VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

INTERAKTIVNÍ GENERÁTOR SYNTAXE HETEROGENNÍCH DATOVÝCH STRUKTUR

INTERACTIVE GENERATOR OF SYNTAX OF HETEROGENEOUS DATA STRUCTURES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MARTIN KOTRAŠ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ALEŠ SMRČKA, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Kotraš Martin**
Program: Informační technologie
Název: **Interaktivní generátor syntaxe heterogenních datových struktur**
Interactive Generator of Syntax of Heterogeneous Data Structures
Kategorie: Web

Zadání:

1. Seznamte se s formáty pro popis stromových struktur a schémat pro JSON a XML. Nastudujte formáty pro popis formálních jazyků (EBNF, ABNF).
2. Navrhněte webovou aplikaci pro interaktivní specifikaci gramatiky řetězců vyjadřující hodnoty běžně užívaných datových struktur. Zaměřte se na řetězce bezkontextových jazyků. Webová aplikace by měla obsahovat editor dokumentů, zvýraznění syntaxe a interaktivní úpravu kódu se zpětnou vazbou. Výstupem aplikace má být gramatika jazyka. Volitelnou výstupní částí je validátor vstupních řetězců.
3. Implementujte navrženou aplikaci jako jednostránkovou webovou aplikaci.
4. Demonstrujte funkčnost aplikace na umělé sadě příkladů.

Literatura:

- Draft standardu IETF pro schéma JSON. Dostupné na URL: https://tools.ietf.org/html/draft-handrews-json-schema-01
- Standard JSON. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259.
- Augmented BNF for Syntax Specifications: ABNF. RFC 5234.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 31. října 2019

Abstrakt

V dnešní době jsou softwarové systémy často skládány z několika komponent předávající si data různými komunikačními kanály. I přes to, že existuje řada standardizovaných formátů kódování dat, vývojáři stále vytváří své vlastní většinou s ohledem na specifické použití jimi vytvářeného softwaru. Jednou ze zásadních částí ověření kvality nebo minimalizace chyb z přenosu dat je validace vstupních dat. Prvním krokem k validaci je formalizace jazyka popisující datové struktury. Nejobecnějším formalismem pro tyto účely je gramatika jazyka ve standardním popisu, např. BNF, ABNF, nebo EBNF. Tvorba gramatiky specifického jazyka však může být krok, který je citlivý na vznik chyby pro nezkušeného vývojáře. Cílem tohoto projektu je jednoduchá aplikace pro tvorbu gramatiky ze vzorku dat.

V práci je řešeno generování gramatiky a následných validačních kousků kódu z nahraného ukázkového řetězce jazyka, např. zdrojového kódu programovacího jazyka. Problém řeší uživatel aplikace postupným označováním částí nahraného řetězce, jejich pojmenováním a přiřazováním vlastností. K tomu mu dopomáhají nástroje na rozdělování pravidel, slučování prefixů a/nebo sufixů pravidla, vytváření seznamů a optimalizaci výsledných pravidel.

V rámci práce vznikla jednostránková webová aplikace, která při testování na jazycích JSON a XML dokázala poměrně dobře obstát a bylo možné tak vytvořit obecnější gramatiku i přes problémy se slabým syntaktickým analyzátorem. Díky této práci tak mohou i méně zkušení uživatelé vytvářet obecnější gramatiky jejich řetězců a používat je pro validaci. Práce navíc dává základ pro další zkoumání v této oblasti a je otevřená pro další vylepšení.

Abstract

Today, software systems are often composed of several components that transmit data through various communication channels. Despite the fact that there are a number of standardized data encoding formats, developers still create their own mostly with regard to the specific use of the software they create. One of the essential parts of quality verification or minimization of data transmission errors is the validation of input data. The first step to validation is to formalize a language describing data structures. The most general formalism for these purposes is the grammar of the language in the standard description, e.g. BNF, ABNF, or EBNF. However, creating a language-specific grammar can be a step that is sensitive to error for an inexperienced developer. The aim of this project is a simple application for creating grammar from a sample of data.

The work solves the generation of grammar and validation code snippets from the sample string of the language, e.g. the source code of the programming language. The user solves the problem by sequentially marking parts of the uploaded string, naming them, and assigning properties to them. This is aided by tools for splitting rules, merging rule prefixes and/or suffixes, creating lists, and optimizing the resulting rules.

As part of the work, a single-page web application was created, which was able to pass relatively well when tested on JSON and XML, and it was possible to create a more general grammar despite the problems with a weak parser. Thanks to this work, even less experienced users can create more general grammars of their strings and use them for validation purposes. In addition, the work provides a basis for further research in this area and is open to further improvement.

Klíčová slova

bezkontextová gramatika, formální jazyky, ABNF, validátor, generátor syntaxe, metadata, React, Redux

Keywords

context-free grammar, formal languages, ABNF, validator, syntax generator, metadata, React, Redux

Citace

KOTRAŠ, Martin. *Interaktivní generátor syntaxe heterogenních datových struktur*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Interaktivní generátor syntaxe heterogenních datových struktur

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Martin Kotraš
27. května 2020

Poděkování

Tímto bych rád poděkoval vedoucímu práce panu Ing. Aleši Smrčkovi Ph.D. za pravidelné konzultace, pomoc a cenné rady, které mi poskytl.

Obsah

Kapitola 1

Úvod

V oblasti vývoje počítačových programů se často stává, že počítačový program potřebuje při své činnosti získat nějaká data z nedůvěryhodného zdroje. Ať už se jedná o vstup od uživatele, odpověď od aplikace třetí strany nebo třeba automaticky vytvářené požadavky webovým prohlížečem, vždy musí vývojář programu pamatovat na možnost obdržení dat, která nebyla očekávána. Častým cílem programátora ale není trávit hodiny nad kontrolou každého jednoho vstupu do aplikace, nýbrž vytvářet aplikace, které něco umí a jsou užitečné jemu samotnému nebo uživatelům dané aplikace.

