Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Program pro práci s prozódií a rýmem v básních

2021

Vladimír Opluštil

Vedoucí práce: Mgr. Petr Osička, Ph.D.

Studijní obor: Aplikovaná informatika, prezenční forma

Bibliografické údaje

Autor:	Vladimír Opluštil
Název práce:	Program pro práci s prozódií a rýmem v básních
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2021
Studijní obor:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	Mgr. Petr Osička, Ph.D.
Počet stran:	40
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Vladimír Opluštil
Title:	A software tool for prosody and rhyme in poetry
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2021
Study field:	Applied Computer Science, full-time form
Supervisor:	Mgr. Petr Osička, Ph.D.
Page count:	40
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Tématem práce je vytvoření programu pro práci s prozodickými vlastnostmi a rýmy v básních. Uživatel může zadat požadované vlastnosti a program je srovná se zadaným úryvkem a vyznačí v něm místa, která odpovídají zadání, čímž může uživateli pomoci se skládáním, překladem nebo studiem básní. Při tvorbě byl použit programovací jazyk C# a technologie WPF.

Synopsis

The topic of this thesis is a creation of a software tool for working with prosody and rhyme in Czech poetry. The user can set the desired properties and the program compares them to a supplied text and highlights parts corresponding to the instructions, thereby helping the user with composing, translating or studying poetry. The program was created using C# and WPF.

Klíčová slova: básně; prozódie; rýmy; C#

Keywords: poetry; prosody; rhymes; C#

Chtěl bych poděkovat Mgr. Petru Osičkovi, Ph.D. za vedení a rady při vytváření této práce.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

1 Úvod

V úvodní části budou popsány teoretické základy prozódie a rýmů, poté následuje několik příkladů aplikací s podobným zaměřením. Druhá sekce se zabývá implementací a popisuje funkčnost nejdůležitějších částí programu. Třetí sekce se zabývá popisem aplikace ze strany uživatelské a pokyny pro správné používání.

1.1 Teoretický úvod

Prozódie je v básnictví pravidelnost veršů podle počtu slabik a jejich přízvučnosti nebo délky. Podle konkrétních vlastností se rozeznává několik druhů prozódie.

1.1.1 Sylabotónická prozódie

Sylabotónická prozódie je založená na pravidelném počtu slabik ve verši a jejich přízvucích. V češtině, stejně jako ve většině moderních evropských jazyků, je nejčastěji používaná.

Přízvučnost je v češtině, na rozdíl od například angličtiny, němčiny nebo ruštiny, velmi pravidelná. Základním pravidlem je, že hlavní přízvuk je na první slabice slova. Vedlejší přízvuky jsou na ostatních lichých slabikách slova. Slova pětislabičná a delší ale mohou mít přízvuky i po třech slabikách (tedy první, čtvrtá, sedmá). Pokud přízvuk vychází na poslední slabiku slova, nemusí tam nutně být (jednoslabičné slovo tedy může být bez přízvuku).

V tabulce 1 uvádím možnosti přízvuku ve slovech o jedné až osmi slabikách. — značí přízvučnou slabiku. ∪ značí nepřízvučnou slabiku. × značí slabiku, která může být přízvučná nebo nepřízvučná (anceps).

Počet slabik	Přízvuky
1	×
2	— ∪
3	— ∪ ×
4	— ∪ — ∪
5	— ∪ — ∪ × — ∪ ∪ — ∪
6	— ∪ — ∪ — ∪ — ∪ ∪ — ∪ ∪
7	— ∪ — ∪ — ∪ × — ∪ ∪ — ∪ ∪ ×
8	— ∪ — ∪ — ∪ — ∪ — ∪ ∪ — ∪ ∪ — ∪

Tabulka 1: Přízvuky podle počtu slabik.

Dále je důležité podotknout, že tato slova se ne vždy shodují se slovy napsanými. Jednoslabičné předložky se většinou berou jako součást následujícího slova,

ale nemusí to být důsledně dodržováno. Podobně příklonky (především slova jako by, byste, bychom, zájmeno se) se berou jako součást slova předchozího.

V následujících dvou verších [14]

> Na topole nad jezerem
> seděl vodník podvečerem:

budou přízvuky stejně, tedy — ◡ — ◡ — ◡ — ◡, protože „Na topole“ a stejně tak „nad jezerem“ se bere dohromady.

1.1.2 Metrika

Konkrétní posloupnost těchto prozodických značek, které má vyhovovat verš označujeme jako metrum nebo metr. Metra mají často své názvy, například právě ukázané — ◡ — ◡ — ◡ — ◡ se označuje jako trochejský tetrametr a v češtině patří k nejběžnějším. Většina metrů v češtině je podobně jednoznačných, ale pokud se například překládá z antických jazyků, kde se používala časoměrná prozódie, jež bude popsána níže, a překladatel chce napodobit antické metrum, vznikají metra s více podobami. První verš z českého překladu Iliady [12]

> O hněvu Péléovce, ó bohyně, Achilla zpívej,

by měl čtení s přízvuky — ◡ ◡ — ◡ — ◡ ◡ — ◡ ◡ — ◡ ◡ — ◡. „O hněvu“ se čte dohromady, „ó“ se čte bez přízvuku. To je ale jen jedna z možných podob. Metrum básně (hexametr) vypadá takto — ◡̲◡̲ — ◡̲◡̲ — ◡̲◡̲ — ◡̲◡̲ — ◡. Zde na místě každého ◡̲◡̲ mohou být dvě nepřízvučné nebo jedna nepřízvučná slabika. Celkem je tedy až 32 různých tvarů.

Další běžný metr je například jambický pentametr ◡ — ◡ — ◡ — ◡ — ◡ —. Rýmované verše jsou někdy kratší nebo delší o poslední slabiku.

Časté jsou případy, že metr nezůstává stejný pro všechny verše, ale opakuje se po slokách. V následující sloce [15]

> Je to chůze po tom světě —
> kam se noha šine:
> sotva přejdeš jedny hory,
> hned se najdou jiné.

mají liché verše metrum — ◡ — ◡ — ◡ — ◡ a sudé — ◡ — ◡ — ◡, stejně tak i v dalších slokách básně. Sapfická strofa, převzatá z antické poezie, by měla metra následující:

— ◡ — ◡ — ◡ ◡ — ◡ — ◡
— ◡ — ◡ — ◡ ◡ — ◡ — ◡
— ◡ — ◡ — ◡ ◡ — ◡ — ◡
— ◡ ◡ — ◡

Každý čtvrtý verš je tedy kratší než předchozí tři.

1.1.3 Časoměrná prozódie

Časoměrná prozódie je založena na pravidelném počtu slabik a střídání dlouhých slabik s krátkými. Pochází z poezie starého Řecka a nejvíce se využívala v latině a staré řečtině, těmto jazykům totiž zcela vyhovovala a jiné druhy prozódie se při básnění v podstatě nevyužívaly. Čeština je jedním z mála moderních jazyků, ve kterém je možné tvořit časoměrné verše. V současné době se ale časoměrná prozódie téměř nevyužívá. Nejvíce se využívala od 16. do začátku 19. století, kdy ji nahradila sylabotónická. V překladu z antických autorů se ale udržela až do konce 19. století. Nyní je na místě vysvětlit, které slabiky jsou dlouhé a které krátké.

Pokud je slabika tvořena dlouhou samohláskou nebo dvojhláskou, říkáme o ní, že je přirozeně dlouhá. Například ve slovech „dlouhá“, „být“, „svítání“ jsou všechny slabiky přirozeně dlouhé.

Je-li je slabika tvořena krátkou samohláskou nebo slabikotvornou souhláskou, dochází k jednomu ze tří případů:

Pokud je její samohláska následována alespoň dvěma souhláskami, které mohou být i v další slabice nebo slově, říkáme o ní, že je pozičně dlouhá. Tak k tomu dochází třeba slovech „most“, „bystrost“

Pokud je však následována jen dvěma souhláskami, z nichž jedna je l, m, n, r nebo ř (takzvané likvidy), jedná se o obojetnou slabiku a můžeme ji brát jako dlouhou nebo krátkou. K tomu dochází třeba ve slovech „jenž“, „park“

Pokud ji následuje nanejvýš jedna souhláska, jedná se o krátkou slabiku. To je například u slov „svoboda“, „kolo“, „ruka“.

Proti těmto pravidlům jsou dvě důležité výjimky. U poslední slabiky verše se neřeší délka a brává se jako obojetná. „Ě“ se většinou bere pouze jako „e“, i když je vyslovováno „je“ nebo „ně“. Slova „země“,„tobě“ tedy tvoří pouze krátké slabiky, i když by se mohlo zdát, že první slabika je pozičně dlouhá.

Dlouhé slabiky značíme —, krátké ∪, obojetné ×.

Pro větší přehled zde jeden verš rozeberu. [16]

Aj, zde leží zem ta, před okem mým smutně slzícím

Délky budou následující: — ∪ ∪ — × × ∪ ∪ × — — ∪ ∪ — ×.
„Aj“ je pozičně dlouhá, po „a“ následují tři souhlásky.
„zde“ je krátká, po „e“ následuje jen jedna souhláska.
„le“ je krátká, po „e“ následuje jen jedna souhláska.
„zem“ je obojetná, po „e“ následují dvě souhlásky, z nich jedna je „m“.
„ta“ je obojetná, po „a“ následují dvě souhlásky, z nich jedna je „ř“.
„pře“ je krátká, po „e“ následuje jen jedna souhláska.
„do“ je krátká, po „o“ následuje jen jedna souhláska.
„kem“ je obojetná, po „e“ následují dvě „m“.
„mým“ má dlouhou samohlásku, tedy přirozeně dlouhá.
„smut“ je dlouhá, po „u“ následují dvě souhlásky.
„ně“ je krátká, po „e“ následuje jen jedna souhláska.

„sl“ je krátká, po „l“ následuje jen jedna souhláska.
„zí“ má dlouhou samohlásku, tedy přirozeně dlouhá.
„cím“ je na konci verše, tedy obojetná.
Konkrétní čtení bude — ⏑ ⏑ — — — ⏑ ⏑ — — — ⏑ ⏑ — —, jedna z variant časoměrného hexametru — ⏓ — ⏓ — ⏓ — ⏓ — ⏓ — ⏔, kde ⏓ znamená buď ⏑ ⏑ nebo —.

Součástí časoměrné prozódie jsou také cézury.
— — ⏑ — — ❘ — ⏑ ⏑ — ⏑ — značí, že po páté slabice musí být mezera.

