Vizualizace grafu matematické funkce

Tomáš Maršálek
marsalet@students.zcu.cz

28. prosince 2011

1 Zadání

Naprogramujte v ANSI C přenositelnou1 konzolovou aplikaci, která jako vstup načte z parametru na příkazové řádce matematickou funkci ve tvaru $y = f(x)$, provede její analýzu a vytvoří soubor ve formátu PostScript s grafem této funkce na zvoleném definičním oboru.

2 Analýza úlohy

Abychom mohli zobrazit graf funkce jako například x ˆ 2 + sin(x), musíme být nejdříve schopni vyhodnotit tuto funkci numericky. Tedy musíme ve výrazu rozpoznat jednotlivá čísla, proměnné nebo operátory a rozpoznat jejich význam. Standardním postupem při parsování vstupního řetězce je rozdělení úlohy na několik po sobě následujících částí tak, aby se případné chyby ve vstupu odhalily v příslušné části a nepropagovaly se do následující. Každá část se tedy stará pouze o chyby, které se jí týkají.

2.1 Lexikální analýza

Místo abychom při vyhodnocování výrazu postupně hledali čísla nebo operátory, ponecháme tuto úlohu specializovanému nástroji, takzvanému scanneru nebo také lexeru. Nadefinujeme jednotlivé symboly a necháme scanner, aby je ve vstupním řetězci rozpoznal, nebo aby případně rozpoznal lexikální chyby. Význam celé této fáze je v zjednodušení nadcházející fáze, která je složitější, a už se nebude muset zabývat problémy typu jestli je symbol skutečně číslo nebo jestli symbol „sin“ je ve skutečnosti „sinh“ a podobně.

2.2 Syntaktická a sémantická analýza

Tyto dvě fáze jsou zde kombinované v jednu. Vytvoříme přeloženou datovou strukturu, se kterou bude vyhodnocování výrazu víceméně triviální. Syntaktická analýza najde chyby týkající se skladby výrazu, jako například dvě čísla nebo znaky následující po sobě, chybějící závorky, a podobně. Jednotlivým symbolům přiřadíme jejich význam, tj. číslům jejich numerickou hodnotu a operátorům jejich příslušnou unární nebo binární funkci.

2.3 Postfixová notace

Výsledná datová struktura je posloupnost symbolů v postfixové nebo také reverzní polské notaci. Výhoda tohoto zápisu oproti klasické infixové notaci je absence závorek a jednoduchost vyhodnocení výrazu. Při vyhodnocování totiž vždy když narazíme na operátor, máme jistotu, že všechny předcházející znaky, které operátor vyžaduje, jsou čísla.

Příklad infixové a postfixové notace.

infix	postfix
1 + 1	1 1 +
1 + 2 * 3	1 2 3 * +
(1 + 2) * 3	1 2 + 3 *
x * sin (x ˆ 2)	x x 2 ˆ sin *
1 - 2 - 3 + 4 ˆ 5 ˆ 6	1 2 - 3 - 4 5 6 ˆ ˆ +

Metoda na převedení infixového do postfixového zápisu je např. Dijkstrův Shunting yard algoritmus.

2.4 Shunting yard

Při konverzi musíme mít na paměti přednosti jednotlivých operátorů. Výraz $1 + 2 * 3$ se musí přeložit jako $1\ 2\ 3\ *\ +$ a ne jako $1\ 2\ +\ 3\ *$. Dále také levou nebo pravou asociaci nekomutativních operátorů, pokud ve výrazů chybí závorky. Například operátor odčítání upřednostňuje levé uzávorkování $(1 - 2 - 3 = ((1 - 2) - 3))$ oproti umocňování, které naopak upřednostňuje uzávorkování zprava $(2\ \hat{}\ 3\ \hat{}\ 4 = (2\ \hat{}\ (3\ \hat{}\ 4)) = 2^{3^4})$.

Algoritmus používá pomocný zásobník na odkládání operátorů, dokud nemá jistotu o jejich správném umístění v postfixu. Čteme infixový výraz zleva doprava po jednotlivých symbolech a vždy když narazíme na symbol typu číslo nebo proměnná, pouze ho uložíme do výsledné výstupní fronty. Pokud narazíme na operátor, všechny operátory, které dosud leží nahoře v zásobníku a vážou se těsněji (mají vyšší precedenci) než právě vytažený symbol, můžeme přidat do výsledné fronty. Právě vytažený symbol pak pouze vložíme na zásobník. Až nám dojdou všechny symboly, pouze vytáhneme všechny operátory ze zásobníku a v tomto pořadí je přidáme do výsledné fronty.

Jako příklad použijme výraz 1 / 2 ˆ 3 + 4.