Práce si klade za cíl prostřednictvím webové aplikace demonstrovat, jak lze z konkrétního příkladu vstupních dat vytvořit za asistence programátora jejich zobecněný popis. Zobecněný popis dat poté může programátor využít ke kontrole většího počtu konkrétních dat – oněch vstupů od uživatele a podobně – nebo dokonce i pro porozumění těmto datům. K tomuto účelu může programátor využít i kousky kódu pro vybrané programovací jazyky, které mu aplikace sama vygeneruje. Naplněním cíle lze tím pádem ušetřit programátorovi přemýšlení nad tím, jak má data kontrolovat, a také zpřístupnit neznalým programátorům prostředky pro formální popis dat.

Kapitola 2 představuje teoretický úvod do současného stavu v oblasti bezkontextových jazyků, gramatik, jejich analýzy, popisu a existujících technologií. Nápad, jak by mohl uživatel při své práci postupovat, a z něj vycházející návrh datového modelu a dalších částí aplikace, prezentuje kapitola 3. Principy generování gramatiky z dat a generátor jakožto jednu z nejdůležitějších součástí aplikace přibližuje kapitola 4. Komponenta pro optimalizaci generované množiny pravidel sloužící nejen pro zmenšení množiny pravidel, ale i pro pomoc uživateli při orientaci v generované gramatice, je představena v kapitole 5. Kapitola 6 ukazuje postup, jakým aplikace převádí vnitřní reprezentaci pravidel na pravidla ve výstupním formátu a na kousky kódu pro různé knihovny a programovací jazyky. Představení webových technologií, které aplikace pro svou funkčnost využívá, je možné nalézt v kapitole 7, přičemž prvky grafického uživatelského rozhraní, které staví zejména na těchto webových technologiích, ukazuje kapitola 8. Testováním výsledné aplikace na bezkontextových jazycích nakonec práci uzavírá kapitola 9.

Kapitola 2

Teorie bezkontextových jazyků a gramatik

Tato kapitola popisuje teoretické základy o bezkontextových jazycích a gramatikách, které jsou zkoumány v této práci. Kromě samotných bezkontextových jazyků a gramatik je nutné si přiblížit i způsoby jejich zpracování – syntaktické analýzy – a použití v programovacích jazycích či formátech, které je možné ve výsledné aplikaci dále zpracovávat. Zároveň je pro účely této práce potřeba představit prostředky, jak lze formálně popsat bezkontextové gramatiky. Užitečné je i seznámení s již existujícími technologie pro práci s bezkontextovými gramatikami, ze kterých je možné dále čerpat inspiraci.

2.1 Bezkontextové jazyky a gramatiky

Následující odstavce představují stručný výtah převážně ze skript Milana Češky a Zdeny Rábové [7] pro uvedení do dané problematiky.

Bezkontextový jazyk je množina řetězců, které lze vygenerovat pomocí bezkontextových gramatik. Bezkontextové gramatiky jsou přitom takové gramatiky, jejichž pravidla mají na levé straně neterminál a na pravé straně libovolnou kombinaci terminálů a neterminálů, přičemž takováto kombinace může být i prázdná, tj. neobsahující ani jeden terminál či neterminál. Bezkontextové gramatiky dostaly svůj název podle skutečnosti, že pravidla lze aplikovat bez ohledu na to, kde se nacházejí, tedy bez ohledu na kontext.

Bezkontextové jazyky a gramatiky jsou důležité zejména z toho důvodu, že pomocí těchto gramatik lze popsat převážnou většinu konstrukcí programovacích jazyků. Mimo to existují i efektivní algoritmy pro analýzu řetězců těchto jazyků.

Že se jedná o alespoň bezkontextový jazyk a nemůže se tak jednat o jazyk regulární, lze často poznat podle výskytu stejného počtu symbolů a a b. Příklady symbolů a a b, které lze často najít v programovacích jazycích, uvádí tabulka 2.1.

a	b	**Příklad**	**Význam**
`(`	`)`	`(1 - (x + y)) / 2`	Vynucení priorit operací
`[`	`]`	`[[1, 2], [3, 4]]`	Zjednodušený zápis polí
`{`	`}`	`if (x) { a(); if (y) { b(); c(); } }`	Seskupení příkazů

Tabulka 2.1: Příklady konstrukcí v programovacích jazycích, kdy je třeba sledovat stejný počet symbolů a a b a jedná se tak o vlastnost alespoň bezkontextového jazyka.

2.2 Syntaktická analýza bezkontextových jazyků

Následující odstavce této sekce stručně shrnují problematiku syntaktické analýzy bezkontextových jazyků tak, jak byla popsána ve skriptech [7].

Libovolný řetězec libovolného bezkontextového jazyka lze syntakticky analyzovat pomocí abstraktního zařízení – zásobníkového automatu. Zásobník jakožto abstraktní datový typ zde hraje klíčovou roli – konečný automat bez zásobníku totiž dokáže analyzovat pouze regulární jazyky. Zásobníkové automaty jsou obecně vzato nedeterministické, existují však ale i deterministické zásobníkové automaty, které mohou v každé situaci provést maximálně jeden příkaz.

Deterministické zásobníkové automaty jsou důležité zejména díky jejich přednostem při výstavbě překladačů programovacích jazyků. Existuje pro ně například algoritmus, který dokáže pracovat v lineárním čase. Toho dosahují skutečností, že mohou, jak už bylo řečeno, vždy provést maximálně jeden příkaz a nepotřebují tak na rozdíl od nedeterministických zásobníkových automatů využívat zpětného návratu, který je časově náročný – vyžadující v nejhorším případně exponenciální čas.

Pro syntaktickou analýzu bezkontextových jazyků existují v zásadě dva hlavní způsoby, kterými může syntaktický analyzátor postupovat:

- syntaktická analýza shora dolů, kdy syntaktický analyzátor postupuje od počátečního symbolu, tedy kořene derivačního stromu, až po terminální symboly, tedy listy derivačního stromu, a
- syntaktická analýza zdola nahoru, kdy syntaktický analyzátor postupuje naopak, tedy od terminálních symbolů až k počátečnímu symbolu.

Jednou z metod syntaktické analýzy shora dolů je i rekurzivní sestup. Ten spočívá v rozdělení programu na funkce, které reprezentují neterminály, přičemž tyto funkce se vzájemně volají a za pomocí lexikálního analyzátoru čtou další symboly na vstupu. Lexikální analyzátor je přitom součást překladače, která se stará o rozdělení vstupního textu na lexémy, tedy například číslo, řetězec, komentář a podobně. Zásobník v tomto případě nemusí programátor sám implementovat, ale je tvořený zásobníkem volání funkcí, tzv. *call stack*.

