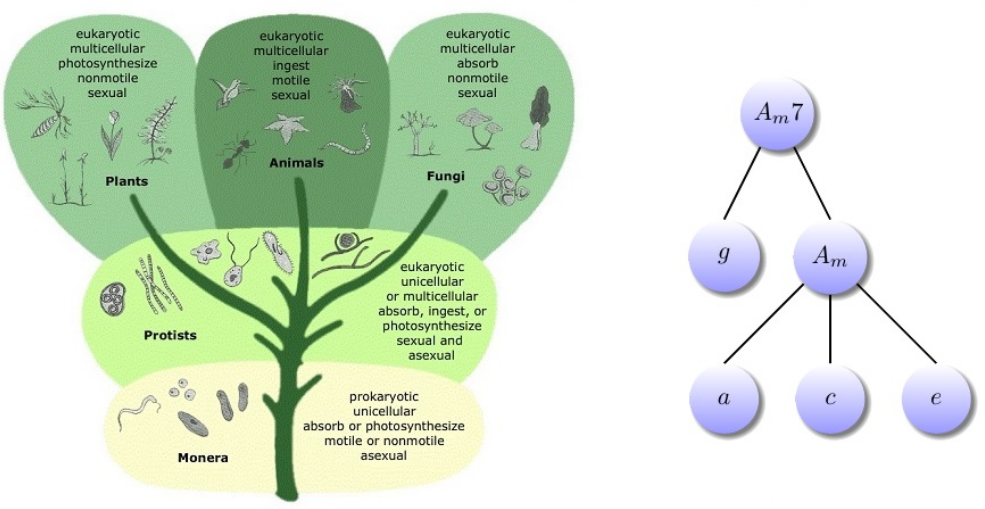
Abstrakt

Cieľom tejto práce je navrhnúť a implementovať v jazyku Java vhodnú dátovú štruktúru schopnú uchovať a pracovať so stromovými štruktúrami vyskytujúcimi sa v symbolickej logike. Výsledná štrukúra je schopná načítať textový vstup - výraz rôzneho charakteru, a v zavislosti od zvoleneho typu stromu transformovať reťazec do stromovej štruktúry. Nad každým načítaným znakom je vykonávaná analýza, na základe ktorej je z databázy každému stromovému uzlu priradená infrozmacia rôzneho charakteru. Aplikácia následne poskytne kompletnú analýzu a ohodnotenie stromu na zaklade zvolenej stromovej štruktúry.

Kľúčové slová: *Stromove štruktúry, Výraz, TeX, Java.*

Úvod

Stromy sú v informatike jednou z najdôležitejších dátových štruktúr. Táto dátová štruktúra je vhodná na ukladanie a spracovanie dát hierarchickej povahy. Pekné príklady stromových štruktúr poskytuje však aj svet okolo nás. Napríklad druhova klasifikácia živočíchov je jednoducho a názorne modelovaná tzv. stromom života(Obr.1), ktorý znázorňuje 5 základných form života. Menej typické stromové štruktúry môžeme nachádzať aj v hudbe. Napríklad zložitejší akord A_m7 je poskladaný z klasického kvintakordu A_m a tónu g. Akord A_m ďalej vieme rozložiť na tóny a, c, e. Tento proces vieme tiež modelovať pomocou jednoduchého stromu(Obr.1).



Obr. 1: Strom života (v ľavo) a Rozklad akordu A_m7 (v pravo)

Aj v tejto práci sa zaoberáme troma stromovými štruktúrami, ktorých hierarchickosť nie je na prvý pohľad tak zjavná.

Prvou štruktúrou bude trieda výraz, ktorá bude stromovo interpretovať aritmeticky, logický, či iný typ výrazu. Inicializácia výrazu prebieha načítaním textového vstupu v TeX-ovskej syntaxi. Výsledkom nie je len ohodnotenie zadaného výrazu ale aj kompletná analýza vlastnosti zadaného vstupu. Vlastnosti skúmaného výrazu sú graficky znázornené do výsledného stromu, ktorý je generovaný len na základe vstupného

reťazca. Táto trieda bude základným stavebným prvkom ďalších, zložitejších stromových štruktúr.

Ďalšou zaujímavou stromovou štruktúrou, ktorou sa budeme zaoberať je sémantické tablo. Ide o jednoduchý a univerzálny teoretický prostriedok k overeniu pravdivosti formúl na sémantickej úrovni. Metóda sémantických tabiel je založená na systematickom postupe transformácie výrokovej formuly do tvaru disjunktívnej normálnej forme (DNF), ktorý má jednoduché podmienky pre kontradikčnosť alebo splniteľnosť.

1 Výraz

1.1 Matematické symboly

Do 15. storočia neexistovala jednotná matematická symbolika. Všetky vety, poučky, rovnice a úlohy sa mohli vyjadriť jedine slovne. Ale tak ako v reálnom svete je pre normálny život človeka potrebná komunikácia, aj v matematike bolo potrebné vybudovať určitý komunikačný prostriedok, čiže jazyk, ktorým môžeme formálne popísať nejaké javy.