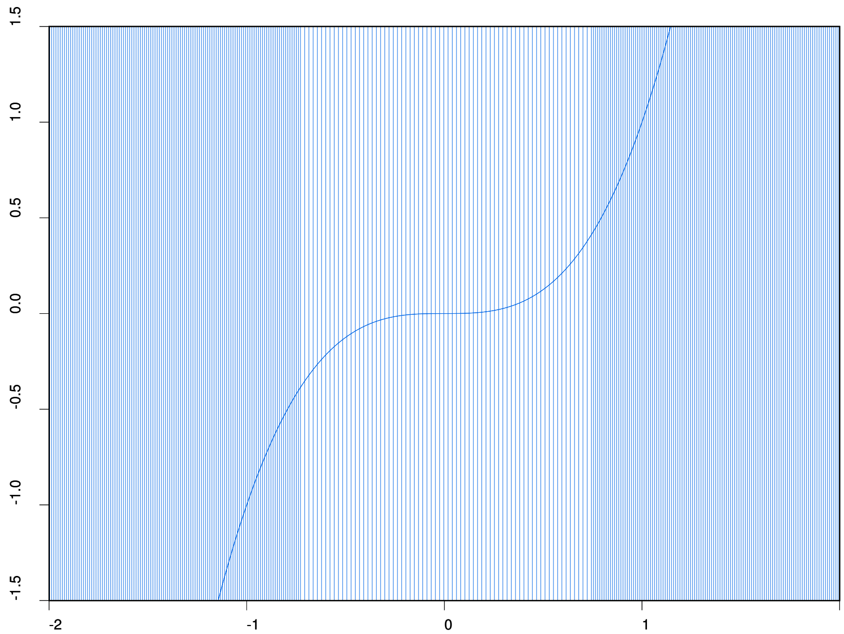
symbol	výsledek	zásobník	akce
1	1		číslo, pouze přidáme do fronty
/	1	/	dosud žádný operátor v zásobníku
2	1 2	/	číslo, pouze přidáme do fronty
ˆ	1 2	/ ˆ	/ má slabší vazbu než ˆ, necháme být
3	1 2 3	/ ˆ	číslo, pouze přidáme do fronty
+	1 2 3 ˆ /	+	ˆ a / mají silnější vazbu než +
4	1 2 3 ˆ / 4	+	číslo, pouze přidáme do fronty
konec	1 2 3 ˆ / 4 +		konec, přidáme všechny operátory

2.5 Zobrazení grafu

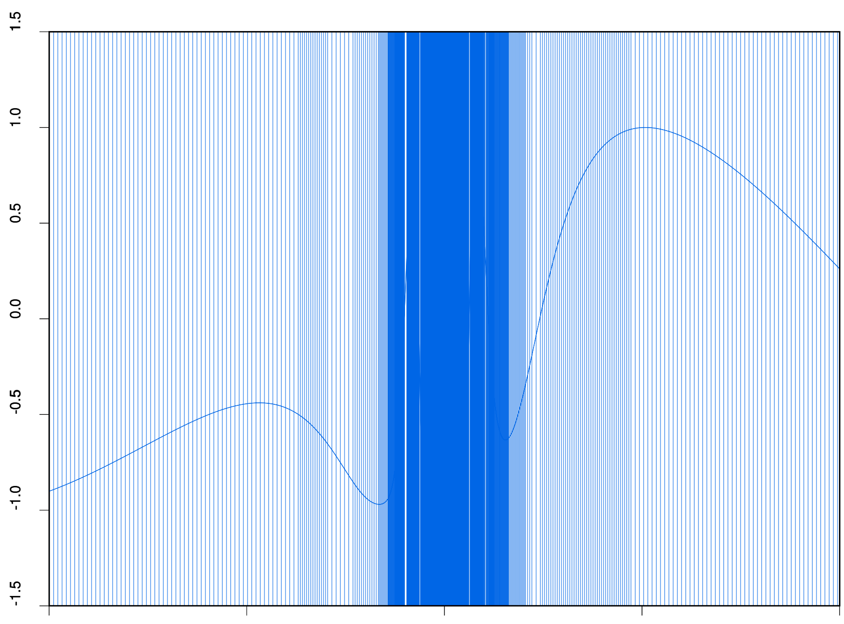
Jednoduchým řešením jak zobrazit funkci je vyhodnotit vždy dva sousední body a spojit je úsečkou. Důležité je pouze zvolit dostatečně detailní rozdělení osy x, aby graf nebyl kostrbatý.