2.3 Různé popisy bezkontextových jazyků

Pro popis bezkontextových jazyků lze využít některý z již existujících nástrojů, tzv. metajazyků. Metajazyk je sám o sobě také jazyk, který se ale využívá pro popis jiného jazyka, ačkoliv je často možné popsat metajazykem sama sebe, čehož může být využito i pro jeho definici. Příkladem metajazyků může být Backusova–Naurova forma (BNF) a její vylepšené varianty rozvinutá Backusova–Naurova forma (EBNF) nebo rozšířená Backusova–Naurova forma (ABNF).

2.3.1 Gramatika v Backusově–Naurově formě

Pro BNF neexistuje jedna konkrétní specifikace [12], nicméně lze definovat některé společné znaky tohoto jazyka, jak ukazuje tabulka 2.2.

Výhody

- poměrně jednoduchá syntaxe,

Zápis	Význam
`<pravidlo>`	neterminální symbol
`::=`	oddělení neterminálu od definice
`|`	oddělení alternativ
␣	zřetězení po sobě jdoucích symbolů
`""` nebo `''`	prázdný řetězec – ϵ
`"ab"` nebo `'ab'`	terminální řetězec

Tabulka 2.2: Přibližný výčet zápisů některých konstrukcí jazyka BNF.

- dobrá podpora v generátorech překladačů.

Nevýhody

- nejednotná syntaxe,
- chybějící speciální operátor pro opakování,
- chybí formální způsob pro zápis speciálních symbolů v terminálech,
- terminální symboly nelze zapsat pomocí rozsahů.

2.3.2 Gramatika v rozvinuté Backusově–Naurově formě

EBNF vychází z jazyka BNF, který vylepšuje o další užitečné konstrukce. Ačkoliv je jeho syntaxe standardizovaná ve standardu ISO/IEC 14977 již od roku 1996 [10], stále existuje mnoho dalších variant, které se liší způsobem zápisu [12]. Zápisy vybraných konstrukcí tak, jak je definuje norma ISO, ukazuje tabulka 2.3.

Zápis	Význam
`pravidlo`	neterminální symbol
`=`	oddělení neterminálu od definice
`|` nebo `/` nebo `!`	oddělení alternativ
`,`	zřetězení symbolů
`;` nebo `.`	ukončení definice
`( ... )`	seskupení pro vynucení priorit
`[ ... ]` nebo `(/ ... /)`	volitelná sekvence
`{ ... }` nebo `(: ... :)`	opakování nula a vícekrát
`n * výraz`	opakování výrazu přesně `n`-krát
`"ab"` nebo `'ab'`	terminální řetězec
`(* ... *)`	komentář

Tabulka 2.3: Výčet vybraných konstrukcí jazyka EBNF podle normy ISO/IEC 14977.

Výhody

- existuje pro něj ISO standard,

- disponuje operátory pro opakování,
- velmi dobrá podpora v generátorech překladačů.

Nevýhody

- značné množství nestandardních implementací,
- chybí formální způsob pro zápis speciálních symbolů v terminálech,
- terminální symboly nelze zapsat pomocí rozsahů.

2.3.3 Gramatika v rozšířené Backusově–Naurově formě

ABNF vychází stejně jako EBNF z jazyka BNF, upravuje jeho syntaxi a zavádí další užitečné konstrukce. Na rozdíl od BNF či EBNF jde díky standardu RFC 5234 o podstatně lépe standardizovaný jazyk [12]. Kromě uživatelsky definovaných pravidel jsou uživateli k dispozici také některá často používaná předdefinovaná pravidla, např. `ALPHA` pro velká a malá písmena nebo `DIGIT` pro čísla [5]. Jak lze zapsat vybrané konstrukce podle standardu RFC ukazuje tabulka 2.4.

Zápis	**Význam**
`pravidlo`	neterminální symbol
`=`	oddělení neterminálu od definice
`|`	oddělení alternativ
␣	zřetězení symbolů
`[ ... ]`	volitelná sekvence
`<min=0>*<max=∞>výraz`	opakování výrazu `<min>`-krát až `<max>`-krát
`n*výraz`	opakování výrazu přesně `n`-krát
`"ab"`	terminální řetězec nerozlišující velikost písmen
`%s"ab"` nebo `%d97.98`	terminální řetězec rozlišující velikost písmen
`%d97-122`	rozsah terminálních symbolů
`( ... )`	seskupení pro vynucení priorit
`;`	komentář

Tabulka 2.4: Výčet vybraných konstrukcí jazyka ABNF podle standardů RFC 5234 a RFC 7405.

Výhody

- standardizováno v RFC,
- disponuje operátory pro opakování,
- speciální symboly v terminálech lze formálně zapsat,
- terminální symboly umožňuje zapsat pomocí rozsahů.

Nevýhody

- velmi malá podpora v generátorech překladačů.

2.4 Existující technologie pro práci s bezkontextovými jazyky

Pro specifické druhy bezkontextových jazyků již existují technologie, které je dokážou popsat a pracovat s nimi. Jedná se například o regulární výrazy nebo JSON či XML schémata. Kromě obecnějších regulárních výrazů dokážou zmíněná schémata pracovat pouze s formátem, pro který jsou určeny. To ale nemusí být na škodu – díky tomu jsou schémata vyladěna právě pro účely daného formátu. Z XML navíc vychází celá řada dalších formátů, které staví na XML schématech, jmenovitě například HTML, SVG, RSS, XAML, FXML a další.

2.4.1 Regulární výrazy

Pokud je řeč o regulárních výrazech, myšleny jsou obvykle regulární výrazy kompatibilní s Perlem, tzv. PCRE[1]. Takovéto regulární výrazy i přes svůj název a valnou většinu regulárních konstrukcí ale obsahují i konstrukce – například zpětné reference – které nejsou dle pumping lemmatu regulární, a dokonce ani bezkontextové [4]. Z regulárních výrazů je možné si vzít pro práci jako inspiraci některé známé způsoby zápisu opakování `?*+` nebo třeba třídy znaků `[a-z]`.

