Gymnázium Christiana Dopplera, Zborovská 45, Praha 5

ROČNÍKOVÁ PRÁCE

Funkcionální programování

Vypracoval: Ivan Žižka
Třída: 8.M
Školní rok: 2018/2019
Seminář: Seminář z programování

Prohlašuji, že jsem svou ročníkovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím s využíváním práce na Gymnáziu Christiana Dopplera pro studijní účely.

V Praze dne 27.2.2019 Ivan Žižka

Obsah

1. Úvod

Cílem práce je vytvořit dva programy s identickou funkcí, ale každý vytvořený v jiném programovacím jazyce a zcela jinou metodou. Oba programy budou převádět prefixový zápis na infixový, s tím že první z těchto dvou bude tvořen procedurálně a to v jazyce C. Druhý program bude tvořen v čistě funkcionálním programovacím jazyce Haskell. Hlavním cílem srovnat odlišné styly programování a zároveň ukázat výhody a nevýhody funkcionálních programovacích jazyků.

2. Teorie funkcionálního programování

2.1 Historie

Čistě funkcionální programovací jazyky vycházejí z konceptu, který navrhl ve 30. letech 20. století britský matematik Alonzo Church. Ten vytvořil lambda kalkul jakožto matematickou teorii funkcí. Church vytvořil také jednu z nejvýznamnějších tezí celého funkcionálního programování, tato teze se nazývá Church-Turingova teze. Ta tvrdí, že každý možný výpočet lze uskutečnit algoritmem běžícím na počítači, je-li dostatek paměti a času.

Lambda kalkul je čistě teoretický model, ale i tak na něm staví většina čistě funkcionálních programovacích jazyků. Lambda kalkul analyzuje funkci a nikoliv z hlediska původního matematického smyslu zobrazení z množiny, ale jako metodu výpočtu. [citace - doplním později - wikipedie Lambda kalkul] Lambda kalkul tvoří páteř takzvaných Turingovýh strojů. Občas je označována jako univerzální matematický programovací jazyk.

Prvním jazykem, který obsahoval funkcionální část, byl LISP, který byl vytvořen pro potřeby IBM v 50. letech minulého století. LISP se do dnes vyučuje na většině vysokých škol v USA, které se zaměřují na studium informatiky, a je brán za jednu z nejdůležitějších schopností každého absolventa takovéto vysoké školy. Prvním čistě funkcionálním programovacím jazykem byla Miranda, tento jazyk byl vyvinut Davidem Turnerem. Byl komerčně podporován a snažil se odpoutat od závislosti na jiných programovacích jazycích. Krátce po vydání jazyku Miranda, proběhla světová konference o funkcionálním programování, ze které vzešel požadavek na standardizaci funkcionálních programovacích jazyků. Z tohoto důvodu byla sestavena speciální komice, která vyprodukovala čistě funkcionální programovací jazyk Haskell.

2.2 Funkcionální programování

Funkcionální programování se spíše než klasickým programovacím jazykům podobá matematickému zápisu funkce. Hlavním princip je založen na zápisu programu ve tvaru výrazu. Ten je následně zjednodušován až do takové míry, kdy už nelze být zjednodušen. Tento tvar je výsledkem tohoto výrazu.
Příklad:
$(2 * 5)$ Tento výraz není nic jiného než funkce násobení (ta vyžaduje dva argumenty, které mezi sebou vynásobí), jehož výsledkem je číslo 10. Číslo 10 nelze dále zjednodušit a proto považujeme tento výraz za výsledek.

Program lze, a dokonce je to vyžadováno, dělit do funkcí. V deklarativním programování mohou funkce přijímat jako argument jiné funkce. To zajišťuje plynulost a absenci pomocných proměnných. Funkce často obsahují jen několik řádků kódu. Zajímavou vlastností funkcionálních jazyků je pohled na funkce jako takové. Ty jsou chápány jako proměnné

v imperativních programovacích jazycích. Například v jazyku C, pokud nebereme v potaz ukazatele, je zcela nemyslitelné vytvořit pole funkcí. Na druhou stranu ve funkcionálním programovaní je tato myšlenka naprosto validní a uskutečnitelná.

2.2.1 Rozdíl mezi funkcionálním a imperativním programováním

Základní jednotkou funkcionálních jazyků je výraz. Ten se dále zjednodušuje až do té doby, kdy ho nelze dále zjednodušit.