Úsporná metoda, která řeší kostrbatost zobrazené funkce je adaptivní vyhlazování. Šetření zobrazenými přímkami oceníme zejména u jednoduchých funkcí, protože rozhodně nepotřebujeme, aby graf funkce $f(x) = x$ byl stejně detailní jako graf funkce $f(x) = sin(1/x)$, který je kolem nuly velmi detailní. Taktika je poměrně jednoduchá. Mějme zvolené nějaké počáteční rozdělení osy x a pokud při zobrazování jednotlivých úseček zjistíme, že by došlo k příliš velkému skoku ve sklonu dvou sousedních úseček (výpočetně by druhá derivace přesahovala jistý zvolený práh), rozdělíme interval mezi x_i a x_{i+1} na polovinu. Samozřejmě musíme mít nějaký limit tohoto půlení, protože u funkcí typu $f(x) = sin(1/x)$ by došlo kolem nuly k nekonečnému půlení. Stejným výpočtem se snažíme zajistit co nejdelší interval, aby nedošlo ke zbytečnému plýtvání.

Rozdělení osy x je o poznání hustší, pokud je graf v tomto místě detailnější, jak je vidět v porovnání dvou různých funkcí.

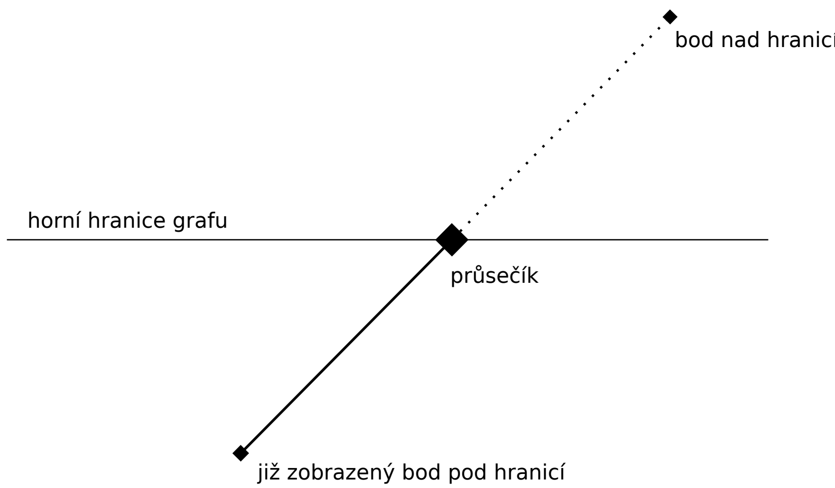


Obrázek 1: Rozdělení osy x grafu funkce x^3



Obrázek 2: Rozdělení osy x grafu funkce $\sin\left(x + \cos\left(\frac{1}{x}\right)\right)$

Zřejmým problémem jsou funkce, které v některých bodech zadaného intervalu rostou limitně do nekonečna, případně nejsou na daném intervalu definovány. Případ, kdy funkce pokračuje mimo zadanou hranici, případně až do nekonečna je celkem jednoduchý. Aby nedošlo k useklým grafům poblíž dolní nebo horní hranice, stačí vypočítat průsečík horní nebo dolní hranice s úsečkou mezi posledním bodem a bodem, který už by byl mimo graf a vést kratší úsečku do tohoto průsečíku.



Obrázek 3: Zamezení úniku grafu mimo hranice

Horší jsou funkce, které nejsou definovány v určitém místě. V některých případech dochází k nežádoucímu vynechání počáteční nebo konečné části grafu. Příkladem je jakýkoliv logaritmus, při rozdělení osy x, které vynechá bod $x = 0.0$ dojde k vynechání první úsečky. Ta pak vzhledem k velkému sklonu logaritmu v tomto místě viditelně schází. Implementace funkcí s nenadefinovanými úseky v matematické knihovně v takových případech vracejí hodnotu NaN. Výpočet průsečíku s takovým číslem nedává smysl a předefinování knihovních funkcí na vlastní, které by vracely alespoň nekonečna namísto NaN způsobuje jiné nežádoucí artefakty. Například předefinování logaritmů pro záporné hodnoty na záporné nekonečno funguje dokud nezobrazíme nějakou transformaci tohoto logaritmu na reálné hodnoty (např. $f(x) = \exp(\ln(x))$). Pro jednoduchost je tedy možná lepší NaN hodnoty pouze nezobrazovat.

3 Implementace

Lexer je konstruován jako deterministický automat, který rozpoznává celá čísla v šestnáctkové, desítkové a osmičkové soustavě, floating point čísla, proměnnou x, operátory a závorky. Mezery rovnou zahazuje. V případě chyby posílá chybový token do další fáze. Token je zde struktura, která si uchovává pozici a délku nalezeného podřetězce a samotný řetězec pro případný výpis chyby.