2.4.2 JSON schéma

JSON schéma nemá v současné době dokončenou specifikaci, přičemž nejnovější návrh specifikace je popsán v příslušném dokumentu organizace IETF[2]. Pro validaci nabízí datové typy jako řetězec, číslo, objekt, pole, pravdivostní typ, prázdná hodnota, a dokonce i regulární výrazy. K těmto datovým typům existují i pravidla, která reflektují potřeby jak syntaktické, tak i sémantické, například délka řetězce, rozsah čísla, unikátnost pole a podobně. Pro práci je schéma inspirací díky nabízeným typům i syntaktickým omezením, které lze vyžadovat a kontrolovat. [6]

2.4.3 XML schémata

XML schémat existuje celá řada, přičemž jedny z nejznámějších jsou DTD[3], XSD[4] nebo Relax NG[5]. Obecně mají všechna schémata za cíl nějakým způsobem popsat XML dokument a lze z nich získat informace o datových typech a omezení jednotlivých položek. Datové typy zahrnují podobné typy jako v případě JSON schématu a stejně tak je tomu i u základních validačních pravidel. Vzhledem k povaze XML jsou ale přítomna i pravidla pro validaci atributů a další datové typy jako například datum a čas. [3]

[1] https://www.pcre.org/
[2] https://www.ietf.org/
[3] https://www.w3schools.com/xml/xml_dtd.asp
[4] https://www.w3schools.com/xml/schema_intro.asp
[5] https://relaxng.org/

Kapitola 3

Návrh podle principu činnosti uživatele

Než bude možné začít programovat aplikaci, je nutné si ujasnit, jak by měla aplikace fungovat, co od ní uživatel očekává, resp. jak s ní bude pracovat a co k tomu bude potřeba. K tomu slouží tato kapitola vysvětlující skrze neformální popis doplněný o diagram případů užití princip činnosti uživatele, ze kterého je poté vycházeno dále v práci. Návrh se neobejde bez datového modelu odrážejícího nastíněný princip činnosti uživatele a také průzkumu a návrhu některých základních částí práce, do nichž patří způsob zadání vstupního textu a z něho pramenící vytvoření počáteční množiny pravidel, kterou může následně analyzovat syntaktický analyzátor.

3.1 Princip činnosti uživatele

Když uživatel přijde na stránku, musí v první řadě nějakým způsobem vložit do aplikace text – řetězec programovacího nebo jiného jazyka, se kterým bude chtít v následujících krocích pracovat. O tom, jak takový text může zadat, a o výběru nejoptimálnější varianty pojednává blíže sekce 3.3.

Při vkládání textu musí být uživateli nějakým způsobem vygenerována startovní množina pravidel, která bude následně na základě jeho instrukcí transformována do cílové podoby. Jaké jsou v tomto možnosti, uvádí sekce 3.4.

Jakmile má aplikace k dispozici uživatelem vložený text, může uživatel začít v textu označovat jeho jednotlivé části, ty začít pojmenovávat a na základě dostupných akcí dále transformovat výslednou množinu pravidel. Co za akce má uživatel k dispozici a jak mu mohou pomoci při generování obecnější množiny pravidel, popisuje samostatná kapitola 4.

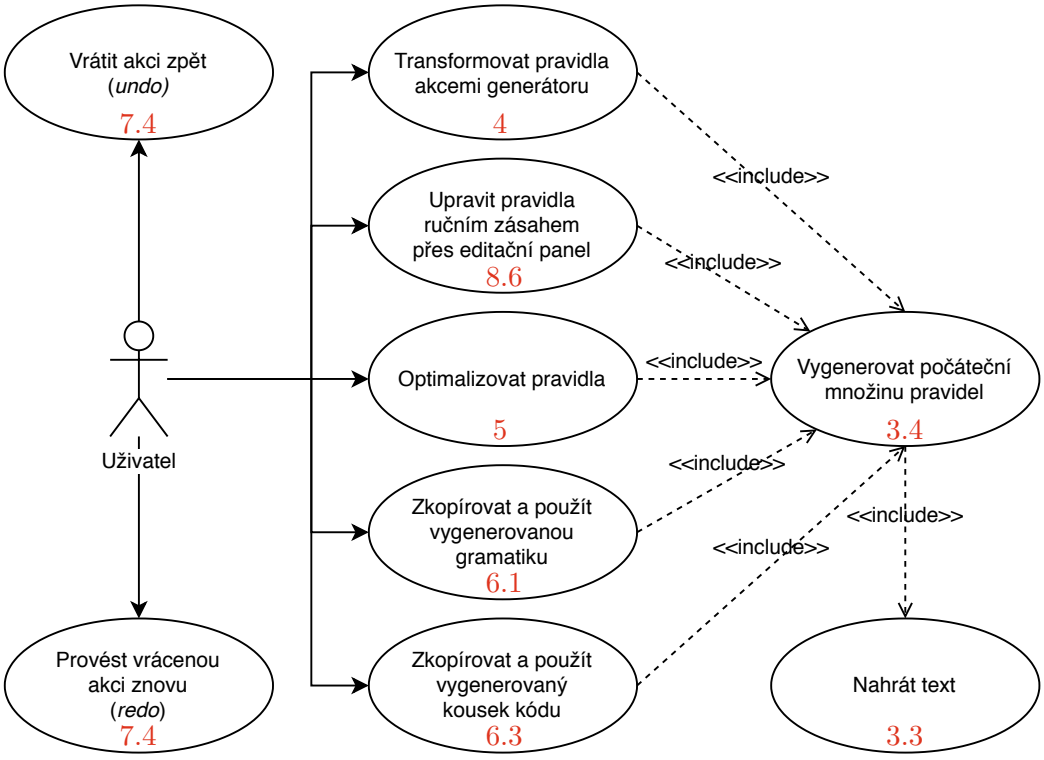
Ne vše lze úplně přesně specifikovat a zobecnit pouhým označováním vstupního textu. Za tímto účelem potřebuje mít uživatel možnost zasáhnout do generované množiny pravidel a dále blíže určit, jak mají jednotlivá pravidla v obecnější formě vypadat. K tomu může uživatel použít formulář, jehož podobu přibližuje sekce 8.6.