Na druhou stranu u imperativních jazyků je základní jednotkou příkaz. Rozdílem je, že tyto příkazy nemají obvykle argument (hodnotu). Proto způsobem, jakým si příkazy vyměňují data, musí být skrze proměnnou. Proměnnou se rozumí stavový vektor, který lze v průběhu běhu programu měnit, hodnota vektoru je dána během programu. Existencí stavů vznikají jeden závažný důsledek.

Příklad:
Funkce sum, která pro každé celé kladné číslo n spočítá součet všech čísel od nuly do čísla n .

```
int sum (int n) {
  int i;
  int s = 0;
  for (i = n; i >= 0; i --)
    {
    s = s + i;
    }
  return s;
}
```

2.1: jazyk C

Důsledkem tohoto programování je změna proměnných i a s v průběhu chodu programu. Při každém otočení cyklu se změní hodnota i a s.

U funkcionálních jazyků se tato vlastnost nevyskytuje. To je dáno z deklarace čistě funkcionálních jazyků, kde jestliže se ve stejném kontextu vyskytuje tentýž výraz, tak bude mít vždy stejný výsledek. Tuto vlastnost nazýváme referenční transparentnost.

```
sum :: Int -> Int
sum n = if n == 1 then 1 else n + sum (n - 1)
```

2.2: jazyk Haskell

Na tomto příkladu můžeme vidět, že n bude v určitém kontextu existovat jen jedno a bude neměnné. I kdybychom chtěli v průběhu programu změnit hodnotu n, tak nemůžeme.

Deklarativní jazyky nemají prostředek, který by jim tuto manipulaci zajistil. Mohli bychom namítnout, že při vynořování z funkce *sum* se mění hodnota proměnné *n*. Opak je ale pravdou, ve funkci *sum* existuje vždy jen jedno konkrétní *n*, které má s předchozím n společný jen název, ale jeho hodnota je jiná.

Zajímavou vlastností imperativních jazyků je například to, že v nich můžeme simulovat podobné procesy jako v deklarativních jazycích.
Příklad:

```
int sum(int n){
    int souc;
    if( n == 1 ){
        return 1;
    }
    else{
        souc = n + sum( n - 1 );
    }
return souc;
```

2.3: jazyk C

I když je tato funkce napsána v imperativním jazyce, tak v ní probíhá stejná posloupnost akcí jako u příkladu s funkcionálním jazykem. Nefunkcionální jazyky mají prostředky, kterými lze simulovat chování funkcionálních jazyků, ale ne všechny funkce se takto dají simulovat.

Deklarativní jazyky neznají pojem proměnné ani přiřazení do ní. Proměnné mohou být pouze vázány na argumenty funkcí. To znamená, že argumentem funkce *sum* může být výraz: $(5+7)$ ten je následně vázán na argument funkce *n*. Z tohoto důvodu můžeme říci, že $n = 5 + 7$, ale nejedná se v tomto případě o proměnnou. Vazbě na argument funkce se říká prostředí. V některých případech může absence stavových vektorů vest ke zkrácení řešení určitého problému.

2.2.2 Lambda kalkul

Základní prvky tohoto univerzálního matematického jazyka jsou tři: proměnná, aplikace a abstrakce. Proměnná je název pro hodnotu, kterou nelze blíže specifikovat. Jsou označeny identifikátory, často to jsou písmena abecedy. Abstrakce, je definice funkce. Zapsání takovéto abstrakce si lze nejlépe představit na příkladu: $f(x) = x-5$, tuto funkci můžeme také zapsat pomocí lambda kalkulu $\lambda x.x-5$ Výraz (proměnná) mezi λ a . je parametr funkce, parametrů může být i více, ale v tomto případě máme jen jeden. Výraz za tečkou značí definici funkce (tělo).

Problém nastává, když máme právě více parametrů než jeden. Jelikož lambda kalkul

dokáže operovat pouze s jednou. I tento problém je jednoduše řešitelný, například funkci o dvou parametrech, která má předpis: $g(x, y) = x * y$ můžeme zcela beztrestně přepsat takto: $\lambda x.\lambda y.x * y$ tento zápis je možno zkrátit na: $\lambda x\ y.\ x * y$. Aplikace, toto je výraz pro volání funkce - $f(x) = x - 5$. Například funkci f zavoláme s parametrem 5, čili se za proměnnou x dosadí číslo 5. Výsledkem této funkce bude 0. Když se koukneme na volání funkce z blízka, tak zjistíme že nejdříve je volána funkce f a následně argument 5. Tento postup se též dá zapsat jako $(\lambda x.x - 5)5$

Často se tyto tři pojmy schovají za značení *E*.