Parser je implementace Shunting yard algoritmu, která navíc kontroluje syntaktické chyby. Výsledek je vrácen jako posloupnost symbolů, které ale musí být po skončení používání dealokovány. Pro zjednodušsení jsou všechny tokeny a přeložené symboly uchovány v pomocných polích, aby na konci došlo k jejich bezchybnému dealokování. Symbol je výsledný přeložený token, který je zde implementován jako polymorfní struktura (uchovává si svůj typ a případně data vztahující se k jednotlivým typům). Mohla by být implementována i efektivněji, například pomocí *union*, ale v ANSI C není podporován anonymní *union*. S pojmenovaným by zdrojový kód vypadal poněkud těžkopádně a navíc k příliš velkému ušetření paměti by rozhodně nedošlo. Zvolil jsem tedy raději čitelnost kódu.

Pomocnou datovou strukturu zde tvoří pouze zásobník, který se využívá při Shunting yard algoritmu a poté při kontrolním průchodu výsledným postfixovým výrazem, kdy se potvrdí, že výraz bude skutečně možné vyhodnotit. Zde se odstraní chyby související s n-aritou operátorů.

Zobrazení grafu je zajištěno dalším odděleným modulem. Ten nejprve zkonstruuje hlavičku postscriptového souboru podle specifikace postscriptu.

Poté je funkce vyhodnocena v klíčových bodech, které jsou poté transformovány na souřadnice postscriptového formátu stránky a spojeny úsečkami tak, jak bylo popsáno v analýze úlohy.

Dalším krokem je vytvoření rámečku grafu a pro přehlednost rozdělení obou os. Velikost jednotlivých dílků je vybrána tak, aby rozdělení osy nebylo příliš husté ani řídké, a podle konvence, že dílky mají velikost pouze 1, 2 nebo 5 krát mocnina deseti.

Nakonec jsou zapsány závěrečné postscriptové komentáře a soubor je uzavřen.

Volitelný parametr popisující definiční obor a obor hodnot je analyzován stejným lexerem, ale má vlastní parsovací funkci.

4 Uživatelská příručka

Překlad ze zdrojových kódů provedeme pomocí příkazu make v kořenovém adresáři. Program spustíme spuštěním souboru graph.exe s parametry funkce, výstupní soubor a volitelně rozsah osy x a y, jak je naznačeno v usage stringu programu: Aritmetický výraz může obsahovat běžné binární operátory (+, -, *, /, ^) jednoduché závorky „()“, proměnnou „x“, a funkce implementované v knihovně math.h (). Konstanty můžou být zadány jako celá čísla v osmičkové, desítkové nebo šestnáctkové soustavě, nebo jako floating point čísla v desítkové soustavě (tak jak jsou rozpoznány jazykem C).

```
usage: graph.exe FUNCTION FILE.ps [LIMITS]
```

Příklad použití:

```
graph.exe "sin(1/x^2)" vystup.ps
Do souboru vystup.ps je zapsán graf funkce
sin(1/x^2) s vestavěným definičním oborem a oborem hodnot x, y \elem [-10, 10].
graph.exe "sin(1/x^2)" vystup.ps -3:3:-2:2
Stejný příkaz, pouze použije jiný
rozsah pro definiční obor (x \elem [-3, 3]) a obor hodnot (y \elem [-2, 2]).
```

Rozsahy jsou jsou specifikovány dobrovolným parametrem, který má formát x1:x2:y1:y2, kde x1 a y1 musí být menší než x2, respektive y2. Vygenerovaný graf je soubor typu encapsulated postscript (.eps) verze 3, může být použit přímo v typografickém nástroji LaTeX.

5 Závěr

Největší překážkou nakonec bylo možná unární mínus, přestože vypadá nevinně. Použitá metoda na parsování výrazu ho zřejmě neumí přeložit jinak, než mu dát nejvyšší precedenci. Přesvědčila mě o tom i skutečnost, že výrobci kalkulátorů a software používající shunting yard algoritmus na vyhodnocování aritmetických výrazů používají právě toto řešení [?]. Možným vylepšením by bylo použít sofistikovanější parsování převedením na strom (arithmetic tree notation), nicměně tento způsob vypadá komplikovaně, co se týče správného zacházení s precedencemi operátorů. Tato metoda by byla náročnější na paměť, což může být v prostředí s omezenou pamětí (kapesní kalkulátor) kritické. Testování proběhlo pro většinu možných případů, všechny dopadly úspěšně. Je zajištěno, aby nedocházelo k únikům paměti, přestože paměťová náročnost programu je minimální a dealokování celého bloku operačním systémem by bylo efektivnější. Stejně jako v každém softwaru, i zde se můžou objevit drobné chyby, prakticky nelze zajistit, aby byl program perfektní. Zadání bylo dle mého subjektivního názoru splněno kompletně. Vygenerované grafy jsou esteticky hezké a díky použitým metodám i úsporné. Je možné je použít jako doprovodné obrázky v nástroji LaTeX.