1.1.4 Sylabická prozódie

Sylabická prozódie je založena na pravidelném počtu slabik ve verši. Na přízvucích či délce slabik nezáleží. Verše se tedy nepopisují metrem, ale jen počtem slabik. Sylabická obdoba sapfické strofy, zmíněné dříve by se popsala jednoduše tak, že první tři verše mají jedenáct slabik a čtvrtý pět slabik.

Rýmované verše mohou být opět kratší nebo delší o slabiku, může tedy být například báseň, kde mají verše jedenáct, někdy deset slabik.

1.1.5 Volná prozódie

Volná prozódie nemá pravidelný rytmus. Nezáleží na počtu slabik, přízvucích, ani délkách. V češtině se využívá v moderní poezii od minulého století.

1.1.6 Tónická prozódie

V tónické prozódii záleží na počtu přízvuků ve verši, ale na rozdíl od sylabotónické nezáleží na jejich rozmístění a celkovém počtu slabik. Využívala se zejména ve staré severské poezii. V češtině se nevyužívá, a proto jsem ji do projektu nezařadil. Zde ji uvádím jen pro úplnost.

1.1.7 Rýmy

Rým je souzvuk slabiky nebo dvou slabik na koncích veršů. Rozeznáváme u nich několik rozdělení, která zde nyní vyložím.

Dle přízvuku dělíme rýmy na mužské, když je poslední slabika přízvučná, a ženský, když není. V mužském rýmu se má rýmovat jedna slabika, v ženském dvě. V následující sloce [14] je rým prvních dvou veršů ženský, druhých dvou mužský.

> Ach nechoď, nechoď na jezero,
> zůstaň dnes doma, moje dcero!
> Já měla zlý té noci sen:
> nechoď, dceruško, k vodě ven.

Dále rozeznáváme kvalitu rýmu. Jako úplný a bohatý se označuje rým, pokud splňuje následující tři podmínky: Rýmovaná slova mají stejný počet slabik. Rýmující se samohlásky mají stejnou délku. Rýmující se souhlásky (všechny za první

samohláskou rýmu) jsou stejné. Pokud rým dodržuje jen jedno nebo dvě z těchto pravidel, označuje se jako úplný a postačující rým. Slabším rýmem je rým překlenutý, kdy jeden verš má na konci proti druhému ještě souhlásku nebo skupinu souhlásek. Ten se může opět dělit na bohatý a postačující.

Kromě toho rozeznáváme rýmy podle rozmístění ve sloce. To se většinou značí přidělením stejného písmene rýmujícím se veršům. Předchozí příklad by se tak označil jako AABB. Tento konkrétně se nazývá sdružený rým. Rýmování v jiných slokách může být složitější, například anglický sonet ABABCDCDEFEFGG.

Existují ale případy, kdy se verše rýmují i napříč slokami. Například v Božské komedii užívá Dante tercíny, které se rýmují ABA BCB CDC... Člověk z tohoto zápisu pochopí, že bude následovat DED, ale pro počítač jsem zavedl vlastní označení pomocí čísel, kdy číslo říká, za kolik veršů následuje rým s aktuálním veršem, 0 pokud se s ním žádný další nerýmuje. Tercína by v tomto značení byla 220, sdružený rým 1010, anglický sonet 22002200220010. V případě, že by byla mezera mezi rýmy více než 9 veršů, je možné čísla oddělovat čárkou, prakticky se to ale neděje. Někdy jsou rýmy nahodile, bez pravidelného řazení.

1.2 Podobné aplikace

Aplikací pro práci s básněmi vzniklo několik. Z těch, se kterými jsem se setkal, mohu jmenovat následující:

Poesialatina.it [7] obsahuje zdrojové texty latinské a řecké poezie, u které je možné nechat vyznačit délky slabik, zjistit jakému metru jednotlivé básně odpovídají a vyhledat některé zvláštnosti.

Romancomedy.wulib.wustl.edu [8] obsahuje databázi veršů latinských dramatiků Plauta a Terentia, kde je možné vyhledávat verše podle konkrétního metra, dramatu, postavy nebo návaznosti.

Howmanysyllables.com [9] je stránka pomáhající s vytvářením anglických básní. Počítá počet slabik v jednotlivých verších a ke slovům nabízí rýmy. Také umožňuje procvičování v počítání slabik.

Rymy.cz [10] je databáze rýmujících se českých slov, která umožňuje vyhledávání s nastavením různých preferencí. Podobných aplikací existuje větší množství v různých jazycích.

Versologie.cz [11] je česká stránka, která obsahuje aplikaci na procvičení poznávání různých meter.

2 Implementace

Funkčnost programu je založena na třech hlavních částech:

Reader zpracovává jednotlivé verše a podle nastavených vlastností vrací strukturu obsahující informace o prozodických vlastnostech.

Rhyme pro dvojici konců veršů zjistí, zda se rýmují a jaká je kvalita rýmu.

Comparator prochází celý text po řádcích, předává je *Readeru* a porovnává získané struktury se vzory. Také vybírá dvojice, které se mají rýmovat a dává je k porovnání *Rhyme*. O špatných nebo správných verších (záleží na volbě uživatele) si ukládá informace.

2.1 Reprezentace prozodických vlastností a struktury

ProsodicElement je výčtový typ, jehož prvky jsou používány k reprezentaci prozodických vlastností verše a metrů ve strukturách.

Teoreticky by bylo možné mít zde jen dva prvky, totiž breve (název znaku ◡) a longum (název znaku —). Mezery by bylo možné reprezentovat až v nadřazených strukturách, například rozdělením verše do podstruktur, které reprezentují jednotlivé části oddělené mezerou. Podobným způsobem mohly být reprezentovány i nejednoznačné části, kde lze dosadit různé skupiny prvků, mezi nimi i anceps, který je buď breve nebo longum.

Takový systém by ale potřeboval mnohem složitější struktury, které by měly časově horší vytváření a porovnávání. Systém, se kterým jsem se rozhodl pracovat má pět prvků, popsaných v tabulce 2.

Hodnota	Význam
Breve	Nepřízvučná slabika v sylabotónické prozódii. Krátká slabika v časoměrné prozódii.
Longum	Přízvučná slabika v sylabotónické prozódii. Dlouhá slabika v časoměrné prozódii.
Anceps	Obojetná slabika. V sylabické prozódii jakákoliv slabika.
Space	Mezera nebo cézura v metru (místo, kde musí být mezera).
Pointer	Odkaz na podstrukturu v místě, kde je více možností.

Tabulka 2: Výčtový typ *ProsodicElement*

Anceps by se dal jednoduše nahradit pointerem na strukturu se dvěma možnostmi, breve nebo longum, ale vyskytuje se tak často, že z hlediska rychlejšího výpočtu je lepší mít jej samostatně.

Celkem lze pomocí těchto prvků značit zadání (metry) i různé způsoby čtení ve trojici prozódií - sylabotónické, časoměrné a sylabické. Poslední volná prozódie však žádnou pravidelnost nemá a tedy není potřeba u ní řešit tyto věci.

List⟨ProsodicElement⟩ je nejjednodušší z prozodických struktur. Jde o seznam typu *ProsodicElement*, nevyužívající pointer. Jedná se o obdobu zápisu značkami

⏑ — × | . V aplikaci však pro jednodušší zápis uživatelem používám znaky u, -, x, |. Kvůli chybějícímu pointeru není možné vyjádřit nejednoznačné části, a tedy je případně nutné použít více seznamů pro všechny alternativy.

V příkladech budu používat časoměrný hexametr (— ⏑⏑ — ⏑⏑ — ⏑⏑ — ⏑⏑ — ⏑⏑ — ⏓, zmíněný již v části o časoměrné prozódii. Pokud bychom chtěli hexametr zadat tímto způsobem, bylo by třeba 32 seznamů o celkem 464 prvcích. V grafických příkladech budu pro velký rozsah celého verše ukazovat jen běžnější variantu (— ⏑⏑ — ⏑⏑ — ⏑⏑ — ⏑⏑ — ⏑ ⏑ — ⏓, která se liší od první tím, že poslední dvojice krátkých slabik není nahrazena dlouhou (tato náhrada se vyskytuje jen zřídka). Celkový počet prvků je tedy přibližně poloviční. Podoba za použití seznamu seznamů je v obrázku 1.

List[0]: — ⏑ ⏑ — ⏑ ⏑ — ⏑ ⏑ — ⏑ ⏑ — ⏑ ⏑ — ×
List[1]: — ⏑ ⏑ — ⏑ ⏑ — ⏑ ⏑ — — — ⏑ ⏑ — ×
...
List[15]: — — — — — — — — — ⏑ ⏑ — ×

Obrázek 1: Tvary časoměrného hexametru v seznamu seznamů.

Druhou možností je umístit prvky do stromové struktury. To obstarává třída *ProsodicTree*. Má atributy *List* typu *List⟨ProsodicElement⟩* a *Children* typu *List⟨ProsodicTree⟩*. *List* obsahuje seznam jednoznačných prvků, *Children* obsahuje podstromy, které mají jako List jednu z nejednoznačných částí společně s navazující jednoznačnou částí. Je to složitější struktura, ale celkový počet prvků se může značně snížit. Uložení stejného metra je ukázané na obrázku 2. Celkem stačí 140 prvků.

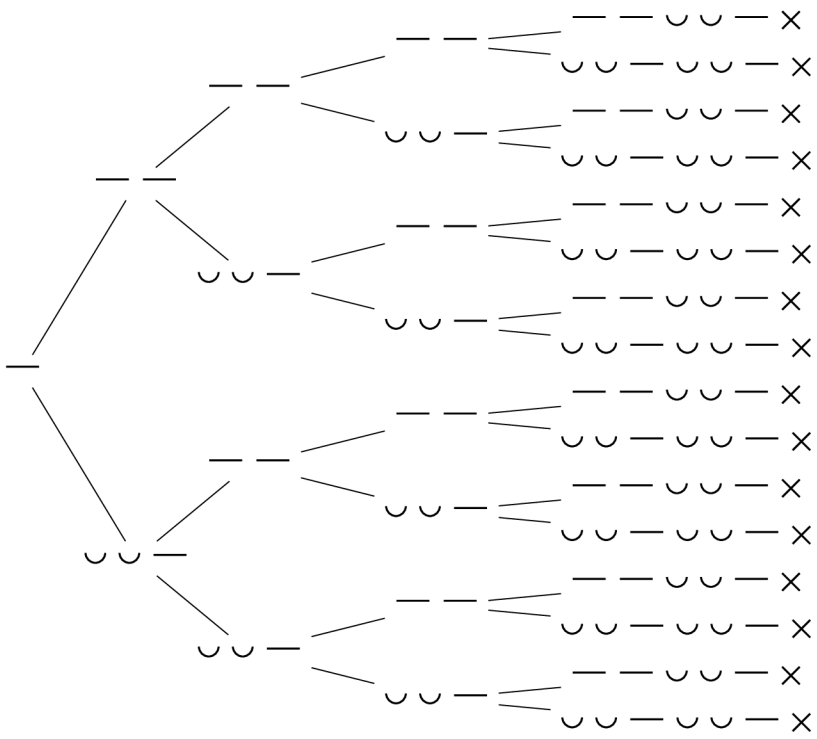
Nejskladnější možností je struktura *List⟨Prosody.Node⟩*. Seznam tvoří objekty struktury *Prosody.Node*, která má atributy *Element* typu *ProsodicElement* a *Alternatives* typu *List⟨List⟨Prosody.Node⟩⟩*. Reprezentace hexametru (pokud ○ je odkaz) je v na obrázku 3. Celkem stačí 25 prvků.