Po dokončení akcí uživatele má uživatel k dispozici množinu pravidel, která ale vzhledem k procesu generování nemusí být optimální. Proto se uživatel může rozhodnout, že chce tuto množinu pravidel ještě dále optimalizovat. To může učinit buď ručně pomocí stejného editačního formuláře, který použil pro specifikaci vlastností jednotlivých pravidel, nebo může využít možnosti nechat si pravidla optimalizovat automaticky od aplikace. O tom,

jak optimalizace probíhá a co za optimalizační pravidla jsou uživateli k dispozici, se lze více informací dozvědět v kapitole 5.

V posledním kroku si uživatel zvolí, jaký výstup z aplikace chce pro své účely použít. Buď si tak může zvolit čistý formální zápis pravidel, o čemž pojednává sekce 6.1, nebo si může vybrat jeden z nabízených kousků kódů podle jazyka či knihovny, o čemž je zase více informací dostupných v sekci 6.3.

Pokud je uživatel spokojený, má v tento moment hotovo a může opustit aplikaci. V případě, že se mu něco nepovedlo, může dále provádět úpravy až dokud se mu nepovede dostat do stavu, kdy je s výsledkem spokojený. Přitom může využít i akcí „zpět“ a „vpřed“ a odvolat tak nechtěné úpravy. Náhled do možností, jak lze tyto akce implementovat, nabízí sekce 7.4.



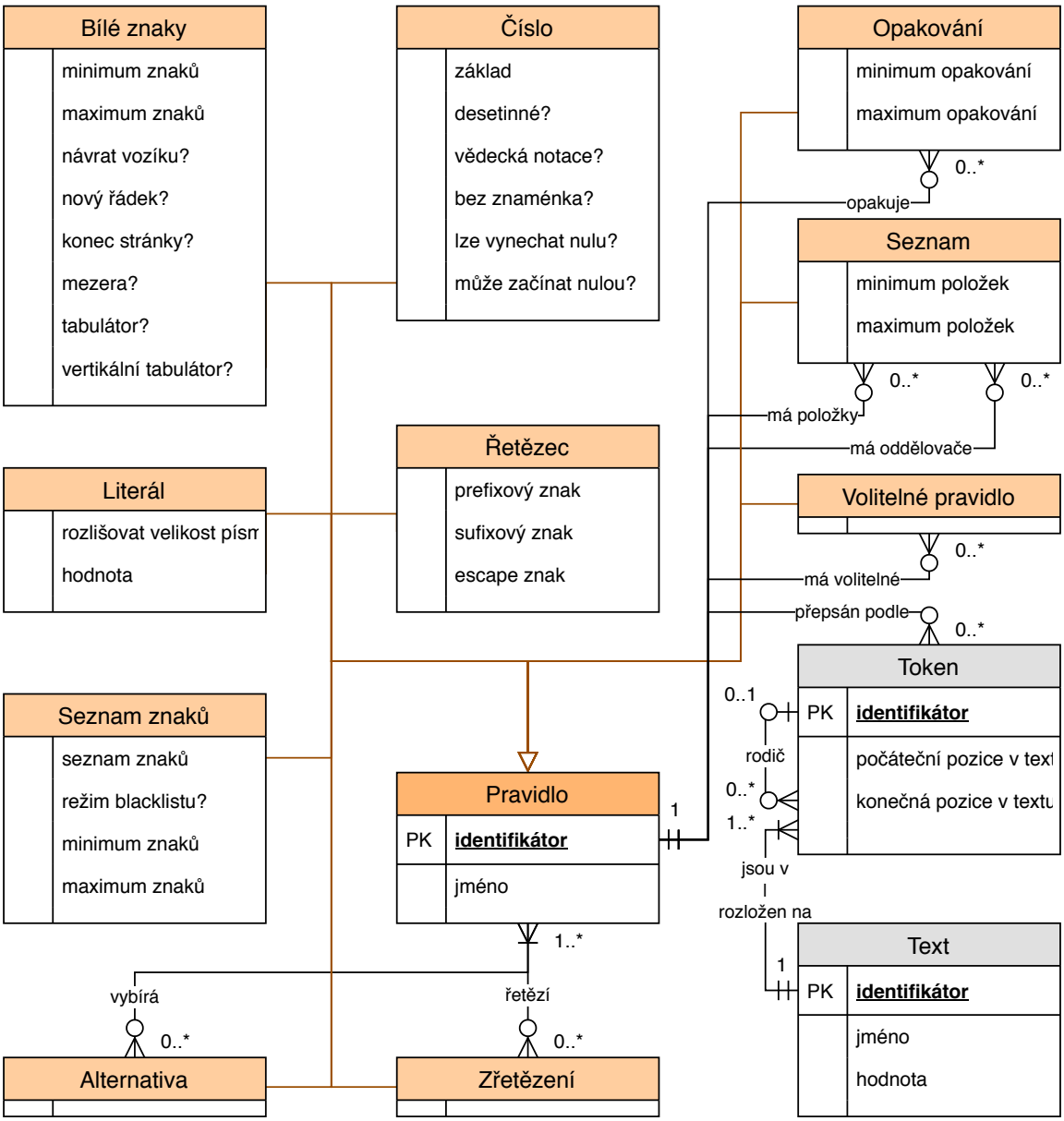
Obrázek 3.1: Diagram případů užití aplikace. Relace `«include»` znázorňují potřebu mít v aplikaci nahraný text, ze kterého aplikace vytvořila počáteční množinu pravidel, aby mohly být příslušné akce vykonány. Čísla u jednotlivých případů užití odkazují do příslušných sekcí či kapitol, které se daným případem užití zabývají.

3.2 Datový model

Na základě průzkumu běžně používaných syntaktických konstrukcí, datových typů, jejich omezujících podmínek a podobně v jazycích JSON a XML a ve schématech pro tyto jazyky, byla pro datový model navržena sada 10 pravidel – 5 terminálních pravidel a dalších 5 neterminálních. Všechna pravidla vychází z jednoho abstraktního pravidla, které specifikuje pouze název pravidla a zbytek ponechává na jeho konkrétní varianty. Typy pravidel a jejich možnou konfigurovatelnost, ukazuje obrázek 3.2.

