Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Automatický překladač jazyka Blason

Plzeň 2023 Bc. Oto Šťáva

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Oto ŠŤÁVA**
Osobní číslo:	**A20N0107P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Téma práce:	**Automatický překladač jazyka Blason**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se se strukturou jazyka Blason a jeho formálními pravidly ve zvoleném jazyce.
2. Seznamte se s problematikou formálních jazyků a s nástroji pro tvorbu vlastního překladače.
3. Seznamte se s problematikou generování vektorových obrázků a formáty pro jejich ukládání.
4. Navrhněte gramatiku pro jazyk Blason, s důrazem na rozpoznání jeho základních struktur. Oddělte od struktury jazyka detaily jako jsou konkrétní jména figur.
5. Implementujte překladač, který na základě navržené gramatiky a popisu v Blasonu vygeneruje odpovídající erbovní znamení.
6. Otestujte vytvořenou implementaci, identifikujte a charakterizujte její omezení.

Rozsah diplomové práce: **doporuč. 50 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce: **Ing. Richard Lipka, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání diplomové práce: **9. září 2022**
Termín odevzdání diplomové práce: **18. května 2023**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 11. října 2022

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 21. června 2023

Bc. Oto Šťáva

Abstract

In heraldry, a blazon is a formal textual description of a coat of arms or another heraldic emblem. The description must be concise and at the same time unambiguous enough so that an author aware of its rules is able to reproduce the described emblem as accurately as possible. Thanks to these requirements, it is possible to recognize patterns and structures in the texts, which may potentially be processed using a machine parser. The aim of this diploma thesis was to analyse these structures and, based on such analysis, to implement a compiler able to process at least a subset of real-world blazons written in the Czech language.

Abstrakt

Blazon, též zvaný blason, je formální slovní popis heraldického znamení. Jedná se o popis dostatečně stručný a zároveň výstižný, aby byl autor kreseb, který je znalý jeho pravidel, schopný znamení věrně reprodukovat. Díky požadavkům na jednoznačnost a stručnost je možné v textech vysledovat pravidelné vzory a struktury, které mohou být potenciálně zpracovány strojovým analyzérem. Cílem této diplomové práce bylo tyto struktury detailně prozkoumat a na jejich základě implementovat překladač, který dokáže zpracovávat alespoň část skutečných blazonů v českém jazyce.

Poděkování

Timto bych rád poděkoval panu Ing. Richardu Lipkovi, PhD., vedoucímu této diplomové práce, za odborné rady, náměty, připomínky a celkovou podporu, které mi poskytoval během jejího vypracování.

Obsah

Úvod

Heraldika je důležitou součástí historických věd, která popisuje erby, znaky a štíty. Ty jsou dodnes nedílnou součástí identit měst a tradičních uskupení – plní prakticky podobnou funkci jako loga firem v moderní době. Štíty, stejně jako loga, by měly být na první pohled rozpoznatelné, aby byl jejich pozorovatel schopen s jistotou poznat, kterému městu, šlechtickému rodu či jinému uskupení nebo jednotlivci patří.

Moderní způsoby reprodukce obrazových informací – zejména pak ty digitální – umožňují přesné vyobrazení loga v prakticky všech situacích. Tak tomu však historicky nebylo vždy. Reprodukce heraldických znamení je zajišťována pomocí *blazonů*, též zvaných *blasonů*. Jedná se o slovní popis znamení, který by jej měl popisovat jasně a zároveň stručně tak, aby byl libovolný autor, který je znalý jeho pravidel, schopen znamení v rozpoznatelné podobě reprodukovat.

Díky požadavku na stručnost a jednoznačnost blazonů lze v jejich struktuře rozpoznat až strojovou pravidelnost. V jednotlivých textech lze vzájemně nalézt totožné vzory, které téměř až nabádají k algoritmickému zpracování; o to více pak v jazycích neobsahujících skloňování a časování slov.

Cílem této diplomové práce je položit základ pro digitální zpracování blazonů v českém jazyce. V jejím rámci je v programovacím jazyce C implementována knihovna funkcí a sada programů CBlazon, která je schopna přečíst text blazonu a vygenerovat z něj strukturovaný datový popis heralidckého znamení, určený k dalšímu zpracování – především pak k automatizovanému vygenerování jeho grafické reprezentace.

I v době rozmachu nejrůznějších generativních systémů postavených na strojovém učení – specificky pak s tématem související generátory obrazových děl na základě textových vstupů – může být vhodné prozkoumat, zda některé úlohy přeci jen nevolají spíše po algoritmickém řešení. Ta totiž budou vždy mít mnoho nesporných výhod – patří mezi ně zejména přesnost, opakovatelnost a výkonnost. Ani velmi pokročilá neuronová síť pravděpodobně nebude nikdy efektivnějším a zároveň přesným řešením logicky jasně vyhraněného problému než vhodně zvolený deterministický algoritmus určený přímo k jeho řešení. Práce se i proto mimo jiné zabývá právě tím, jak sestavit algoritmicky zpracovatelnou formální gramatiku pro popis blazonů – tedy snaží se co nejjasněji logicky vyhranit problém jejich zpracování.

Text této diplomové práce je členěn do šesti kapitol.

První kapitola stručně uvádí do problematiky blazonů. Čerpá především z *Klíče ke znakům měst České republiky* autora Petra Houzara [9], jež byla při vypracování hlavním zdrojem informací o této problematice z pohledu heraldiky.

Druhá kapitola se zabývá formální gramatikou, jež byla experimentálně sestavena na základě mnoha příkladů blazonů. Gramatika v žádném případě nepopisuje kompletní jazyk blazonů – ten je nesmírně rozsáhlý a vyžaduje další výzkum, který přesahuje rozsah diplomové práce. Nicméně dle autorova nejlepšího vědomí pokládá pevný základ, na němž je možné v budoucnosti dále stavět.

Ve třetí kapitole prozkoumáme nástroje pro lemmatizaci, tj. převod slov na jejich základní tvary. Ta je pro strojové zpracování blazonů nezbytná, neboť český jazyk a tudíž český blazon obsahuje flexi slov, která by bez lemmatizace představovala obrovskou překážku.

V kapitole čtvrté je popsána problematika figur a vektorové grafiky. Nachází se zde i návrh principu fungování databáze, z níž by program generující grafickou podobu štítů čerpal soubory obsahující obrázky figur. Implementace takové databáze může být vhodným tématem pro navazující práce.

Pátá kapitola popisuje samotnou implementaci sady CBlazon. Společně s programátorskou dokumentací obsaženou přímo ve zdrojových souborech sady je tato kapitola důležitým vodítkem pro čtenáře, který by měl v úmyslu práci dále rozšiřovat.

Šestá a poslední kapitola popisuje, jak byla sada CBlazon testována a poskytuje výsledky tohoto testování. Popisuje také použitá testovací data.

1 Blazon

Blazony jsou texty slovně popisující heraldické znaky. Jejich studium je součástí heraldiky, pomocné vědy historické zabývající se právě znaky měst, států, rodů, rytířských řádů a dalších historických uskupení. O heraldice obecně se lze více dočíst v Houzarově *Klíči ke znakům České republiky* [9], ze kterého tato práce hojně čerpá informace právě o blazonech samotných včetně jejich plných znění, ale i v široké škále další literatury, například *Klíči k našim městům* autorů Lišky a Muchy [13], *Encyklopedii heraldiky* autora Bubna [5] nebo v anglické knize *Heraldry, historical and popular* autora Boutella [4]. Zejména poslední zmiňovaná kniha je velmi obsáhlá a popisuje množství různých figur a detailů, které se v heraldických znacích mohou objevovat.

Text blazonu popisuje dělení znaku, obsažené figury a použité barvy či kovy (společně odborně zvané tinktury). Pokud má znak štítonoše nebo se u horního okraje znaku nachází přilba, může být popis těchto rovněž zahrnut v blazonu.

Znak je vždy popisovaný z pohledu jeho držitele, tedy ve výsledné kresbě jsou strany vyobrazeny obráceně, než jak je text popisuje – pokud text obsahuje popis, že se objekt nachází *vpravo*, v kresbě se bude nacházet po levé straně. Tímto se budeme řídit v celém textu této práce i v kódu programu – tj. strany budou popisovány dle heraldických pravidel. ([9], kapitola 6)

Štít může být členěný na více částí. Není-li členěný, jeho blazon nejprve uvádí tinkturu štítu, poté popisy figur, přilbu s klenotem apod. Pokud štít členěný je, uvádí se jako první druh heroldské figury. Jednotlivá pole štítu se popisují zprava doleva a shora dolů. Polcené, dělené či čtvrcené štíty lze na začátku blazonu zkráceně označit barvami, např. *„červeno-zlatě polcený štít“*. ([9], kapitola 6)

Součástí štítu může být také malý srdeční, střední, čestný nebo pupeční štítek, případně další menší štítky. Střední štítek má při popisu přednost před hlavním štítem. Přilby na štítu se mohou popisovat od pravé k levé, nebo, pokud jich je lichý počet, může být popsána nejprve prostřední a ostatní zprava doleva. I v případě lichého počtu však lze popisovat přilby zprava doleva. ([9], kapitoly 6 a 10)

Štít může obsahovat patu či hlavu. Patou štítu může být například i *trávník*, *trojvrší* apod. Z paty může vycházet některá figura, například strom, přičemž skutečnost, že figura z paty vychází, není nutno explicitně specifiko-

vat. Kupříkladu, pokud text zní *„v modrém štítě s trávníkem vlevo vyrůstající lípa“*, vyplývá z něj, že lípa vyrůstá z trávníku. ([9], kapitola 6)

Houzar [9] vysvětluje, že blazon musí být dostatečně věcný a stručný. Libovolný blazonu znalý autor si na jeho základě musí být schopen popisovaný znak představit a nakreslit bez markantních rozdílů oproti originálu. Přípustné jsou rozdíly dané autorovým stylem, znak však musí vždy být na první pohled rozpoznatelný.

Při popisování štítu je zpravidla dbáno na to, aby barvy byly kladeny na kovy či kovy na barvy. Nikoliv tedy barvy na barvy a kovy na kovy. Jednou z výjimek je znak města Plzně, kde v pravé horní části štítu nalezneme zlaté klíče na stříbrném poli. [13]

Pro představu zde uvádíme blazon znaku města Poděbrad. Znak je vyobrazený na obrázku 1.1 a jeho blazon může znít například následovně:

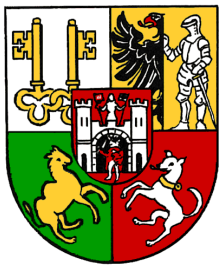
> *Ve zlatém štítě zvýšená kvádrovaná hradba s červeně krytým cimbuřím a prolomenou bránou se zlatou vytaženou mříží se stříbrnými hroty a zlatými vraty se stříbrným kováním. Mezi bránou a cimbuřím černo-stříbrně dělený štítek se dvěma stříbrnými břevny v prvním poli. Na hradbě černo-červeně polcená orlice se zlatou zbrojí a stříbrným do špičky zakončeným perizoniem přeložená z hradby vynikající nekvádrovanou věží s obdélníkovým oknem v každém patře odděleném římsou, cimbuřím a červenou kuželovou střechou se zlatou makovicí; stavby stříbrné a okna černá.* ([9], kapitola 10)

Obrázek 1.1: Znak města Poděbrad[1]

[1] zdroj (veřejná doména): https://commons.wikimedia.org/wiki/File:Znak_m%C4%9Bsta_Pod%C4%9Bbrady.gif

Druhým příkladem je blazon znaku města Plzně pro představu, jak vypadá popis štítu čtvrceného se středním štítkem. Jeho podoba bez štítonoše a okolních dekorací je vyobrazena na obrázku 1.2. Blazon štítu zní:

> *Ve čtvrceném štítě červený střední štítek, v něm vynikají z hnědé skály dvě věže, každá s nízkým přízemím s bránou a cimbuřím, ve vysokém patře vysoké 'okno s vimperkem, podsebitím a' jehlanovou střechou se stříbrnou makovicí a červeným patriarším křížem. Stavby stříbrné nekvádrované, brány a okna černá gotická. Mezi věžemi zvýšená hradba s bránou, v ní stojící doleva natočený ozbrojenec jako český král - ve stříbrné plátové zbroji s červeným pláštěm, stříbrným mečem v pravé ruce, červeným štítem s českým lvem na levé a černým položeným orličím křídlem na přilbě; po stranách brány dvě slepá trojlistá okna. Na hradbě vyskakující stříbrná žena držící dvě přivrácené korouhve - pravá červená s českým lvem, levá modrá s moravskou orlicí.*
>
> *V prvním stříbrném poli hlavního štítu dva zlaté vztyčené dole svázané odvrácené papežské klíče vedle sebe; ve druhém zlatém poli pravá půle císařského orla před doprava natočeným ozbrojencem ve stříbrné plátové zbroji s kolčí přilbou, držícím v pravé ruce orla a v levé stříbrný meč se zlatou záštitou a hlavicí; ve třetím zeleném poli zlatý obrácený vykračující dvouhrbý velbloud; ve čtvrtém červeném poli stříbrná chrtice se zlatým obojkem.* ([9], kapitola 10)



Obrázek 1.2: Znak města Plzně[2]

[2]zdroj (veřejná doména): https://commons.wikimedia.org/wiki/File:Plzen_small_CoA.png

2 Formální gramatika

Následující kapitolu zahájíme zavedením několika základních pojmů, se kterými budeme pracovat. Tato úvodní část kapitoly čerpá z článku *On certain properties of grammars* Noama Chomského [6].

Prvním pojmem je *abeceda* – množina znaků či symbolů, určených k utváření řetezců, tj. uspořádaných n-tic představujících nějaký text. Výukové materiály jako abecedy často využívají různé množiny písmen a/nebo číslic, nicméně je vhodné se při vývoji analyzátoru nezaměřit pouze na znaky ze vstupního textu. Pro překladače textových vstupů je totiž často užitečné analýzu textu rozdělit na dvě fáze – na analýzu lexikální a analýzu syntaktickou.

Při *lexikální analýze* se vstupní text převádí na seznam či proud *tokenů*. Těmi mohou být například slova, interpunkční znaménka či technické symboly (např. `EOF` symbol značící konec vstupu). Implementačně je takový token zpravidla řešen datovou strukturou obsahující typový příznak a další informace, které jsou použitelné pro vygenerování užitné informace ze vstupního řetězce – může se jednat například o ukazatel na paměť obsahující textový obsah tokenu (např. u řetězcového literálu), jeho číselnou hodnotu (např. u číselného literálu) apod. Typový příznak rozlišuje, zda token představuje právě slovo, interpunkční znaménko apod. a je používán k rozpoznávání během *syntaktické analýzy*.

Při *syntaktické analýze* dochází k rozpoznávání, zda tokeny vygenerované lexikálním analyzátorem tvoří řetězec náležící zpracovávanému *jazyku* a zpravidla také k sestavení datové struktury, která zjednodušuje další práci s informací, kterou vstupní text nese – této datové struktuře se říká *abstraktní syntaktický strom*, či zkráceně *AST*.

Jazyk definujeme jako množinu řetězců nad abecedou. Jedná se o podmnožinu množiny *všech* řetězců, které lze z abecedy vyprodukovat[1]. Důležitou vlastností řetězce tedy je, zda *patří* či *nepatří* do daného jazyka. Jelikož je jazyk ze své definice jednoduše množinou, je potřeba nějakým způsobem popsat, jak tato množina vypadá. Lze to provést například *výčtem*, *regulárním výrazem* nebo *formální gramatikou*. Každý z těchto způsobů může mít svá využití.

Výčet je velmi triviální způsob popisu jazyka a v praxi se s takovým jazykem setká snad každý programátor, aniž by věděl, že vlastně teoreticky

[1] Takové množině *všech* řetězců tvořených znaky abecedy se říká *iterace*

s nějakým jazykem pracuje. Může se jednat například o seznam argumentů, které přijímá nějaký program; povolené řetězcové hodnoty pro klíč v konfiguraci apod.

Regulární výraz je dalším generátorem jazyka. Je často používán k vyhledávání vzorů v textu či ke kontrole uživatelských textových vstupů. Příkladem použití může být kontrola, zda uživatel zadává číslo, jednoslovné jméno, e-mailovou adresu apod.

V případě zpracování blazonů volíme způsob generování gramatikou. *Formální gramatika* je soubor přepisovacích pravidel generujících jazyk. S pomocí vhodně zvolené gramatiky můžeme vytvořit program, který určuje, zda daný vstupní řetězec patří do daného jazyka, či nikoliv, a uvnitř její softwarové implementace z něj pak získat informace v podobě AST, které daný vstupní text nese. V našem případě získáváme digitální popis štítu, který lze dále využít například k vygenerování jeho grafické podoby.

Formální definice gramatiky je následující:

$$G = (N, \Sigma, P, S) \tag{2.1}$$

N je abeceda neterminálních symbolů. Σ je abeceda tokenů, ve smyslu gramatik rovněž zvaných terminálních symbolů. $S \in N$ je symbol označený jako počáteční – tj. analýza se vždy začíná s řetězcem délky jedna obsahující právě jeden symbol S. P je konečná množina produkčních pravidel majících následující podobu:

$$\begin{aligned} &\alpha \to \beta \\ &\alpha \in N \\ &\beta \in (N \cup \Sigma)^*, \end{aligned} \tag{2.2}$$

kde operátor * značí iteraci množiny – tj. množinu všech řetězců nad danou množinou.

Slovy řečeno: α je neterminální symbol, který se přepíše na řetězec terminálních a neterminálních symbolů β. Všechny neterminální symboly v řetězci β pak lze dále přepisovat jakožto symboly α, čímž dochází k postupnému nahrazení všech neterminálních symbolů řetězci symbolů terminálních. Řetězec β patří do jazyka generovaného gramatikou G, vzniknul-li postupnou aplikací přepisovacích pravidel na neterminální symboly od řetězce obsahující pouze symbol S a neobsahuje-li žádné neterminální symboly.

Při aplikaci jednotlivých přepisovacích pravidel vzniká tzv. *derivační strom*. Ten pro daný vstupní text reprezentuje, které jeho části odpovídají kterým částem gramatiky. Uzly tohoto stromu jsou symboly z $N \cup \Sigma$, jeho

kořenem je počáteční symbol S gramatiky G. Potomci každého neterminálního uzlu jsou ty symboly, na které se symbol z rodičovského uzlu přepsal při zpracovávání vstupního textu. Terminální uzly jsou listy. Příklady vizualizovaných derivačních stromů lze nalézt v přílohách.