Každý takýto jazyk je založený na tom, že najprv sa vymedzia iste samostatné objekty či deje a potom sa každému z nich pridelí element tohto jazyka, ktorý naň ukazuje, je jeho symbolom čiže akýmsi abstrahovaným zástupcom. Vhodnou kombináciou takýchto symbolov potom môžeme vyjadriť pozorovane vzťahy medzi nimi a tieto vyjadrenia efektívne sprostredkovať ostatným.

Jazyk matematiky vždy do istej miery musí kopírovať prirodzený jazyk, pretože ten odráža proces myslenia. Skúsime teda podľa týchto postupov definovať matematické symboly, no urobíme to tak striktne aby jednotlivým symbolom, neskôr aj výrazom rozumel aj „hlúpy“ počítač, keďže jednoznačnosť pri strojom spracovaní symbolov je veľmi dôležitá.

Všetky matematické symboly, s ktorými budeme pracovať, by sme mohli rozdeliť do niekoľkých kategórií, podľa toho, čo vyjadrujú:

- Symboly `3`, `4`, `5,2`, `e`, `Venuša` či `Mars`, sú menami konkrétnych objektov, istých konštant. Môžeme ich preto nazvať ***konštantové***.

Ako však môžeme vidieť, posledné dva symboly majú k sebe bližšie než ostatné štyri, sú totiž na rozdiel od nich reťazcami znakov slovenskej abecedy. Budeme preto po vzore mnohých programovacích jazkyov rozlišovať dátové typy objektov, ako napríklad `Integer` (ten majú prvé dva symboly z nášho zoznamu), `Real` (druhé dva) či `String` (posledné dva). Hneď však narážame na istú nejednoznačnosť, veď napríklad čislo `3` je nielen celé, ale aj reálne, ba dokonca ho môžeme chápať ako reťazec. Tento problém však vieme veľmi rýchlo vyriešiť: namiesto jedného symbolu 3 môžeme pracovať s jeho troma verziami – 3_{Integer}, 3_{Real} a 3_{String}, pričom každá bude mať príslušný typ.

- Symboly `+`,`-`, `sin`, $\sum_{\dots}^{\dots}$, $\lim_{\dots \to \dots}$ `CONCAT` či `SUBSTR` vyjadrujú isté funkcie, preto ich budeme nazývajú ***funkciové***.

Funkcia, ktorú takýto symbol označuje, spracúva istý počet vstupujúcich objektov do jedného výsledného objektu. Hodnoty vstupných objektov, samozrejme, dopredu nepozná, aby ich však mohla spracovať, požaduje od nich, aby mali dopredu predpísaný typ. Podobne je to s výslednou hodnotou funkcie, aj tu vieme dopredu určiť len jej dátový typ. Počet a dátové typy vstupov a dátový typ výstupu tak môžeme považovať za neoddeliteľnú súčasť samotného funkciového symbolu. Napríklad vstup pre symbol `sin` (presnejšie pre funkciu sínus ním symbolizovanú) má dátový typ `Real` a výsledok tiež Real. Pri symbole `SUBSTR` máme zasa tri vstupy, v poradí s dátovými typmi `String`, `Integer` a `Integer`, výstupná hodnota bude mať typ `String`.

- Symboly `=`, $\leq$, $\in$ či $\mid$ znamenajú relácie, a preto ich nazývame ***reláciové***.

- Významnou kategóriou symbolov sú ***logické spojky***. V matematike sú používané hlavne unárna negácia $\neg$ (nie je pravda, že ...) a binárnevkonjunkcia $\wedge$ (... a zároveň ...), *disjunkcia* $\vee$ (... alebo ...), *implikácia* $\Rightarrow$ (z toho, že ..., vyplýva ...) a *ekvivalencia* $\Leftrightarrow$ (to, že ..., platí práve vtedy, keď platí, že ...).

Názov tejto kategórie spojky, korešponduje s názvom slovného druhu, lebo aj ony vezmú jednu či dve vety (čiže formuly) a vytvoria z nich súvetie.

- Na záver našej malej prehliadky matematických symbolov si všimnime ***kvantifikátory***, ktorých zmyslom je, ako naznačuje ich názov, kvantifikovať, koľko objektov z danej množiny musí vyhovovať podmienke, na ktorú sa kvantifikácia vzťahuje.

Veľký alebo *všeobecný* kvantifikátor $\forall_{\dots}$ (pre každé ...) vyžaduje, aby kvantifikovanej podmienke vyhovovala každý uvažovaný objekt.

Pri použití *malého* alebo *existenčného* kvantifikátora $\exists \dots$ (existuje ...) stačí, že podmienka je splnená aspoň v prípade jedného objektu.

Je potrebné si všimnuť, že napríklad kvantifikátory nikdy neexistujú samostatne, každý z nich je neoddeliteľne spätý s akýmsi parametrom, značkou, ktorá je napísaná bezprostredne za ním. Už slovné spojenie „pre každé“ naznačuje, že táto značka nemá určenú pevnú hodnotu, Nazýva sa preto *premenná* a hrá podobnú úlohu ako v prirodzenom jazyku zámená. Premenná však môže meniť svoju hodnotu iba v istom rámci, má preto zmysel aj pri nej uvažovať dátový typ. Z tohto hľadiska sa veľmi podobajú na konštantové symboly (majú nulový počet vstupov), symbolmi však nie sú, lebo nič konkrétne nesymbolizujú.