Vázané a volné proměnné

Proměnná je v λ-výrazu vázaná, pokud se jedná o parametr nějaké funkce, takže např. ve výrazu ($\lambda x.\ y\ x$) je x vázaná proměnná. [citace `https://publications.petrzemek.net/articles/PZ_-_IPP-Lambda-Kalkul.pdf`] Protikladem k vázané proměnné je volná, například: ($\lambda x.\ y\ x$). Zde je y volnou proměnnou. Obecně řečeno proměnná se vždy váže k nejbližší lambdě k ní vlevo. V lambda výrazech je nutno respektovat závorky, ty určují nejen oddělení parametru, ale také říkají, ke které lambdě se proměnná bude vázat. To si lze jednoduše představit na příkladu: ($\lambda x.\ (\lambda x.\ x - 2)\ z$), kde x je proměnná vázaná a z proměnnou volnou. Taktéž x se váže k λ které je v závorce s ním.

2.2.3 Higher-order funkce

Tímto pojmem se označují takové funkce, které splňují alespoň jednu ze dvou podmínek. Těmito podmínkami se rozumí, že funkce může přijímat jako argument jinou funkci, druhou podmínkou je že výstup z funkce je jiná funkce. Příklad, který je nám blízký, můžeme nalézt v matematice a to u funkcí derivace a integrace. Tyto funkce mají za argument jinou funkci a výsledkem je funkce, která vychází z původního argumentu.

Příkladem takovéto funkce může být (v Haskellu) funkce *map*, která vyžaduje dva argumenty, z níž je jeden funkce *f* a druhý list *l*. Výsledkem je list, na který byla aplikována funkce *f*.

2.2.4 Monády

Monády jsou druhem abstrakce datového konstruktoru (type class), tyto konstruktory uzavírají určitou část logiky programu. Monády umožňují programátorovi vytvářet složité funkce, tím je myšleno řetězení funkcí. V minimalistickém principu jde o programovou strukturu, která umožňuje určitý výpočet. Monády jsou často zaváděny z důvodu toho, aby se předešlo vedlejším efektům funkcí.

Čistě obecně jde o konstrukci na úrovni teorie kategorií. Důležité je zmínit, že monády jsou často použity v kódu, aniž bychom si to uvědomovali. Jedno z takových to použití je řešení vstupu a výstupu programu, například do konzole. Myšlenka za touto funkcí je taková,

že funkci IO je předáván stav programu a následně opět vrácen vnější svět. Mohlo by se zdát, že se jedná o jakési implementování vlastností z imperativních programovacích jazyků. Není tomu tak vždy, jen v kontextu IO se tato představa nabízí. Monády jsou zárukou toho, že je zachována čistá funkcionalita jazyka bez vedlejších efektů, ale zároveň nám umožňují využívat vlastností, které mají vedlejší efekty.

Třída monád, podporuje dva operátory jeden z nich je operátor `bind` ten se značí (>>=) a druhou `return` ten vrací monádu.

IO - vstup a výstup

Touto funkcí se rozumí monadická funkce, která může být použita různými způsoby. Prvním z nich je IO s datovým typem například: *IO Char*. Druhým způsobem použití může být *IO()* takto definovaná funkce nevrací žádnou hodnotu. Práce ve funkci, která je definována pomocí *IO()*, musí probíhat až za klíčovým slovem *do*. Toho se dá využít například při načítání ze souboru nebo vypisování do konzole.

Maybe

Monáda *Maybe* má tu vlastnost, že dokáže vracet dva různé výstupy, jedním z nich je *Just x* a druhým *Nothing*. Je definována takto:

```
data Maybe x = Just x | Nothing
return x = Just x
Nothing >>= f = Nothing
Just x >>= f = f x
```

2.4: Definice Maybe

Je-li výstupem funkce *f* - *Nothing*, tak se vypíše jen *Nothing*, jeli výsledkem funkce vylidní výsledek přidá se před tento výsledek *Just*. Funkce *return* vrací ono *Just x* zpět. Důležité k této definici je zmínit to, že *x* může být jakéhokoliv typu.