2.1 Formát pro popis gramatiky

Pro zpracování strukturovaných blazonů byla navržena gramatika popsaná vlastní textovou obdobou *Backus-Naurovy formy* (zkr. *BNF*). Tato obdoba byla vytvořena pro konkrétní potřeby této diplomové práce a vznikla vlastní iniciativou rozšířením původní BNF ve chvíli, kdy se ukázalo, že není zcela odpovídajícím vodítkem pro ruční přepis na analyzátor rekurzivním sestupem (vizte sekci 5.4) v jazyce C, neboť některé konstrukty je zjevně mnohem vhodnější vyjádřit například pomocí smyčky, spíše než rekurzí – ušetří se tak paměť na zásobníku i výpočetní režie spojená s četnými voláními funkcí.

Pozdější průzkum pramenů ukázal, že obdobné výhody by poskytla *rozvinutá Backus-Naurova forma* (zkr. *EBNF* z angl. *extended Backus-Naur form*) [10], avšak v té době již byla gramatika popsána vlastním formátem a další přepis by pravděbodobně nepřinesl mnoho výhod. Existuje i několik nestandardních variant EBNF [11] – některé jsou formátu použitému v této práci velmi blízké, ale žádná mu zcela neodpovídá.

Ve vlastním formátu popisu gramatiky jsou jednotlivé symboly popisovány následovně:

- Názvy neterminálních symbolů jsou opatřeny špičatými závorkami (např. `<Shield>`, `<Blazon>`, `<Object>`);
- terminální symboly v podobě konkrétních slov, resp. jejich lemmat (vizte kapitolu 3), jsou opatřeny dvojitými uvozovkami (např. `"vlevo"`, `"štít"`, `"barva"`);
- terminální symboly reprezentované druhem slova jsou značeny názvem druhu slova v anglickém jazyce ohraničeném hranatými závorkami (např. `[noun]`, `[numeral]`, `[adjective]`).

Pravidlo pro přepis je popsáno právě jedním neterminálním symbolem na levé straně a jedním či více řetězci symbolů na straně pravé. Jednotlivé možné pravostranné řetězce jsou od sebe odděleny svislou čarou (`|`) a jejich seznam je zakončený středníkem (`;`). Formát na pravých stranách pravidel nepřipouští prázdné řetězce (jinde typicky označované symbolem ϵ). Na levé

straně pravidel se každý neterminální symbol smí objevit nejvýše jednou. Na pravých stranách je povoleno se odkazovat na symboly, které jsou v textu gramatiky zavedeny níže.

Formát zavádí jednoznakové kvantifikační operátory, které se mohou vyskytnout za symboly či skupinami symbolů. Slouží k vyjádření možného opakování či nepovinnosti výskytu symbolů nebo jejich skupin. Skupiny symbolů jsou ohraničeny kulatými závorkami – jedná se o řetězce symbolů, na které lze kvantifikační operátory aplikovat jako na celek (kulaté závorky bez operátoru jsou povoleny, ale nemají žádný funkční význam, mohou však být potenciálně použity k uskupování symbolů za účelem zpřehlednění zápisu). Mezi kvantifikační operátory popisovaného formátu patří:

- `?` – žádný či jeden výskyt symbolu či skupiny,
- `+` – jeden či více výskytů symbolu či skupiny,
- `*` – žádný nebo více výskytů symbolu či skupiny.

Symboly bez operátoru jsou implicitně povinné a jednočetné – musejí se tedy na daném místě vyskytnout právě jednou.

Formát umožňuje označení počátečního neterminálního symbolu gramatiky tak, že levá strana jeho pravidel pro přepis počátečního neterminálního symbolu je označena klíčovým slovem `start`.

Formát je navržen tak, aby bylo možné jej zpracovávat strojově, např. pro potřeby budoucího vývoje nástrojů, které by byly schopny v ní provádět analýzu chyb.

2.2 Gramatika blazonu

V této sekci je popsána gramatika, jež byla použita pro implementaci syntaktického analyzátoru. Nutno podotknout, že gramatika v žádném případě negeneruje kompletní jazyk všech možných blazonů – takovou gramatiku pravděpodobně není v lidských silách sestavit i proto, že se jedná o podmnožinu přirozeného jazyka, byť značně omezenou. Aktuální verze gramatiky podchycuje pouze velmi základní varianty blazonů. Umožňuje popisy nedělených, polcených, čtvrcených i komplexněji dělených štítů a v nich umístěných figur. V současnosti gramatika nerozpoznává popisy klenotů ani štítonošů. Nicméně dle autorova mínění představuje dobrý základ pro další rozšiřování.

Analyzátor popsaný v kapitole 5 je přímým přepisem této gramatiky do jazyka C a zcela jí tedy odpovídá. Gramatika se rovněž nachází v Git

úložišti (vizte sekci 5.12) této práce v souboru `grammar.bnf` a je udržován, aby byl aktuální vůči zdrojovému kódu programu umístěném tamtéž – dojde-li později k dalšímu rozšiřování programu, bude pravděpodobně vhodnější studovat gramatiku tam. Pro úplnost následuje kompletní znění gramatiky i zde v textu práce.

```
<Blazon> start       ::= <Shield> <Helmet>? <Bearer>? [sentence_end]? ;

<Shield>             ::= "v" <SimpleOrComplex>
                         | "štít" <ComplexSplits> <SectionSep> <ComplexesIn> ;
<SimpleOrComplex>    ::= <Colour> "štít" <FieldAttrsOpt>? <Objects>?
                         <ObjectAddAttrs>?
                         | <Complexes> ;

# Základ štítu - dělení, barvy atd.
# Pozn.: Gramatika umožňuje vícekrát specifikovat štít, což znemožníme
# sémantickou analýzou
<Complexes>          ::= <ComplexFirst> (<SectionSep>? <ComplexesIn>)? ;
<ComplexesIn>        ::= (<SectionSep>? "v" <Complex>)+ ;
<ComplexFirst>       ::= [numeral] <Colour>? <FieldDesignation>
                         <FieldAttrsOpt>? <ComplexShield> <Objects>?
                         <ObjectAddAttrs>? [sentence_end]?;
<Complex>            ::= [numeral] <Colour>? <FieldDesignation>
                         <FieldAttrsOpt>? <Objects>?
                         <ObjectAddAttrs>? [sentence_end]?;
<ComplexShield>      ::= <ComplexSplits>? "štít" ;

<FieldDesignation>   ::= "pole" | "polovina" | "čtvrtina" ;

<ComplexSplits>      ::= <ComplexSplit> (<And>? <ComplexSplits>)* ;
<ComplexSplit>       ::= <SplitStyle>? <SplitWhere>? <Colours>? <SplitType> ;

<SplitStyle>         ::= [adjective] "řez" ;
<SplitWhere>         ::= "vlevo" | "vpravo" | "nahoře" | "dole" ;
<SplitType>          ::= "dělený" | "půlený" | "polcený" | "čtvrcený" ;

# Atributy pole/štítu - hlavy, paty, objekty v pozadí...
<FieldAttrsOpt>      ::= "s" <FieldAttrsBeg> | "na" <FieldAttrsBeg> ;
<FieldAttrsBeg>      ::= <FieldAttr> (<And> <FieldAttrsOptWith>)? ;
<FieldAttrsOptWith>  ::= <FieldAttrOptWith> (<And> <FieldAttrsOptWith>)* ;

<FieldAttrOptWith>   ::= "s" <FieldAttr> | "na" <FieldAttr> | <FieldAttr> ;

<FieldAttr>          ::= <FieldObject> ;

# Objekty v pozadí štítu - brané jako vlastnost štítu (např. 'štít s trávníkem')
<FieldObject>        ::= <ObjectAdjs>? [noun] <NaturalColours>? <ObjectPostAttrs>? ;

# Základní definice objektů ve štítu
<Objects>            ::= <Object> (<ObjectConnector>? <Object>)* ;
<Object>             ::= [numeral]? <ObjectAdjs>? <ObjectName>
                         <NaturalColours>? <ObjectPostAttrs>? ;
<ObjectConnector>    ::= "přeložený" | "provázený" | <And> ;
```

```
# Objekt může mít jméno z více slov, např. "Panna Marie"
<ObjectName>         ::= [noun]+ ;

<ObjectAdjs>         ::= <ObjectAdj>+ ;
<ObjectAdj>          ::= [adverb] [adjective] | [adjective] ;

<ObjectPostAttrs>    ::= <WithOrIn> <ObjectAttrsBeg> ;
<ObjectAttrsBeg>     ::= <ObjectAttr> (<And>? <ObjectAttrsOptWith>)? ;
<ObjectAttrsOptWith> ::= <ObjectAttrOptWith> (<And>? <ObjectAttrOptWith>)* ;

<ObjectAttrOptWith>  ::= <WithOrIn> <ObjectAttr> | <ObjectAttr> ;
<ObjectAttr>         ::= [numeral]? <ObjectAdjs>? <ObjectName> | <NaturalColours> ;


# Utilities ##############################################################

<WithOrIn>           ::= "s" | "v" | "bez" ;

<Colours>            ::= <Colour> ("-" <Colours>)* ;
<Colour>             ::= [noun] | [adjective] ;
<NaturalColours>     ::= "přirozený" "barva" ;

<And>                ::= "a" | "," ;
<SectionSep>         ::= "." | ";" ;
```

3 Lemmatizace a tagging

Jazykem blazonu je v českém prostředí nepříliš překvapivě čeština. Skutečnost, že se jedná o přirozený jazyk (byť pro potřeby co možná nejstručnějšího popisu značně okleštěný a strukturalizovaný), skýtá mnohá úskalí pro možnosti strojového zpracování.

Předpokladem pro úspěšné strojové zpracování je získání co možná nejvíce informací o jednotlivých slovech v textu. Mezi ty patří například určení druhu slova, jeho mluvnických kategorií a o jaké konkrétní slovo se vlastně jedná. Tato data jsou nezbytná pro implementaci syntaktického analyzéru (též zvaného parseru; vizte sekci 5.4). Ten pak vygeneruje počítačový model štítu, ze kterého je možné například vykreslit jeho grafickou podobu.

Byla vyvinuta řada nástrojů, tzv. *taggerů*, které jsou schopny strojově převádět slova na jejich základní tvary. Často rovněž poskytují dodatečná data, která lze využít k dalšímu zpracování vstupního textu – mezi ně může patřit informace o slovním druhu; o tvaru slova, tj. jak je dané slovo skloněno/časováno; zda se jedná o vlasní jméno, jméno osoby, název barvy aj.; pokud se jedná o číslovku, může vyjádřit její numerickou hodnotu apod.

V této sekci si popíšeme několik dostupných lemmatizérů, zhodnotíme je a vybereme jeden, který použijeme jako knihovnu v programu, který je výstupem této diplomové práce. Důležitými kritérii jsou rychlost zpracování, přítomnost programátorské dokumentace a možnost integrace s programem napsaným v nízkoúrovňovém programovacím jazyce, zejména v C.

3.1 Ajka

Ajka[1] je morfologický analyzátor češtiny, který vznikl jako výsledek diplomové práce na Fakultě informatiky Masarykovy univerzity v Brně v roce 1999. Je napsaná v jazyce C a poskytuje lemmatizaci a tagging českých slov na základě vyhledávání pomocí vyhledávací struktury *trie*, která je navíc paměťově optimalizována díky analogii mezi ní a *deterministickým konečným automatem*. [16]

Kvůli výkonnostním potížím byla Ajka nahrazena novější Majkou, popsanou v následující sekci.

[1]Ke stažení zde: https://nlp.fi.muni.cz/projekty/ajka/ajkacz.htm

3.2 Majka

Majka[2] je nástupcem Ajky z předchozí sekce. Je implementována v jazyce C++ a vznikla na popud jejích uživatelů (mezi nimi společnost Seznam a Informační systém Masarykovy univerzity), kteří ji využívali jako součást svých fulltextových vyhledáváčů či pro detekci plagiátorství. Pro jejich výkonnostní požadavky byla Ajka příliš pomalá. Majka využívá stejnou sadu dat jako Ajka, ale v revidovaném formátu. Zcela nový je i její implementovaný analyzátor. [21]

Zdrojový kód Majky není příliš detailně zdokumentován. Její webové stránky obsahují pouze stručný příklad použití v kódu a dokumentaci odlišností oproti Ajce. Celkově se dokumentace Majky zaměřuje spíše na použití v podobě samostatného spustitelného programu, což není způsob, jakým bychom lemmatizér chtěli v této práci používat.

Majka je k dispozici pod svobodnou licencí *GNU General Public License 2.0*. Dodávané morfologické databáze jsou k dispozici pod licencí *Creative Commons Attribution-ShareAlike 3.0 Unported*.

3.3 Morče

Morče[3] neboli ***Mor****fologie* ***Če****štiny* je samostatný morfologický tagger založený na průměrovaném perceptronu. Původně vzniklo jako semestrální práce, dále pokračovalo jako diplomová práce na Matematicko-fyzikální fakultě Univerzity Karlovy v Praze. [20]

Podobně jako u Majky, i dokumentace Morčete se zabývá spíše používáním již přeloženého programu. Ani zdrojové, resp. hlavičkové soubory neobsahují mnoho dokumentace, která by mohla být návodná k použití Morčete v rámci většího programu.

Základní verze Morčete je k dispozici bez registrace pod svobodnou licencí *GNU General Public License 2.0*. Po registraci zdarma je možné získat i vývojovou verzi, která podává dle autorů přesnější výsledky, nicméně takové podmínky se zdají být problematické pro integraci v dalším svobodně licencovaném programu.

[2]Ke stažení zde: https://nlp.fi.muni.cz/ma/

[3]Ke stažení zde: https://ufal.mff.cuni.cz/morce/index.php

3.4 MorphoDiTa

MorphoDiTa[4] neboli *Morphological Dictionary and Tagger* je nástroj pro morfologickou analýzu, morfologickou generaci, tagování a tokenizaci. Vznikla – stejně jako Morče – na půdě Matematicko-fyzikální fakulty Univerzity Karlovy v Praze. Je distribuována jako samostatný nástroj nebo softwarová knihovna pro programovací jazyk C++. Součástí balíku jsou i natrénované lingvistické modely. V českém jazyce MorphoDiTa dosahuje propustnosti až ve statisících slov za sekundu. [17]

Tagger knihovny MorphoDiTa přijímá jako vstup seznam skloňovaných forem slov, pričemž při analýze bere v potaz okolní kontext slova. Díky tomu jsou lemmatizace velmi přesné – některá slova totiž sama o sobě mohou být formami několika různých lemmat. Zároveň je však možné nechat vygenerovat všechny analýzy daného slova, což je výhodné v případě, že MorphoDiTa vybere lemma, které v daném kontextu nedává smysl, ale díky omezením daným pravidly gramatiky (kapitola 2) je možné tuto chybu odhalit a opravit. Podrobněji je tato funkcionalita popsána v implementační části práce v sekci 5.3.

Na webových stránkách knihovny se nachází podrobná programátorská i uživatelská dokumentace. Zejména pak ta programátorská je velmi užitečným nástrojem pro úspěšnou integraci knihovny do dalšího programu.

MorphoDiTa je k dispozici pod svobodnou licencí *Mozilla Public License 2.0*. Dodávané lingvistické modely jsou dostupné pro nekomerční použití pod licencí *Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International*.

Pro své pokročilé funkcionality a přímočarost integrace knihovny do programu napsaném v jayzce C byla MorphoDiTa zvolena jako nejvhodněší k použití v této práci. Veškerými zmínkami o lemmatech a lemmatizaci dále v textu této práce se míní ty, které byly vygenerovány knihovnou MorphoDiTa.

[4]Ke stažení zde: https://ufal.mff.cuni.cz/morphodita

4 Figury a vektorová grafika

Pro sestavování vizuálních reprezentací štítů se jeví jako vhodný nástroj *vektorová grafika*. Ta je tvořena obrazci, které jsou ohraničeny sadami úseček a křivek. Jednou z vlastností vektorové grafiky je její prakticky neomezená škálovatelnost – rozměry obrázku lze libovolně měnit, aniž by se vyskytlo rozostření známé z grafiky rastrové. Je však nutné vzít v potaz, že obrazce nemohou být neomezeně komplexní. S počtem křivek totiž narůstá výpočetní náročnost jejich vyobrazení. Při volbě mezi rastrovou a vektorovou grafikou je tedy vždy nutné zvážit, jaký druh vizuální informace má být zobrazován.

Vyobrazení heraldických znaků bývá převážně symbolické. Ve středověkých bitvách měly znaky především rozlišovací význam – vojákův štít má zřetelně ukazovat, na kterou z bojujících stran voják patří. Příliš velké množství detailů může rozlišovací schopnost značně snižovat. Pro symbolická vyobrazení je použití vektorové grafiky ideální, neboť takové obrazce bývá celkem snadné vyjádřit právě pomocí relativně malého počtu křivek.

Další výhodou vektorové grafiky je možnost parametrizace různých vlastností (například barev) vyobrazených ploch. Díky tomu lze vytvořit předdefinovanou databázi figur reprezentovaných vektorovými obrázky, jejichž provedení lze snadno dynamicky upravovat, aby odpovídalo danému popisu štítu blazonem. V databázi figur by bylo možné možné definovat takzvané *přirozené barvy* — což je pojem, který se v blazonech často vyskytuje — jako jakési defaultní barvy, a ty by pak bylo možné přetížit na základě popisu.