Kvantifikátory nie sú jediné symboly, ktoré takýmto spôsobom viažu premenné, stačí si všimnúť napríklad $\lim_{x \to \dots}$ (kde je zväzovaná premenná x), či $\sum_{i,j=\dots}^{\dots}$ (tu sú to i a j). Tieto premenné budeme považovať za neoddeliteľnú súčasť svojich symbolov. Pri každom symbole tak získavame novú charakteristiku zoznam premenných, ktoré sú ním *zväzované*, hoci drvivá väčšina symbolov má tento zoznam prázdny.

1.1.1 Konštrukcia potrebných prvkov logického jazyka

Zadefinujme si množinu `Abc` (Abeceda), ktorá je rozdelená na dve disjunktné spočítateľné množiny – množinu `Var` tzv. premenných a množinu `Sym` tzv. symbolov.

Označme množinu dátových typov `Typ` množinu `Sym` a množinu `Var`. Podľa predchádzajúcich odsekov nás budú zaujímať takéto zobrazenia množiny `Abc` (`Sym` $\cup$ `Var`):

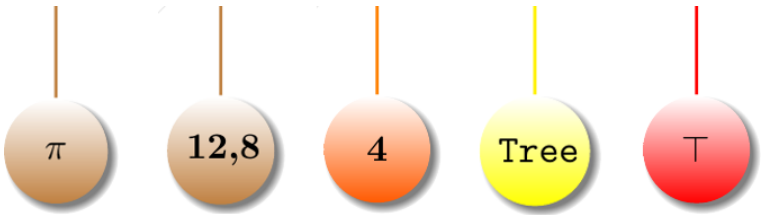
- ins (skratka pre inputs - vstupy), ktoré každému znaku priradí konečnú postupnosť dátových typov
- out (skratka pre output - výstup), ktoré každému znaku priradí dátový typ,
- bva (skratka pre bounded variables - konečná postupnosť zväzovaných premenných) ktoré každému znaku priradí konečnú postupnosť rôznych premenných.

1.1.2 Grafická interpretácia prvkov logického jazyka

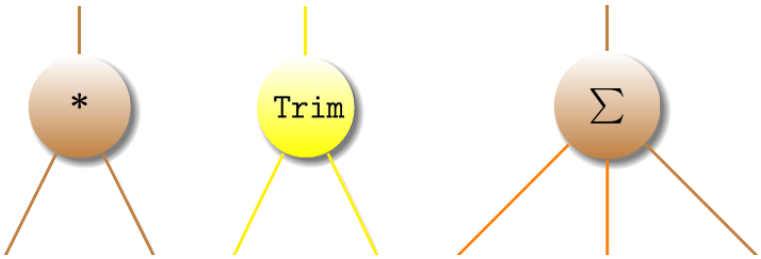
Každý symbol či premennú c a jeho charakteristiky $\mathsf{ins}(c)$ a $\mathsf{ins}(c)$ môžeme znázorniť aj graficky. Dátové typy budú znázornené farbami, pričom každému z nich bude prislúchať iná farba. Napríklad dátový typ `Boolean` bude červená farba, `Integer` žltá, `Real` oranžová, `String` hnedá. Každý symbol či premenná bude mať podobu akéhosi okrúhleho uzla (farby zodpovedajuceho typu) obsahujúceho c, z ktorého vychádzajú akési ***synapsie*** - smerom nahor jedna čiara farby zodpovedajúca typu $\mathsf{out}(c)$ a smerom nadol zľava doprava čiary, ktoré zopovedajú tici $\mathsf{int}(c)$.

Dostávame tak napríklad takéto reprezentácie príslušných symbolov:

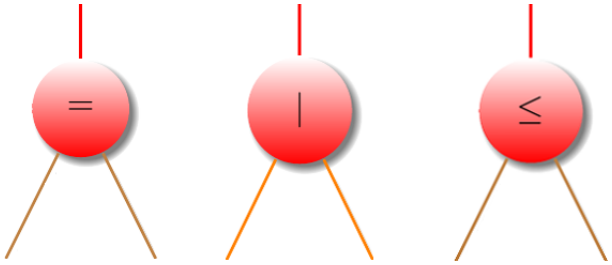
- Premenné a konštantové symboly nemajú žiadne synapsie smerom nadol:



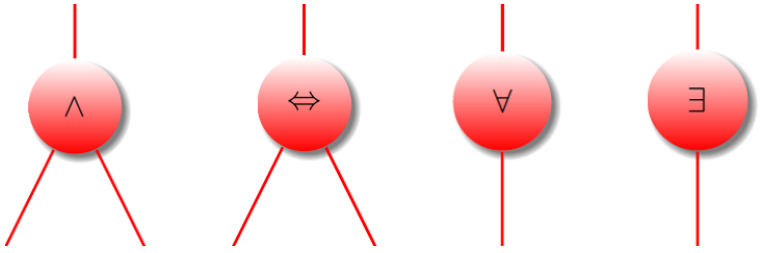
- Funkciové symboly majú nadol kladný počet synapsií, a zároveň tá nahor nie je nikdy červená:



- Pri reláciových symboloch je naopak synapsia nahor vždy červená:



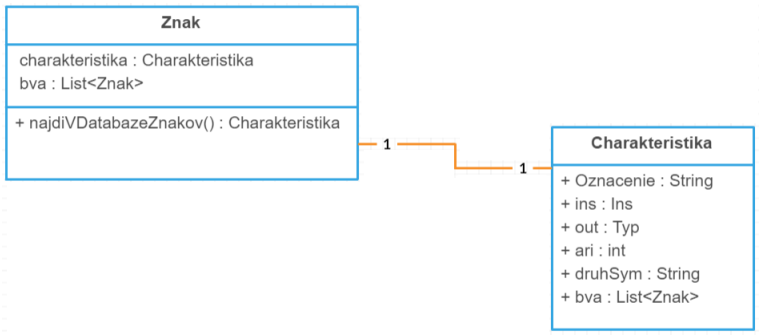
- A napokon logické spojky a kvantifikátory majú nahor i nadol len červené synapsie:



1.1.3 Implementácia znaku

Po načítaní každého znaku z textového vstupu je pre ďalšiu prácu potrebné mu priradiť vyšie spominané vlastnosti a charakteristiky, tj. zobrazenie ins a out, ktoré pomôžu pri vyššie prezentovanéj grafickej interpretácii, ale iné vlastnosti ako arita znaku, druh symbolu resp. identifikacia premennej a zobrazenie bva, ktoré neskôr poslúžia na konštukciu samotného výrazu.

Z textového znaku sa stava (v reči OOP) objekt typu *Znak*, so znázornenou štruktúrou:



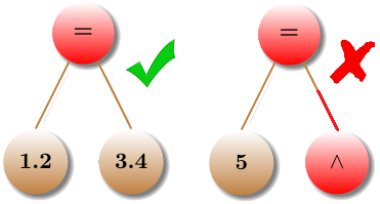
Samozrejme, všetky tieto vlastnosti sa k načítanému znaku nepridajú náhodne. Samotnému priradeniu, predchádza iterácia databázou, ktorá obsahuje informácie o všetkých „stavebných prvkoch“, o ktorých predpokladáme, že sa môžu v reťazci nachádzať. Následné sa porovnávaním vyberie prislúchajúci záznam z databázy a z nášho predtým „nič netušiaceho“ stringu sa stáva objekt triedy Znak, z ktorým možno ďalej pracovať.

Dalo by sa povedať, že na tomto mieste sa pridá doteraz len „bezduchému“ znaku interpretácia, resp. sémantickosť.

1.2 Konštrukcia výrazu

V predchádzajúcich riadkoch sme sa venovali popisu a interpretácii matematických symbolov, teda v našom jazyku sme si definovali písmenka. Tie však na prenos informácii, alebo na popis nejakého deja nestačia, a teda sme motivovaný poskladať z týchto písmenok slová.

Keďže písmenkami sú matematické symboly, „pod slovom“ v našom prípade budeme rozumieť ***výraz***. Spomeňme si, že každý symbol vieme vyjadriť graficky vo forme uzlu a z neho vychádzajúcich synapsií. Je asi prirodzené, že ak chceme konštruovať výraz, má dôjsť k prepojeniu uzlov. Príslušné synapsie sa zlúčia do jednej, pričom jedna ide zdola (je teda výstupom nižšieho uzla) a druhá zhora (je vstupom vyššieho). Aby sa tak naozaj mohlo stať, farba zlučujúcich sa synapsií musí byť rovnaká, teda sinapsie musia byť rovnakého typu.

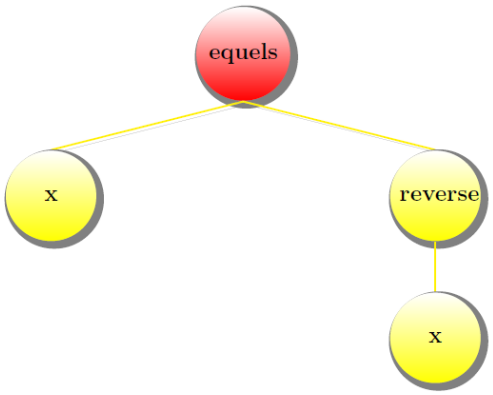


Definícia 1.1. Pod ***výrazom*** budeme rozumieť *Abc*-ohodnotený strom e taký, že pre každé d z $\mathsf{Dsc}(e)$ platí

$$\mathsf{ins}(\mathsf{roo}(d)) = (\mathsf{out}(\mathsf{roo}(\mathsf{chd}(d,i))))_{i<chn(d)}$$

