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 22.2.2019 Ivan Žižka

Poděkování

Chtěl bych poděkovat Mgr. Janu Hamáčkovi za vedení mé práce, rady a věcné připomínky. Dále bych chtěl poděkovat RnDr. Rudolfovi Krylovi za odbornou konzultaci při zpracování této práce.

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. Druhým cílem je vysvětlit základní pojmy funkcionálního programování, 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. [1] Tato teze srovnává člověka, který následuje algoritmus, a Turingův stroj.

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. [2] Lambda kalkul tvoří páteř takzvaných Turingových 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 dodnes 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. [1] Z tohoto důvodu byla sestavena speciální komise, která vyprodukovala čistě funkcionální programovací jazyk Haskell.

2.2 Funkcionální programování

Funkcionální programování se spíše než klasickým programovacím jazykům podobá matematickému zápisu funkce. Hlavní 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 vzniká 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 $\lambda x.x - 5$. [3] 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. [3] 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:

1. funkce může přijímat jako argument jinou funkci
2. výsledkem této funkce může být 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 Vedlejší efekty

Nastávají tehdy, kdy funkce během výpočtu mění stav procesu, než je návratová hodnota funkce. Vedlejší efekty jsou častým zdrojem chyb v kódu, na druhou stranu může správně použitý vedlejší efekt usnadnit implementaci kódu.

Příkladem neúmyslného vedlejšího efektu může být toto:

```
void and(int a, int b)
        {
        int sum = a + b;
        printf("%d", sum);
        return;
}

int main(){
        int n = 1;
        add(n++, n);
        return 0;
}
```

2.4: jazyk C

Zde se jedná o zcela zjevný vnější efekt. Jelikož nevíme jestli se zavolá funkce *add* s argumenty *2 a 1* nebo s argumenty *2 a 2*. Toto rozhodnutí leží na konkrétním překladači a dokumentaci daného jazyka. V tomto případě se vypíše číslo: 3.

Úmyslným použitím vedlejšího efektu může být funkce *sleep(500)*, která nám na základě hardwarového času uspí program.

Obecně se může říci, že za vedlejší efekty můžeme považovat změnu hodnoty proměnné, ukládání na disk nebo načítání vstupu.

2.2.5 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 užití je řešení vstupu a výstupu programu, například do konzole. Myšlenka za touto funkcí je taková, že funkci IO je předáván stav programu jako argument (stav = momentální vyhodnocení výrazu programem). IO následně například vypíše daný výsledek do konzole. 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
```

2.5: Definice Maybe

Je-li výstupem funkce *f* - *Nothing*, tak se vypíše jen *Nothing*, jeli výsledkem funkce validní výsledek přidá se před tento výsledek *Just*. Funkce *return* vrací ono *Just x* zpět.[4] 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
```

2.6: 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.[5] 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 (nemění se = vstup) se značí stejně jako funkce malým písmenem

3. Aritmetické výrazy

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. [6] 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

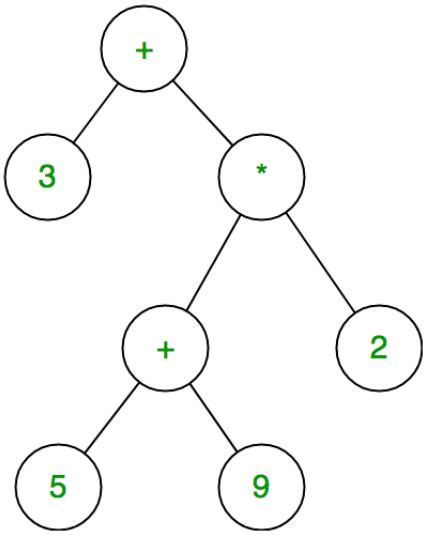
Postfix je velice podobný prefixu. Jak již název napovídá jedná se o zápis, kde se operátory píší za operandy. Stejně jako u polské notace není důležité používat závorky, jelikož pozice operátorů je fixní. Vyhodnocení probíhá naprosto stejně jako u polské notace, ale s tím rozdílem, že čteme výraz zleva 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 [7] algoritmu a ná-

sledně 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

Zdroj: `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.

4. Dokumentace

Oba níže popsané programy mají stejnou funkci, převádějí prefixovou notaci na infixovou. Toho docilují za pomoci binárních nevybalancovaných stromů, které jsme si představili výše.

Ten jen postaven z prefixové notace. Stavba probíhá pomocí rekurze, která se zanořuje pokaždé, když je načteno znaménko. Znaménka jsou vždy otcové a čísla výhradně syny, kteří nemají potomky.