2.2 Vznik a převod struktur

Struktury vznikají ve dvou případech, pokaždé jako *List⟨Prosody.Node⟩*, jako vzor zadaný uživatelem nebo při čtení verše pomocí *Readeru*.

O přeložení vzoru zadaného uživatelem v textové podobě se stará třída *Interpret*. Vzor zadaný uživatelem může mít například tento tvar:
b = (uu,-)
-b-b-b-b-uu-x
-b-b-|-uu-uux

Některé symboly byly popsány už dříve: - je longum, u je breve, x je anceps, | je mezera. Závorka je náhradou za pointer, posloupnosti symbolů v ní, oddělené čárkou značí různé alternativy na tom místě. Pokud je v závorce jen jedna



Obrázek 2: Tvary časoměrného hexametru ve stromové struktuře.

posloupnost, chápe se jako tento obsah nebo nic. (u,) i (u) značí breve nebo nic. Závorky se mohou zanořovat i do sebe, například (-(u,uu),u-,uuu).

Na prvním řádku je deklarace symbolu. Každé b na dalších řádcích je tak nahrazeno (uu,-). Symbolem je jeden znak, kromě těch, které byly už dříve využity. Řádků s deklarací může být více, ale každý symbol lze deklarovat jen jednou. Symboly se mohou zadat na jakémkoliv řádku před prvním použitím.

Další řádky už obsahují vzory samotné. Pokud je jich více, znamená to, že se postupně střídají. V tomto případě by měl každý lichý verš básně odpovídat vzoru na 2. řádku a každý sudý verš vzoru na 3. řádku.

Přeložení prozodického vzoru zajišťuje metoda ve zdrojovém kódu 1. Metoda vrací seznam typu *Prosody.Node* odpovídající jednomu řádku zadání. Nejdříve se vytvoří prázdný seznam (3), pak se prochází string po znacích (4-23). Pokud je aktuální znak levá závorka (6), je k ní metodou *FindRightBracket*, která přihlíží k tomu, že závorky mohou být zanořené, nalezena odpovídající pravá (8). Řetězec v závorce je metodou *Split* rozdělen podle čárek na podřetězce. Tato metoda ignoruje čárky ve vnořených závorkách (9). Je vytvořena *Prosody.Node* s *Elementem* Pointer a do *Alternatives* jsou vloženy rekurzivně vypočtené vzory odpovídající podřetězcům závorky (10-12). Pokud závorka obsahovala jen jeden podřetězec, je do alternativ vložen ještě prázdný seznam (13-14). Nově vytvořená *Prosody.Node* se vloží do seznamu (15) a závorka se přeskočí (16). Pokud aktuální znak není závorka, vyhledá se mu odpovídající seznam a připojí k vy-

List (Elementy): — ○ — ○ — ○ — ○ — ∪ ∪ — ×
List[1].Alternatives[0]: ∪ ∪
List[1].Alternatives[1]: —
List[3].Alternatives[0]: ∪ ∪
List[3].Alternatives[1]: —
List[5].Alternatives[0]: ∪ ∪
List[5].Alternatives[1]: —
List[7].Alternatives[0]: ∪ ∪
List[7].Alternatives[1]: —

Obrázek 3: Tvary časoměrného hexametru v seznamu s odkazy.

tvářenému seznamu (18-22). Po projití celého řetězce je vrácen hotový seznam (24).

Symboly jsou uloženy v kolekci typu *Dictionary⟨char, List⟨Prosody.Node⟩⟩*. Na začátku obsahuje jen symboly -, u, x, | a jim odpovídající jednoprvkové seznamy. Uživatel může přidat symbol pro jakýkoliv znak kromě již definovaných symbolů, kulatých závorek, čárky, rovná se nebo mezery.

Výše popsané vzory se používají pro Sylabotónickou a Časoměrnou prozódii, pro Sylabickou prozódii se vzory zadávají číslem, odpovídajícím počtu slabik, například 10, rozsahem, odděleným pomlčkou, například 8-10, nebo více hodnotami či rozsahy, oddělenými čárkami, například 6,8,10-12.

Převod mezi strukturami je využíván ve dvou případech. První je převod z *List⟨Prosody.Node⟩* na *ProsodicTree*, který se používá u vzorů zadaných uživatelem, které jsou svým způsobem zadání velmi blízké *List⟨Prosody.Node⟩*, ale pro další srovnání je výhodné použít strom.

Ve zdrojovém kódu 2 je metoda na převod z *List⟨Prosody.Node⟩* na *ProsodicTree*. Postupně je procházen seznam v argumentu (4-15) a všechny elementy, které nejsou *Pointer* se přidávají do seznamu *List* (atribut stromu) (6-7). Když se narazí na *Pointer*, do seznamu *Children* (atribut stromu) se přidají stromy rekurzivně vytvořené pro každou z alternativ této *Node* s *Pointerem*, rozšířenou o zbytek seznamu z argumentu (10-12). Protože zbytek seznamu je už obsažen v potomcích, cyklus skončí (13).

Převod z *List⟨Prosody.Node⟩* na *List⟨List⟨ProsodicElement⟩⟩* je o něco složitější. Funkce je ve zdrojovém kódu 3. *MainList* obsahuje nezpracovanou část, *buildingList* zpracovanou část a *doneList* hotové seznamy. *BuildingList* a *doneList* jsou na začátku prázdné. Pokud *mainList* neobsahuje žádný *Pointer*, převede se na seznam typu *ProsodicElement*, připojí za *buildingList* a přidá do *doneList* (3-5). Jinak se do *buildingList* přidá převedená část před *Pointerem*, v *mainList* se nechá jen část po *Pointeru* (8-10). Pro každou z alternativ, co byla v *Node* s *Pointerem* se metoda volá rekurzivně, tak že za *mainList* se dosadí alternativa rozšířená o *mainList*, za *buildingList* jeho kopie, *doneList* zůstane stejný (11-16). Vrací se *doneList*, obsahující všechny alternativy (18).

```
1  public List<Prosody.Node> ParseProsodicPattern(string s)
2  {
3     List<Prosody.Node> list = new List<Prosody.Node>();
4     for (int i = 0; i < s.Length; i++)
5     {
6        if (s[i] == '(')
7        {
8           int idx = FindRightBracket(s, i);
9           List<string> substrings = Split(s.Substring(i + 1, idx - i
                - 1));
10          Prosody.Node n = Prosody.MakePointer();
11          foreach (string substring in substrings)
12             n.Alternatives.Add(ParseProsodicPattern(substring));
13          if (substrings.Count == 1)
14             n.Alternatives.Add(new List<Prosody.Node>());
15          list.Add(n);
16          i = idx;
17       }
18       else
19       {
20          var v = symbols[s[i]];
21          list.AddRange(v);
22       }
23    }
24    return list;
25 }
```

Zdrojový kód 1: *Interpret*: Přeložení vzoru prozódie.

```
1  public ProsodicTree(List<Prosody.Node> prosodicNodeList)
2  {
3     List = new List<ProsodicElement>();
4     for (int i = 0; i < prosodicNodeList.Count; i++)
5     {
6        if (prosodicNodeList[i].Element != ProsodicElement.Pointer)
7           List.Add(prosodicNodeList[i].Element);
8        else
9        {
10          Children = new List<ProsodicTree>();
11          foreach (List<Prosody.Node> alternative in prosodicNodeList
                [i].Alternatives)
12             Children.Add(new ProsodicTree(alternative.Concat(
                   prosodicNodeList.Skip(i + 1)).ToList()));
13          break;
14       }
15    }
16 }
```

Zdrojový kód 2: *ProsodicTree*: Vytvoření podle $List\langle Prosody.Node\rangle$.

```
1  public static List<List<ProsodicElement>> ListAllAlternatives(List<
       Node> mainList, List<ProsodicElement> buildingList, List<List<
       ProsodicElement>> doneList)
2  {
3     int pointerIdx = mainList.FindIndex(n => n.Element ==
          ProsodicElement.Pointer);
4     if (pointerIdx == -1)
5        doneList.Add(buildingList.Concat(mainList.ConvertAll(n => n.
             Element)).ToList());
6     else
7     {
8        buildingList.AddRange(mainList.Take(pointerIdx).ToList().
             ConvertAll(n => n.Element));
9        List<List<Node>> alternatives = mainList[pointerIdx].
             Alternatives;
10       mainList = mainList.Skip(pointerIdx + 1).ToList();
11       foreach (List<Node> alternative in alternatives)
12       {
13          List<Node> newMainList = new List<Node>(alternative.Concat(
                mainList));
14          List<ProsodicElement> buildingListCopy = new List<
                ProsodicElement>(buildingList);
15          doneList = ListAllAlternatives(newMainList,
                buildingListCopy, doneList);
16       }
17    }
18    return doneList;
19 }
```

Zdrojový kód 3: *Prosody*: Vytvoření $List\langle List\langle ProsodicElement\rangle\rangle$.

2.3 Readery

Readery slouží k převodu verše v textovém tvaru na *List⟨Prosody.Node⟩* obsahující informace o jeho prozódii. V programu jsou tři třídy - *Reader*, který zajišťuje počítání slabik, a jeho potomci, *SyllabotonicReader*, který navíc určuje přízvuky slabik, a *WeightBasedReader*, který určuje délku slabik.