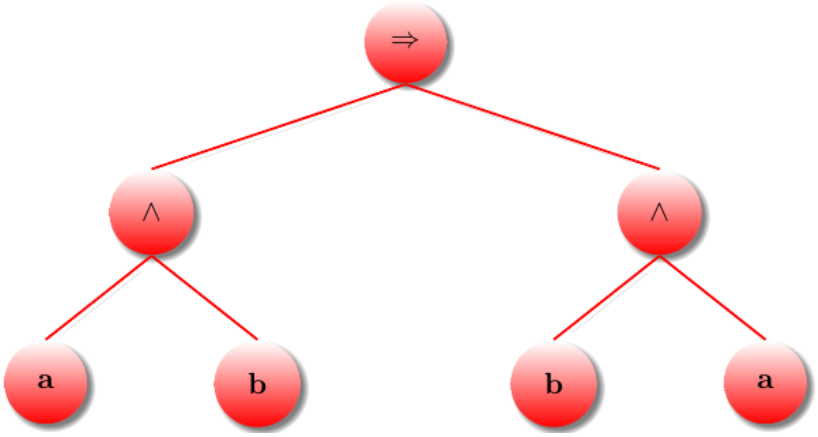
kde $\mathsf{Dsc}(e)$ je množina potomkov stromu e, $\mathsf{chd}(d,i)$ je i-te dieťa potomka d.
Množinu všetkých výrazov budeme označovať ***Exp***.

- Nasledujúci príklad je ukážkou výrazu, ktorý modeluje funkciu ***jePalindrom(x)***, ktorá true ak slovo x sa číta rovnako spredu ako zo zadu(kajak, level, ťahať...):



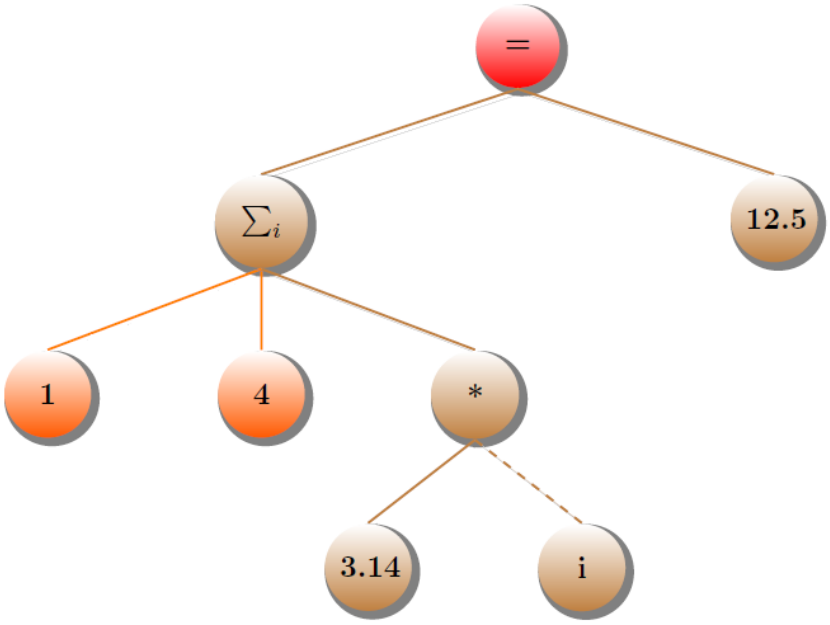
Takým to spôsobom môžeme modelovať priebeh rôznych programov. Ak takýto výraz patrí do množiny *Exp*, znamená to vlastné, že program, ktorý strom znazrňuje je napísaný syntakticky správne, resp. každá funkcia v strome dostala do argumentu hodnotu žiadaného datového typu.

- Za výraz v zmysle definiecie 1.1 môžeme považovať aj logickú formulu:



Ako môžeme vidieť, celý strom je červený, čo znamená, že všetky použite symboly maju na vstupe a na vystupe jeden dátový typ, a to `Boolean`.

- Ako posledný príklad uvedieme strom aritmetického výrazu:



Overme, že tento výraz naozaj patri do množiny *Exp* a teda či je to správny výraz v zmysle našej definície. Rozoberieme všetky uzly:

- ins(=)=(`Real`, `Real`)=(out($\sum_i$),out(12.5))
- ins($\sum_i$)=(`Integer`, `Integer`, `Real`)=(out(1), out(4), out(*))
- ins(*)=(`Real`, `Real`)=(out(3.14),**(cast)**out(i))
- ins(12.5)=ε
- ins(1)=ε
- ins(4)=ε
- ins(3.14)=ε
- ins(i)=ε

Ako môžme vidieť, symbol *,ktorý má zadefinované vstupy (`Real`,`Real`) dostane na vstup prirodzené číslo, teda typu `Integer`. Sú dve možnosti ako túto situáciu vyriešiť:

- môžeme zadefinovať novy symbol *, ktorý ma zadefinované vstupy (`Integer`,`Integer`)
- alebo premennu i jednoducho pretypujeme na `Integer`. Castovanie budeme graficky znázorňovať prerušovanou čiarou (ako na poslednom obrázku).

Definícia 1.2. Definujme zobrazenie bve (bounded variable of the expression - viazané premenné výrazu) z Exp do $Pow^{Fin}(\texttt{Var})$ indukciou:

1 Ak $e = v$, kde $v \in \texttt{Var}$, tak $\mathsf{bve}(e) = \emptyset$.

2 Ak $e = s(e_i)_{i<n}$, kde $s \in (Sym)$ a $e = s(e_i)_{i<n} \in InR^s$, tak

$$\mathsf{bve}(e) = \left(\bigcup_{i<n} \mathsf{bve}(e_i) \right) \cup \mathsf{bva}(s).$$

Ak e je výraz, tak množinu $\mathsf{bve}(e)$ nazveme jeho viazanými premennými.