Důležitou věcí, která je potřeba zmínit, je ta že oba programy jsou přizpůsobeny na převádění výrazů, které obsahují pouze **jednociferná čísla**.

Získání zdrojového kódu

Existují dvě cesty jak lze získat zdrojový kód. První možností je přistoupit na url adresu (viz. příloha), zde kliknout na tlačítko *Clone or download*, a následně vybrat možnost *Download ZIP*. Tento soubor je poté nutno extrahovat.

Druhou možností je využití nástroje *Git*. Zde je opět nutné přejít na url adresu (viz. příloha), zde kliknout na tlačítko *Clone or download* a následně zkopírovat odkaz, který se nachází v nově vzniklém boxu. Poté je nutné se přesunout do vašeho počítače, ve kterém je nainstalován program *Git*. Otevřít konzoli Git (*Git Bash*), zde je nutné se navigovat do složky, kam chcete uložit repositář, a poté užít příkaz *git clone (adresa, kterou jsme si již dříve zkopírovali)*.

V tuto chvíli máte přístup ke zdrojovému kódu jednotlivých programů.

4.1 Dokumentace procedurálního programu

4.1.1 Vývoj a editace

Program byl vyvíjen v nástroji *Code Blocks*, v repositáři se nachází nastavený projekt. Editace vstupu probíhá přímo v kódu, v souboru *main.c*. Rozhodli jsme se zvolit tento, ne až tak praktický způsob, jelikož jsme chtěli primárně ukázat rozdíl mezi imperativním a neimperativním způsobem programování.

4.1.2 Princip fungování

Program vezme vstup, který mu zadal uživatel (modifikoval pole *vstupniData*), převede ho do pole datového typu *prefix*. Datový typ prefix obsahuje dvě proměnné z nichž první je znak a druhá číslo. Pokud je zadáno číslo, znak má nastavenou hodnotu na *NULL*. Tento výše popsaný proces se děje pomocí funkce *poleDoPrefix* (viz strana 16.).

Ve funkci *main* je následně vytvořena hlava stromu (datový typ *strom*) a její všechny části jsou nastaveny na *NULL*. Poté je volána funkce *UkladaniDoStromu* (viz strana 16.), která zajišťuje inicializaci stavby stromu.

Po ukončení stavby stromu je zavolána funkce *infix* (viz strana 17.), která vypíše do konzole infixový tvar.

poleDoPrefix

Tato funkce vyžaduje dva argumenty, prvním z nich je vstupní pole (zadaná data) a druhým argumentem je počet prvků pole, ten je získán podílem, kde prvním operandem je velikost vstupního pole a druhým je velikost datového typu *char*. Výstupem funkce *poleDoPrefix* je ukazatel na pole datového typu *prefix*.

UkladaniDoStromu

Inicializuje ukládání do stromu, tím že kontroluje, jestli nebyl zadán prázdný vstup a následně volá funkci *ZapisDoLeva* (viz strana 16.). Argumenty funkce *UkladaniDoStromu* jsou vstupní data (datový typ prefix), ukazatel na hlavu (datový typ strom) a velikost pole (datový typ int). Výstupem není nic, jelikož ukazatel na hlavu stromu si uchováváme ve funkci *main*.

ZapisDoLeva

Argumenty této funkce jsou: ukazatel na hlavu, vstupní pole a index (ten říká kolikáté pole vstupních dat se má uložit do stromu). Funkce si následně vytvoří proměnnou syn, která je ukazatelem na datový typ *strom* (viz strana 17.). Poté je kontrolováno, zda se má uložit znaménko či číslo. Pokud se jedná o znaménko je do struktury *syn* uloženo a pomocí ukazatelů je vytvořen odkaz z otce na syna a opačně. Následně je rekurzivně volána opět funkce *ZapisDoLeva* s vyšším indexem o jedna.

Jedná-li se o číslo, je uloženo do struktury a opět je vytvořeno propojení mezi otcem a synem. Poté je zavolána funkce *ZapisDoPrava* (viz strana 17.), ale je jí předán odkaz na otce.

ZapisDoPrava

V principu funguje stejně jako předchozí funkce, akorát s tím rozdílem, že pokud se jedná o znaménko netypicky volá funkci *ZapisDoLeva*.