Na obrázku 3.2 lze kromě pravidel samotných vidět i entity text a token. Entita text slouží jako skladiště pro vstupní texty zadané samotným uživatel. Aplikace už od začátku počítá s tím, že textů může být v aplikaci v jeden okamžik více než jeden. Díky tomu lze aplikaci v budoucnu rozšířit o texty, které se mohou použít pro kontrolu, zda je možné z vygenerované gramatiky odvodit tento nově nahraný text nebo také pro dodefinování dalších částí gramatiky, které nejsou z prvního zadaného textu zřejmé.

Entita token slouží pro reprezentaci jednoho uzlu derivačního stromu, který vzniká na základě syntaktické analýzy vstupního textu. Tyto entity jsou zvláštní tím, že nemusejí být, na rozdíl od ostatních prezentovaných entit, uloženy v relativně trvalém úložišti, protože je lze kdykoliv znovu vytvořit pomocí syntaktické analýzy.



Obrázek 3.2: ER diagram zachycující nejdůležitější prvky datového modelu – tmavou oranžovou je zachyceno abstraktní pravidlo, světlejší oranžovou poté jeho konkrétní varianty.

3.3 Způsoby zadání vstupního textu

Vstupní text může uživatel zadat do aplikace v zásadě dvěma způsoby: buď text napíše nebo vloží do nějakého textového pole či editoru kódu, nebo text nahraje jako soubor.

Psaní vstupního textu do textového pole by uživateli poskytlo mnohem větší svobodu než prosté a jednorázové nahrání souboru s textem. Text by mohl kdykoliv a jakkoliv upravovat a opravovat a nemusel by se vázat na jednou nahraný a možná i chybný text. Na druhou stranu by to znamenalo, že aplikace musí v každém kroku promítat tyto změny ze strany uživatele do generované gramatiky, jinak by se aplikace mohla dostat do stavu, kdy vygenerovaná gramatika neodpovídá vstupnímu řetězci. Problém by to nebyl v případě, že by byla gramatika dostatečně obecná, aby dokázala zachytit všechny validní řetězce, které si může uživatel vymyslet. Ale vzhledem k tomu, že gramatika začíná s jedním konkrétním řetězcem a transformuje se do obecnější gramatiky až s postupnými zásahy uživatele, není reálné takovou synchronizaci nějak rozumně zajistit. Složitost takové synchronizace roste tím více, čím více zásahů a kroků pro vytváření cílové gramatiky uživatel provede.

Pokud uživatel již na začátku nahraje neměnný soubor, může si aplikace okamžitě vytvořit startovní pravidlo, které bude přesně odpovídat nahranému souboru, a toto pravidlo poté může bez nutnosti jakékoliv synchronizace dále transformovat dle potřeb uživatele. Nevýhodou tohoto přístupu je naopak ztráta svobody editace na uživatelově straně. Na druhou stranu to může přimět uživatele si pečlivě rozmyslet, jak má jeho vstupní text vypadat, což mu umožní snadnější vytváření gramatiky.

Z důvodu jednoduchosti implementace byla v práci použita druhá zmíněná možnost, tedy nahrání neměnného souboru. I přes použití této možnosti ale práce zůstává i nadále otevřená případnému rozšíření o první zmíněnou možnost.

3.4 Vygenerování počáteční množiny pravidel

Inicializaci počáteční množiny pravidel lze provést několika způsoby. Aplikace se může například pokusit podle obsahu nahraného textu zjistit, co za formát uživatel nahrál a podle něj doplnit počáteční množinu pravidel. Jelikož ale není cílem mít hned ze začátku obecnou množinu pravidel a je více méně na uživateli, aby pomocí aplikace takovou množinu vytvořil, je možné využít nejjednodušší varianty, kdy bude počáteční množina pravidel obsahovat jediné pravidlo, které bude odpovídat přesně vstupnímu textu. Tzn. například ze vstupního textu `{x: 1}` vznikne pravidlo `start = "{x: 1}"` v ABNF[1] notaci.

Varianta, kdy se přímo z textu vytvoří pravidlo, možnosti vytvoření počáteční množiny pravidel neuzavírá. V budoucnu by tak bylo možné přidat například první variantu, kterou by si uživatel mohl zvolit jako alternativu při inicializaci aplikace.

3.5 Syntaktická analýza vstupního textu

O syntaktickou analýzu vstupního textu se stará syntaktický analyzátor, v kódu také nazývaný jako *parser*. Úloha syntaktického analyzátoru je pomoci následující komponentě – generátoru – najít ve vstupním textu část pravidla, ve které bude generátor pracovat a ze které bude vytvářet nová pravidla.

Jelikož nebylo cílem práce implementovat pokročilý syntaktický analyzátor, byla pro syntaktickou analýzu zvolena implementačně jednoduchá metoda rekurzivního sestupu.

[1] ABNF viz sekce 2.3.3.

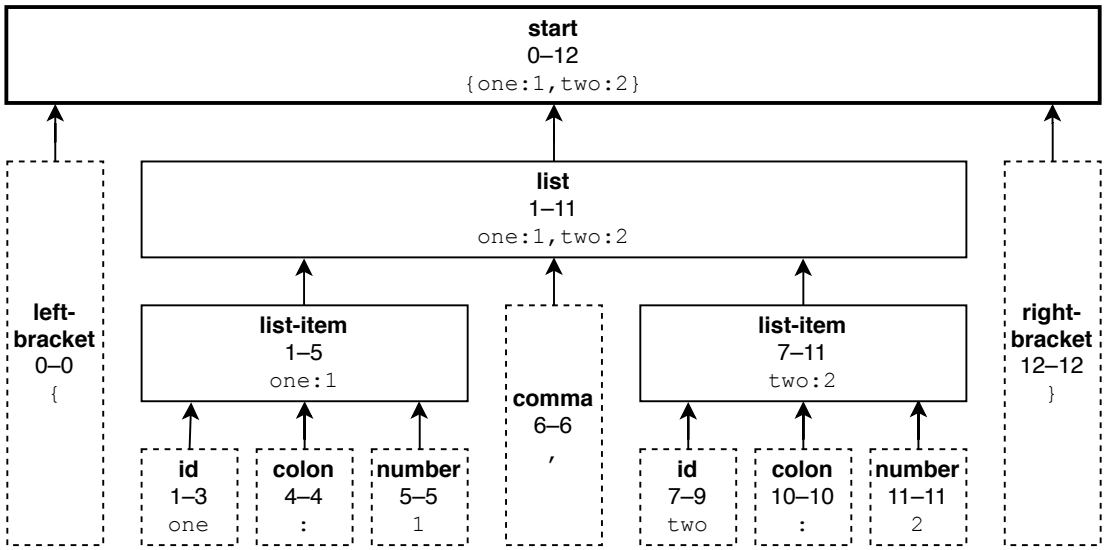
V neterminálních pravidlech, kde není předem jasné, jaké pravidlo se má aplikovat, využívá syntaktický analyzátor metodu pokus–omyl, resp. určitou formu omezeného zpětného návratu. Například v pravidlu alternativy jsou postupně zleva doprava zkoušeny všechny alternativy, až dokud se syntaktický analyzátor nedostane k prvnímu pravidlu, které lze aplikovat. Toto nalezené pravidlo syntaktický analyzátor použije a žádné jiné už nikdy jindy nevyzkouší.