Pro rychlejší určení, o jaký typ znaku se jedná, vznikl výčtový typ *CharacterType*, který obsahuje *Vowel* - samohláska, *SyllabicConsonant* - souhláska, která může tvořit slabiku, *NonSyllabicConsonant* - jiná souhláska, *Space* - mezera, *Comment* - znak uvozující komentář a *Other* - jiný znak.

V *Readeru* je kolekce typu *Dictionary⟨char, CharacterType⟩*, která znakům přiřazuje jejich typ. V základním nastavení jsou znaky z české abecedy, ale pro případ, že by uživatel potřeboval použít některé jiné, třeba kvůli cizím jménům, je možnost je přidat. Komentáře jsou uvozovány znakem @, ale je opět možné znak změnit. *Dictionary* je implementované pomocí hashovací tabulky, vyhledání je tedy velmi rychlé.

Dále *Reader* obsahuje čtyři kolekce typu *HashSet⟨string⟩*, na uložení dvojhlásek, předložek, oddělitelných předložek a příklonek. *HashSet* je také implementovaná pomocí hashovací tabulky, a tak je zjištění, zda řetězec do množiny patří, opět rychlé.

Reader má dvě nastavení: *DivideDiphtongs* na rozdělování dvojhlásek. Pokud je zapnuto, dvojhlásky mohou tvořit jednu nebo dvě slabiky, jinak tvoří jen jednu. *ReadAllPrepositionsAsSeparate* na oddělení předložek. Jednoslabičná předložka se obyčejně bere jako součást následujícího slova. Některé předložky ale mohou být zároveň i jiným slovním druhem, například „se“ je častěji zájmeno. Takové se uvedou mezi oddělitelnými předložkami a budou zvažovány obě možnosti, tedy že jsou součástí následujícího slova, i že jsou samostatné. V některých básních ale není pravidlo o jednoslabičných předložkách dodržováno přísně a proto je možné vybrat tuto možnost, která bude brát jako oddělitelné všechny zadané předložky.

Před čtením jsou verše převedeny pomocí třídy *StringPreprocessor*. Ta dělá čtyři věci: Převede text na malá písmena. Odstraní znaky, které nejsou definovány jako písmena, mezery nebo uvozovky komentářů. Na začátek a konec verše přidá znak mezery pro jednodušší zpracování. Provede případné další změny, aby text odpovídal výslovnosti.

Funkce zajišťující přečtení jednoho verše pomocí *Readeru* je ve zdrojovém kódu 4. V atributu *list* je vytvořen nový seznam (3). Do atributu *line* se uloží řetězec z argumentu (4). *Reader* poté volá funkci *LineStart*, která u potomka *WeightBasedReader* resetuje některé atributy (5). Poté se prochází řetězec po znacích a podle jejich typu se volají příslušné metody (6-28). Na konec se zavolá funkce *LineEnd*, která u potomků připojí případné poslední *Prosody.Node* (30). Nakonec je vrácen list (31). Tyto hlavní rysy jsou všem *Readerům* společné, liší se funkcemi pro jednotlivé typy znaků.

Reader pouze počítá slabiky, jeho chování je tedy nejjednodušší. Při samohlásce zkontroluje, zda netvoří dvojhlásku. Pokud ano, přesune se na její konec a přidá jeden anceps, nebo jeden až dva, pokud je nastaven příznak *divideDiph-*

```
1  public List<Prosody.Node> ReadString(string s)
2  {
3     list = new List<Prosody.Node>();
4     line = s;
5     LineStart();
6     for (idx = 1; idx < line.Length - 1; idx++)
7     {
8        if (characters.TryGetValue(line[idx], out CharacterType
             characterType))
9           switch (characterType)
10          {
11             case CharacterType.vowel:
12                EvaluateVowel();
13                break;
14             case CharacterType.syllabicConsonant:
15                EvaluateSyllabicConsonant();
16                break;
17             case CharacterType.nonSyllabicConsonant:
18                EvaluateNonSyllabicConsonant();
19                break;
20             case CharacterType.space:
21                EvaluateSpace();
22                break;
23             case CharacterType.comment:
24                goto LineEnd;
25             default:
26                break;
27          }
28    }
29    LineEnd:
30    LineEnd();
31    return list;
32 }
```

Zdrojový kód 4: *Reader*: Přečtení verše.

tongs. Pokud dvojhlásku netvoří, přidá jeden anceps. U souhlásky, která může tvořit slabiku zkontroluje, zda ji tvoří (před ní je souhláska a za ní souhláska nebo konec slova), a pokud ano, přidá anceps, jinak nic nedělá. U jiných souhlásek nic nedělá. U mezery přidá mezeru, pokud není po jednoslabičné předložce. U znaku uvozujícího komentář přestane vyhodnocovat zbytek slova.

SyllabotonicReader zjišťuje i přízvuky slabik, takže hned při zjištění slabiky nepřidává *Prosody.Node* do seznamu, ale počítá, kolik slabik slovo tvoří a započítává je až později. Protože nastavením *divideDiphtongs* vzniká nejasnost, zda dvojhláska tvoří jednu nebo dvě slabiky, počítá se i počet slabik které mohou být přidány navíc. Jednoslabičné předložky a příklonky mohou být se slovem dohromady, co se týče přízvuku, a tedy se počítají také, ale zvlášť. Vznikají dva seznamy čísel, z nichž jeden obsahuje počty slabik a druhý počty případných

dalších slabik a z těchto se pak vypočítávají možnosti, když se *Reader* dostane na mezeru a předchozí slovo nebylo jednoslabičnou předložkou, ani následující není příklonkou. Jak se počítají možné přízvuky z délky slova jsem ukázal v úvodu 1. *SyllabotonicReader* si při vytvoření předpřipraví seznamy pro až desetislabičná slova, které pak používá opakovaně, aby je nemusel vypočítávat při každém slově. *SyllabotonicReader* na rozdíl od druhých dvou počítá i ukončení veršů pro rýmy.

```
1  public List<List<Prosody.Node>> MakeLists(List<int> syllables, List<
       int> extraSyllables)
2  {
3     List<List<Prosody.Node>> lists = new List<List<Prosody.Node>>();
4     if (syllables.Count == 1 && extraSyllables[0] == 0 && syllables
          [0] < 10)
5        lists.Add(premadeLists[syllables[0]]);
6     else
7        for (int i = 1; i <= syllables.Count(); i++)
8           for (int extra = 0; extra <= extraSyllables.Take(i).Sum();
               extra++)
9           {
10             List<Prosody.Node> list = MakeList(syllables.Take(i).Sum
                   () + extra);
11             List<List<Prosody.Node>> sublists = MakeLists(syllables.
                   Skip(i).ToList(), extraSyllables.Skip(i).ToList());
12             if (sublists.Count == 1)
13                list = list.Concat(sublists[0]).ToList();
14             else if (sublists.Count > 1)
15             {
16                Prosody.Node n = Prosody.MakePointer();
17                n.Alternatives = sublists;
18                list.Add(n);
19             }
20          lists.Add(list);
21          }
22    return lists;
23 }
```

Zdrojový kód 5: *SyllabotonicReader*: Vytvoření seznamů podle počtu slabik.

Ve zdrojovém kódu 5 je funkce na vytvoření seznamů typu *List⟨Prosody.Node⟩* odpovídající počtu slabik v podslovech. Vytvoří se seznam seznamů typu *Prosody.Node* (3). Pokud se výraz skládá jen z jednoho podslova a nemůže mít slabiky navíc, bude výsledkem jen jeden seznam, který se může vzít z předchystaných, má-li méně než 10 slabik (4-5). Jinak se vykonává cyklus pro 1 až počet podslov (7-21). Pro každý průchod je vnitřní průchod pro 0 až počet možných slabik navíc v dané části (8-21). Nejdříve se vytvoří jednoduchý seznam pro prvních i podslov s přidanými slabikami (10). Pak je rekurzivně vytvořen seznam seznamů pro zbytek výrazu (11). Pokud obsahuje jen jeden seznam, připojí se (12-13). Pokud obsahuje více seznamů, vloží se do nové *Prosody.Node* typu *Poin-*

ter a ta se připojí (14-19). Vytvořený seznam se vloží do seznamu seznamů (20). Po ukončení cyklu je seznam seznamů vrácen (22).

I když druhá větev je poměrně složitá a může dosahovat exponenciální časové složitosti, je využívána poměrně zřídka, a i tehdy pro jednodušší případy.

WeightBasedReader zjišťuje i délky slabik a funguje podobným způsobem, jak bylo popsáno v úvodu. Při samohlásce se případně přidá *Prosody.Node* z poziční délky předchozí samohlásky. Pokud je tato „ě“ a je to nastaveno, počítá se k poziční délce i tato. Pak se zkontroluje, zda tvoří dvojhlásku, pokud ano, přidá se longum. Pokud je však nastaveno *divideDiphtongs*, začne se počítat poziční délka (pak se započítá buď jako longum, nebo breve se slabikou podle poziční délky). Pokud dvojhlásku netvoří, zkontroluje se, zda je dlouhá nebo krátká. Za dlouhou se přidá longum, krátké se začne počítat poziční délka. Podobně se děje u slabikotvorných souhlásek. Ty se berou většinou jako krátké a počítá se u nich poziční délka, ale nastavením je možné je počítat i jako obojetné.

Poziční délku počítám tak, že za likvidu připočtu 2, za jinou souhlásku 3, za ě, pokud je to nastaveno, také 3. Spřežky, ve výchozím nastavení jen ch, se počítají jako jedna souhláska, tedy 3. Počet menší než 4 je krátká slabika, 4 nebo 5 obojetná, více než 5 dlouhá. Poziční délka se může počítat buď jen v rámci slova nebo ze začátku slova dalšího. Pokud se počítá jen v rámci slova, může dojít k vyhodnocení poziční délky při mezeře. Pokud se počítá i z dalšího slova, musí se nastavit příznak, že mezera se musí přidat při vypočtení poziční délky.

V aplikaci jsou celkem čtyři druhy prozódie. Se sylabotónickou pracuje *SyllabotonicReader*, s časoměrnou *WeightBasedReader*, se sylabickou *SyllabotonicReader* nebo *Reader*, pokud je bez rýmů, s volnou *SyllabotonicReader* (ta je jen s rýmy, bez rýmů by neměla žádnou pravidelnost a tedy by v této aplikaci neměla smysl).