V případě, že se jedná o číslo provede naprosto stejné kroky, jako funkce *ZapisDoLeva*, ale rozdílem je to, že funkci *ZapisDoPrava* je předán ukazatel na otce otce syna (Otec->Otec).

strom

Základní stavební jednotkou celého programu je struktura *strom*. Ta tvoří základ aritmetického stromu, obsahuje ukazatel na levého a pravého syna, ukazatel na svého otce a dvě proměnné, které uchovávají znaménko nebo číslo. Pokud ukazatel nikam nevede, je mu nastavena hodnota *NULL*. Příkladem může být ukazatel na otce u kořene celého stromu. Ten logicky nikam nevede, a tak je nastavena hodnota *NULL*.

infix

Funkce inicializuje vypsání infixového tvaru výrazu. To je zajištěno tím, že vypíše nejdříve levou větev (*vypisL* viz strana 17.) a následně pravou (*vypisR* viz strana 17.). Mezi ně je vypsáno znaménko.

vypisL

Pokud je v hlavě (ta je předána v argumentu) znaménko, zavolá se funkce *vypisL*, poté se vypíše znaménko v hlavě a následně se volá funkce *vypisR*.

Je-li v hlavě uloženo číslo, vypíše se začátek závorky a následně toto číslo.

vypisR

Pokud je v hlavě (ta je předána v argumentu) znaménko, zavolá se funkce *vypisL*, poté se vypíše znaménko v hlavě a následně se volá funkce *vypisR*.

Je-li v hlavě uloženo číslo, vypíše se toto číslo a následně konec závorky.

4.2 Dokumentace neprocedurálního programu

4.2.1 Vývoj a editace

Program byl vyvíjen v jazyce Haskell na platformě linux, jako vývojové prostředí jsme zvolili Atom. Hlavním důvodem bylo to, že existuje velké množství doplňků, které například zvýrazňují syntax Haskellu. Další nástroj, který je nutný pro kompilaci programu, je *stack*. Který je dostupný z oficiálních stránek Haskellu [8], ten obsahuje nástroj pro vytváření projektů, je schopen automaticky stahovat velké množství knihoven a navíc již sám o sobě obsahuje nejpoužívanější knihovny.

Editace je možná přímo v souboru *Main.hs*, proč tomu tak je jsme již vysvětlili.

4.2.2 Spuštění a kompilace

Pokud se rozhodneme spustit program na platformě Windows a máme již nainstalován kompilátor i s doplňky, stačí jen otevřít složku *App*, v ní spustit soubor **Main.hs** a napsat příkaz **main**. Tím se spustí hlavní funkce programu a vypíše se výsledek. Pokud budeme editovat vstupní data, stačí upravenou verzi jen uložit a kompilátor sám zkompiluje daný program.

4.2.3 Princip fungování

Ve funkci *main* je volána funkce *printIndex*, jíž argumentem je další funkce. Tato funkce je *buildTree* (viz strana 18.) s argumentem *prefix*.

Již z definice Haskellu víme, že program začne vyhodnocovat nejdříve funkci *buildTree* a následně vyhodnotí funkci *printIndex* s argumentem, který je roven výsledku *buildTree*. Tímto argumentem bude datová struktura stromu. Následně může být vypsán infixový tvar výrazu.

buildTree

Funkce nejdříve postaví levou (*buildLeft* viz strana 18.) část stromu a následně pravou část (*buildRight* viz strana 19.). Výstupem této funkce je datový typ Tree Char. Argumentem, který dostává je list znaků, v programu zvaný *prefix*.

buildLeft

Dostává dva argumenty, jedním z nich je list znaků (*prefix*) a druhým je pomocný datový typ Builder Char (viz strana 19.). Ten zajišťuje to, aby byl vždy vzat správný prvek z listu. Dalším důvodem jeho použití je to, že si dokáže pamatovat strukturu strom. Pokud-li chceme pomocí této funkce uložit do stromu znaménko, zavolá se opět tato funkce, ale s o jedna větším indexem. Poté co se vynoří z této rekurze, opět se zanoří do rekurze (*buildRight*

viz strana 19.), ale tentokrát do pravé větve. Obě rekurzivní zanoření končí tu chvíli, kdy chceme do stromu uložit číslo. Výstupem této funkce je Builder Char, který obsahuje číslo dalšího indexu, který by se měl načítat z listu. Druhou důležitou hodnotu, kterou Builder uchovává je celá struktura stromu pod ním.

buildRight

Je naprosto stejná jako *buildLeft*. Dala by se nahradit předešlou funkcí, jen s tím rozdílem, že by se volala s jiným argumentem.

Tree a

Tato datová struktura může mít hned dvě podoby. První podobou je ta, ve které strom nikam nevede, a proto je nastavena hodnota na **Empty** (to je námi zadefinované pojmenování). Druhou podobou je taková, kdy strom obsahuje hodnotu typu *a* (stejná jako datový typ stromu) další hodnotou je struktura stromu (můžeme si ji označit jako levá větev) a poslední hodnotou datové struktury *Tree* je opět strom (pro jednoduchost ji můžeme nazývat pravá větev).