4.1 Scalable Vector Graphics

Standardizovaným a široce podporovaným formátem pro vektorovou grafiku je *Scalable Vector Graphics* (zkr. SVG) vyvinutý konsorciem W3C. Formát je nadstavbou nad *eXtensible Markup Language* (zkr. XML) a umožňuje popisovat množství vektorových primitiv (obdélníky, elipsy, apod.) a rovněž vlastní obrazce popisované uspořádanými n-ticemi křivek (ty jsou zvané *cesty*). Formát rovněž podporuje různé transformace jako posunutí, otočení a škálování, a to jak na úrovni jednotlivých obrazců, tak na úrovni jejich uskupení, která lze definovat obalením definic obrazců do příslušného tagu. [2]

Obrazce v SVG lze stylovat. Mimo jiné lze nastavovat výplně, jejich barvy a chování na základě překryvu jednotlivých částí cesty; a obrysu, jeho barvy, tloušťky čáry, stylu (tj. zda se jedná o plnou či přerušovanou

čáru) a dalších vlastností. Stylování lze provést přímo uvnitř SVG dokumentu pomocí příslušných atributů tagů nebo externím kaskádovým stylem ve formátu CSS. [2]

Formát SVG je podporován širokou škálou editačního softwaru a všemi předními webovými prohlížeči. [7] Jeho přenositelnost a všudypřítomnost je tedy významným argumentem pro jeho použití.

4.2 Databáze figur

Pro generování vizuálních reprezentací štítů na základě blazonů je nutné mít zdroj útvarů, které se na štítu a okolo něj mohou vyskytnout. Z důvodu vyšší než očekávané složitosti návrhu samotného parseru a datového modelu není tento zdroj součástí softwarové implementace této diplomové práce. Nicméně, jelikož alespoň základní analýza jeho řešení byla provedena, popíšeme si v této sekci nástin, jak by takové řešení mohlo vypadat.

Předkládaným návrhem řešení je použití databázového softwaru, který spravuje vektorové grafiky reprezentující figury dostupné k použití. Grafiky v takovéto databázi by byly nejspíše kategorizovány podle sady slovních *tagů*, které by navíc mohly mít různé úrovně na základě jejich důležitosti. Slovní tagy by měly podobu lemmat, aby bylo možné je porovnávat s daty získanými z blazonu.

Nejdůležitější sadou tagů figury by byla uspořádaná n-tice podstatných jmen tvořících její název, tedy například *kůň*, *orlice* či *Panna Marie*. Pro nalezení vhodné figury by u názvu byla vyžadována *přesná shoda*. Na základě názvu by bylo možné získat z databáze figuru v různých variantách, rozlišovaných podle druhé sady tagů.

Tato druhá sada by obsahovala různé atributy, zpravidla přídavná jména s možností modifikace příslovci, případně celé popisy dalších objektů. Příkladem může být například ***otevřená*** *brána*, ***Aescalupův*** *had* či *Panna Marie* ***držící v náručí Ježíška***. Jelikož databáze pravděpodobně nebude vždy obsahovat přesně popisovanou figuru, byly by figury vybírány na základě *nejbližší shody*. Pokud je nalezena přesná shoda (tj. všechny tagy odpovídají hledaným, žádný nepřebývá, žádný nechybí), výběr je jednoznačný a použije se nalezená zcela odpovídající figura. Jsou-li nalezeny pouze částečné shody (tj. figury nejsou označeny všemi hledanými tagy nebo mají nějaké tagy navíc), použije se ta, která je hledaným tagům nejbližší – vzdálenost mezi nalezenou a hledanou sadou tagů by pravděpodobně byla kvantifikována nějakou číselnou hodnotou, aby bylo možné kandidáty porovnávat.

5 Implementace sady CBlazon

V rámci diplomové práce byla v programovacím jazyce C implementována sada knihoven a programů pro zpracování vstupních blazonů. Tyto komponenty jsou kolektivně nazvané CBlazon. Tímto názvem se na implementovaný software budeme dále odkazovat. Sada je implementována podle normy C11 s použitím některých rozšíření od GNU. Sada je volně šířitelná pod licencí *Mozilla Public License 2.0*.

Jazyk C byl zvolen pro svou výkonnost a přenositelnost (prakticky pro každou platformu včetně WebAssembly – virtuální stroj pro spouštění nízkoúrovňových programů uvnitř webových prohlížečů – existuje překladač jazyka C). Dalším kritériem pro jeho výběr byly autorovy aktivně se rozrůstající zkušenosti a preference právě pro tento jazyk.

Co se týče stylu kódu, ten je silně inspirovaný tím z jádra operačního systému Linux [18]. Zásadnějším rozdílem je, že názvy *funkčních maker* (tj. maker, která přijímají seznam parametrů v závorce) jsou **vždy** ve formátu `MULTI_WORD_MACRO_NAME(...)` (tj. velkými písmeny, oproti klasickým funkcím). To proto, aby bylo v místech jejich použití zcela zřejmé, že se jedná o makro, a může tedy být narušena sémantika, která platí obecně pro funkce v jazyce C. Této vlasnosti maker však nezneužíváme, snahou je udržet čitelnost kódu na co nejvyšší úrovni – zejména v makrech nepoužíváme data z jejich vnějšku, která jim nejsou explicitně předána.

5.1 Struktura programu

Hlavní součástí sady je knihovna funkcí, jejíž zdrojové soubory se v přiloženém archivu nacházejí v adresáři `lib`. Knihovna je členěná na několik modulů – tak zde nazýváme páry souborů typu `.c` (popř. `.cpp`[1]) a `.h`.

Rozhraní v hlavičkových souborech modulů je důsledně popisováno dokumentačními komentáři v anglickém jazyce. Snahou je dokumentovat i interní neveřejné funkce uvnitř modulů, nicméně na ty není kladen takový důraz, jako na ty veřejné.

Komentáře jsou teoreticky kompatibilní se systémem *Doxygen* [19], avšak toto použití není nijak testováno – zamýšlený způsob použití je pro čtení hlavičkových souborů programátorem přímo, což je běžný postup ve světě

[1] Pro adaptaci knihovny napsané v jazyce C++ k použití s kódem v jazyce C

knihoven jazyka C[2]. Kontrakt je popisovaný přirozeným jazykem a mimo formátování inspirované značkovacím jazykem *Markdown* [8] neobsahuje žádné speciální značky. Příklad programátorské dokumentace lze vidět na listingu 5.1.

```
/** Parses tokens from the specified `tokenizer` and constructs a blazon
 * structure. Takes an optional `config`, supplying additional information to
 * the parser. If `out_tree` is not `NULL`, writes a pointer to the constructer
 * parse tree into it.
 *
 * On success or a syntax error (and no other error), a pointer to a blazon
 * structure will be written into `out_blazon`. `out_blazon` must not be `NULL`.
 *
 * See `CBL_PARSE_STATUS_MAP` for details on individual status codes that this
 * function returns. */
enum cbl_parse_status cbl_parse(struct cbl_tokenizer *tokenizer,
                                const struct cbl_parse_config *config,
                                struct cbl_parse_tree **out_tree,
                                struct cbl_blazon **out_blazon);
```

Listing 5.1: Příklad dokumentované funkce `cbl_parse()`

Centrálním modulem knihovny je `cblazon`. Ten obsahuje definice struktur pro digitální popis štítu a procedury pro práci s těmito strukturami. Jedná se tedy o *datový model* sady, dá se rovněž mluvit o *abstraktním syntaktickém stromě* (vizte kapitolu 2). Podrobněji jsou struktury definované tímto modulem popsány v následující sekci 5.2.

Modul `parser` obsahuje kompletní logiku a datové struktury pro zpracování vstupního textu podle gramatiky popsané ze sekce 2.2. Jedná se o implementaci syntaktické analýzy *rekurzivním sestupem*. Na základě proudu vstupních tokenů modul generuje digitální popis štítu a rovněž je schopen předat volajícímu sestavený derivační strom. Modul je dále podrobněji popsán v sekci 5.4.

Knihovna MorphoDiTa, kterou sada CBlazon využívá k lemmatizaci, a která je popsaná v sekci 3.4, je implementována v jazyce C++. Ten je do značné míry kompatibilní a interoperabilní s jazykem C, který byl zvolen pro implementaci této diplomové práce. Jazyk C však nepodporuje objektově orientované programování a abstraktní datové typy ze standardní knihovny jazyka C++, které MorphoDiTa využívá. K propojení programu napsaného v jazyce C s knihovnou MorphoDiTa slouží modul `morphoadapt`, který je jako jediný ze sady implementovaný v jazyce C++, přičemž své funkce a struktury umožňuje využívat z jazyka C pomocí konstrukce `extern "C"`. Kromě

[2]Vizte populární knihovny jako `cairo`, `SDL2`, `zlib` apod.

adaptace knihovny MorphoDiTa má tento modul na starosti dělení textu na slova a interpunkční znaménka.

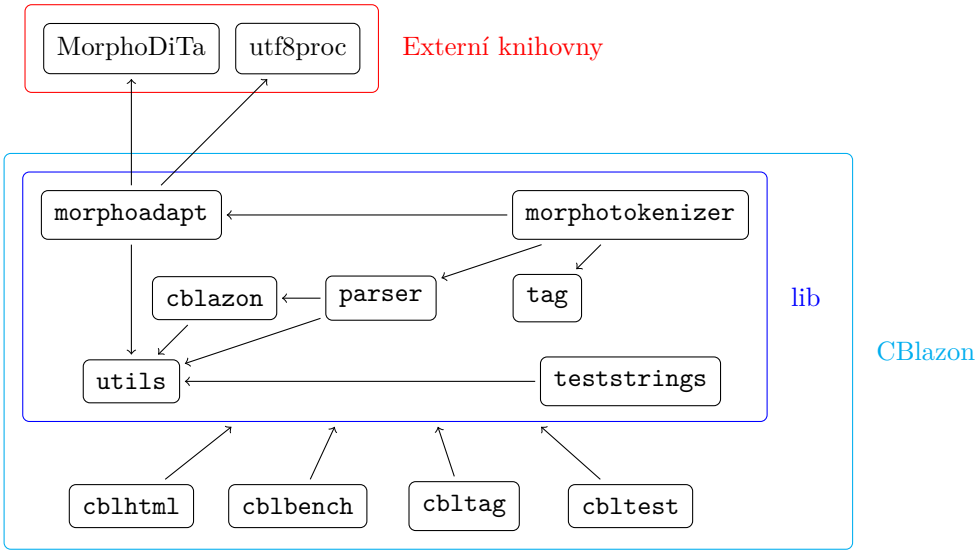
Jelikož se očekává, že vstupní texty budou obsahovat českou diakritiku, je nutné podporovat také kódování, které s ní počítá – standardem dnešní doby pro reprezentaci textu v libovolném jazyce je Unicode, konkrétně v podobě kódování UTF-8. Pro jeho korektní zpracování modul `morphoadapt` využívá knihovnu `utf8proc` [12].

Moduly `morphotokenizer` a `tag` pak zajišťují tokenizaci vstupního textu za pomoci této knihovny.

Modul `utils` obsahuje obecné utilitní funkce použitelné napříč sadou. Nacházejí se zde například makra pro definici dynamických polí[3], struktura `string_array` pro uchovávání polí řetězců, logovací a ladicí funkce apod. Vybrané součásti modulu jsou popsány v sekci 5.6.

Posledním modulem je `teststrings`. Ten obsahuje referenční řetězce, které byly použity k vývoji a testování sady CBlazon. Jedná se především o skutečné blazony znaků českých měst převzatých převážně z Houzarova *Klíče* [9], ale i několik upravených tak, aby je současně implementovaný analyzátor knihovny CBlazon byl schopen zpracovat – zpravidla se jedná o úpravy odstraňující některé prvky.

Závislosti mezi jednotlivými moduly a programy jsou naznačeny diagramem na obrázku 5.1.



Obrázek 5.1: Diagram závislostí mezi moduly v sadě CBlazon

[3]Obdoba např. typu `ArrayList` ze standardní knihovny jazyka Java nebo typu `std::vector` z STL jazyka C++

Sada obsahuje spustitelné programy `cblbench`, `cbltag`, `cblhtml` a `cbltest`.

Program `cblbench` slouží k zátěžovému testování. Provádí opakovaná zpracování několika testovacích blazonů a měří časy jednotlivých úkonů.

Program `cbltag` zpracuje určený blazon a vypíše požadované informace na základě uživatelského nastavení. Mimo jiné dokáže vypsat rozpoznané symboly (tokeny), rozpoznaný *derivační strom*, nebo sestavenou datovou strukturu štítu (resp. *abstraktní syntaktický strom*). Některé z těchto výpisů využívají escape kódy ANSI pro barevné rozlišování jednotlivých částí. Program slouží především k ladění sady CBlazon.

Program `cblhtml` má podobnou funkci jako `cbltag`, ale namísto jednoduchých terminálových výpisů generuje report ve formátu HTML. Za zmínku zde stojí o něco přehlednější bloková vizualizace derivačního stromu oproti programu `cbltag` a vizualizace dělení štítu ve formátu SVG včetně obarvení jednotlivých sekcí.

Program `cbltest` slouží k jednotkovému testování. V době psaní práce obsahuje testy pro ověření správné funkčnosti algoritmu očíslování sekcí štítu, který je popsán v sekci 5.5.

Způsob použití jednotlivých programů sady včetně příkladů jejich výstupů je popsán v uživatelské příručce v přílohách.

5.2 Datový model

Jak bylo zmíněno v předchozí sekci, hlavní datový model knihovny, tj. deklarace struktur popisujících daný štít, je obsažen v modulu `cblazon`. Model je součástí veřejného API knihovny a jeho struktury a funkce lze tedy využívat v externích programech na knihovnu napojených.

Některé z těchto struktur využívají principy objektového programování. Jazyk C sice nemá žádné zvláštní konstrukce pro objektové programování (jako je deklarace `class` v jiných jazycích), je však možné snadno docílit podobných vlastností aliasingem struktur. V praxi to vypadá tak, že je deklarována rodičovská struktura (předek; v našem případě například `struct cbl_obj`) a její potomci (zde např. `struct cbl_obj_shield` či `struct cbl_obj_figure`). Potomci mají předka vloženého jako svůj první členský atribut, díky čemuž se k nim lze chovat v zásadě jako k rodiči, především v kontextu přetypovávání ukazatelů na tyto struktury.

Pro určení konkrétního typu potomka slouží atribut v předkovi, jehož hodnota značí jeho typ. Zpravidla se jedná jednoduše o výčtový typ, v našem případě `enum cbl_obj_type` pro atribut `type`. Polymorfismus funkcí se pak odvíjí od tohoto atributu. Například je v programu definovaná funkce `cbl_`

`obj_fprint()`, která podle hodnoty atributu vypíše konkrétní typ objektu do standardního proudu (typ `FILE` ze standardního souboru `stdio.h`). Docíleno je toho jednoduše pomocí příkazu `switch` nad atributem `type`.

Pokud se čtenář zároveň dívá do zdrojových souborů, může si všimnout, že typ `enum cbl_obj_type` a několik dalších výčtových typů je generováno pomocí maker. Díky tomu je snazší vytváření některých funkcí, které s výčtovým typem souvisejí, například funkce pro výpis hodnot v podobě řetězců vyžaduje méně psaní a je méně náchylná na chyby, když je třeba přidána nová hodnota do výčtu. Tomuto generování je věnována samostatná sekce 5.7.

Struktura `struct cbl_obj_shield` v modelu popisuje, jak vypadá štít: jeho dělení, jednotlivá políčka, jejich barvy, seznamy obsažených objektů apod. Jak bylo zmíněno, je potomkem `struct cbl_obj` – to proto, že znak může teoreticky obsahovat další znaky, tedy se sám může nacházet uvnitř seznamu objektů. Tento případ v době psaní není zakotven v gramatice, takže nelze napsat blazon, pro který bude digitální popis štítu takto vygenerován, nicméně model je na to připraven a stačí pouze vhodně rozšířit gramatiku a implementovat potřebnou logiku do parseru.

Struktura obsahuje atributy `global_colours`, `splits` a `fields`.

Atribut `global_colours` obsahuje globální seznam barev štítu. Ten je naplněn názvy barev v případě, že blazon obsahuje frázi ve stylu *„štít červeno-stříbrně polcený“* – tj. seznam bude obsahovat slova `červená` a `stříbrná`. Tyto barvy mohou být přetíženy barvami explicitně zmíněnými přímo v popisu pole.

Atribut `splits` obsahuje popis dělení štítu pomocí binárního stromu reprezentovaného polem. Každý uzel může být typu `HORIZONTAL` nebo `VERTICAL` v případě, že je dané políčko rozdělené, nebo `UNSPLIT` v případě, že se jedná o list reprezentující naplnitelné políčko štítu. Po vyhodnocení, jak je štít dělen, je všem listům přiřazen index políčka z atributu `fields` tak, aby byly číslovány zprava doleva a shora dolů. Algoritmus pro toto očíslování je detailněji popsán později v sekci 5.5.

Nakonec atribut `fields` obsahuje popisy jednotlivých políček štítu reprezentovaných strukturou `struct cbl_field`. Ta obsahuje nepovinný název barvy (pokud není přítomen, je použita barva z `global_colours`) a seznam ukazatelů na objekty typu `struct cbl_obj`.

Struktura `struct cbl_obj_figure` popisuje obecnou figuru uvnitř štítu seznamy slov, která jsou určena k interpretaci v podobě tagů. Myšlenkou zde je, že bude existovat databáze obrázků figur, v níž bude možné vyhledávat pomocí těchto tagů, aby bylo možné vytvořit grafickou podobu popisovaného znaku.

Atribut `name` obsahuje jedno či více podstatných jmen popisujících figuru. Příkladem takovýchto jmen může být například *kůň*, *orlice* či *Panna Marie*.

Atribut `tag_attrs` obsahuje seznam přídavných jmen, jejichž účelem je modifikovat figuru. Příkladem mohou být slova jako *stojící*, *vyrůstající*, *obdélníkový* apod. Mohou se zde nacházet i barvy.

Dalším atributem je `obj_attrs`, který obsahuje ukazatele na další objekty, zpravidla popisy figur. Například figura postavy může držet zbraň – ta je popsána objektem, jehož ukazatel bude uložen v tomto seznamu.

Posledním atributem je booleovská hodnota `natural_colours`. Ta je nastavena na `true` v případě, že se v popisu figury nachází sousloví *„přirozených barev“* (a jinak `false`). Zde je myšlenkou, že figura v databázi bude mít předdefinované defaultní barvy, které budou použity, pokud je tato hodnota `true` nebo pokud nejsou v seznamu atributů žádné jiné barvy.