Definícia 1.3. Definujme zobrazenie fve (free variable of the expression - voľné premenné výrazu) z Exp do $Pow^{Fin}(\texttt{Var})$ indukciou:

1 Ak $e = v$, kde $v \in \texttt{Var}$, tak $\mathsf{fve}(e) = v$.

2 Ak $e = s(e_i)_{i<n}$, kde $s \in (Sym)$ a $e = s(e_i)_{i<n} \in InR^s$, tak

$$\mathsf{fve}(e) = \left(\bigcup_{i<n} \mathsf{fve}(e_i) \right) \setminus \mathsf{bva}(s).$$

Ak e je výraz, tak množinu $\mathsf{fve}(e)$ nazveme jeho voľnými premennými.

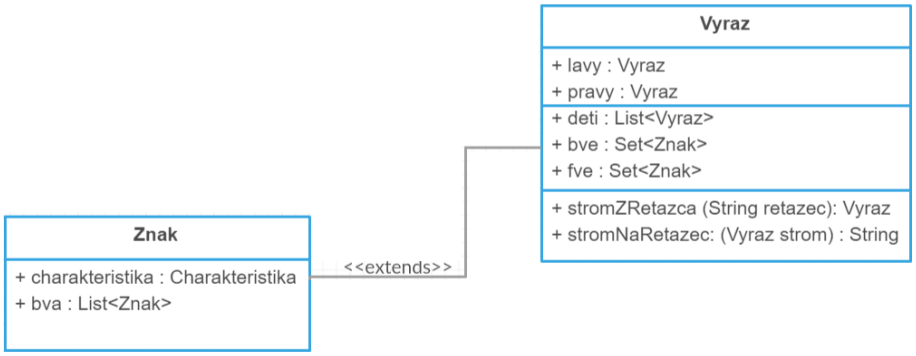
Príklad 1.1. Majme výraz e:

$$\exists a : (\sum_{n=1}^{4}(a+n) - y = 14 + x),$$

- $\mathsf{bve}(e)$=(a, n)
- $\mathsf{fve}(e)$=(x, y).

1.2.1 Implementácia výrazu

Samozrejme keďže výraz je vlastne strom, hlavna rovnomenná entita musí mať stromovú štruktúru:



Ako môžeme vidieť na obrázku, trieda `Vyraz` je implementovaná spôsobom dovoľujúcim reprezentovať ako binárny, tak aj všeobecný (viac-árny) stom. Symboly s ktorými budeme pracovať sú väčšinou binárne, a teda pri písaní metód je jednoduchšie s nimi pracovať ako s binárnym stromom (parsovanie reťazca, prechadzanie stromom, atď.) a však nájdu sa aj symboly ktoré maju vyšší požadovaný počet paramtrov (suma, určitý integral, atď.) a preto je nutné mať štruktúru schopnú reprezentovať aj všeobecný strom.

1.2.1.1 Algoritmy na získanie bve a fve

Výhodou definícii, formulovaných matematickou indukciou, je ich veľmi jednoduchý prepis do rekurzívnych metód:

Algoritmus 1 Získanie viazaných pemenných.

```
1: procedure GetBve(e)                          ▷ e je výraz typu Vyraz
2:     List = ∅                                 ▷ Zoznam znakov
3:     if isVariable(e) then      ▷ Ak sa je e premenna, vrátime prazdny zoznam
4:         return List
5:     for e_i : Deti(e) do
6:         List ← GetBve(e_i) + bva(roo(e))     ▷ Rekurzívne volanie
7:     return List                              ▷ Vraciame List bve
```

Ako môžeme vidieť, báza algoritmu (3, 4) je len akýsi prepis 1. indukčného kroku do pseudokódu a rekurzívne volanie algoritmu zas prislúcha 2. indukčnému kroku z definície 1.2.

Algoritmus na získanie voľných pemenných výrazu e má skoro zhodnú štruktúru:

Algoritmus 2 Získanie voľných pemenných.

```
1: procedure GetFve(e)                          ▷ e je výraz typu Vyraz
2:     List = ∅                                 ▷ Zoznam znakov
3:     if isVariable(e) then      ▷ Ak sa je e premenna, vraciame e
4:         List ← e
5:         return List
6:     for e_i : Deti(e) do
7:         List ← GetFve(e_i) − bva(roo(e))     ▷ Rekurzívne volanie
8:     return List                              ▷ Vraciame List fve
```

1.2.1.2 Algoritmus typovej kontroly

Majme výraz e:

$$\exists a : (\sum_{n=1.1}^{4}(a+n) - y + \frac{\sqrt[3]{64}}{2} = 14 + \sum_{n=1}^{10}(b+n))$$

Pozornému čitateľovy isto neújde, že hodnota n v prvej $\sum_{\dots}^{\dots}$ je typu `Real`. To však symbolu $\sum_{\dots}^{\dots}$ nevyhovuje, keďže $\mathsf{ins}(\sum_{\dots}^{\dots})$=(`Integer`, `Integer`, `Real`).