Builder a

Je pomocná stavební struktura, která obsahuje číslo a datovou strukturu stromu. Je využita k uchovávání indexu dalšího členu a vrácení stromu z rekurze.

printInfix

Argumentem této funkce je struktura stromu a výstupem je vypsání do konzole. Nejdříve vypisuje levou část stromu (*printLeft* viz strana 19.), následně vypíše hodnotu uloženou v začátku stromu a jako poslední vypíše pravou větev (*printRight* viz strana 20.)

printLeft

Dostává v argumentu strukturu stromu. Když chceme vypsat hlavu stromu, v níž je uloženo znaménko, nejprve vypíše otevřenou závorku a následně volá funkce sama sebe s argumentem levého syna, poté se vypíše hodnota znaménka v hlavě a následně se vypíše pravá větev (*printRight* viz strana 20.).

Když chceme vypsat hlavu, kde je uloženo číslo vypíšeme nejdříve otevřenou závorku a poté číslo uložené v hlavě. Tím se vracíme o krok zpět v rekurzi.

printRight

Stejně jako předchozí funkce dostává jako argument strukturu stromu. Vypisujeme-li znaménko, nejdříve voláme funkci na vypsání levého stromu, poté vypíšeme hodnotu hlavy a následně vypíšeme pravou větev. Vše ukončíme vypsáním uzavřené závorky.

Vypisujeme-li číslo, nejprve zobrazíme do konzole číslo a následně ukončenou závorku.

5. Srovnání

V této kapitole se zaměříme na subjektivní srovnání funkcionálního a nefunkcionálního programování založeného na zkušenosti, které jsme nabyli při vytváření převodu prefixové notace na infixovou.

5.1 Srovnání rozdílného přemýšlení

Rozdílné přemýšlení ve funkcionálních jazycích, v našem případě přímo v Haskellu, bylo pro nás nejtěžším problémem. Zažité postupy a způsoby řešení z imperativních programovacích jazyků, zde mnohdy nefungují. Hlavní rozdíl jsme mohli spatřit již v samotné funkci **main**, která u obou programů plní stejnou funkci. Hlavní rozdíl je v přemýšlení nad posloupností příkazů. Například v jazyce C přemýšlíme chronologicky a snažíme se postupovat od vstupu až po výsledný algoritmus. Samozřejmě, že to není vyžadováno, ale mnoho programátorů to takto dělá. Ve funkcionálním programování je situace opačná, je zde nepřímo vyžadováno přemýšlení od výstupu ke vstupu, alespoň takto to na nás působilo.

Dalším velkým rozdílem je, že při tvorbě v Haskellu jsme velmi často pracovali s rekurzí. Již dříve jsme měli zkušenosti s rekurzí, ale nikoliv v tak velkém množství. V podstatě tam, kde bychom v imperativním jazyce napsali cyklus, zde píšeme rekurzivní funkci.

5.2 Formální stránka kódu

Hned na první pohled si můžeme všimnout toho, že ve funkcionálním programu je skoro až absence čísel. Kód, který jsme vyprodukovali v Haskellu, více připomíná anglický jazyk než program, na který jsme zvyklí. V tom je částečně krása funkcionálního programování, je zde úplná absence proměnných jaké známe z nefunkcionálních jazyků. Jak jsme již zmínili v teorii, nelze za běhu programu měnit stav proměnné. Proměnné v čistě funkcionálních programech mají spíše funkci substituce. Z této vlastnosti plyne absence vedlejších efektů funkcí, tím je zajištěno to, že programy vytvořené ve funkcionálních jazycích nemají často chyby (ve smyslu programátorské). Proto častěji než klasické krokování programu řádek po řádku se v Haskellu užívají testy. Kde zadáme vstupní data a očekávaný výsledek, následně se vypíše, jestli se shoduje naše očekávání s reálným výstupem.

Dalším charakteristickým znakem funkcionálního programování je to, že si programátor může kód dost přizpůsobovat k obrazu svému. Nádherným příkladem mohou být datové struktury. Zde si můžeme podle libosti vytvářet pojmenování, příkladem takového pojmenování je struktura stromu.

```
data Tree a = Empty
    | Node a (Tree a) (Tree a)
    deriving (Show)
```

5.1: Definice datového typu Tree

Zde bychom mohli nahradit *Empty* za *NULL* a výsledek by byl stejný.