5.2.1 Textové výstupy

Program `cbltag` ze sady CBlazon generuje několik různých druhů textových výstupů. Výstup jeho příkazu `ast` (vizte uživatelskou příručku v přílohách) je generován funkcemi, které knihovna CBlazon poskytuje veřejně v rámci API modulu `cblazon` – tyto funkce pro generování uživatelsky čitelných řetězců jsou tedy dostupné i pro programy mimo sadu CBlazon.

V hlavičce `cblazon.h` se jedná o funkce, jejichž názvy mají sufix `_fprint`. Každá struktura deklarovaná v této hlavičce má k sobě přidruženou takovouto funkci buď přímo, anebo, pokud struktura objektově dědí od `struct cbl_obj`, použije se k jejímu textovému výpisu obecná funkce `cbl_obj_fprint()`.

Funkce `_fprint()` přijímají obecně tyto parametry:

- ukazatel na strukturu k výpisu
- ukazatel `file` na standardní typ `FILE` – musí ukazovat na zapisovatelný proud, do kterého bude struktura vypsána
- celé číslo `depth` – určuje odsazení – použito pro hierarchické výstupy
- booleovská hodnota `indent_first` – určuje, zda se má i první řádek výpisu odsadit – hodí se, pokud se již na dosud vypsaném řádku nachází například název atributu, který ukazuje na vypisovanou strukturu

Jejich návratová hodnota je standardního typu `size_t` a značí, kolik bajtů bylo celkem zapsáno do proudu `file`.

Některé struktury obsahují atributy výčtových typů, pro něž jazyk C sám o sobě neposkytuje funkcionalitu pro uživatelsky čitelný výpis jejich hodnot. Jak již bylo zmíněno, sada CBlazon využívá preprocesorová makra pro pohodlné generování řetězcových vyjádření těchto výčtových hodnot – v modulu `cblazon` mají takovéto výčtové typy přidružené funkce se sufixem `_name` pro získání řetězcových názvů jednotlivých hodnot. Detailní popis tohoto využití preprocesoru lze nalézt dále v sekci 5.7.

Funkce `_fprint()` provádějí strukturovaný výstup, který je podobný formátu JSON nebo formátu, jakým se v programovacím jazyce C inicializují struktury. Je určen čistě ke čtení člověkem především za účelem kontroly, zda byl vstupní blazon správně zpracován.

5.3 MorphoDiTa a tokenizace

Sada CBlazon obsahuje modul `morphoadapt`, který je zodpovědný za převod vstupních a výstupních dat tak, aby v ní bylo možné využívat knihovnu MorphoDiTa. Primárním účelem tohoto modulu je umožnění interoperability kódu napsaného v programovacím jazyce C++ s kódem napsaným v programovacím jazyce C.

Druhotným účelem modulu je rozdělení vstupního textu na jednotlivá slova a interpunkční znaménka. Tagger knihovny MorphoDiTa přijímá seznamy (typu `std::vector<ufal::morphodita::string_piece>`) skloňovaných forem slov – s interpunkcí vůbec nepracuje. Modul utváří jak seznam slov pro knihovnu MorphoDiTa, tak seznam interpunkčních znamének. Struktury typu `struct punct_entry` reprezentující interpunkční znaménka obsahují mimo jiné index slova, kterému předcházejí – ten je poté použit ke spojení výsledků taggingu s interpunkcí do jednoho sekvenčního seznamu, ze kterého později modul `morphotokenizer` generuje proud tokenů.

V hlavičkovém souboru je deklarována *neprůhledná struktura*[4] `struct morpho_tagger`. Tu lze obecně v kódu, který hlavičku používá, použít pouze jako ukazatel, který je předáván relevantním funkcím. Prakticky se jedná o koncept zapouzdření známý z objektového programování.

Uvnitř modulu je tato struktura použita jako alias pro třídu `ufal::morphodita::tagger`, která poskytuje funkcionalitu pro lemmatizaci a tagging slov ze vstupního textu.

[4] Angl. *opaque structure* – deklarovaný název struktury bez deklarace jejího obsahu; používáno pro typování ukazatelů na struktury se skrytým interním obsahem v místech, kde je taková vlasnost žádoucí

Výstup lemmatizace je struktura `struct morpho_tagged` obsahující seznam otagovaných lemmat a rozpoznaných interpunkčních znamének. Tento seznam pak slouží tokenizeru z modulu `morphotokenizer` ke generování tokenů, což jsou terminální symboly, nad nimiž parser (popsaný v následující kapitole) provádí syntaktickou analýzu textu.

Tokeny vygenerované jako výsledek lemmatizace zahrnují *varianty*, které vycházejí z několika možností, které poskytuje tagger z knihovny MorphoDiTa. Konkrétní podobě slova lze totiž často bez okolního kontextu přisoudit různé mluvnické kategorie a lemmata. Tagger knihovny MorphoDiTa sice s okolním kontextem slova pracuje, ne vždy je však schopen slovo určit správně. CBlazon si proto kromě dat statisticky určených přímo taggerem uchovává až tři další varianty rozpoznané třídou `morpho` rovněž z knihovny MorphoDiTa. Parser pak může při rekurzivním sestupu a rozpoznávání konkrétních symbolů tyto varianty použít.

V praxi zavedení variant pomohlo například při chybném rozpoznání slova `se`, kdy tagger určil, že se jedná o zvratné zájmeno, avšak ve skutečnosti se v blazonu jednalo o skloněnou předložku `s` – zvratné zájmeno v daném kontextu nedávalo smysl. Dalším příkladem bylo slovo `brána` v jednom z blazonů, kde jej tagger rozpoznal jako časované sloveso `brát`, ale mělo se jednat o podstatné jméno. Díky použití variant v tokenu je parser schopen tyto nepřesnosti taggeru tolerovat.

Modul `morphotokenizer` slouží k převodu seznamu lemmat a interpunkčních znamének na proud tokenů kompatibilních s implementovaným parserem. Jedná se prakticky o dopředný iterátor nad tímto seznamem.

5.4 Parser

Jak bylo dříve zmíněno, syntaktický analyzátor blazonů je implementován v modulu `parser`. Jedná se o nejkomplexnější část celé implementace, a proto je její vnitřní fungování zejména pro potřeby navazujících prací podrobně popsáno v této sekci. Dalším důležitým zdrojem informací jsou dokumentační komentáře obsažené přímo ve zdrojovém kódu.

5.4.1 Rekurzivní sestup

Parser provádí analýzu textu *rekurzivním sestupem*. To znamená, že pro každý neterminální symbol z gramatiky (kapitola 2) je v programu definována funkce, která má za úkol zpracovat skupinu terminálních symbolů (zde rovněž zvaných *tokenů* – ve smyslu softwarové implementace se jedná o jednu a tutéž věc) na základě jeho přepisovacích pravidel.

Všechny funkce neterminálů jsou stejného společného typu, který je v kódu deklarován jako `typedef` s názvem `nonterm_cb`. Jako parametry tyto funkce přijímají ukazatele na globální kontext parseru (typu `struct cbl_parse_context`), na kontext volání (typu `struct cbl_level_context`) a na paměť, do které může funkce zapsat svá výstupní data. Návratový typ funkce je `struct cbl_nonterm_result`, udávající informaci o úspěšnosti rozpoznání neterminálního symbolu na daném místě v textu. Tento vnitřní datový model parseru je podrobněji popsán v podsekci 5.4.2.

Pro zjednodušení práce při implementaci rekurzivního sestupu jsou jednotlivé deklarace funkcí neterminálů generovány pomocí maker s prefixem `NONTERM_DECL`. Tato makra generují hlavičky funkcí, jejichž účelem je provést syntaktickou analýzu pro daný neterminální symbol. Příkladem budiž listing 5.2, kde vidíme použití maker pro vygenerování hlaviček funkcí neterminálů `FieldDesignation`, `ComplexSplits` a `ComplexSplit`.

```
struct cbl_complex_splits_ctx {
        struct cbl_obj_shield *shield;
        ssize_t current_split_ix;
};

NONTERM_DECL0(FieldDesignation);
NONTERM_DECL(ComplexSplits, struct cbl_obj_shield);
NONTERM_DECL(ComplexSplit, struct cbl_complex_splits_ctx);
```

Listing 5.2: Příklad deklarací neterminálních symbolů

Lze si všimnout, že v souboru `parser.c` jsou nejprve vypsány všechny deklarace funkcí neterminálů (tj. pouze hlavičky bez těla) a až po nich následují jejich definice (včetně těla). Jedná se o klasický případ *dopředných deklarací*[5] v jazyce C, aby se funkce směla odkazovat na jinou funkci, jejíž definice se nachází v kódu za ní. To je nutné, protože jazyk C je historicky navržen pro jednoprůchodovou analýzu zdrojových souborů. Snahou je udržovat totožné pořadí deklarací a definic neterminálů mezi textovou podobou gramatiky a jejím přepisem do kódu programu, přičemž textová podoba má jakousi pocitově dobře čitelnou strukturu, která vznikla ještě před implementací programu. Pro zkušeného programátora v jazyce C toto není žádná novinka, nicméně kvůli tomu, že jsou zde pro generování hlaviček použita makra, což může být na první pohled matoucí, vyvstala potřeba tuto skutečnost explicitně zmínit.

[5] Angl. *forward declaration*

K volání funkcí neterminálů jsou určena makra s prefixem `DESCEND_INTO`. Makro automatizovaně volá inline funkci `descend_into_impl()`, pro kterou generuje zejména ukazatel na funkci neterminálu a řetězcovou podobu názvu neterminálu. Řetězec může být dále využit například pro debugovací výpisy derivačního stromu, čehož využívá program `cbltag`.

Samotná definice funkce neterminálu, tj. hlavička funkce generovaná makrem `NONTERM_DECL` a pod ní tělo této funkce, může vypadat například tak, jak lze vidět na listingu 5.4. Pozornému čtenáři přitom jistě neunikne určitá korelace mezi definicí funkce a textovou podobou přepisovacích pravidel pro neterminální symbolu z gramatiky, jejíž odpovídající úryvek lze vidět na listingu 5.3.

Volání přes makro je zde použito opět za účelem zkrácení zápisu a snížení redundance informací obsažených v kódu – bez něj by totiž bylo nutné psát název neterminálu dvakrát: jednou v názvu funkce a jednou jako řetězcovou hodnotu pro zaznamenání derivačního stromu, což může být náchylné na chyby ve chvíli, kdy během vývoje dojde ke změně názvu. Díky makru jsou spolu tyto hodnoty pevně spjaty a pokud je v názvu chyba (tedy pokud se název odkazuje na neexistující neterminál), překlad programu bude vždy ukončen se srozumitelným chybovým hlášením.

Makra `DESCEND_INTO` také – tentokrát za běhu programu – zajišťují zvláštní kontrolu v případě, že je rozpoznání daného neterminálu v daném místě vstupního textu vyžadováno, což je signalizováno booleovskou hodnotou `match_required`. V případě, že je neterminál v daném místě vyžadován, ale není rozpoznán, nastavuje program atribut `correct` v návratové hodnotě funkce na `false`, čímž se oznámí syntaktická chyba ve vstupním blazonu.

Co se týče hodnoty `match_required`, ta je zpravidla `true`. Hodnoty `false` nabývá, pokud je daný neterminál prvním symbolem daného přepisovacího pravidla nebo nepovinného opakovacího výrazu (vizte sekci 2.1 o formátu popisu gramatiky).

Neterminály mohou přijímat ukazatel na paměť, do které zapisují svůj výstup. Tímto výstupem jsou zpravidla struktury z modulu `cblazon`. Tímto způsobem se ze vstupního textu generuje abstraktní syntaktický strom. Nevyžaduje-li neterminál žádnou výstupní paměť, použijí se v kódu makra definovaná se sufixem `0` (tj. `NONTERM_DECL0()` a `DESCEND_INTO0()`). V makrech je pomocí triku s operátorem `sizeof` zajištěna typová kontrola pro tyto ukazatele, čímž se snižuje šance na zavolání nesprávného neterminálu či poskytnutí nesprávného typu struktury – trik spočívá v tom, že do parametru operátoru `sizeof` lze vložit výraz, který se ve výsledném programu neprovede, nicméně překladač nad ním provede běžnou typovou kontrolu.

```
<FieldAttrOptWith> ::= "s" <FieldAttr>
                     | "na" <FieldAttr>
                     | <FieldAttr> ;
```

Listing 5.3: Přepisovací pravidla pro symbol `FieldAttrOptWith`

```
NONTERM_DECL(FieldAttrOptWith, struct cbl_field)
{
        struct cbl_nonterm_result r;
        const struct cbl_token *t;
        size_t ti;

        if ((r = DESCEND_INTO(FieldAttr, ctx, lvctx, false, out)).match) {
                if (!r.correct)
                        return nonterm_syntax_error();

                return nonterm_success();
        } else if (VARIANT_MATCHES((t = ctx_get_token(ctx, 0)), v,
                                   v->keyword == CBL_TOKEN_KEYWORD_WITH, &ti))
        {
                ctx_match_consume_tokens(ctx, lvctx, &ti, 1);

                r = DESCEND_INTO(FieldAttr, ctx, lvctx, true, out);
                if (!r.correct)
                        return nonterm_syntax_error();

                return nonterm_success();
        } else if (VARIANT_MATCHES(t, v,
                                   v->keyword == CBL_TOKEN_KEYWORD_ON, &ti))
        {
                ctx_match_consume_tokens(ctx, lvctx, &ti, 1);

                r = DESCEND_INTO(FieldAttr, ctx, lvctx, true, out);
                if (!r.correct)
                        return nonterm_syntax_error();

                return nonterm_success();
        }

        return nonterm_no_match();
}
```

Listing 5.4: Převod symbolu `FieldAttrOptWith` na funkci v jazyce C

Makra s prefixem `NONTERM_DECL` generují hlavičky funkcí s názvem ve formátu `cbl_grammar_nonterm_*`, kde hvězdička je nahrazena názvem neterminálu. Použit je přitom název neterminálu z popisu gramatiky – tedy například funkce neterminálu `Shield` má uvnitř programu po rozložení maker název `cbl_grammar_nonterm_Shield`. Nicméně jelikož jsou všechny tyto funkce volány pomocí maker, která jako parametr přijímají čistě název neterminálního symbolu, není pro většinu interakcí s kódem (mimo úpravu maker samotných) tento detail o názvu funkce příliš důležitý.

Parser ve své současné podobě netoleruje syntaktické chyby. Zpracování vstupního textu je vždy ukončeno okamžitě ve chvíli, kdy je ve vstupním textu nalezen neočekávaný symbol. Není-li definované konkrétní chybové hlášení, vypíše parser generickou zprávu *Neočekávaný výraz ‘`slovo`‘*. V kódu parseru je připraveno makro `CTX_ERROR()`, pomocí něhož je možné uživateli poskytnout přesnější popis chyby.

5.4.2 Vnitřní kontext a reakce na chyby

Každý jednotlivý průchod parserem – tj. každé volání funkce `cbl_parse()` – si uchovává globální data v podobě struktury `struct cbl_parse_context`. Tato struktura uchovává počáteční konfiguraci parseru, ukazatel na používaný tokenizer, buffer tokenů apod.

Buffer tokenů slouží k dočasnému uchování tokenů, jejichž význam dosud nebyl rozpoznán. Tokenizer je totiž implementovaný tak, že text zpracovává jako sekvenční proud: opakované volání jeho členské funkce `next_cb()` vždy vrací další rozpoznaný token v pořadí – není možné se v něm vracet zpět. Parser ale často potřebuje uchovat alespoň jeden token v paměti, protože při volání funkce neterminálu prakticky „zkouší“, zda daná část textu odpovídá danému neterminálu, tj. zda se v ní nachází očekávaných prvních n terminálních symbolů.

Těchto n terminálních symbolů můžeme nazvat *souslovím*. Můžeme například v některém z neterminálů chtít rozpoznat sousloví `přirozené barvy` – každé z těchto dvou slov je jedním terminálem, ale jednotlivě v daném místě nedávají smysl (ale při rozpoznávání jiného neterminálu by mohly). Naše funkce proto načte dva tokeny do bufferu a zkontroluje, zda se jedná o přesně tato dvě klíčová slova. Pokud ano, jsou z bufferu vyjmuta a uvolňuje se místo pro další tokeny. Pokud ne, neterminál se označuje jako nerozpoznaný, nicméně tokeny jsou v bufferu ponechány, aby se funkce jiného neterminálu mohla pokusit je rozpoznat jiným způsobem.

Kromě globálního kontextu si ještě každé volání neterminálu vytváří svůj kontext v podobě struktury `struct cbl_level_context`. Ta je hierarchická

– každá instance si uchovává seznam potomků a (kromě kořenové) také ukazatel na rodičovskou instanci. Potomkem může být další instance `struct cbl_level_context` nebo rozpoznaný token. Ve chvíli, kdy je neterminál či token rozpoznán, je zařazen do seznamu potomků rodiče. Vzniká tak reprezentace derivačního stromu, kterou pak lze využít k jeho výpisu.

Každá funkce neterminálního symbolu vrací hodnotu typu `struct cbl_nonterm_result`. Tato struktura obsahuje pouze dva booleovské atributy: `match` a `correct`.

Atribut `match` indikuje, zda byl rozpoznán první token v daném neterminálu: pokud ano, hodnota je `true`, neterminál se stává součástí derivačního stromu textu a pokračuje se ve vyhodnocování jeho syntaktické korektnosti; pokud ne, hodnota je `false` a mohou nastat dvě situace:

- **není-li** neterminál vyžadován (zpravidla, pokud se nachází na začátku jiného neterminálu nebo na začátku opakující se sekvence), pokračuje se vyhodnocováním dalšího potenciálního neterminálu;
- **je-li** neterminál vyžadován, je zároveň nastaven na hodnotu `false` atribut `correct` popsaný níže.

To, zda neterminál je, či není v daném místě vyžadován, je určeno parametrem `match_required`, který je předáván makrům s prefixem `DESCEND_INTO` popsaným v předchozí podsekci.

Hodnota `true` atributu `correct` značí, že neterminál byl rozpoznán (tj. musí být `true` i hodnota `match`) a text je zároveň podle gramatiky syntakticky korektní. Je-li jeho hodnota `false`, značí se tím, že došlo k syntaktické chybě. Ta může být pomocí makra `CTX_ERROR()` textově popsána, což je propsáno do nakonfigurovaného bufferu.