V niektorých prípadoch, keď dĺžka výrazu nie až tak veľká, sa dá chyba relativne rýchlo odhaliť. No ak su výrazy príliž dlhé, je dobré mať v rukách neomylný nástroj, ktorý takéto výrazy skontroluje za nás:

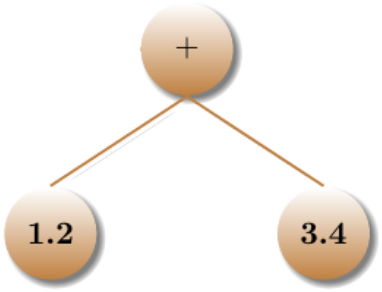
Algoritmus 3 Alogoritmus typovej kontroly.

```
1: procedure PatriDoExp(e)                          ▷ e je výraz typu Vyraz
2:     if isVariable(e) Or isConstant(e) then       ▷ Báza rekurzie
3:         return True
4:     List = ins(roo(e))                           ▷ Zoznam vstupych dat. typov koreňa
5:     for e_i : Deti(e) do
6:         if out(e_i) ≠ List.get(i) Or PatriDoExp(e_i) = False then   ▷ Rekurzia
7:             return False
8:     return True                                  ▷ Nenašla sa nezhoda typov, vraciame True
```

Algoritmus kontroluje strom od spodu. Ak nájde nezhodu typov vráti $False$, ak však cestou ku koreňu žiadnu nezhodu nenajde tak vráti $True$.

1.3 Hodnota výrazu - Ohodnotenie stromov

Keď už rozumieme všetkým symbolom (tj. máme ich interpretáciu), nič nám nebráni priradeniu nejakej hodnoty aj výrazu. Majme jednoduchý výraz `1.2+3.4`. Strom výrazu vyzera takto:



Z obrazku môžeme jednoducho vyčítať, že symbol `+` dostal na vsup symboly žiadaného dátoveho typu. V tomto prípade je ohodnotenie stromu prebieha jednoducho. Aplikujeme funkciu + symbolizovanú symbolom `+` Na dvojicu hodnôt 1.2 a 3.4, ktoré sú symbolizované symbolmi `1.2` a `3.4`, a dostávame výslednú hodnotu 4.6.

Rozoberme si iný prípad, majme výraz `4a-3b`. V tomto prípade nám nestačí poznať interpretáciu v ňom zučastnených sybolov `4, 3, -, *`, ale vplyv na hodnotu výrazu maju aj hodnoty jeho premenných `a` a `b`. Ale keďže premenne nie su syboly ani typy (na ktorých sa typy vzťahjú), informaciu o ich hodnotách potrebujeme osobitne. Hodnota výrazu sa môže v závislosti od ohodnotenia jeho premenných meniť. Ak `a`=1.1 a `b`=2.2, tak hodnota celého vyrazu bude iná ako keby platilo, že `a`=0.5 a `b`=$\frac{3}{4}$. Aby sa však naozaj dala zistiť, hodnoty príslušných premenných musia byť v súlade s interpretáciou ich typov - ak sú teda `a` a `b` typu `Real`, ich hodnoty musia byť z množiny, ktorá je interpretáciou tohto typu, čiže v obvyklom prípade z $\mathbb{R}$ (prípadne môžu mať defaultnú hodnotu pre typ `Real`).

Ohodnotenie premenných je teda vlastne zobrazenie, ktoré každej premennej z istej množiny priradí istú hodnotu v súlade s interpretáciou dátového typu tej-ktorej premennej.

Hodnoty výrazov teda závisia od interpretácie, ale aj od konkrétneho ohodnotenia premenných.

Definícia 1.4. Nech $\mathscr{I}$ je interpretácia a $A \subseteq Var$. Pod označením $EvV_A^{\mathscr{I}}$ (evaluations of variables ohodnotenia premenných) budeme rozumieť množinu $\prod_{v \in A} \mathscr{I}^{Typ}(\mathsf{out}(v))$.

Ľubovoľný prvok niektorej z množín $EvV_A^{\mathscr{I}}$, kde $A \subseteq Var$, budeme nazývať ***$\mathscr{I}$-ohodnotenie premenných***.

Množinu všetkých čiastočných $\mathscr{I}$ ohodnotení označíme $EvV^{\mathscr{I}}$ (evaluations of variables - ohodnotenia premenných).

Príklad 1. Inými slovami, E je $\mathscr{I}$-ohodnotenie premenných práve vtedy, keď pre každú premennú v z $\mathsf{dom}(E)$ platí

$$E(v) \in \mathscr{I}^{Typ}(\mathsf{out}(v)).$$

Nech $\mathscr{I}$ je klasická interpretácia, $\mathsf{typ}(x) = \texttt{Integer}, \mathsf{typ}(y) = \texttt{Real}$ a $\mathsf{typ}(\alpha)$=`Boolean`. Nech E je zobrazenie z množiny $\{x, y, \alpha\}$ také, že platí:

- $E(\alpha) = 1$
- $E(x) = 2$
- $E(y) = 3.4$

Potom E je $\mathscr{I}$ -ohodnotenie premenných.