List

I list, který je jedním z nejpoužívanějších vlastností Haskllu, je monáda. List je definován takto:

```
return x = [x]
xs >>= f = concatMap f xs
```

2.5: Definice List

Když by byl zadán jen jeden prvek, funkce *return* vrátí jednoprvkový list. Další řádek definice je o trošku zajímavější, tento kus definice se stará o to, aby všechny prvky byly vypsány v

listu. Toho je docíleno, tak že na každý prvek je volána funkce f a následně funkcí *concat* jsou všechny prvky sloučeny do jednoho listu.

2.3 Haskell

Tento programovací jazyk řadíme do kategorie jazyků s líným vyhodnocováním. Což znamená, že program vyhodnotí funkci nebo výraz jen v tom případě, že je nutná k dosažení konečného výsledku. Haskell také pečlivě lpí na mezerách, které mají syntaktický význam. Existují dva hlavní kompilátory pro Haskell, prvním z nich je Hugs - v dnešní době se již moc nepoužívá. Druhým, ale daleko významnějším kompilátorem je GHC = Glasgow Haskell Compiler. Tento kompilátor je multiplatformní a velmi dobře aktualizován. Operátory funkcí se chovají stejně jako v matematice, existuje zde několik možných zápisů. Prvním z nich je infixový tvar [8 ‘mod‘ 2 ukázka infixu], který je nutné ohraničit zpětným apostrofem. Dalším způsobem je prefixový tvar [mod 8 2 - ukázka prefixu], zde si lze lépe představit, že funkci **mod** jsou předány dva argumenty (8 a 2). Volba zápisu je samozřejmě možná i u základních operací jako jsou například sčítaní nebo odčítání. Haskell dokáže velice dobře vyhodnocovat prioritu operací, také platí, že volání funkce má přednost před operací. Prioritu operací můžeme upravovat pomocí závorek. Syntax tohoto programovacího jazyka je velice striktní v pojmenovávání funkcí, proměnných a datových typů:

- funkce se značí malými písmeny
- datové typy se značí prvním velkým písmenem (např: String, Char)
- proměnné se značí stejně jako funkce malým písmenem

3. Vyhodnocování aritmetických výrazů

3.1 Notace

3.1.1 Prefix

Prefixová notace, v některé literatuře zvaná též polská notace, je varianta zápisu algebraických výrazů. V této variantě se operátory zapisují před operandy. Protože je pořadí operátorů fixní, syntaxe nepotřebuje žádné závorky a zápis je tedy jednoznačný. [citace - `https://cs.wikipedia.org/wiki/Prefixov%C3%A1_notace`]. Tento způsob je velice oblíbený v informatice, jelikož jeho vyhodnocení trvá lineární množství času a je poměrně jednoduché.

K vyhodnocení využíváme zásobník (stack) - do tohoto zásobníku postupně vkládáme celý výraz zprava doleva: $+\ 26\ *\ 3\ \ 6$

Do zásobníku se nejdříve vloží číslo 6, následně se vloží číslo 3, jako další by se měl vložit operátor *, ale místo toho se tato operace (*) provede na poslední dva členy v zásobníku (jsou vyjmuty ze zásobníku) a výsledek operace $3 * 6 = 18$ je uložen opět do stejného zásobníku. V dalším kroku je do zásobníku vloženo číslo 26 a poté operátor +. Operace (+) se provede opět na poslední dva členy v zásobníku (26 + 18) a výsledek je opět uložen do zásobníku. Výraz je vyhodnocen tehdy, když v zásobníku zůstane jen jediná hodnota, která je výsledkem výrazu.