Takovýto návrh syntaktického analyzátoru umožní jeho velmi snadnou implementaci, která je ale bohužel kompenzována jeho omezenou sílou, kdy může odmítnout jako syntakticky nesprávný vstupní řetězec, který ve skutečnosti syntaktický správný je. Příkladem může být řetězec `x` generovaný gramatikou `start = ["x"] "x"`, kdy syntaktický analyzátor přečte symbol `x` jako volitelnou část a dále mylně očekává zde nepřítomný druhý symbol `x`. Stejný případ může nastat i pro jinak zapsanou gramatiku, např. `start = "x" / "xx"` pro vstupní řetězec `xx` nebo `start = ("x" / "xx") "x"`, kde nepomůže pro přijímání obou vstupních řetězců ani prohození pořadí alternativ jako v předchozím případě.

Návrh syntaktické analyzátoru v této podobě zároveň může být při vysokém využití takovýchto neterminálních pravidel poměrně pomalý čili s vysokou asymptotickou složitostí a ani zdaleka tak nemusí dosahovat rychlostí nabízené prediktivním syntaktickým analyzátorem.

Na druhou stranu, budoucí implementace kompletního zpětného návratu a s tím související zvýšení síly syntaktického analyzátoru může být pro aplikaci žádoucí. Uživatelem nahrané texty totiž nemusejí být nutně popsané či popsatelné deterministickými bezkontextovými gramatikami a stejně tak ani generátor nemusí nutně vygenerovat deterministickou variantu, existuje-li.

Možný příklad derivačního stromu, který je výstupem syntaktického analyzátoru, tak, jak ho dostanou pro práci další komponenty, ukazuje obrázek 3.3.



Obrázek 3.3: Příklad možného derivačního stromu pro řetězec `{one:1,two:2}` v podobném formátu jako JSON. Uzly obsahují jméno pravidla, podle kterého se symbol přepsal, rozsah v původním řetězci číslovaný od nuly a reprezentovanou část řetězce. Kořen stromu a zároveň startovní pravidlo je vyznačeno tučným ohraničením. Listy stromu či terminály jsou označeny přerušovaným ohraničením.

3.5.1 Zpracování terminálních symbolů

Syntaktický analyzátor podle tohoto návrhu nevyužívá lexikální analyzátor, ale stará se o čtení terminálních symbolů sám. Pravidla jsou postupně čtena od startovního pravidla až po první pravidlo, které je chápáno jako terminální. V obvyklých případech by následující terminální symbol získal a vrátil lexikální analyzátor, ale vzhledem k jeho absenci rozhoduje o přečtení toho kterého terminálního symbolu syntaktický analyzátor a pravidlo, které je aktuálně na řadě.

Terminální symboly lze číst jakožto regulární části pomocí stavového automatu [7], ale vzhledem k tomu, že stejnou práci dokážou zaujmout i regulární výrazy, využívá syntaktický analyzátor jich. Před přečtením terminálu je tak z aktuální konfigurace pravidla sestaven příslušný regulární výraz, který se syntaktický analyzátor pokusí přečíst.

V aplikaci je momentálně pro zpracování k dispozici 5 terminálních symbolů:

- **bílé znaky** v rozmezí `<min,max>` odpovídající sekvenci `\s` či `[\f\n\r\v ]` z regulárních výrazů,
- **číslo** o určitém základu zapsané různými způsoby včetně vědecké notace,
- **literál** jakožto přesná posloupnost znaků, volitelně s možností nerozlišovat velikost písmen,
- **řetězec** libovolných znaků začínající a končící právě jedním znakem a s *escape* sekvencí a
- **seznam znaků** v rozmezí `<min,max>` sestávají z libovolné kombinace uživatelem zadaných znaků, případně z libovolné kombinace znaků, které uživatel nezadal.

3.5.2 Zpracování neterminálních symbolů

Pro neterminální symboly se, na rozdíl od symbolů terminálních, nevytváří regulární výrazy, ale instruuje se syntaktický analyzátor, aby rekurzivně zpracoval další neterminální symboly. Přitom záleží na typu a konfiguraci pravidla dané uživatelem – některá pravidla zpracují právě jedno další pravidlo (alternativa) a některá pravidla mohou zpracovávat teoreticky i nekonečně mnoho dalších pravidel (seznam, opakování).

V aplikace je momentálně k dispozici pro zpracování 5 neterminálních symbolů:

- **alternativa** pro výběr jednoho z několika alternativních pravidel,
- **opakování** jediného pravidla v rozmezí `<min,max>`,
- **seznam pravidel** jednoho typu oddělený oddělujícím pravidlem,
- **volitelné pravidlo** jakožto rychlá volba specializace pravidla opakování pro interval `<0,1>` a
- **zřetězení** několika pravidel za sebou.

Kapitola 4

Principy generování gramatiky z dat

Jednou z nejdůležitějších částí aplikace je komponenta nazvaná generátor, která se stará o generování nových pravidel a jejich transformaci přesně podle požadavků uživatele. Jednotlivé akce generátoru, které může uživatel využívat při práci s pravidly, jsou popsány v sekcích 4.2, 4.3, 4.4 a 4.5. Před každou akcí je ale nutné nejprve vyhledat místo, kde si uživatel přeje požadovanou akci vykonat. Jak toho lze docílit popisuje sekce 4.1.