Při každém volání neterminálu musí být důsledně dbáno na kontrolu atributu `correct`, aby zpracování textu bylo v případě chyby řádně zastaveno.

Takto nastavenou sémantikou atributů mohou instance `struct cbl_nonterm_result` nabývat pouze tří možných hodnot. Pro pohodlnější zacházení a lepší čitelnost jsou proto definovány inline funkce `nonterm_no_match()`, `nonterm_syntax_error()` a `nonterm_success()`, které tyto tři hodnoty generují.

5.4.3 Formát tagu

Knihovna MorphoDiTa pro vstupní slova generuje kromě lemmat rovněž tagy ve formátu popsaném manuálem pro morfologickou anotaci [14]. Tag je

reprezentován řetězcem 15 symbolů, kdy každý popisuje jednu mluvnickou kategorii.

Symbolem je nejčastěji písmeno anglické abecedy (např. `P` jako plurál), popř. jiný symbol (např. `f` pro infinitiv či `^` pro spojku). Kategorie, které nejsou pro dané slovo aplikovatelné, se značí pomlčkou `-` (např. čas pro podstatné jméno nedává smysl).

Popisovanými kategoriemi jsou v tomto pořadí: *slovní druh*, *slovní poddruh*, *rod*, *číslo*, *pád*, *rod vlastníka*, *číslo vlastníka*, *osoba*, *čas*, *stupeň*, *negace*, *slovesný rod*, *vid*, *agregát* a *varianta*.

CBlazon tyto tagy aplikuje na jednotlivá slova tak, že jejich hodnoty převádí na hodnoty výčtových typů, které jsou součástí struktury `struct cbl_token`. Například slovní druh je mapován na datový typ definovaný v modulu `parser`, zvaný `enum cbl_token_subtype`. Tato hodnota je pak použita při syntaktické analýze pro rozpoznání druhu tokenu, a tedy pro rozhodnutí, do které větve rekurzivního sestupu má program v dané části textu vstoupit.

5.4.4 Vnější API

Modul `parser` poskytuje ve svém hlavičkovém souboru rozhraní (API), které lze pomyslně rozdělit na dvě části. První částí je definice datových typů a prototypů funkcí pro tokenizaci textu. Část druhá je tvořena funkcí `cbl_parse()` pro vyvolání samotné syntaktické analýzy a jí náležícími datovými typy, pomocí kterých je možné ji parametrizovat.

Tokenizace

V první části API je tedy deklarována struktura `struct cbl_token` představující rozpoznaný terminální symbol v gramatice. Kromě ukazatele na přímou podobu slova ve vstupním textu obsahuje například lemma rozpoznané knihovnou MorphoDiTa, pozici ve zdrojovém textu (řádek a sloupec), tag v podobě popsané výše v podsekci 5.4.3 či kategorizaci tokenu.

Token může být kategorizován až čtyřmi atributy nesoucími hodnoty výčtových datových typů. Primárním členem je `type` – ten rozlišuje, zda token představuje slovo (hodnota `WORD`) či některé z interpunkčních znamének; také může nabývat hodnoty `END_OF_FILE` značící konec vstupního textu. Je-li token typu `WORD`, bude mít vyplněnu alespoň jednu variantu, která obsahuje atributy `subtype`, `keyword` a `nametype`. Hodnota atributu `subtype` obsahuje identifikátor slovního druhu. Atribut `keyword` může označovat konkrétní klíčové slovo, např. *štít*, *řez*, *hlava*, *pata*, nejrůznější předložky a podobně. Atribut `nametype` rozlišuje, zda token obsahuje nějaký název – může se jednat

o jméno osoby, název města, národost, barvu apod.; podle manuálu pro morfologické tagování [14] – pro CBlazon je zde zejména důležitá hodnota `COLOUR` pro rozpoznávání barev.

Jednotlivé instance struktury `struct cbl_token` jsou proudově generovány *tokenizerem* (lexikálním analyzérem). Ten je definován obecnou strukturou `struct cbl_tokenizer`, která sama o sobě obsahuje pouze atribut `next_cb`. Ten je ukazatelem na funkci, která na opakovaná volání proudově vrací rozpoznané tokeny. Tato struktura je určena k použití formou objektového programování. Konkrétní implementace tedy bude definovat vlastní strukturu obsahující `struct cbl_tokenizer` jako prvního člena a vlastní stav jako členy následující.

V rámci práce byla implementována jediná implementace takového tokenizeru. Ta se nachází v modulu `morphotokenizer` a k tokenizaci využívá knihovnu MorphoDiTa. Implementace deklaruje strukturu `struct cbl_morpho_tokenizer` přetypovatelnou na `struct cbl_tokenizer`, kterou lze použít právě výše popisovaným způsobem. Detaily fungování implementovaného tokenizeru byly popsány v sekci 5.3.

Syntaktická analýza

Tato druhá část API je tvořena funkcí `cbl_parse()` a parametry a datovými typy, které jí náleží. Je určena k použití uživatelskými programy, které za její pomoci mohou získat ze zdrojového blazonu strukturovaný datový popis štítu.

Funkce `cbl_parse()` je vstupním bodem pro spuštění rekurzivního sestupu. Jako první povinný parametr přijímá vstupní ukazatel `tokenizer`. Ten se odkazuje na instanci `struct cbl_tokenizer`, která má být použita jako zdroj symbolů pro gramatiku.

Druhým parametrem je ukazatel `config`, který je nepovinný a může tedy nabývat hodnoty `NULL`. Struktura `struct cbl_parse_config`, na kterou se odkazuje, obsahuje dodatečnou konfiguraci pro parser. Jedná se především o ukazatele na debugovací *callback funkce* a ukazatel na paměťový buffer, do kterého může parser vypisovat chybová hlášení.

Posledním parametrem je ukazatel na ukazatel (tedy v jazyce C označený dvěma hvězdičkami `**`) `out_blazon`. Jeho vyplnění je povinné a jeho účelem je předání ukazatele na vygenerovaná data štítu na určenou adresu volající funkci. Parser se sám stará o alokaci paměti pro strukturovaný popis štítu – volající pouze předává pozici ukazatele, kam bude zapsána příslušná adresa.

Funkce vrací hodnotu výčtového typu `enum cbl_parse_status`, značící, zda bylo zpracování vstupního blazonu úspěšné, a pokud ne, pak z jakého důvodu.

5.5 Algoritmus pro očíslování sekcí štítu

Pro očíslování jednotlivých sekcí děleného štítu byl v rámci práce navržen algoritmus, který je očísluje v pořadí opticky zprava doleva a shora dolů (přičemž, jak je popsáno v kapitole 1 o blazonech, strany jsou vnímány z pohledu štítonoše, tedy pravá je na obrázku štítu vlevo a naopak).

Dělení štítu je v datovém modelu popsáno binárním stromem, jehož listy (tj. uzly bez potomků) reprezentují políčka štítu, ve kterých se mohou nacházet nějaké objekty.

Algoritmus využívá prioritní frontu, do které vkládá listy stromu na základě vygenerovaných celočíselných proměnných `righter` a `higher` – v tomto pořadí vyjadřují, zda se políčko nachází více vpravo a více nahoře. Obě proměnné mohou nabývat i záporných hodnot. Tyto proměnné jsou generovány pro každý uzel stromu podle rekurzivní funkce popsané pseudokódem na listingu 5.5.

```
function zařaď_uzel(fronta, uzel, righter, higher)
{
        if (uzel.typ == UNSPLIT) {
                vlož_do_fronty(fronta, uzel, righter, higher);
        } else if (uzel.typ == VERTICAL) {
                zařaď_uzel(fronta, uzel.pravý_potomek,
                        righter * 2 + 1, higher);
                zařaď_uzel(fronta, uzel.levý_potomek,
                        righter * 2 - 1, higher);
        } else if (uzel.typ == HORIZONTAL) {
                zařaď_uzel(fronta, uzel.horní_potomek,
                        righter, higher * 2 + 1);
                zařaď_uzel(fronta, uzel.dolní_potomek,
                        righter, higher * 2 - 1);
        }
}
```

Listing 5.5: Pseudokód vkládání listů do fronty

Uzel v prioritní frontě se nachází blíže začátku než ostatní v případě, že:

- jeho hodnota `higher` je vyšší

  NEBO

- jeho hodnota `higher` je stejná **A** jeho hodnota `righter` je vyšší

Funkce `cbl_split_tree_enumerate_fields()`, `cbl_split_tree_enqueue_leaves()` a `splitqueue_put()` v kódu modulu `cblazon` implementují popisovaný algoritmus. Byly pro ně rovněž napsány jednotkové testy v programu `cbltest`, které ověřují správnost očíslování několika uměle vytvořených situací.

5.6 Vybrané součásti utilitního modulu

Modul `utils` obsahuje definice obecně užitečných struktur a funkcí, které se nehodí do žádného jiného konkrétního modulu. Často je jeho rozhraní využíváno napříč více moduly sady CBlazon. V této podsekci si popíšeme několik zajímavějších částí tohoto modulu.

Dynamicky alokovaný formátovaný řetězec

První částí jsou funkce `xasprintf()` a `vxasprintf()`. Ty jsou náhradou za funkce `asprintf()` a `vasprintf()`, které jsou součástí překladačů rodiny GCC, ale nepatří do standardu jazyka C a jiné překladače (resp. implementace knihovny `libc`, která obsahuje standardní knihovnu jazyka C a její možná rozšíření) je tudíž nemusí implementovat.

Funkce slouží k vytvoření formátovaného řetězce, který je automaticky alokován na haldě – není tedy nutné dopředu znát výslednou délku řetězce, alokace je vždy provedena tak, aby se do vzniknuvšího bufferu celý řetězec (včetně ukončovacího znaku `NUL`) vešel. Jedná se o variaci na funkce `snprintf()` a `vsnprintf()` ze standardní knihovny, které však požadují předání předem alokovaného paměťového bufferu, do kterého je výsledný řetězec zapsán.

Každý takto alokovaný řetězec je po ukončení jeho používání potřeba uvolnit pomocí standardní funkce `free()`, aby nedošlo k úniku paměti.

Dynamické pole

Další část modulu implementuje *dynamická pole*. Jedná se o šablony datových struktur, kterými program spravuje pole (tedy sekvenční kontinuální

skupiny dat stejného typu), do kterých je možné dynamicky přidávat další členy, aniž by bylo nutné dopředu vědět, kolik členů bude nakonec pole obsahovat.

Pomocí makra `ARRAY()` tak lze deklarovat datový typ dynamického pole, které obsahuje členy dalšího datového typu, které je tomuto makru předáno jeho jediným parametrem.

Tento datový typ splňuje idiom *zero-is-initialization* (zkr. *ZII*) ražený vývojářem počítačových her a jiných nízkoúrovňových programů Caseym Muratorim. [15] Tento idiom říká, že strukturu lze inicializovat jednoduše nastavením celé její paměti na nulové hodnoty (typicky zápisem standardní funkcí `memset()` nebo alokací standardní funkcí `calloc()`), čímž se stane platnou a připravenou k použití za pomoci k tomu určených funkcí či maker. Tento idiom zjednodušuje nakládání se strukturou, která jej splňuje tím, že není nutné mít speciální funkci určenou k její inicializaci.

Další členy je do dynamického pole možné přidávat makry `ARRAY_ADD()` (k přidání člena v podobě určené parametrem) či `ARRAY_ADD_ZERO()` (k přidání vynulovaného člena – hodí se, pokud struktury obsažené v dynamickém poli rovněž splňují ZII). Obě makra přitom zajišťují, že na dalšího člena je v poli místo voláním makra `ARRAY_RESERVE()`. Je-li paměť dynamického pole příliš malá, dojde k jejímu zvětšení pomocí standardní funkce `realloc()`.

Další makra pro nakládání s dynamickými poli jsou `ARRAY_TAIL()` k získání ukazatele na posledního člena pole, `ARRAY_CLEAR()` k vyprázdnění pole (tj. nastavení počtu členů na nulu) a `ARRAY_DEINIT()` k uvolnění prostředků struktury.

Pole řetězců

Nadstavbou nad dynamickými poli deklarovanými makrem `ARRAY()` jsou pole řetězců, která představuje struktura `struct string_array`. Struktura slouží k uchovávání skupin řetězců, například názvů barev, které se hodí při popisování sekcí štítu. Obsahuje ukazatel na dynamickou oblast v paměti, v níž jsou za sebou uloženy vložené řetězce a dynamické pole záznamů, které na řetězce v oblasti ukazují. Na jednotlivé řetězce je odkazováno ofsety – tj. indexy bajtů relativními k začátku dynamické oblasti. Struktura rovněž splňuje ZII.

Řetězce lze do struktury přidávat pomocí funkce `string_array_add()`, která přijímá jako parametry ukazatel na řetězec a počet bajtů délky řetězce. Poskytnutý řetězec je zkopírován na konec dynamické oblasti spravované strukturou `struct string_array` a odkaz na něj a jeho délka je zapsána do

jejího dynamického pole se záznamy. Je rovněž možné do pole přidat řetězec nulové délky pomocí funkce `string_array_add_empty()`.

K získání řetězců slouží funkce `string_array_get()`, která jako parametry přijímá index záznamu a místa v paměti, kam má uložit ukazatel na řetězec a jeho délku.

Mezi další funkce patří `string_array_size()`, která vrací počet záznamů v poli řetězců; `string_array_fprint()`, která do poskytnutého proudu vypíše všechny řetězce obsažené v daném poli; a `string_array_deinit()`, která uvolní prostředky pole.

5.7 Generování výčtových typů

Datovému typu, který umožňuje použití konečného množství programátorem určených a pojmenovaných konstantních hodnot, říkáme *výčtový typ*. V jazyce C je výčtový typ deklarován pomocí klíčového slova `enum`, volitelně s názvem typu, a *výčtem* povolených identifikátorů. Volitelně lze také jednotlivým identifikátorům určit explicitní hodnoty. Není-li hodnota identifikátoru určena explicitně, určí ji kompilátor automaticky sekvenčně – defaultně od nuly, nebo od poslední explicitně určené hodnoty, existuje-li taková. Výčtový typ je v jazyce C aliasem pro celočíselný typ s předdefinovanými konstantními hodnotami.

Výhodou oproti starším standardům jazyka, které výčtové typy nepodporovaly, a kde byly výčty zpravidla realizovány pomocí typu `int` a sadou preprocesorových direktiv `#define`, je především lepší statická analýza. Například při použití příkazu `switch` je kompilátor schopen programátora uporoznit, pokud je v něm vynechána větev pro některou z hodnot daného výčtového typu.

Jazyk sám o sobě nedefinuje žádné speciální operace nad výčtovými typy. Zejména pro ladění programu však může být výhodné, aby pro výčtový typ existovaly specializované funkcionality, jako je například vypsání člověkem čitelného názvu hodnoty. Aby nebylo nutné tuto funkcionalitu psát ručně – což může ztěžovat udržovatelnost, pokud dojde k nějaké změně ve výčtu (přidání či odebrání hodnoty, změně názvu atd.) – generuje sada CBlazon takovéto výčtové typy pomocí preprocesorových maker, jejichž definice mohou vypadat například tak, jak je vyobrazeno na následujícím listingu 5.6.

```
#define CBL_OBJ_TYPE_MAP(XX) \
        XX(BEARER, bearer) \
        XX(HELMET, helmet) \
        XX(SHIELD, shield) \
        XX(FIGURE, figure) \
        //
```

Listing 5.6: Definice makra pro generování výčtu

`XX` je parametrem předávané další makro, které v tomto konkrétním případě přijímá dva parametry zvané `cid` a `vid` (zkratky pro *constant identifier* a *variable identifier*). Tyto parametry mohou být využity k vygenerování dalšího kódu.

Makro `CBL_OBJ_TYPE_MAP()` samo o sobě nedeklaruje žádný datový typ, ani nedefinuje žádné hodnoty. Může k tomu však být využito. Pro vygenerování skutečně použitelného výčtového typu lze definovat makro `XX()` a předat jej makru `CBL_OBJ_TYPE_MAP()` následujícím způsobem:

```
enum cbl_obj_type {
        CBL_OBJ_TYPE_NULL = 0,
#define XX(cid, vid) CBL_OBJ_TYPE_ ## cid,
        CBL_OBJ_TYPE_MAP(XX)
#undef XX
        CBL_OBJ_TYPE_COUNT
};
```

Listing 5.7: Generování výčtového typu makrem `CBL_OBJ_TYPE_MAP()`

Operátor `##` v makrech jazyka C značí konkatenaci identifkátorů, tj. identifikátor před operátorem je spojen bez mezer s identifikátorem za operátorem, přičemž ještě před konkatenací dojde k rozvinutí parametrů makra. Vzniknou tak identifikátory výčtových hodnot `CBL_OBJ_TYPE_BEARER`, `CBL_OBJ_TYPE_HELMET`, `CBL_OBJ_TYPE_SHIELD` a `CBL_OBJ_TYPE_FIGURE`. Je vhodné podotknout, že makro `XX()` na tomto místě může mít libovolný název, neboť je makru `CBL_OBJ_TYPE_MAP()` předáváno jako parametr.

Obdobným způsobem je pak možné vygenerovat funkci, která při poskytnutí hodnoty výčtového typu vrátí její název v podobě řetězce. Podobu funkce přímo ve zdrojovém kódu lze vidět na listingu 5.8, její pravděpodobnou podobu po rozvinutí maker lze pak vidět na listingu 5.9. Přesná podoba

po rozvinutí záleží na konkrétní implementaci překladače jazyka C – rozdíly však budou pouze v detailech, které nebudou mít vliv na výslednou podobu přeloženého programu.