2.4 Rhyme

Třída *Rhyme* slouží k zjištění, zda se konce dvou veršů rýmují, a zjištění kvality tohoto rýmu. Při vytvoření objektu této třídy se nastaví požadovaná kvalita. Tu je možné zadat třemi způsoby. Buď stačí asonance, tedy shoda poslední nebo dvou posledních samohlásek, bez ohledu na délku. Druhou možností je výskyt jedné až tří z vlastností - shodná délka posledních slov, shodnost souhlásek v rýmu, shoda délek posledních samohlásek. Třetí možností je výskyt konkrétních z těchto tří vlastností. Kromě toho se nastavuje, zda se uznávají překlenuté rýmy.

Výběrem konkrétních dvojic ke srovnání se zabývá třída *Comparator*, určením konců veršů ke srovnání se zabývá třída *SyllabotonicReader*, počet slabik posledního slova určuje třída *Prosody*. *Rhyme* dostává ke srovnání až dvojici ukončení veršů a počet slabik posledních slov.

Ve zdrojovém kódu 6 je funkce na porovnání dvou ukončení veršů a zjištění kvality rýmu. Nejprve se konce převedou podle jejich výslovnosti (3-4). Tento převod je důkladnější než ten, který prováděla třída *StringPreprocessor*. Zde se zvažuje i ta výslovnost, která mění jen jednotlivé souhlásky, zatímco *StringPre-*

```
1  public bool Compare(string end1, string end2, int length1, int
       length2)
2  {
3     ending1 = ToPhonetic(end1);
4     ending2 = ToPhonetic(end2);
5     rhymeInfo = new RhymeInfo();
6     rhymeInfo.Asonance = CompareAsonance();
7     rhymeInfo.Overlapping = CompareOverlapping();
8     rhymeInfo.VowelLength = CompareVowelLengths();
9     rhymeInfo.Phonemes = ComparePhonemes();
10    rhymeInfo.WordLength = length1 == length2;
11    if (rhymeInfo.Asonance)
12    {
13       if (desiredQuality == -2)
14          return true;
15       else if (CompareEndingConsonants() || (allowedOverlapping &&
             rhymeInfo.Overlapping))
16       {
17          if (desiredQuality == -1)
18             return CompareSpecific();
19          else
20             return CompareNumber();
21       }
22    }
23    return false;
24 }
```

Zdrojový kód 6: *Rhyme*: Porovnání na rýmy.

processor se zabýval jen tou, která může změnit počet nebo délku slabik. Poté následují jednotlivá srovnání (5-10). Ukládají se do objektu třídy *RhymeInfo*, která má pouze tyto vlastnosti a metodu na jejich přepis do textového tvaru. Pak se srovnají získané vlastnosti s požadovanou kvalitou (11-22).

```
1  public bool CompareAsonance()
2  {
3     return ShortenVowels(GetVowels(ending1)).Equals(ShortenVowels(
          GetVowels(ending2)));
4  }
```

Zdrojový kód 7: *Rhyme*: Porovnání asonance.

Porovnání na asonanci 7: z obou ukončení se vyberou samohlásky, zkrátí se a musejí se rovnat.

Porovnání na překlenutí 8 : z obou ukončení se vezme část po poslední samohlásce. K překlenutí dojde, pokud se tyto části nerovnají, ale jedna začíná druhou.

```
1 public bool CompareOverlapping()
2 {
3     string ec1 = ending1.Substring(ending1.LastIndexOfAny(vowels.
          ToCharArray()) + 1);
4     string ec2 = ending2.Substring(ending2.LastIndexOfAny(vowels.
          ToCharArray()) + 1);
5     return ec1 != ec2 && (ec1.StartsWith(ec2) || ec2.StartsWith(ec1))
          ;
6 }
```

Zdrojový kód 8: *Rhyme*: Porovnání na překlenutí.

```
1 public bool ComparePhonemes()
2 {
3     string es1 = ShortenVowels(RemoveSpaces(ending1));
4     string es2 = ShortenVowels(RemoveSpaces(ending2));
5     return (es1.Equals(es2) || (allowedOverlapping && rhymeInfo.
          Overlapping && (es1.StartsWith(es2) || es2.StartsWith(es1))))
          ;
6 }
```

Zdrojový kód 9: *Rhyme*: Porovnání na shodu hlásek.

Porovnání na shodu hlásek 9: Z obou ukončení se odstraní mezery a zkrátí samohlásky. Takto upravené části se musejí rovnat nebo pokud se jedná o překlenutí a to je uznávané, jedna začínat s druhou.

Ostatní porovnání jsou jednoduchá.

2.5 Comparator

Třída *Comparator* slouží k procházení básně po řádcích, používá *StringPreprocessor*, *Reader* a *Rhyme* pro získávání informací, srovnává je se vzory a ukládá do objektů třídy *VerseInfo*.

Pro jednodušší práci s ukončeními veršů používá strukturu *EndingData*, která má vlastnosti *int LineIndex* - číslo řádku, *string Ending* - ukončení verše, *int WordLength* - počet slabik posledního slova, *bool Rhymed* - zda se našel verš rýmující se s tímto.

Ve zdrojovém kódu 10 je hlavní funkce *Comparatoru* na průchod jednotlivých řádků básně. Nejdříve se nastaví hodnoty některých proměnných (3-6). Pokud mají být vyhodnocovány i rýmy, vytvoří se struktura pro *EndingData* (7). Pak se prochází celá báseň po řádcích (8-33). Pomocí *Readeru* je z aktuálního řádku získán seznam typu *Prosody.Node* (11-12). Pokud je prázdný, přejde se na další řádek (13-14), jinak se jedná o verš a proměnná počítající verše se zvýší (15). Seznam se převede na seznam seznamů typu *ProsodicElement* a uloží do *readAlternatives* (16). Pokud se jedná o volný verš (bez pravidelné prozódie), zjistíme

informace o ukončení verše podle prvního ze seznamů (není možné zjistit, který je správný, je-li jich více) (17-18). Jinak zjistíme, zda prozódie vyhovuje vzoru (21). Podle nastavení proměnné *passing* se pak do *currentVerseInfo* uloží příslušné informace. Je-li *passing true* pro vyhovující verše, *false* pro nevyhovující verše (22-23). Pokud má báseň mít rýmy, ale nebyl nalezen seznam vyhovující prozódii, určí se informace o ukončení verše podle prvního seznamu (24-25). Poté se jednou ze dvou metod vyhodnotí rýmy, pokud mají v básni být (27-30). Pokud aktuální *VerseInfo* obsahuje nějaké informace, přidá se (31-32). V případě básně s nepravidelnými rýmy se vyhodnotí posledních několik rýmů (34-35). Nakonec se vrátí *verseInfoList*, ze kterého byly odebrány neúplné položky (36-37).

Srovnání prozódie probíhá tím způsobem, že se vybere ze vzoru *ProsodicTree* odpovídající danému verši a s ním se srovnávají postupně seznamy typu *ProsodicElement*, které se získaly převedením ze seznamu *Prosody.Node*, vygenerovaného *Readerem*, dokud není nalezen odpovídající. Pro ten jsou případně zjištěny informace o ukončení verše.

Srovnaní jednoho seznamu se stromem vzoru ukazuje zdrojový kód 11. Prochází se část *List* ve stromu (4-24). Pokud je seznam na konci, ale ve stromu jsou ještě další prvky, srovnání končí negativně (6-7). Jinak probíhá srovnání podle aktuálního prvku v seznamu (8-23). Pokud je mezera a ve stromu na aktuálním místě mezera není, sníží se index stromu o 1, tím bude při příštím průchodu tato mezera přeskočena (10-13). Pokud je prvek breve nebo longum, musí být ve stromu ten stejný nebo anceps (14-18). Pokud je prvek anceps, může být ve stromu jakýkoliv kromě mezery (19-22). V případě, že má strom podstromy, srovnají se rekurzivně se zbytkem seznamu (25-31). Jinak se vrátí pravda, pokud seznam došel na konec nebo obsahuje už jen mezery (32).

Na srovnání rýmů existují dvě metody, podle toho, zda mají být rýmy pravidelně nebo ne.

Metodu pro rýmy nepravidelné ukazuje zdrojový kód 12. Atribut *rhymed = false* znázorňuje, že aktuální verš se zatím s žádným nerýmuje (3). Procházejí se data o ukončení několika posledních veršů a srovnají se s aktuálním, pokud se rýmují, nastaví se příslušné proměnné (4-9). Pro nejvzdálenější verš uložený v *endingData* se zjistí, zda se rýmoval (10) a podle toho se buď upraví *VerseInfo* odpovídající danému verši (12-14) nebo se vytvoří *VerseInfo* nové (15-16). Nakonec se informace o ukončení veršů posunou (18-19) a vloží se nové odpovídající aktuálnímu verši (20).

Část odpovídající řádkům 10-17 pro několik posledních veršů provede metoda *ProcessFinalRhymes*, která se v případě rýmů bez pravidelného vzoru zavolá ke konci metody *ProcessStrings*.

Rýmy s pravidelným vzorem se vyhodnocují podobně, s tím, že při každém zavolání metody je nanejvýš jedno srovnání rýmů a informace do *endingData* se ukládají podle toho, za jak dlouho se má objevit verš, který se má s tímto rýmovat. Obdoba metody *ProcessFinalRhymes* není v tomto případě potřeba.

```
1  public List<VerseInfo> ProcessStrings(string[] s)
2  {
3     verseInfoList = new List<VerseInfo>();
4     verse = -1;
5     lines = s;
6     convertedLines = preprocessor.ConvertStrings(lines);
7     MakeEndingData();
8     for (lineIdx = 0; lineIdx < lines.Length; lineIdx++)
9     {
10       currentVerseInfo = null;
11       line = convertedLines[lineIdx];
12       List<Prosody.Node> prosodicNodeList = reader.ReadString(line);
13       if (prosodicNodeList.Count == 0)
14          continue;
15       verse++;
16       readAlternatives = Prosody.ListAllAlternatives(
            prosodicNodeList);
17       if (FreeVerse())
18          FindEndingAndWordLength(readAlternatives[0]);
19       else
20       {
21          bool prosodyPassing = CompareProsody();
22          if (prosodyPassing == passing)
23             currentVerseInfo = new VerseInfo(lineIdx, lines[lineIdx
                  ], stanzaPattern[verse \% stanzaPattern.Count],
                  prosodicNodeList);
24          if (HasRhymes() && !prosodyPassing)
25             FindEndingAndWordLength(readAlternatives[0]);
26       }
27       if (RhymesNoPattern())
28          ProcessRhymesNoPattern();
29       else if (rhymePattern != null)
30          ProcessRhymes();
31       if (currentVerseInfo != null)
32          verseInfoList.Add(currentVerseInfo);
33    }
34    if (RhymesNoPattern())
35       ProcessFinalRhymes();
36    verseInfoList.RemoveAll(vi => !vi.IsComplete());
37    return verseInfoList;
38 }
```