1.3.1 Implementácia ohodnotenie stromov

Aby sme mohli ohodnotiť nejaky výraz, napríklad `2+x`, musíme najprv prideliť prisluchajúcu hodnotu lisom stromu. Listy stromu sú uzly, ktoré namajú žiadných potomkov, teda nevedu do nich žiadne synapsie. V našom stromovej štruktúre, listami môžu byť:

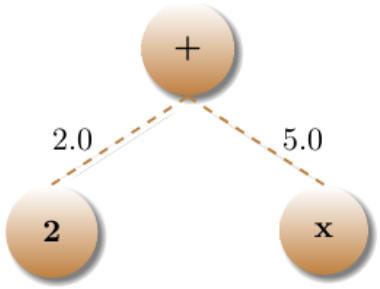
- konštantové symboly(`2`), v tomto prípade je hodnota pridelená hneď pri načitávaní znaku a je uložená v inštančnej premenne triedy `Charakteristika`.
- premenné(`x`), ktorých hodnotu najdeme v textovom súbore, ktorý je reprezentáciou funkie E, $\mathscr{I}$ -ohodnotenia premenných.

Súbor, kde ju uložené hodnoty jednotlivých premenných je formatovaný takto:

*ohodnotenia.txt

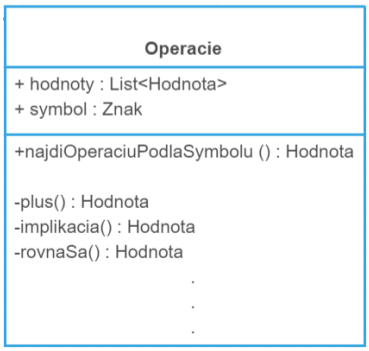
```
1 a:0:
2 b:1:
3 x:5:
4 i:Symbolicka:
5 y:Logika:
```

Keď pri načítavaní znaku narazíme na nejakú premennu, nastáva iterácia cez prvý stĺpec súboru(môžeme ho chápať ako $\mathsf{dom}(E)$) a po najdení zhody s načítaným znakom sa premenne priradí hodnota:



Keď už teda má náš strom ohodnotené všetky listy, môžeme pristúpiť k ohodnocovaniu zložitejších symbolov, resp. k ohodnoteniu celého výrazu. Každému symbolu je jeho hodnota pridelená na základe jeho vstupov. To znamená, že v našom výraze bude koreň stromu, znazorňujúci symbol `+`, ohodnotený na základe už vypočítaných, do neho vchádzajúcich, vsupujúcich hodnôt. To, ako zo vstupných hodnôt vypočítame hodnotu, ktorá ma byť pridelená uzlu reprezentujécemu určity symbol, zaléži na konkretnom symbole, resp. akú funkciu symbol reprezentuje. Napríklad symbol `+` zoberie hodnotu z ľaveho uzla a pripočíta k nej hodnotu z pravého uzla, čiže `+(2,x)=7`(keďže `x` bolo ohodnotene na hodnotu `5`).

O takéto pridelenia hodnôt symbolom sa nám postará trieda `Operacie`:



Trieda `Operacie`, vypočíta hodnotu uzla na základe vstupov a symbolu, ktorý ohodnocovaný uzol reprezentuje. Metóda triedy `najdiOperaciuPodlaSymbolu()` má v sebe jednoduchý switch, ktorý na základe symbolu volá prisluchajúce metódy.

Samotný algoritmus ohodnocovania výrazu vyzerá takto:

Algoritmus 4 Algoritmus ohodnotenia výrazu.

1: **procedure** OHODNOTVYRAZ(e) ▷ e je výraz typu `Vyraz`
2: **if** PatriDoExp(e) = $False$ **then** ▷ Ak výraz nie je skonštruovaný spravne, nemá zmysel ho ohodnotiť
3: **return** $Null$
4: **if** isVariable(e) **Or** isConstant(e) **then** ▷ Báza rekurzie
5: **return** $e.Charakteristika.Hodnota$
6: $List = \emptyset$ ▷ Vstupujúce hodnoty
7: **for** $e_i : Deti(e)$ **do**
8: $List.add$(OhodnotVyraz(e_i)) ▷ Rekurzívne volanie
9: **return** $Operacie(roo(e), List).najdiOperaciuPodlaSymbolu()$ ▷ Vrati hodnotu, na základe symbolu a vstupnych hodnôt

Záver. Hotová je kompletná implementácia triedy Vyraz, ktorá je schopná načítať aritmeticky výraz, či logickú formulu, ktorú následne vie previesť do stromovej štruktúry. Objekt triedy Vyraz, teda vybudovaný strom sa vie typovo skontrolovať a ohodnotiť (počítanie rovníc, vyhodnotenie matematického výroku). Objekt vie ďalej generovať podľa požiadavky s vyššie popísanými vlastnosťami.

Na čom sa pracuje. Pracujem na entite dôkaz, ktorá by mala byť schopná priblížiť sa ku konštrukcii formálneho dôkazu pomocou odvodzovaniach pravidiel. Zatiaľ program vie povedať, či je alebo nie je skúmaná formula axiómou.

Literatúra. 1. http://ics.upjs.sk/ krajci/skola/vyucba/ucebneTexty/logika-stromy.pdf.

2. Goldstern Martin and Judah Haim, The incompleteness phenomenon, A new course in mathematical logic, A K Peters, Wellesley, Mass., 1995, xiii + 247 pp.