V makru je použit operátor `#`, který značí převod identifikátoru na řetězec (anglicky se tento operátor nazývá *stringizing operator*). Operátor provede rozvinutí parametru za ním a výsledek obklopí dvojitými uvozovkami (`"`), čímž ve výsledném vygenerovaném kódu vznikne řetězcová konstanta.

```
const char *cbl_obj_type_name(enum cbl_obj_type type)
{
        switch (type) {
        case CBL_OBJ_TYPE_NULL:
                return "(null)";
#define XX(cid, vid) case CBL_OBJ_TYPE_ ## cid: return #cid;
        CBL_OBJ_TYPE_MAP(XX)
#undef XX
        default:
                return "(invalid)";
        }
}
```

Listing 5.8: Generování funkce pomocí makra `CBL_OBJ_TYPE_MAP()`

```
const char *cbl_obj_type_name(enum cbl_obj_type type)
{
        switch (type) {
        case CBL_OBJ_TYPE_NULL:     return "(null)";
        case CBL_OBJ_TYPE_BEARER:   return "BEARER";
        case CBL_OBJ_TYPE_HELMET:   return "HELMET";
        case CBL_OBJ_TYPE_SHIELD:   return "SHIELD";
        case CBL_OBJ_TYPE_FIGURE:   return "FIGURE";
        default:                    return "(invalid)";
        }
}
```

Listing 5.9: Funkce `cbl_obj_type_name()` po rozvinutí maker

5.8 Paměťové náležitosti

Jazyk C neposkytuje sám o sobě žádné automatizované nástroje pro správu paměti, jako například *garbage collector*, *chytré ukazatele* apod. To znamená, že o alokaci a následnou dealokaci paměti se musí postarat programátor. Ruční správa paměti má svá specifická úskalí. Je nutné alespoň pomyslně definovat životní cyklus alokované paměti, aby bylo jasné, kdy je jaká paměť k dispozici a kdy má zaniknout.

Co se týče paměti pro digitální popis štítu, modul `cblazon` se stará pouze o paměť, kterou sám vytvořil – tj. dokáže alokovat a následně řízeně dealokovat struktury v něm definované. Je nutné mít na paměti, že parser nevytváří kopie dat, u kterých to není vysloveně nutné – je to z důvodu výkonnostních úspor, aby parser mohl být co možná nejrychlejší.

Důsledkem je, že napříč stromem představovaným strukturou **`struct cbl_blazon`** lze nalézt ukazatele na paměť, která byla poskytnuta jako vstupní data pro parser. V současné implementaci se jedná zejména o buffer obsahující kompletní vstupní text blazonu a o výstupní data analýzy knihovnou MorphoDiTa. Jedná se zejména o řetězcové buffery obsahující slova z textu, lemmata, tagy apod.

Při nakládání se strukturou **`struct cbl_blazon`** je tedy nutné ponechat vstupní data předaná parseru v paměti po celou dobu zpracování a neuvolňovat je, dokud není jisté, že se již s touto strukturou nebude dále pracovat. Správné nakládání s daty je naznačeno následujícím útržkem kódu:

```
char *blazon_text = load_text(...);
struct morpho_tagger *tagger = morpho_tagger_load(...);
struct morpho_tagged *tagged = morpho_tag(tagger,
        blazon_text, strlen(blazon_text));
struct cbl_blazon *blazon = NULL;
struct cbl_morpho_tokenizer *tokenizer = cbl_morpho_tokenizer_new(tagged);
cbl_parse((struct cbl_tokenizer *)tokenizer, NULL, NULL, &blazon);

/* Zde lze pracovat s daty, na která se odkazuje ukazatel `blazon`. Je přitom
 * NUTNÉ, aby v tuto dobu zůstala zachována i data, na která se odkazují
 * proměnné `blazon_text`, `tagger`, `tagged` a `tokenizer`, protože některé
 * struktury v `blazon` se na ně budou odkazovat! */

cbl_blazon_free(blazon);
cbl_morpho_tokenizer_free(tokenizer);
morpho_tagged_free(tagged);
morpho_tagger_free(tagger);
free(blazon_text);
```

Listing 5.10: Ilustrace práce s pamětí v knihovně CBlazon

5.9 Spustitelné programy

Sada CBlazon obsahuje celkem čtyři spustitelné programy – `cblbench`, `cblhtml`, `cbltag` a `cbltest`. Knihovna CBlazon (tj. moduly v adresáři `lib`) je závislostí všech těchto programů a je jimi využívána různými způsoby. Následující podsekce popisují implementaci a účel jednotlivých programů. Způsob jejich použití je pak popsán v uživatelské příručce v přílohách.

5.9.1 `cblbench`

Program `cblbench` je prvním z implementovaných programů sady CBlazon. Jeho původním účelem bylo testování výkonu knihovny MorphoDiTa.

Implementace programu je velmi jednoduchá. Program má definovánu konstantu `NUM_TRIES` o hodnotě 1000, která značí, kolikrát má ve smyčce proběhnout výkonnostní testování. Test probíhá nad všemi dostupnými testovacími řetězci z modulu `teststrings`.

Čas každé iterace je ukládán do globální matice `time_matrix` o šířce `NUM_TRIES` a výšce `CBL_TESTSTR_COUNT`, což je konstanta obsahující počet dostupných řetězců. Měří se zvlášť čas lemmatizace a čas zpracování syntaktickým analyzátorem. Po ukončení měření je vypočten aritmetický průměr časů pro každý řetězec a ten je poté vypsán na obrazovku společně s časem načtení modelu knihovny MorphoDiTa a celkového času průběhu testu.

5.9.2 `cblhtml`

Program `cblhtml` slouží k vygenerování reportu o zadaném blazonu ve formátu HTML. Report obsahuje vstupní text, vizualizaci jeho derivačního stromu v podobě vnořených bloků, vizualizaci dělení štítu včetně použitých barev a, pokud analyzátor narazil na syntaktickou chybu, popis chyby.

Vizualizátor derivačního stromu je rovněž opatřen syntaktickým zvýrazňováním – barevně označuje některé důležité prvky vstupního textu. Bloky derivačního stromu lze skrýt – poté je vidět pouze zvýrazněný text.

Vizualizace dělení štítu je vygenerována ve formátu *Scalable Vector Graphics*. Dělení je prováděno nad obdélníkem, jehož tvar je poté pomocí *masky* upraven, aby připomínal heraldický štít.

Pro účely generování reportu byl vytvořen velmi jednoduchý šablonovací engine. Ten sekvenčně čte vstupní soubor `template.html` a zapisuje jeho obsah do souboru výstupního, dokud není nalezen identifikátor ve formátu `#{název}`, tj. řetězec ohraničený složenými závorkami, před nimiž se nachází znak křížku. Jakmile je takovýto identifkátor nalezen, je vyhledán jeho název

v poli `placeholder_processors`. Pokud existuje, je zavolána názvu přiřazená funkce, které je jako parametr předán výstupní proud a zpracovaná data blazonu. Funkce pak může do proudu zapsat kód požadované vizualizace. Po návratu z funkce pokračuje čtení vstupního souboru, dokud není nalezen další identifkátor anebo není dosaženo jeho konce.

Příkladem identifikátoru může být `#{parse_tree}` – jeho název `parse_tree` je vyhledán v poli, kde je mu přiřazena funkce `parse_tree_cb()`. Ta po zavolání přečte vygenerovaný derivační strom a na jeho základě do výstupu zapíše hierarchii HTML tagů typu `<div>`, na něž je poté webovým prohlížečem aplikován kaskádový styl (CSS) pro lepší přehlednost vizualizce.

Několik příkladů výstupních reportů programu `cblhtml` lze nalézt v přílohách na straně v.

5.9.3 `cbltag`

Program `cbltag` je primárním nástrojem pro manuální testování funkčnosti knihovny CBlazon. Jedná se o program ovládaný z příkazového řádku, který může být spouštěn v několika různých řežimech podle toho, jaké informace o vstupním blazonu (či více blazonech) mají být zobrazeny.

Prvním příkazem je `tag`, který byl do programu přidán po první úspěšné integraci knihovny MorphoDiTa do sady CBlazon. Vypisuje po řádcích jednotlivá slova a interpunkční znaménka ze vstupního textu společně se seznamy variant lemmat rozpoznaných knihovnou MorphoDiTa, včetně jejich tagů a dodatečných dat, která knihovna poskytuje.

Velmi obdobným příkazem je `tokenize`, nicméně ten vypisuje tagy převedené na tokeny, které jsou připraveny na zpracování syntaktickým analyzátorem. Zejména se zde nacházejí názvy typů a podtypů tokenů, popř. také informaci, o jaké se jedná klíčové slovo. Najdeme zde také číselnou hodnotu slova, jedná-li se o číslovku.

Po první implementaci syntaktického analyzátoru byl přidán příkaz `tree`, který vypisuje strukturu derivačního stromu, který vyjadřuje, jak analyzátor sestupoval do jednotlivých funkcí neterminálních symbolů formální gramatiky. Výstup tohoto příkazu je jako jediný barevně zvýrazněný pomocí ANSI escape kódů[6], neboť bylo potřeba jeho výstup zpřehlednit pro usnadnění dalšího vývoje gramatiky. Zejména jsou zde zvýrazněny větve stromu, kde program nalezl syntaktickou chybu.

Posledním příkazem je `ast`, který vypisuje vygenerovaný digitální popis štítu. Využívá k tomu funkce se sufixem `_fprint()` z modulu `cblazon`,

[6]Netisknutelné posloupnosti znaků, které mění chování terminálového výstupu, například právě barev, ale i podtržení, pozici kurzoru apod.

které jsou podrobněji popsány v podsekci 5.2.1. Příklady takto vypsaných digitálních popisů lze vidět v přílohách na straně viii.

5.9.4 `cbltest`

Program `cbltest` je určen k jednotkovému testování sady CBlazon. Aktuální implementace obsahuje pouze tři testy, pomocí kterých byla ověřena správná funkčnost algoritmu pro očíslování sekcí štítu, jehož dělení je reprezentováno binárním stromem.

5.10 WebAssembly

Jedním z neformálních požadavků na práci bylo eventuální vyvinutí webové aplikace – z tohoto důvodu bylo při implementaci dbáno, aby byl napsaný kód vždy přeložitelný do *WebAssembly* (zkr. WASM).

WebAssembly [3] je binární formát instrukcí pro zásobníkový virtuální stroj. Primárně se využívá ve webových prohlížečích, které v minulosti za účelem tvorby interaktivních webových aplikací obsahovaly pouze běhové prostředí pro jazyk JavaScript. V současné době lze díky překladu do formátu WASM uvnitř webových prohlížečů spouštět i kód psaný v tradičních kompilovaných programovacích jazycích, jako jsou například C, C++, Rust, apod.

WASM nemůže přímo interagovat s webovým obsahem – k tomu je stále nutné napsat obalový kód v jazyce JavaScript, který tyto interakce zprostředkovává. Překlad do WASM je tak vhodný především pro tvorbu výkonných multiplatformních výpočetních jader, která lze ze stejného zdrojového kódu překládat jak do nativního strojového kódu procesorů, tak právě do WASM.

5.10.1 Emscripten

Emscripten je kompletní sada nástrojů pro vytváření programů běžících pod WebAssembly. K překladu ze zdrojového kódu využívá LLVM, díky čemuž podporuje širokou škálu programovacích jazyků, mezi které patří i jazyky C a C++ použité v této práci. Emscripten poskytuje implementace standardních knihoven jazyků C a C++ a je schopno emulovat virtuální souborový systém, čehož práce využívá pro načítání natrénovaného modelu taggeru MorphoDiTa a šablony pro program `cblhtml`.

Součástí Emscripten je i obalový skript *emcmake* umožňující snadný překlad projektu ve formátu sestavovacího systému CMake. Ten je rovněž využit v této diplomové práci.

5.11 Výstupy kompilace

O nativně kompilovaných programovacích jazycích na předních operačních systémech se dá obecně říci, že výstupy jejich linkování můžeme dělit na tři kategorie.

První kategorií je *statická knihovna funkcí* (soubor `.a` na Linuxu; `.lib` na Windows). Ta sama o sobě není spustitelná. Obsahuje pouze pojmenované funkce a svá globální data v takové podobě, aby mohla být později staticky slinkována s dalším programem. Statické linkování znamená, že výsledný program bude obsahovat kompletní kopii této statické knihovny.

Druhou kategorií je *dynamická knihovna funkcí* (soubor `.so` na Linuxu; `.dll` na Windows). Podobně jako knihovna statická není přímo spustitelná a obsahuje pouze pojmenované funkce a globální data. S dalšími programy se však linkuje dynamicky, což znamená, že výsledný program neobsahuje její kopii, ale je načtena při spuštění nebo za běhu programu dynamickým linkerem. Dynamické linkování umožňuje například nahrazení knihovny její odlišnou verzí, pakliže mají obě verze vzájemně kompatibilní binární rozhraní (ABI). Na operačních systémech odvozených od GNU/Linux je dynamické linkování hojně využíváno pro širokou řadu programů, které tak spolu sdílejí kopie knihoven nainstalované v daném systému.

Knihovny (statické i dynamické) obsahují pozičně nezávislý strojový kód (v angličtině *position-independent code*; zkr. *PIC*). To je strojový kód v němž jsou data a funkce adresovány relativně, tedy není předem pevně určeno, na jakých adresách se globální data a funkce nacházejí. U statických knihoven jsou adresy převedeny na absolutní v době linkování s dalším programem. U dynamických knihoven k převodu dochází při jejich načtení.

Poslední kategorií je *spustitelný program* (na Linuxu typicky bez přípony; na Windows soubor `.exe`). Ten obsahuje pevně určený vstupní bod, ve kterém začíná provádění programu (v jazyce C funkce `main()`).

Sada CBlazon kombinuje všechny tři tyto kategorie výstupů. Jakým způsobem je to provedeno, je popsáno v následující podsekci.

5.11.1 Členění CMake projektu

Sada CBlazon je sestavována pomocí systému CMake. K popisu projektu se v tomto systému používá soubor `CMakeLists.txt`, který je součástí přiložených zdrojových souborů. Je psán ve specializovaném skriptovacím jazyce, pomocí kterého lze definovat a konfigurovat jednotlivé výstupy, v terminologii systému CMake zvané *targets*. Těmi jsou výše popisované spustitelné programy a sady funkcionalit v podobě dynamických či statických knihoven.

Jako statické jsou v sadě překládány externí knihovny *utf8proc* a *MorphoDiTa*.

Knihovna *utf8proc* přímo poskytuje vlastní soubor `CMakeLists.txt`, který je v projektu sady CBlazon importován pomocí příkazu `add_subdirectory`, čímž dochází k sestavování knihovny společně se zbytkem sady.

Oproti tomu MorphoDiTa je původně sestavována pomocí nástroje *Make*. Ten byl původně během vývoje sady používán, avšak tento přístup se ukázal jako nepříliš spolehlivý. V některých případech nedocházelo k automatickému znovusestavování knihovny a API systému CMake neposkytovalo nástroje k napravení těchto nedostatků. Proto byl nakonec zvolen přístup vlastní definice knihovny přímo v systému CMake pomocí příkazu `add_library`.

Knihovna sady CBlazon je definována jako dynamická. Je sdílena mezi jednotlivými spustitelnými programy sady pro účely úspory místa na disku. Knihovna se sestává z modulů obsažených v adresáři `lib` a ze staticky přilinkovaných knihoven *utf8proc* a *MorphoDiTa*. Při použití hlavičkových souborů z adresáře `lib` je umožněno použití tohoto sdíleného objektu i v externích programech, které nejsou součástí sady.

Odděleným nepovinným výstupem je modul `teststrings`. Ten obsahuje referenční řetězce, které byly použity k vývoji a testování sady CBlazon. Rovněž obsahuje funkce pro nakládání s těmito řetězci. V projektu je definován jako oddělená dynamická knihovna.

Jako spustitelné soubory jsou v projektu definovány programy `cblbench`, `cblhtml`, `cbltag` a `cbltest`. Každý z programů má svůj hlavní zdrojový `.c` soubor v kořenovém adresáři sady CBlazon a váže se na sdílené objekty popisované výše.

5.11.2 Assety

K programům ze sady CBlazon jsou rovněž přiloženy externí datové soubory, zvané *assety*. Patří sem soubor obsahující natrénovaná data taggeru knihovny MorphoDiTa; a soubor `data/template.html`, který slouží jako šablona programu `cblhtml` k vygenerování reportu ve formátu HTML.

Pro zajištění konzistence jsou soubory při sestavení připraveny pro použití překládanými programy. K tomu slouží v `CMakeLists.txt` definovaná funkce `copy_file()`. Ta přijímá jako parametr relativní cestu k souboru – cesta je zároveň relativní k adresáři se zdrojovými soubory a k adresáři výstupnímu. V případě běžného překladu sady – tj. pro GNU/Linux – se soubor jednoduše překopíruje do výstupního adresáře. Pokud je sada překládána pro

WebAssembly pomocí Emscripten, soubor je přidán do virtuálního souborového systému pomocí příznaku `--preload-file` překladače.

Programy k přiloženým souborům přistupují pomocí funkce `program_file()` v modulu `utils`. Pod operačním systémem GNU/Linux tato funkce získá cestu k souboru relativně k cestě, na které se nachází spustitelný soubor s aktuálně běžícím programem, a to bez ohledu na aktuální pracovní adresář (tj. zpravidla adresář, ze kterého byl program spuštěn pomocí příkazového řádku). Ve WebAssembly funkce jednoduše vrací absolutní cestu k souboru ve virtálním souborovém systému.

5.12 Získání zdrojových souborů

Zdrojové soubory sady CBlazon byly odevzdány jako součást příloh diplomové práce. Jsou rovněž k dispozici na GitLabu Katedry informatiky a výpočetní techniky (KIV) FAV ZČU a na GitLab SaaS (`gitlab.com`). V době psaní práce je GitLab SaaS autoritativním úložištěm pro zdrojový kód sady CBlazon – KIV GitLab je nastaven jako její zrcadlo.

V Git úložišti je použito rozšíření *Large File Storage* (LFS) pro uložení jazykových modelů knihovny MorphoDiTa v rámci prevence rapidního nárůstu velikosti úložiště v případě, že se tyto soubory budou měnit. To by totiž znamenalo, že uživatel bude mít lokálně stažené veškeré verze těchto souborů, i když bude skutečně používat pouze jednu z nich. Git LFS namísto toho velké soubory stahuje až ve chvíli, kdy je načítán daný commit.

Git LFS je oddělené rozšíření pro Git a běžně nebývá součástí jeho distribuce. Na většině systémů je tedy nutné jej mít explicitně nainstalované.