Zdrojový kód 10: *Comparator*: Průchod básně.

```
1  public bool CompareListToTree(List<ProsodicElement> list,
       ProsodicTree tree)
2  {
3     int idxList = 0;
4     for (int idxTree = 0; idxTree < tree.List.Count; idxTree++,
          idxList++)
5     {
6        if (idxList == list.Count)
7           return false;
8        switch (list[idxList])
9        {
10          case ProsodicElement.Space:
11             if (tree.List[idxTree] != ProsodicElement.Space)
12                idxTree--;
13             break;
14          case ProsodicElement.Breve:
15          case ProsodicElement.Longum:
16             if (!(tree.List[idxTree] == list[idxList] || tree.List[
                   idxTree] == ProsodicElement.Anceps))
17                return false;
18             break;
19          case ProsodicElement.Anceps:
20             if (tree.List[idxTree] == ProsodicElement.Space)
21                return false;
22             break;
23       }
24    }
25    if (tree.Children != null)
26    {
27       foreach (ProsodicTree child in tree.Children)
28          if (CompareListToTree(list.Skip(idxList).ToList(), child))
29             return true;
30       return false;
31    }
32    return idxList == list.Count || list.Skip(idxList).All(e => e ==
          ProsodicElement.Space);
33 }
```

Zdrojový kód 11: *Comparator*: Srovnání prozódie.

```
1  private void ProcessRhymesNoPattern()
2  {
3     rhymed = false;
4     for (int i = 0; i < maxGap; i++)
5        if (endingData[i].LineIndex != -1 && rhyme.Compare(ending,
            endingData[i].Ending,finalWordLength, endingData[i].
            WordLength))
6        {
7           endingData[i].Rhymes = true;
8           rhymed = true;
9        }
10    if (endingData[0].Ending != null && endingData[0].Rhymes ==
         passing)
11    {
12       VerseInfo verseInfo = verseInfoList.Find(vi => vi.LineNumber
            == endingData[0].LineIndex);
13       if (verseInfo != null)
14          verseInfo.FoundRhyme = passing;
15       else
16          verseInfoList.Add(new VerseInfo(endingData[0].LineIndex,
               lines[endingData[0].LineIndex], passing));
17    }
18    for (int i = 0; i < maxGap - 1; i++)
19       endingData[i] = endingData[i + 1];
20    endingData[maxGap - 1] = new EndingData(this);
21 }
```

Zdrojový kód 12: *Comparator*: Zpracování nepravidelných rýmů.

2.6 MainWindow

MainWindow představuje hlavní okno aplikace. Jsou zde funkce na načtení a ukládání souborů a nastavení, na sestavení *Comparatoru* a ostatních nástrojů podle zvoleného nastavení. Po vyhodnocení básně se vedle textového pole objeví puntíky, které odpovídají (ne)vyhovujícím veršům. Zajímavá je metoda volaná při změně v textovém poli, která zajišťuje správné posunutí puntíků 13.

```
1  private void TbText_TextChanged(object sender, TextChangedEventArgs
       e)
2  {
3      if (verseInfo == null || !verseInfo.Any() || previousLineCount ==
           tbText.LineCount)
4          return;
5      int diff = tbText.LineCount - previousLineCount;
6      int caretIdx = tbText.CaretIndex;
7      int lineIdx = tbText.GetLineIndexFromCharacterIndex(caretIdx);
8      int startIdx = lineIdx - diff;
9      if (diff > 0 && tbText.GetLineText(startIdx).All(c => char.
           IsWhiteSpace(c)))
10         startIdx--;
11     else
12     {
13         string line = tbText.GetLineText(lineIdx);
14         int caretInlineIdx = caretIdx - tbText.
               GetCharacterIndexFromLineIndex(lineIdx);
15         if (verseInfo.Any(v => v.LineNumber == lineIdx) && line.Take(
               caretInlineIdx).Any(c => !char.IsWhiteSpace(c)))
16             lineIdx++;
17         else if (verseInfo.Any(v => v.LineNumber == startIdx) && line.
               Skip(caretInlineIdx).Any(c => !char.IsWhiteSpace(c)))
18             startIdx--;
19         verseInfo = verseInfo.Where(v => v.LineNumber < lineIdx || v.
               LineNumber > startIdx).ToList();
20     }
21     verseInfo.Where(v => v.LineNumber > startIdx).ToList().ForEach(v
           => v.LineNumber = v.LineNumber + diff);
22     if (diff < 0)
23         LinesRemovedFix();
24     MakeDots();
25     previousLineCount = tbText.LineCount;
26 }
```

Zdrojový kód 13: *MainWindow*: Změna lišty při změně počtu řádků textu.

Pokud objekt *verseInfo* nebyl vytvořen, neobsahuje žádné záznamy nebo se nezměnil počet řádků, funkce skončí (3-4). Do *diff* se uloží změna v počtu řádků (5), do *caretIdx* pozice kurzoru (6), do *lineIdx* pozice aktuálního řádku (7), do *startIdx* pozice řádku před změnou (8). Pokud řádky přibyly a původní řádek obsahuje jen bílé znaky, chápe se to, že byl odsazen celý řádek, a tedy se sníží

startIdx, aby se případně posunul i puntík odpovídající tomu řádku (9-10). Pokud naopak řádky ubyly (11-20), načte se do *line* text aktuálního řádku (13). Do *caretInlineIdx* pozice kurzoru v aktuálním řádku (14). Pokud má aktuální řádek odpovídající *VerseInfo* a před kurzorem obsahuje jiné než bílé znaky, zvýší se *lineIdx*, aby toto *VerseInfo* zůstalo zachováno (15-16). Pokud má naopak původní řádek odpovídající *VerseInfo* a řádek po kurzoru obsahuje jiné než bílé znaky, sníží se *startIdx*, aby jeho *VerseInfo* zůstalo zachováno (17-18). Z *verseInfo* se odstraní všechny záznamy ze smazané oblasti (19). Záznamy ve *verseInfo* za *startIdx* se posunou (21). Pokud řádky ubyly, zavolá se metoda *LinesRemovedFix*, která zobrazí nový počet záznamů a vyřeší situaci, pokud byl smazán aktuálně zvolený záznam (22-23). Metoda *MakeDots* vytvoří puntíky odpovídající novému rozmístění záznamů (24). Do proměnné zaznamenávající počet řádků se uloží jejich aktuální počet (25).

2.7 Nápověda

Součástí okna je i nápověda, která se aktivuje kliknutím na nápovědu v menu.

```
1  private void MenuHelp_Click(object sender, RoutedEventArgs e)
2  {
3     Mouse.OverrideCursor = Cursors.Help;
4     helpMode = true;
5  }
6
7  private void Grid_PreviewMouseDown(
8     object sender, MouseButtonEventArgs e)
9  {
10    if (helpMode)
11    {
12       helpMode = false;
13       Mouse.OverrideCursor = null;
14       Control source = (Control)e.Source;
15       Help.ShowHelp(source.Name);
16       e.Handled = true;
17    }
18 }
```

Zdrojový kód 14: *MainWindow*: Použití nápovědy

Funkce zajišťující její zapojení do uživatelského rozhraní jsou ve zdrojovém kódu 14 Při kliknutí na nápovědu v menu se změní podoba kurzoru a hodnota booleanu *helpMode* se nastaví na pravda.

Při kliknutí kamkoliv do okna s nastaveným *helpMode* se *helpMode* nastaví zpět na nepravdu, podoba kurzoru se vrátí na původní a je zavolána metoda *ShowHelp* třídy *Help* s názvem vybrané položky formuláře jako argumentem. Běžné vyhodnocení kliknutí neproběhne.

```
1  static Help()
2  {
3    dictionary = new Dictionary<string, string>();
4    Uri uri = new Uri("HelpFiles/help.txt", UriKind.Relative);
5    StreamResourceInfo sri = Application.GetResourceStream(uri);
6    StreamReader sr = new StreamReader(sri.Stream);
7    char[] separator = new char[] { ':' };
8    while (!sr.EndOfStream)
9    {
10     string[] parts = sr.ReadLine().Split(separator, 2);
11     foreach (string controlName in parts[1].Split(','))
12       dictionary.Add(controlName, parts[0]);
13   }
14 }
```

Zdrojový kód 15: *Help*: Vytvoření.

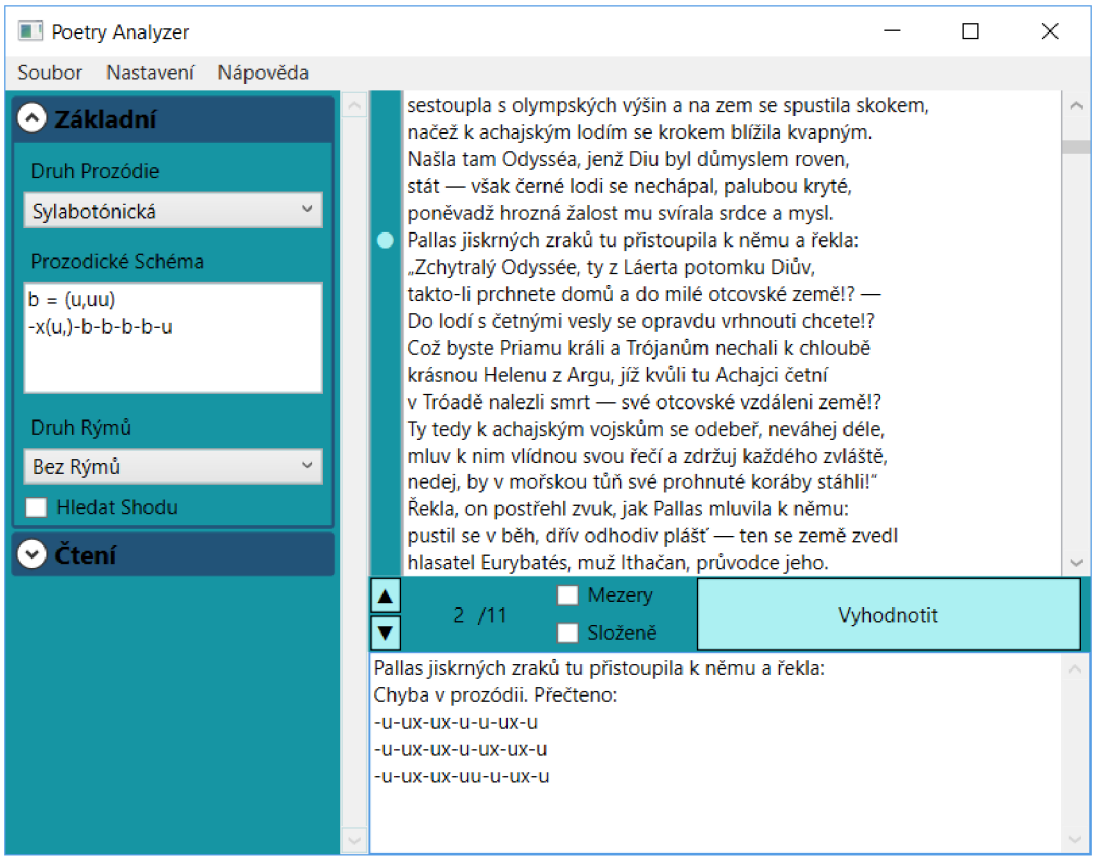
Vytvoření statické třídy *Help* při prvním použití nápovědy ukazuje zdrojový kód 15. Soubor *help.txt* je čten po řádcích a ukládán do objektu typu *Dictionary*. Řádky mají tvar *soubor:položka1,položka2,...,položkaN*. Soubor je název souboru s nápovědou bez přípony a položky jsou názvy položek formuláře, kterým odpovídá tento soubor.