5.3 Srovnání programů

Nejdůležitější rozdíl vidíme ve stavbě stromu. U programu, který byl tvořen v jazyce C jsme si mohli dovolit používání ukazatelů a z toho důvodu také předávat ukazatel například na svého otce. Tento způsob jsme zvolili záměrně, abychom demonstrovali možnosti jazyka C. Na druhou stranu jsme museli řešit nečekané programátorské chyby, které se při psaní programy vyskytly. Nejčastěji se jednalo o předávání špatného indexu nebo ukazatele.

V Haskellovém programu jsme strom museli stavět rekurzivně a řešení nebylo tak intuitivní jako v neprocedurálním jazyku C. Haskell využívá místo ukazatelů strukturu, která si uchovává celý strom.

5.4 Výhody

Funkcionální jazyky umožňují obrovskou míru abstrakce, snazší rozdělení do funkcí nebo modulů. Dají se zde využívat funkce vyšších řádů, a může tak vzniknout kód, který je srozumitelnější a často i kratší nežli jeho imperativní varianta. Výhodou Haskellu je líné vyhodnocování, které mnohdy urychlí algoritmus. V poslední řadě spatřujeme výhodu ve velké striktnosti datových typů.

5.5 Nevýhody

Jelikož většina počítačů není navržena pro funkcionální jazyky, může být i papírově efektivnější algoritmus paradoxně pomalejší než jeho imperativní ekvivalent. Některé postupy které známe z nefunkcionálních jazyků lze velice těžko implementovat do funkcionálních programů. Příkladem může být například hašovací tabulka, kterou můžeme znát například z jazyka C#. Další nevýhodou je složitost překladačů a při využití líného vyhodnocení i těžce odhadnutelná paměťová složitost.

6. Závěr

V této práci se nám povedlo vytvořit dva rozdílné programy, které jsou však vytvořeny jinou programovací metodou. Oba programy jsou zcela funkční, ale dokáží převádět takové výrazy, které obsahují jen jednociferná čísla. Tyto dvě implementace převodu prefixového zápisu na infixový, nám však poskytli dostatečné znalosti k tomu, abychom mohli porovnat imperativní a neimperativní programovací jazyky. V našem případě konkrétně jazyk C s jazykem Haskell.

Největší rozdíl spatřujeme ve způsobu myšlení, který je u neimperativního naprosto rozdílný. Výhodou funkcionálních jazyků je abstrakce a jednoduchý zápis některých algoritmů. Hlavní nevýhodou je to, že většina dnešních počítačů není konstruována pro zpracování funkcionálního kódu.

Literatura

[1] Kolektiv autorů, 2018. Wikipedia.org: Funkcionální programování [online]. Poslední změna 1.12.2018 [cit. 2.1.2019]. Dostupné z: https://cs.wikipedia.org/wiki/Funkcion%C3%A1ln%C3%AD_programov%C3%A1n%C3%AD#Historie

[2] Kolektiv autorů, 2018. Wikipedia.org: Lambda kalkul [online]. Poslední změna 24.12.2018 [cit. 12.2.2019]. Dostupné z: https://cs.wikipedia.org/wiki/Lambda_kalkul

[3] Petr Zemek, 2008. λ-kalkul rychle a pochopitelně [online]. Poslední změna 28.5.2008 [cit. 12.12.2018]. Str. 2. Dostupné z: https://publications.petrzemek.net/articles/PZ_-_IPP-Lambda-Kalkul.pdf

[4] Miran Lipovača, 2011. learnyouahaskell.com: Monad laws [online]. [cit. 28.12.2018]. Dostupné z: http://learnyouahaskell.com/a-fistful-of-monads#monad-laws

[5] Miran Lipovača, 2011. learnyouahaskell.com: So what's Haskell? [online]. [cit. 12.1.2019]. Dostupné z: http://learnyouahaskell.com/introduction

[6] Kolektiv autorů, 2018. Wikipedia.org: Prefix [online]. Poslední změna 8.8.2018 [cit. 1.12.2018]. Dostupné z: https://cs.wikipedia.org/wiki/Prefixov%C3%A1_notace

[7] Nik1996 a Shrikanth13, 2013. geekforgeeks.org: Expression Evaluation [online]. [cit. 19.12.2018]. Dostupné z: https://www.geeksforgeeks.org/expression-evaluation

[8] Happy: The Parser Generator for Haskell, 2018. haskell.org: Downloads [online]. [cit. 18.2.2019]. Dostupné z: https://www.haskell.org/downloads

Přílohy

1. **Git repositář imperativního programu:**

   `https://github.com/Zizkai/funkcionalni_programovani_imperativni_program`

2. **Git repositář neimperativního programu:**

   `https://github.com/Zizkai/funkcionalni_programovani_neimperativni_program`