Další specialitou je použití Git submodulů pro získání zdrojových souborů použitých knihoven – MorphoDiTa a utf8proc. Pro stažení Git úložiště včetně submodulů lze použít následující příkaz:

```
$ git clone <git_url> --recurse-submodules
```

V případě, že je úložiště již staženo bez submodulů, lze tyto dodatečně stáhnout následujícím příkazem:

```
$ git submodule update --init --recursive
```

Adresy stránek úložišť se sadou CBlazon jsou následující:

- SaaS: https://gitlab.com/Spiffyk/cblazon
- KIV: https://gitlab.kiv.zcu.cz/ostava/cblazon

5.13 Sestavení sady

Sada CBlazon je implementována primárně v jazyce C podle normy C11 s rozšířeními od GNU. Některé části vyžadují rovněž překladač jazyka C++ podle normy C++20. Doporučenými a aktivně testovanými překladači jsou GNU GCC či Clang. Kvůli použití GNU rozšíření není možné použít například překladač MSVC, který tato rozšíření nepodporuje.

Zdrojové soubory jsou vytvořeny pro překlad pro systém GNU/Linux či pro běhové prostředí WebAssembly (resp. Emscripten). Jiná běhová prostředí (např. OS Microsoft Windows) nebyla testována a sada pro ně nemusí být v současné podobě přeložitelná. Zároveň sada ale není na běhové prostředí aktivně nijak vázána a zajištění přeložitelnosti pod novými prostředími by nemělo být zásadně problematické.

Pro překlad jsou zapotřebí následující prerekvizity:

- Překladače jazyků C a C++
  - GNU GCC (`gcc` a `g++`; testováno na verzi 12.2.1)
  - Clang (`clang` a `clang++`; testováno na verzi 15.0.7)
- Sestavovací systém CMake (alespoň verze 3.0; testováno na 3.26.3)
- Emscripten (pouze pro překlad do WebAssembly)

Mimo tyto nástroje není nutné mít nainstalované žádné další závislosti. Ty závislosti, které sada využívá, jsou obsaženy v archivu přiloženému k diplomové práci, popř. jsou přidány jako submoduly v Git úložišti projektu.

Projekt je sestavitelný na operačním systému GNU/Linux následujícími příkazy spuštenými v adresáři se zdrojovými soubory:

```
$ mkdir build && cd build
$ cmake ..
$ cmake --build .
```

Tímto dojde k sestavení knihovny CBlazon a všech přidružených knihoven a programů pro GNU/Linux. Způsob použití programů ze sady CBlazon je popsán v uživatelské příručce v přílohách.

Pro sestavení pro WebAssembly (vyžadováno Emscripten, vizte výše) lze použít následující příkazy:

```
$ mkdir build && cd build
$ emcmake cmake ..
$ cmake --build .
```

Systém CMake defaultně sestavuje programy bez překladových optimalizací a se zapnutými ladicími symboly. Pro zapnutí optimalizací a odstranění ladicích symbolů lze za první příkaz `cmake` (resp. `emcmake`) přidat řetězec `-DCMAKE_BUILD_TYPE=Release`.

6 Testování sady CBlazon

Knihovna funkcí pro analýzu blazonů, která je součástí sady CBlazon, byla testována převážně manuálně pomocí implementovaných programů, zejména pak pomocí programu `cbltag`. Manuální testování bylo stěžejním nástrojem pro vývoj knihovny, neboť během něj poukazovalo na nedostatky vznikající gramatiky v různých oblastech.

Pro vývoj a ladění gramatiky se jako velmi užitečná ukázala vizualizace derivačního stromu – zprvu textová, generovaná příkazem `tree` programu `cbltag`, později i grafická v programu `cblhtml`. Ta vždy velmi přesně ukázala, v jaké části gramatiky došlo k chybnému rozpoznání aktuálně zpracovávaného tokenu.

Výpis derivačního stromu je samozřejmě vhodný spíše pro sestavování gramatiky a není příliš užitečný pro uživatele překladače – pro ty je lepší vytvořit na vhodných místech specifická chybová hlášení, nejlépe obsahující rady, jak danou chybu opravit. Nicméně právě pro vytvoření uživatelsky přívětivých hlášení je nutné nejprve provést analýzu možných chyb například právě pomocí výpisu derivačního stromu k nalezení takových vhodných míst a vytvoření vhodných návodných kroků. Na přidávání takových hlášení je knihovna připravena (vizte popis makra `CTX_ERROR()` v sekci 5.4.2), byť jich je v současné verzi implementováno pouze málo.

6.1 Testovací data

Jako testovací data obsahuje knihovna CBlazon sadu vstupních blazonů převzatých a adaptovaných zejména z desáté kapitoly Houzarova klíče [9]. Seznam textů kategorizovaných podle zdrojů je následující:

- **Klíč ke znakům měst České republiky [9]:** Bělá nad Radbuzou, Jičín, Letohrad, Letovice, Mariánské Lázně, Meziměstí, Miroslav, Nový Bydžov, Oloví, Plzeň, Poděbrady, Přerov, Příbram, Přimda, Rousínov, Ústí nad Labem, Vamberk, Vrchlabí, Vysoké Mýto
- **Heraldry of the world [1]**: Podůlší
- **Upravené ze zdrojů výše:** Mariánské Lázně (adapt.), Poděbrady (první část)
- **Původní:** Řetězec k testování dělení

Ne všechny tyto blazony dokáže současná implementace korektně zpracovat, nicméně představují sadu dat, u kterých bylo autorem vyhodnoceno, že jsou zajímavá pro další rozšižování gramatiky.

Mezi funkční blazony, tedy ty, které implementovaný syntaktický analyzátor zpracuje bez chyb a vygeneruje digitální popis, patří následující:

- Letohrad, Letovice, Mariánské Lázně (adapt.), Nový Bydžov, Poděbrady (první část), Podůlší, Rousínov, Vamberk, Řetězec k testování dělení.

Tento seznam je součástí modulu `teststrings` knihovny CBlazon a lze jej využít například k hromadnému spuštění zpracování nad jeho členy, což podporuje program `cbltag`.

Programy `cbltag` a `cblhtml` rovněž přijímají vstupní texty mimo popisovanou testovací množinu. Lze tak učinit čtením ze standardního vstupu – postup, jak toho docílit, je popsán v uživatelské příručce sady v přílohách práce.

6.2 Výsledky testování

V současné době překladač podporuje pouze velmi základní texty blazonů. Ukazuje se, že jazyk blazonů dokáže být komplikovaný a vytvoření formální gramatiky, která jej popíše z větší než malé části, je zdlouhavým procesem s časovou náročností přesahující rozsah diplomové práce. Nicméně, jak je zřejmé ze seznamu již funkčních vstupních textů, přinejmenším část takovéto gramatiky při vhodném použití lemmatizačních nástrojů sestavit lze.

Z vizualizací programu `cblhtml` (příklady rovněž v přílohách) je zřejmé, že základní dělení štítů a barvy jejich jednotlivých sekcí díky implementaci algoritmu pro jejich očíslování (vizte sekci 5.5) odpovídají popisům a předepsaným pravidlům.

Syntaktický analyzátor byl otestován na *pokrytí kódu* – pomocí automatického nástroje GCov, který je součástí překladače GCC, bylo zjištěno, jaké procento řádků zdrojového kódu je skutečně provedeno. Testován byl průběh příkazu `ast` programu `cbltag` nade všemi testovacími řetězci. Pro soubor `parser.c` nástroj hlásí pokrytí ze 74.89 %. Jistou část nepokrytého kódu zde tvoří utilitní funkce, které nemají vliv přímo na analýzu textu, nicméně nacházejí se zde i neterminální symboly s malým či žádným pokrytím jako `FieldAttrOptWith` (0.0 %), `FieldAttrsBeg` (42.86 %), `FieldDesignation` (41.67 %) apod. To naznačuje, že by bylo v budoucnu vhodné přidat ještě další testovací případy, které by tyto části kódu pokryly.

6.2.1 Omezení analyzátoru a datového modelu

V současné gramatice a datovém modelu byla identifikována některá omezení, která mohou být napravena navazujícími pracemi, má-li být sada CBlazon schopna zpracovávat širší množinu blazonů. Identifikovaná omezení lze rozdělit do několika kategorií. Následuje jejich seznam doprovázené názvy testovacích blazonů, u kterých se daná omezení vyskytují:

- **Chybějící podpora pro vnořené štíty:** Plzeň, Vrchlabí
- **Chybějící podpora pro dodatečné modifikující věty pro figury:** Vysoké Mýto
- **Chybějící podpora typu předložky (např. „o“) pro atribut nacházející se za názvem figury:** Bělá nad Radbuzou
- **Chybějící podpora pro atributy násobně aplikované na více figur:** Jičín, Mariánské Lázně, Přerov
- **Chybějící podpora pro násobení figur pomocí předložek (např. „z každé strany medvěd“):** Přimda
- **Chybějící podpora pro dodatečné informace v závorkách:** Jičín
- **Chybějící podpora pro modifikaci umístění atributu figury (např. „v náručí“):** Mariánské Lázně, Oloví, Vysoké Mýto
- **Nesprávná asociace atributů mezi sebou (např. střecha se přisoudí celé budově namísto její věže):** Mariánské Lázně, Mariánské Lázně (adapt.)
- **Chybějící podpora pro hlavy a paty štítů:** Meziměstí
- **Chybějící podpora pro modifikaci umístění figur (např. „vedle sebe“):** Meziměstí, Poděbrady
- **Chybějící podpora pro popis dělení štítu samostatně na začátku první věty blazonu:** Plzeň
- **Chybějící podpora pro alianci štítů:** Příbram
- **Chybějící podpora pro štítonoše a helmy:** Plzeň
- **Chybějící podpora pro upřesňující popis figury oddělený pomlčkou:** Ústí nad Labem

- **Přídavná jména se neaplikují na více figur:** Vamberk
- **Nekompletní podpora pro příslovce:** Poděbrady, Poděbrady (první část)
- **Nevyhovující lemmatizace:** Miroslav, Poděbrady, Vysoké Mýto

V případě blazonu znaku Vamberka a adaptované verze blazonu znaku Mariánských Lázní se jedná o sémantickou nepřesnost výsledného digitálního podpisu – gramatika tyto blazony přijímá korektně, ale výsledná data nejsou zcela smysluplná.

Například u znaku Vamberka blazon popisuje, že jednorožec je *se zlatou zbrojí a jazykem* (tj. přídavné jméno *zlatý* se vztahuje jak na *zbroj*, tak na *jazyk*), ale ve vygenerovaném digitálním popisu je zlatá pouze zbroj. Řešením by zde mohlo být dodání fáze post-processingu výsledné datové struktury, která by takovéto nedostatky heuristicky opravovala. Heuristika pro opravy by musela vycházet z podrobnější analýzy českého jazyka – u některých přídavných jmen totiž nemusí nutně dávat smysl, aby byla aplikována na více podstatných jmen, která je následují.

Soubor `grammar-todos.bnf` v archivu se zdrojovými soubory sady CBlazon naznačuje, kudy dále mohou vést cesty k rozšíření implementované gramatiky. Nejedná se však o jediný způsob – je stále nutné provést důkladnou dodatečnou analýzu vstupních textů a aktuálního chování parseru při jejich zpracování.

Závěr

Cílem této diplomové práce bylo položit základ softwarového překladače *blazonů* – formálních slovních popisů heraldických znaků a štítů. První kapitola práce uvádí do problematiky blazonů z pohledu heraldiky.

Pro syntaktickou analýzu blazonů byla experimentálně sestavena formální gramatika popsaná ve druhé kapitole práce. Gramatika v žádném případě nepopisuje kompletní jazyk blazonů. Ten je totiž odvozen od jazyka přirozeného, kvůli čemuž je nesmírně rozsáhlý. Sestavení gramatiky, která by jej pokryla z valné části, by vyžadovalo další výzkum, který by pravděpodobně přesáhl rozsah diplomové práce.

Nejrozsáhlejší kapitolou práce je kapitola pátá, která detailně popisuje, jak byla implementována sada CBlazon. Ta obsahuje knihovnu funkcí pro strojový překlad blazonů a doprovodné programy, které tuto knihovnu používají. Sada byla pro zajištění co nejlepší možné výkonnosti a přenositelnosti implementována v programovacím jazyce C. Analyzér, který je součástí knihovny, je schopný přečíst text vstupního blazonu a vygenerovat z něj strukturovaný datový popis heraldického znamení. Ten může být dále využit k dalšímu zpracování, zejména pak k vygenerování grafické reprezentace znamení. Pátá kapitola může být zejména užitečná pro autory navazujících prací – je vodítkem k orientaci ve zdrojovém kódu sady a vysvětluje mnoho technik specifických pro jazyk C, které byly při psaní programu použity. Je zde rovněž popsán datový model použitý pro strukturovaný popis štítu, který překladač generuje.

Práce využívá softwarovou knihovnu MorphoDiTa pro lemmatizaci slov – bylo totiž nutné vyřešit problém skloňování a časování v českém jazyce, které by značně znesnadňovalo rozpoznávání klíčových slov a slovních druhů uvnitř textů. Popis funkcionalit knihovny MorphoDiTa a její porovnání s dalšími lemmatizéry se nachází ve třetí kapitole této práce.

Nutno podotknout, že sada CBlazon zahrnuje generátor pouze pro velmi základní grafickou reprezentaci heraldického znamení – vizualizuje dělení popisovaného štítu včetně barev jednotlivých jeho sekcí. Kompletní vizualizace by vyžadovala další rozsáhlejší shromažďování podkladů a dodatečnou implementaci zejména v oblasti databázového softwaru, který by poskytoval obrázky figur vkládaných do štítů. Rovněž by bylo potřeba vyřešit pozicování a modifikace figur na základě jejich popisu. Taková rozšíření jsou vhodnou oblastí k zaměření navazujících prací. Základní popis formátu SVG, ve kte-

rém je dosavadní vizualizace generována a nástin fungování databáze figur se nachází ve čtvrté kapitole práce.

V šesté a poslední kapitole jsme si nastínili, jak byly funkcionality implementované knihovny testovány a jak toto testování ovlivnilo vývoj formální gramatiky. Kapitola rovněž poukazuje na aktuální nedostatky navrženého řešení a naznačuje směr, jakým by se jeho vývoj mohl dále ubírat, vzniknou-li další navazující práce zabývající se právě touto oblastí.

Dle autorova mínění tato práce pokládá dobrý základ pro algoritmické zpracovávání českých blazonů. Vzniknuvší knihovna sice v současné chvíli nepodporuje velké množství existujících blazonů, nicméně se jedná spíše o problém nedostatku času k vytvoření skutečně rozsáhlé gramatiky, než o zásadní nedostatky v samotném návrhu řešení. Gramatika i datový model jsou snadno rozšiřitelné, čímž může být pokrytí zpracovatelných blazonů značně vylepšeno.

Literatura

[1] *Podůlší (Erb - znak - Coat of arms - crest) – Heraldry of the world* [online]. Dostupné z: https://www.heraldry-wiki.com/heraldrywiki/index.php?title=Pod%C5%AFl%C5%A1%C3%AD.

[2] Scalable Vector Graphics (SVG) 1.1 (Second Edition). Standard, World Wide Web Consortium, Cambridge, Massachusetts, US, August 2011. Dostupné z: https://www.w3.org/TR/SVG11/.

[3] *Overview — WebAssembly 2.0* [online]. WebAssembly Community Group, 2022. Dostupné z: https://webassembly.github.io/spec/core/intro/overview.html.

[4] Boutell, C. *Heraldry, historical and popular*. Bentley, 1864.

[5] Buben, M. *Encyklopedie heraldiky*. Libri, 2003. ISBN 80-7277-135-3.

[6] Chomsky, N. On certain formal properties of grammars. *Information and Control*. 1959, 2, 2, s. 137–167. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(59)90362-6. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0019995859903626.

[7] Deveria, A. *SVG (Basic support) – Can I use... Support tables for HTML5, CSS3, etc* [online]. 2023. Dostupné z: https://caniuse.com/svg.

[8] Gruber, J. *Markdown* [online]. 2004. Dostupné z: https://daringfireball.net/projects/markdown/.

[9] Houzar, P. *Klíč ke znakům měst České republiky*. Moravská zemská knihovna, 2016. ISBN 978-80-7051-221-0.

[10] ISO 14977:1996(E). Information technology — Syntactic metalanguage — Extended BNF. Standard, International Organization for Standardization, Geneva, CH, December 1996. Dostupné z: https://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf.

[11] Jinks, P. *BNF/EBNF variants* [online]. The University of Manchester, Department of Computer Science, March 2004. Dostupné z: http://www.cs.man.ac.uk/~pjj/bnf/ebnf.html.

[12] JuliaStrings. *utf8proc* [online]. 2022. Dostupné z: https://juliastrings.github.io/utf8proc/.

[13] Liška, K. – Mucha, L. *Klíč k našim městům.* Práce, vydavatelství a nakladatelství ROH, 1979.

[14] Mikulová, M. et al. Manual for Morphological Annotation. Revision for Prague Dependency Treebank – Consolidated. Technical report, 2020. Dostupné z: https://ufal.mff.cuni.cz/techrep/tr64.pdf.

[15] Muratori, C. *Handmade Hero Day 341 – Dynamically Growing Arenas* [online]. 2016. Dostupné z: https://guide.handmadehero.org/code/day341/#1684.

[16] Sedláček, R. Morfologický analyzátor češtiny, 1999. Dostupné z: https://nlp.fi.muni.cz/projekty/ajka/ajka.pdf.

[17] Straková, J. – Straka, M. – Hajič, J. Open-Source Tools for Morphology, Lemmatization, POS Tagging and Named Entity Recognition. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, s. 13–18, Baltimore, Maryland, June 2014. Association for Computational Linguistics. Dostupné z: http://www.aclweb.org/anthology/P/P14/P14-5003.pdf.

[18] Torvalds, L. *Linux kernel coding style* [online]. Dostupné z: https://docs.kernel.org/process/coding-style.html.

[19] Heesch, D. *Doxygen* [online]. 2022. Dostupné z: https://www.doxygen.nl/.

[20] Votrubec, J. Morphological Tagging Based on Averaged Perceptron. In *WDS'06 Proceedings of Contributed Papers, Part I*, s. 191–195, Praha, 2006. MATFYZPRESS. Dostupné z: https://www.mff.cuni.cz/veda/konference/wds/proc/pdf06/WDS06_134_i3_Votrubec.pdf. ISBN 80-86732-84-3.