Metoda *ShowHelp* funguje tak, že pro název položky formuláře v argumentu vyhledá odpovídající název souboru v *Dictionary* a pokud soubor existuje, zobrazí okno s jeho obsahem.

Pro soubory s nápovědou jsou použity *resources*, tedy jsou zkompilovány do spustitelného souboru, a při spuštění programu i s nápovědou stačí jen tento jeden soubor.

3 Aplikace

Grafické rozhraní aplikace můžeme rozdělit do pěti oblastí. Nahoře je menu, vlevo je nastavení a vpravo jsou pod sebou vstupní textové pole s lištou pro puntíky, ovládací panel a výstupní textové pole.



Obrázek 4: Vzhled aplikace

3.1 Ovládání

Do vstupního textového pole se zadává text básně k vyhodnocení. Tento je možné zadat ručně, zkopírovat nebo lze využít záložku soubor v menu, umožňující text načíst z textového souboru. Dále záložka soubor umožňuje uložení aktuálního obsahu vstupního textového pole nebo jeho vymazání.

Nastavení ovlivňuje okolnosti vyhodnocení. Pro větší přehlednost je rozděleno do sekcí, které lze kliknutím na hlavičku otevírat a zavírat. Samy mizí části nastavení, které při aktuálně vybraném postrádají smysl, například pokud je zvolen druh rýmů *bez rýmů*, ostatní nastavení k rýmům jsou skrytá. Nastavení je možno zadávat ručně nebo je možné využít záložku nastavení v menu, která umožňuje nastavení načítat ze souborů. Dále umožňuje nastavení ukládat nebo

vrátit do výchozího stavu. Co dělají jednotlivé položky v nastavení a jak je správně zadávat bude popsáno v samostatné podsekci.

Ovládací panel obsahuje tlačítko *vyhodnotit*, které báseň vyhodnotí podle zadaného nastavení, nebo uživatele upozorní, že v nastavení jsou chyby. V případě, že nastavení je v pořádku, báseň je vyhodnocena a uživateli je buď oznámeno, že nebyly nalezeny žádné výsledky, pokud hledal chybné verše a všechny byly správné nebo naopak, nebo se vedle vstupního textového pole zobrazí puntíky, které označují chybné nebo správné verše (podle nastavení). Verš je možné vybrat kliknutím levým tlačítkem na puntík nebo odstranit kliknutím pravým. O vybraném verši se ve výstupním textovém poli dole zobrazí informace. Mezi vybranými verši lze přepínat také šipkami. Vedle šipek je indikátor, který ukazuje, o kolikátý z uložených záznamů se jedná. Přepisem první části lze také přepínat výběr verše.

Pokud je zaškrtnuto políčko *mezery*, budou zobrazovány mezery (znak |) v prozodických výrazech z vyhodnocených veršů. Například
Při zaškrtnutí:

-ux|-u-u|x|-ux|-ux|-u

Bez zaškrtnutí:

-ux-u-ux-ux-ux-u

Pokud je zaškrtnuto políčko *složené*, bude se při více možnostech vyhodnocení prozódie výraz zobrazovat na jeden řádek, s tím, že v částech, kde je více možností se objeví závorka s těmito možnostmi. Například
Bez zaškrtnutí:

-ux-u-ux-ux-ux-u

-ux-u-ux-u-u-ux-u

Se zaškrtnutím:

-ux-u-ux(-ux,-u-u)-ux-u

3.2 Nastavení

V této podsekci budou popsány všechny části nastavení.

3.2.1 Základní

Druh prozódie je výběr prozódie v básni, podrobněji byly druhy prozódie popsány v teoretickém úvodu 1.1, zkráceně výběry odpovídají následujícímu:

Sylabotónická: záleží na počtu slabik a přízvučnosti.

Sylabická: záleží na počtu slabik.

Časoměrná: záleží na počtu a délce slabik.

Volná: bez pravidelnosti.

Prozodické schéma je vzor prozódie, se kterým má být báseň srovnávána. Při sylabotónické a časoměrné prozódii se vzor zadává následujícím způsobem:

- pro přízvučnou nebo dlouhou slabiku.

u pro nepřízvučnou nebo krátkou slabiku.

x pro libovolnou slabiku.

| pro mezeru.

Pokud je potřeba uvést více možných vzorů, použijí se závorky. Jeden výraz v závorce se chápe jako obsah závorky nebo nic.

-(u) tedy znamená -u nebo -.

Více výrazů v závorce, oddělených čárkou se chápe jako více možnosti na tom místě.

(uu,-) tedy znamená uu nebo -.

Je-li třeba uvést na nějakém místě více možností nebo nic, může se použít následující:

((uu,-)) a (uu,-,) oba znamenají uu, - nebo nic.

Závorky lze libovolně vnořovat.

(u-u,-(-,uu)) znamená u-u, – nebo -uu.

Pokud mají různým veršům odpovídat různé vzory, napíše se více řádků.

-u-u-u-u

-u-u-u

znamená, že liché verše mají vyhovovat vzoru -u-u-u-u, sudé vzoru -u-u-u.

Mohou se definovat symboly ve tvaru symbol = výraz.

a = (uu,-)

Každé „a“ na dalších řádcích bude nahrazeno výrazem (uu,-).

Symboly mohou být jakýkoliv jeden znak, co už není použitý jinde. Definovat se mohou na jakémkoliv řádku před prvním použitím.

V případě sylabické prozódie se vzor zadává:

číslem reprezentujícím počet slabik „10“

výčtem hodnot, oddělených čárkou „8,10“

rozsahem „8-10“

nebo kombinací obojího „4-6,8-10,12“

Opět je možné napsat více vzorů, kterým budou odpovídat různé řádky.

8

6

znamená, že liché řádky mají mít osm slabik, sudé šest.

Jako *druh rýmů* se vybere:

Podle schématu, pokud existuje přesné pravidlo, které verše se mají spolu rýmovat.

Bez schématu, pokud se verše mají rýmovat, ale není pravidlem určeno které se kterými.

Bez rýmů, pokud není vyžadováno rýmování.

Rýmovací schéma se zadává v případě, že je vybrán druh rýmů podle schématu. Jsou dva způsoby zadání:

Pomocí písmen. Každé písmeno odpovídá řádku. Stejná písmena znamenají, že se řádky rýmují. ABAB značí, že se rýmují první s třetím. a druhý se čtvrtým veršem a stejně i v rámci všech následujících čtveřic.

Pomocí čísel. Každé číslo značí, po kolika verších následuje verš, který se má s tímto rýmovat. 0 znamená, že žádný další verš se s aktuálním nemusí rýmovat, 1 znamená následující a tak dále. ABAB by v tomto způsobu značení bylo 2200. AABB by bylo 1010. Pomocí tohoto způsobu jde zapsat více možností, co prvním způsobem nejde, například 1 značí, že se rýmují úplně všechny verše. Pokud je mezi rýmy vzdálenost větší než 9, lze čísla oddělit čárkou, například 11,1,0,1,0,1,0,1,0,1,0,0 by odpovídalo abbccddeeffa.

Maximální mezera mezi rýmy se zadává v případě, že je vybrán druh rýmů bez schématu. Zadává se číslo 0 až 99, které značí, kolik jiných veršů může být nanejvýš mezi dvěma rýmujícími se verši, aby se rým uznal.

0 znamená, že se verš musí rýmovat s bezprostředně následujícím.

1 znamená, že mezi nimi může být jeden jiný verš a tak dále.

Pokud je zaškrtnuto políčko *hledat shodu*, vyznačí se po vyhodnocení verše vyhovující prozodickým a rýmovým vlastnostem, jinak se vyznačí nevyhovující.

3.2.2 Pokročilé rýmy

Pokud je zaškrtnuto políčko *povolit překlenutí*, budou za rýmy uznány (pokud splňují i ostatní vlastnosti) rýmy s překlenutím. Překlenutí je, pokud se verše rýmují, ale jeden z nich má na konci navíc souhlásku nebo skupinu souhlásek. Například pole a bolest. -ole se rýmuje, ale druhé slovo má navíc -st.

Kvalita rýmu může být:

Podle výběru, tedy přesně které vlastnosti musejí být splněny.

Podle počtu, tedy kolik vlastností musí být přinejmenším splněno (0-3).

Pouze asonance znamená, že stačí, aby se shodovaly samohlásky bez ohledu na délku.

Tři vlastnosti, které ovlivňují kvalitu rýmu jsou:

Shodná délka samohlásek, znamenající, že samohlásky tvořící rým jsou stejné a mají stejnou délku, například slova chvíli a pílí tvoří rým bez této vlastnosti, slova nese a třese tuto vlastnost mají.

Shodné hlásky, znamenající, že výslovnost všech hlásek, které tvoří rým, je shodná, přičemž se nebere ohled na délku samohlásek. Například slova stal a svár tuto vlastnost nemají, slova nese a třese tuto vlastnost mají.

Shodná délka slov, znamenající, že slova, která tvoří rým, tedy nejen jejich konce, mají stejný počet slabik. Slova chvíli a vyměnili tuto vlastnost nemají, slova nese a třese tuto vlastnost mají.

Do *délek* se zadávají dlouhé samohlásky a odpovídající krátké. Samohlásky jsou odděleny mezerou, dvojice čárkou.

