integrační funkce

Pro numerickou integraci je vyžadován aktivní režim úhlové jednotky RAD.

Textová reprezentace a aliasy funkce	Popis a příklad použití
diff	**analytická derivace** → diff(výraz, proměnná) *diff(sin(x),x) = cos(x)*
integ, integral	**numerická integrace** → integ(výraz, proměnná, od, do) *integ(sin(x),0,PI) = 2*