[21] Šmerk, P. Fast Morphological Analysis of Czech. In *Proceedings of Third Workshop on Recent Advances in Slavonic Natural Language Processing, RASLAN 2009*, s. 13–16, Brno, 2007. Masaryk University. Dostupné z: https://nlp.fi.muni.cz/raslan/2009/papers/13.pdf. ISBN 978-80-210-5048-8.

Přílohy

Uživatelská příručka sady CBlazon

V rámci sady CBlazon byly implementovány celkem tři spustitelné programy: `cblbench`, `cbltag` a `cblhtml`. Všechny programy jsou určeny k obsluze pomocí příkazového řádku a nedisponují žádným grafickým uživatelským rozhraním.

Knihovna CBlazon obsahuje množinu testovacích řetězců, které jsou určeny ke zpracování jejím parserem. V množině se nachází směs řetězců, které je parser schopný bezchybně zpracovat (pro ně je definována podmnožina), i těch, které aktuální verze parseru korektně zpracovat neumí. Jedná se převážně o skutečné blazony znaků českých měst. K dispozici je i několik upravených blazonů – jedná se o deriváty skutečných blazonů, ale jsou v nich provedeny takové změny, aby je aktuální implementace parseru byla schopna zpracovat a vygenerovat z nich platný digitální popis štítu.

Množina řetězců je přístupná všem třem implementovaným programům. Programy `cbltag` a `cblhtml` kromě zpracování předdefinovaných blazonů umožňují také čtení textů ze standardního vstupu.

Následující sekce popisují funkce jednotlivých programů a návody k jejich použití.

`cblbench`

Program `cblbench` slouží k zátěžovému testování – původně byl implementován a využit ke zkoumání výkonnosti knihovny MorphoDiTa před jejím použitím k implementaci práce. V současné době je ale doplněn i o měření času zpracování textu parserem.

Program nepřijímá žádné parametry. Po spuštění načte model knihovny MorphoDiTa a provede několikanásobně zpracování nad předdefinovanou množinou vstupních blazonů. Program měří časy jednotlivých úkonů zvlášť pro každý vstupní text i celkové časy zpracování. Tyto časy vypisuje do standardního výstupu. Příklad výpisu programu `cblbench` lze vidět na listingu 6.1.

```
$ ./cblbench
Loading tagger file...
Tagging...
Done.

Total results:
         Tagger load:   3806 ms
  Total tagging time: 219963 ms

String averages:
  BELA_NAD_RADBUZOU: tag:  14.967 ms | parse: 0.098 ms
              JICIN: tag:   6.865 ms | parse: 0.090 ms
           LETOHRAD: tag:   0.905 ms | parse: 0.115 ms
           LETOVICE: tag:   1.679 ms | parse: 0.079 ms
    MARIANSKE_LAZNE: tag:  18.515 ms | parse: 0.026 ms
          MEZIMESTI: tag:   2.350 ms | parse: 0.023 ms
           MIROSLAV: tag:   2.570 ms | parse: 0.022 ms
        NOVY_BYDZOV: tag:   0.492 ms | parse: 0.087 ms
              OLOVI: tag:   8.522 ms | parse: 0.047 ms
              PLZEN: tag:  68.818 ms | parse: 0.037 ms
          PODEBRADY: tag:   8.802 ms | parse: 0.312 ms
       PODEBRADY_FP: tag:   2.015 ms | parse: 0.272 ms
            PODULSI: tag:   2.581 ms | parse: 0.186 ms
             PREROV: tag:   8.062 ms | parse: 0.095 ms
            PRIBRAM: tag:  39.267 ms | parse: 0.019 ms
             PRIMDA: tag:   4.729 ms | parse: 0.083 ms
           ROUSINOV: tag:   2.069 ms | parse: 0.120 ms
     USTI_NAD_LABEM: tag:   3.425 ms | parse: 0.215 ms
            VAMBERK: tag:   2.518 ms | parse: 0.204 ms
           VRCHLABI: tag:   6.452 ms | parse: 0.099 ms
        VYSOKE_MYTO: tag:  11.967 ms | parse: 0.053 ms
```

Listing 6.1: Výstup programu `cblbench`

cbltag

Program `cbltag` slouží k testování jednotlivých částí knihovny CBlazon. Umožňuje vypisování předdefinovaných testovacích blazonů; jejich slov otagovaných knihovnou MorphoDiTa; tokenů generovaných lexikální analýzou; derivačního stromu vstupního textu; a abstraktního syntaktického stromu, resp. strukturovaných dat popisujících rozpoznaný štít.

Formát argumentů pro spuštění programu je následující:

```
$ ./cbltag <příkaz> [název_textu]
```

Argument `příkaz` je vždy povinný. Určuje, jakou operaci má program provést. Platné operace jsou následující:

- `print` – pouze vypíše daný vstupní text;
- `tag` – zpracuje vstupní text taggerem knihovny MorphoDiTa a vypíše informace o získaných otagovaných slovech;
- `tokenize` – rovněž zpracuje text taggerem a navíc jednotlivá slova převede na symboly (tokeny) přijímané parserem
- `tree` – zpracuje vstupní text a vygeneruje a vypíše jeho derivační strom;
- `ast` – zpracuje vstupní text a vygeneruje a vypíše jeho abstraktní syntaktický strom, resp. strukturovaný popis rozpoznaného štítu;
- `teststrings` – vypíše názvy dostupných testovacích textů;
- `help` – vypíše popisy jednotlivých dostupných příkazů.

Pro všechny příkazy kromě `teststrings` a `help` je rovněž povinný i argument `název_textu`. Ten může být název předdefinovaného textu (zjistitelný pomocí příkazu `teststrings`) nebo může nabývat následujících hodnot:

- `all` – provede zpracování nad všemi předdefinovanými texty;
- `good` – provede zpracování nad texty, které program umí bezchybně zpracovat, a které tak byly ručně označeny;
- `stdin` – provede zpracování nad textem ze standardního vstupu (tj. z konzole nebo z přesměrovaného souboru).

Argument `název_textu` nerozlišuje mezi malými a velkými písmeny.

cblhtml

Program `cblhtml` je obdobou programu `cbltag`, který však o jednom zvoleném vstupním blazonu vygeneruje report ve formátu HTML do zadaného souboru.

Formát argumentů pro spuštění programu je následující:

```
$ ./cblhtml <název_textu> [název_výstupu]
```

Argument `název_textu` je povinný a může jím být název předdefinovaného textu (zjistitelný pomocí příkazu `./cbltag teststrings` z předchozí sekce), nebo `stdin` pro zpracování textu ze standardního vstupu (tj. z konzole nebo z přesměrovaného souboru). Oproti programu `cbltag` zde není podporováno zpracování více textů najednou použitím argumentů `all` či `good`.

Nepovinný argument `název_výstupu` umožňuje uživateli určit cestu k výstupnímu HTML souboru. Není-li přítomen, je použit název vstupního textu s přidanou příponou `.html` (např. `PRIBRAM.html`), který je umístěn do aktuálního pracovního adresáře, ze kterého je program spuštěn.

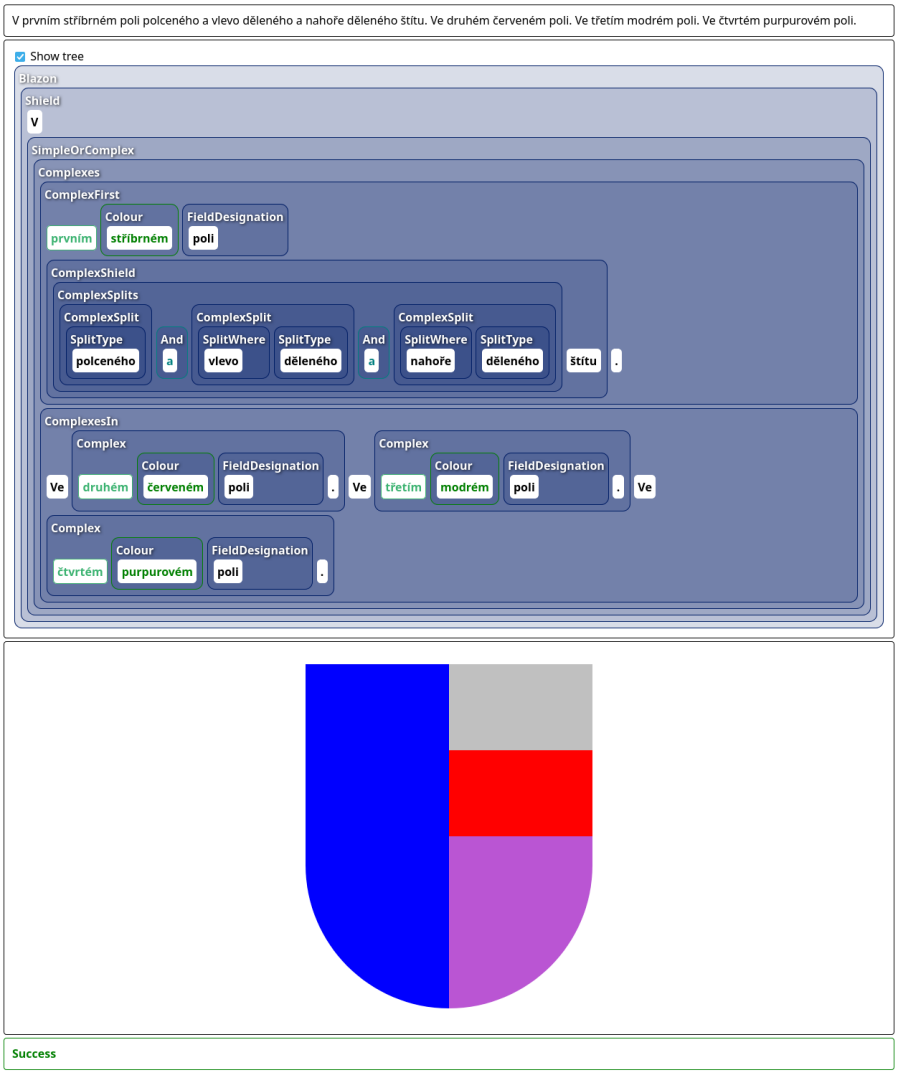
cbltest

Program `cbltest` slouží k jednotkovému testování. V současnosti obsahuje pouze tři testy, kterými byla ověřována korektnost algoritmu pro očíslování sekcí štítu popsaného v sekci 5.5.

Program vypisuje a ověřuje jednotlivé testovací případy a signalizuje jejich úspěšnost či neúspěšnost. Nepřijímá žádné parametry.

Výstupy programu `cblhtml`

SPLIT_TEST_1



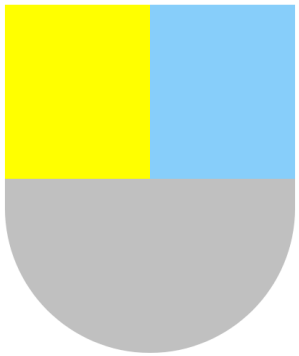
Vizualizace dělení štítu a derivačního stromu uměle vytvořeného testovacího blazonu

MARIANSKE_LAZNE_ADAPT

V prvním zlatém poli děleného a nahoře polceného štítu čelně vyrůstající Panna Marie v červeném rouše s modrým pláštěm a stříbrným závojem. Ve druhém poli stříbrný kulatý pavilon se čtyřmi černými románskými okny a červenou kuželovou vydutou střechou a zlatým patriarším křížem. Ve třetím stříbrném poli s trávníkem stříbrná fontána.

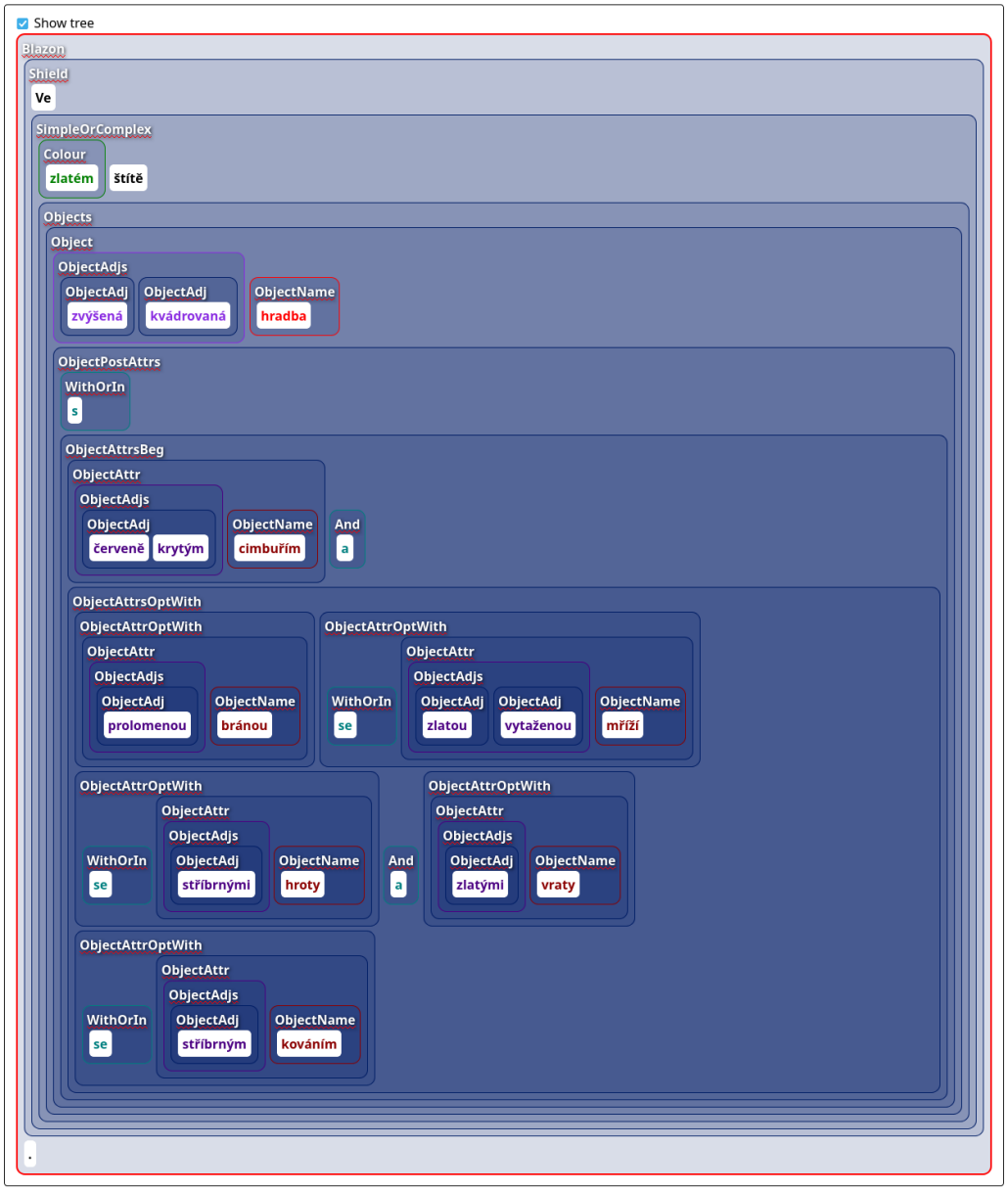
☐ Show tree

V prvním zlatém poli děleného a nahoře polceného štítu čelně vyrůstající Panna Marie v červeném rouše s modrým pláštěm a stříbrným závojem . Ve druhém poli stříbrný kulatý pavilon se čtyřmi černými románskými okny a červenou kuželovou vydutou střechou a zlatým patriarším křížem . Ve třetím stříbrném poli s trávníkem stříbrná fontána .



Success

Vizualizace dělení štítu města Mariánské Lázně



Vizualizace derivačního stromu nepodporovaného blazonu štítu města Poděbrady (obsahuje zvýraznění větve stromu, kde došlo k chybě)

Abstraktní syntaktické stromy

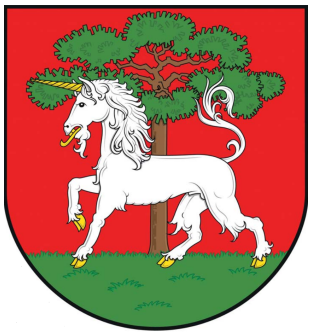
```
cbl_blazon {
  shield = cbl_obj_shield {
    split = cbl_split {
      style = (null)
      where = (null)
      colours = [ "červený" ]
      type = UNSPLIT
    }
    fields = [
      cbl_field {
        objects = [
          cbl_obj_figure {
            name = [ "lev" ]
            tag_attrs = [ "český" ]
            obj_attrs = [ (empty) ]
            natural_colours = false
          },
        ]
      },
    ]
  }
  bearer = (null)
  helmet = (null)
  valid = true
}
```

V červeném štítě český lev. (Houzar, kapitola 10 [9])

Znak Nového Bydžova[1], jeho strukturovaný popis a blazon

[1] Zdroj: https://commons.wikimedia.org/wiki/File:Coat_of_arms_of_Nov%C3%BD_Byd%C5%BEov.png

```
cbl_blazon {
  shield = cbl_obj_shield {
    split = cbl_split {
      colours = [ "červený" ]
      type = UNSPLIT
    }
    fields = [
      cbl_field {
        objects = [
          cbl_obj_figure {
            name = [ "trávník" ]
            natural_colours = false
          },
          cbl_obj_figure {
            name = [ "borovice" ]
            tag_attrs = [ "vyrůstající" ]
            natural_colours = true
          },
          cbl_obj_figure {
            name = [ "jednorožec" ]
            tag_attrs = [ "stříbrný", "kráčející" ]
            obj_attrs = [
              cbl_obj_figure {
                name = [ "zbroj" ]
                tag_attrs = [ "zlatý" ]
                natural_colours = false
              },
              cbl_obj_figure {
                name = [ "jazyk" ]
                natural_colours = false
              },
            ]
            natural_colours = false
          },
        ]
      },
    ]
  }
  valid = true
}
```



V červeném štítě s trávníkem vyrůstající borovice přirozených barev přeložená stříbrným kráčejícím jednorožcem se zlatou zbrojí a jazykem. (Houzar, kapitola 10 [9])

Znak Vamberku[2], jeho strukturovaný popis[3] a blazon

[2]Zdroj: http://vamberk.cz/wp-content/uploads/2016/01/znak.png

[3]Prázdné a `NULL` elementy jsou vynechány pro zkrácení výpisu