á a,é e,í i,ó o,ú u

znamená, že „á“ po zkrácení odpovídá „a“, „é“ odpovídá „e“ a podobně. Zkracování probíhá až po převodu podle výslovnosti, proto je zbytečné zde uvádět například „ů“, „y“, „ý“.

Do *výslovnosti* v části *Pokročilé rýmy* se zadávají dvojice pro převod na skutečnou výslovnost. Pokud výslovnost ovlivňuje podstatnější věci, jako je počet slabik nebo délka, měly by být uvedeny v části *Čtení*. Zde se uvádí jen výslovnost, kterou je nutné posoudit při vyhodnocení rýmu. Pokud chce uživatel, aby byly podobné hlásky (například d-t, m-n, b-p,...) chápány jako stejné při porovnání na shodnost hlásek, může je zde uvést také.

Správný zápis je skupina znaků a jejich výslovnost, oddělená pomlčkou. Dvojice se oddělují čárkou.
bě-bje,dě-ďe
znamená, že „bě“ se má číst jako „bje“ a „dě“ jako „ďe“.

3.2.3 Čtení

Samohlásky, *slabikotvorné souhlásky*, *ostatní souhlásky*, *mezery* a *znaky uvozující komentáře* se píší bez oddělovače. Mezi mezerami musí být uveden znak mezery „ “. Znak nemůže být uveden ve více kategoriích zároveň.

Dvojhlásky se skládají ze dvou písmen, z nichž první je samohláska. Jednotlivé dvojhlásky se oddělují mezerou.

Pokud je zaškrtnuto políčko *rozdělovat dvojhlásky*, bude uvažováno, že dvojhláska může tvořit jednu nebo dvě slabiky. Například ve slově poučit „ou“ tvoří dvě slabiky, zatímco běžně jen jednu.

Příklonky jsou slova, která se vyslovují zároveň s předchozím, jako by mezi nimi nebyla mezera a může na nich tedy být přesunutý přízvuk.
Ve výraze „my bychom“ je přízvuk na první a třetí slabice, jako by to bylo jedno slovo. Kdybychom však „bychom“ nechápali jako příklonku, byl by na první a druhé.
Zadávají se oddělené mezerou. Musejí se skládat z písmen definovaných výše.

Jednoslabičné *předložky* přesouvají přízvuk následujícího slova. Slovní spojení „na zemi“ má přízvuk na první a třetí slabice, pokud bychom však „na“ nechápali jako předložku, byl by na první a druhé Víceslabičné předložky tuto vlastnost nemají, a tedy by se zde zadávat neměly.

Některé předložky mohou být zároveň i jiným slovním druhem. Například „se“ je ve výrazu „se sestrou“ předložkou, ve výrazu „viděli se“ zájmenem. Předložky, které se mohou vyskytnout i jako jiný slovní druh by měly být uvedeny mezi *oddělitelnými předložkami*. Předložky i oddělitelné předložky se zadávají

oddělené mezerou. Musejí se skládat z písmen definovaných výše.

Pokud je zaškrtnuto políčko *oddělovat předložky*, bude se u všech předložek počítat s možností, že se mohou vyslovovat odděleně.

Do *výslovnosti* v části *Čtení* se zadává ta výslovnost, která ovlivňuje počet slabik nebo jejich délku. Výslovnost, která toto nemění, nechť je případně uvedena v části *Pokročilé rýmy*.

Správný zápis je skupina znaků a jejich výslovnost, oddělená pomlčkou. Dvojice se oddělují čárkou.

x-ks,qu-kv

znamená, že „x“ se má číst jako „ks“ a „qu“ jako „kv“.

3.2.4 Časoměrná prozódie

Pokud je zaškrtnuto políčko *poziční délka přes mezeru*, budou se do poziční délky poslední slabiky slova počítat i souhlásky ze začátku dalšího slova. Ve verši „zkázyplném, na tisíce který zplodil útrap Achajským,“ [13] bude poslední slabika slova „tisíce“ chápána jako dlouhá, pokud je políčko zaškrtnuto, jinak bude považována za krátkou.

Pokud je zaškrtnuto políčko *slabikotvorné souhlásky obojetné*, bude slabika tvořená slabikotvornou souhláskou chápána vždy jako obojetná (může být dlouhá nebo krátká), jinak u ní bude počítána poziční délka, jako u krátké samohlásky. Ve verši „Srbů větve tiché, Obodritské říše potomci,“ [16] bude první slabika chápána jako obojetná pokud je políčko zaškrtnuté, jinak krátká.

Pokud je zaškrtnuto políčko *počítat ě do poziční délky*, bude ě v slabikách, kde se čte jako je nebo ně počítáno do poziční délky. Jinak bude bráno pouze jako e. Ve verši „jenžto mnohé, bujaré v oběť duše Hádovi střítil“ [13] bude první slabika slova „oběť“ dlouhá, pokud je políčko zaškrtnuto, jinak krátká.

Dlouhé samohlásky tvoří dlouhé slabiky. Zapisují se bez oddělení. Musejí být uvedeny v samohláskách v sekci *Čtení*.

Spřežky jsou hlásky, které jsou zapsány více znaky, jako ch v češtině, v cizích slovech třeba th. Zapisují se oddělené mezerou.

Likvidy jsou souhlásky, které se chovají jinak při dělení slova na slabiky, a tedy ovlivňují poziční délku.

Slovo „bratři“ se většinou rozdělí jako bra-tři, protože ř je likvida. Jindy se ale může dvojice souhlásek s likvidou rozdělit mezi dvě slabiky, jako čer-ná. Dvojice souhlásek, z nichž žádná není likvida se rozdělí mezi dvě slabiky, jako hos-té.

Zadávají se bez oddělení. Musejí být zadané v souhláskách v sekci *Čtení*.

Závěr

Cílem práce bylo vytvořit program pro práci s prozodickými vlastnostmi a rýmy v básních. Uživatel jej může použít ke srovnání textu se zadanými vlastnostmi a vyznačení odpovídajících veršů.

Tato funkčnost může být užitečná příležitostným básníkům, kteří nemusejí mít dobrý cit pro rytmus. Překladatelé mohou ocenit rychlou kontrolu formální správnosti u rozsáhlejšího díla. Program by mohl nalézt využití i při zkoumání básnických děl, konkrétně ve vyhledání veršů, splňujících některé vlastnosti.

V budoucnu by bylo možné dodělat funkci na vyhledání metra, jehož pravidla báseň nejblíže dodržuje, v souboru nebo databázi.

Další zajímavé rozšíření by bylo přizpůsobit program i jiným jazykům. Sice byl vytvářen pro češtinu, ale při správném nastavení funguje i se slovenštinou a částečně latinou.

Conclusions

The objective of this thesis was to create a program for working with prosodic properties and rhymes in poetry. The user can use it to compare a text with set properties and highlight corresponding verses.

This functionality could be beneficial to casual poets, who don't necessarily have a good sense for rhythm. Translators can appreciate the fast control of formal correctness in the case of a larger work. The program could also find usage when studying poetic works, as it can search for verses having certain properties.

In the future a function could be added to search for a meter, the rules of which are most closely followed by a given poem, in a file or a database.

Another interesting addition would be to expand the program to work with other languages. Although it was created with Czech poetry in mind, with appropriate settings, it can also work with Slovak and partially Latin.

A Obsah přiloženého CD/DVD

bin/
: Program PoetryAnalyzer, spustitelný přímo z CD/DVD.

doc/
: Text práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech příloh, a všechny soubory potřebné pro bezproblémové vygenerování PDF dokumentu textu (v ZIP archivu), tj. zdrojový text textu, vložené obrázky, apod.

src/
: Kompletní zdrojové texty programu PoetryAnalyzer se všemi potřebnými (příp. převzatými) zdrojovými texty, knihovnami a dalšími soubory potřebnými pro bezproblémové vytvoření spustitelných verzí programu / adresářové struktury pro zkopírování na webový server.

readme.txt
: Instrukce pro instalaci a spuštění programu PoetryAnalyzer, včetně všech požadavků pro jeho bezproblémový provoz.

data/
: Testovací data pro program PoetryAnalyzer. Jednotlivé podadresáře obsahují spolu související texty a nastavení. Texty mají příponu .txt. Nastavení mají příponu .cfg.

Literatura

[1] Michael McMillan. *Data Structures and Algorithms in C#.* First edition. Cambridge University Press, 2007.

[2] Marcin Jamro. *C# Data Structures and Algorithms.* Packt, 2018.

[3] Donald E. Knuth. *The art of computer programming I.* Third edition. Addison-Wesley Professional, 1997.

[4] Donald E. Knuth. *The art of computer programming III.* Second edition. Addison-Wesley Professional, 1998.

[5] Matthew MacDonald. *Pro WPF 4.5 in C# Windows Presentation Foundation in .NET 4.5, 4 edition.* APRESS, 2013.

[6] Alex Khang. *Professional WPF and C# Programming: Practical Software Development Using WPF and C#.* Independently published, 2019.

[7] Giuseppe Frappa. *Voci dal mondo antico.* http://www.poesialatina.it/.

[8] Timothy J. Moore. *The Meters of Roman Comedy.* Washington University in St. Louis. http://romancomedy.wulib.wustl.edu/.

[9] *How Many Syllables* https://www.howmanysyllables.com/.

[10] Pavel Šrubař. *Veršovací slovník* https://www.rymy.cz/.

[11] Petr Plecháč. *Úvod do teorie verše – cvičebnice.* https://versologie.cz/.

[12] Homér. *Ílias.* Přeložil Otmar Vaňorný. V MKP 1. vydání. Městská knihovna v Praze, 2018. https://search.mlp.cz/cz/titul/ilias/4403453/.

[13] Homér. *Ilias.* Přeložil Antonín Škoda. Praha: vlastním nákladem překladatele, 1886. https://cs.wikisource.org/wiki/Hom%C3%A9rova_Ilias_(%C5%A0koda).

[14] Karel Jaromír Erben. *Kytice.* V MKP 1. vydání. Městská knihovna v Praze, 2011. https://search.mlp.cz/cz/titul/kytice/3370042/.

[15] František Ladislav Čelakovský. *Ohlas písní českých.* V MKP 1. vydání. Městská knihovna v Praze, 2011. https://search.mlp.cz/cz/titul/ohlas-pisni-ceskych/3370052/.

[16] Ján Kollár. *Slávy dcera.* V MKP 1. vydání. Městská knihovna v Praze, 2011. https://search.mlp.cz/cz/titul/slavy-dcera/3370649/.