3.1.2 Postfix

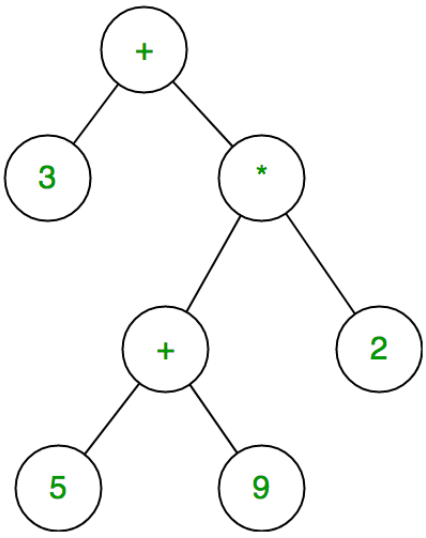
Postfix je velice podobný prefixu. Jak již název napovídá jedná se o zápis, kde se operátory píší za operandy. Stejně jako u polské notace není důležité používat závorky, jelikož pozice operátorů je fixní. Vyhodnocení probíhá naprosto stejně jako u polské notace, ale s tím rozdílem, že čteme výraz zleva doprava. Vkládáme je do zásobníku a provádíme na nich operace, mezivýsledky následně taktéž vkládáme do zásobníku.
Příklad postfixového zápisu: $5\ \ 1\ \ 2\ \ +\ \ 4\ \ *\ \ +\ 3\ -$
Výhodou může být jednoduchost vyhodnocení, čtení zleva doprava, rychlost vyhodnocení.
Nevýhodami jsou neintuitivnost zápisu výrazu a dodržování mezer.

3.1.3 Infix

Infixová notace je běžný způsob zápisu výrazu ve vzdělávacích systémech a i v běžném životě. Je velice intuitivní a jednoduše čitelný. Operátory se píší mezi operandy, z toho vyplývá nutnost používání závorek, když chybí závorky je priorita operací zřízena operátory. I když je pro člověka tato notace velice jednoduchá, tak pro počítačové zpracování není. Je tu hned několik možností vyhodnocení tohoto výrazu. Jedním z možností je převést infixový

tvar na postfixový nebo prefixový. Toho lze dosáhnout za pomoci Shunting-yard algoritmu a následně velice jednoduše vyhodnotit tento nový zápis. Další možností je vyhodnocení pomocí formálních gramatik, tento způsob je například využíván operačními systémy. Třetí způsob je převedení výrazu do binárního stromu, kde každý otec je znaménko a jeho poslední syn je vždy číslo. Následně je výraz vyhodnocen rekurzivně.

Teorie převedení výrazu zapsaného v infixové podobě:
Strom je tvořen pouze operátory a operandy. V první fázi, musí program najít operaci s nejnižší prioritou, při tomto procesu musí respektovat pořadí operandů, ale také rozmístění závorek. Když takovou operaci nalezne, stává se toto znaménko s nejnižší prioritou dělící čarou, která rozděluje výraz na dvě podmnožiny. Stejná funkce je zavolána na obě vzniklé části a celý proces pokračuje, dokud nezůstane jen jedna operace mezi dvěma operandy. Strom je stavěn rekurzivně, s tím že se do každého otce (pokud má syny) je uloženo znaménko (operátor). Do otců, které nemají žádné syny jsou uloženy operandy. Ukázka stromu pro výraz $3 + ((5 + 9) * 2)$ je vidět v následujícím obrázku (Binární strom).



Obrázek 3.1: Binární strom

Obrázek 3.2: `https://www.geeksforgeeks.org/expression-tree/`

Převod notací

Převod notací z binárního stromu je velice jednoduchá záležitost, závisí na pořadí výpisu synů a otce.

Převod z infixu na postfix:

Za pomocí rekurze se vypíše nejdříve levý syn, následně pravý syn a nakonec otec těchto synů.

Převod z infixu na prefix:

Za pomocí rekurze se vypíše nejdříve otec, následně levý syn a nakonec pravý syn. Jelikož v tomto typu zápisu nejsou vyžadovány závorky je takto vzniklý výraz validní.

Převod z binárního stromu zpět na infixový zápis:

Opět za pomoci rekurze vypíšeme vždy nejdříve levého syna, následně otce a poté pravého syna. Před každým vypsáním levého syna vypíšeme závorku a po každém vypsání pravého syna ukončíme tuto závorku. Tím nám vznikne původní výraz, který bude obsahovat více závorek nežli původní, ale výsledek výrazu zůstane nezměněn.

3.2 Dokumentace procedurálního programu

3.3 Dokumentace neprocedurálního programu

3.4 Srovnání

4. Závěr

****** 4546546

Literatura

[1] Birge J. R., Wets R. J.-B. (1987): Computing bounds for stochastic programing problems by means of a generalized moment problem. *Mathematics of Operations Research* **12**, 149-162.

Přílohy