4.1 Vyhledání pravidla podle vybrané části textu

Když uživatel dokončí vybírání částí vstupního textu a zvolí požadovanou akci, potřebuje aplikace v první řadě zjistit, co uživatel vybral a v jakém pravidle má aplikace vůbec provést změnu. Pro tento účel musí aplikace nahlédnout do derivačního stromu, který jí dodá syntaktický analyzátor popsaný v sekci 3.5. Úkolem je pak vyhledat v tomto derivačním stromu takový uzel, jehož interval zcela obsahuje uživatelem vybraný interval a zároveň je jeho velikost nejmenší možná.

Jednoduchým řešením tohoto problému je projít všechny uzly stromu, zjistit, které uzly obsahují uživatelův interval, a z těchto vybraných intervalů vybrat ten nejmenší. Toto řešení pochopitelně vyžaduje pro svůj běh lineární čas, ale vzhledem k jeho implementační nenáročnosti bylo zvoleno právě ono.

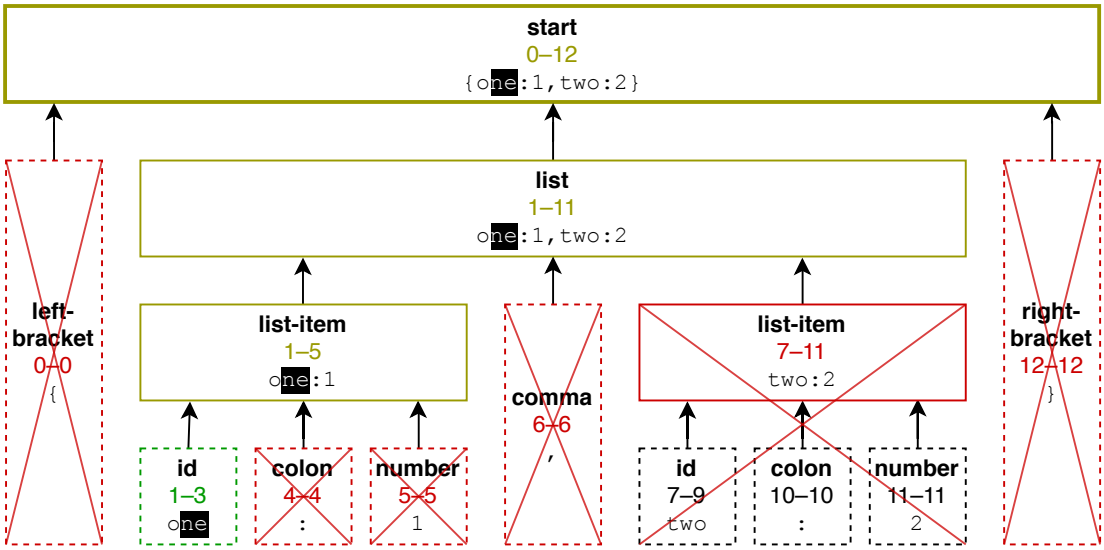
Vzhledem k tomu, že vrácený derivační strom splňuje vlastnosti vyhledávacího stromu, je možné ho použít k efektivnějšímu vyhledávání intervalů uživatele. Derivační strom má v kořeni interval o rozsahu celého vstupního textu, který se dále větví na menší a menší podintervaly, a to do chvíle, než narazí na listové uzly – terminální symboly. Optimalizovanějším řešením by tak bylo prohledávat jenom tu část stromu, která zcela jistě obsahuje daný interval a přeskočit tak větve stromu, které zcela jistě interval obsahovat nemohou.

Díky tomu, jak funguje syntaktický analyzátor, jsou uzly a intervaly stromu navíc seřazeny zleva doprava od nejlevější pozice v textu po nejpravější pozici v textu. To nahrává další možné optimalizaci – když se totiž má vyhledat, ve kterém podintervalu bude vyhledávání uživatelova intervalu pokračovat, muselo by být v neseřazené množině podintervalů využito lineárního vyhledávání. V seřazené množině je ale možné využít metody půlení intervalů a vyhledávání tak dále urychli.

V ideálním světě uživatel označuje text tak, aby vždy zasáhnul do maximálně jednoho listového uzlu stromu. Když se ale uživateli například chybou podaří vybrat takový interval,

který zasahuje do dvou a více uzlů stromu, nastává problém. V takovém případě vyhledávání najde nejbližšího společného předka všech vybraných uzlů a v tomto uzlu musí aplikace nějak dále operovat. Zřejmě nejjednodušším, a i použitým řešením je odstranit všechny potomky daného uzlu a transformovat pravidlo daného uzlu podle zvolené akce, jako by tyto potomky pravidlo nikdy nemělo. Chytřejším řešením by mohlo být nalezení a sloučení pouze těch přímých potomků, kterých se to týká. Doplňkem k těmto akcím by mohlo být vyžádání přímého souhlasu uživatele, který mohl udělat chybu, ale také nemusel, a naopak mohlo být jeho záměrem přepsat potomky ve stromu, protože udělal chybu v jiné fázi odvozování pravidel.

Ilustraci efektivnější varianty vyhledávání v derivačním stromu ukazuje na konkrétním příkladu obrázek 4.1.

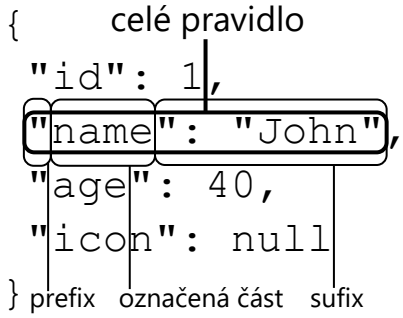


Obrázek 4.1: Ilustrace efektivnějšího vyhledávání uživatelem označeného textu `ne` v rozsahu 2–3, kdy se při vyhledávání nemusí procházet celá větev `list-item`, u které je jasné, že neobsahuje hledaný text na základě vyloučení rozsahu. Červeně jsou označeny vyloučené části, žlutozeleně uzly, které sice text obsahují, ale nejsou minimální a zeleně uzel, který text obsahuje a zároveň má nejmenší možný rozsah.

4.2 Odvození pravidla rozdělením stávajícího

Nejtriviálnější akcí, kterou může uživatel s gramatikou provést, je odvodit nové pravidlo tím, že rozdělí nějaké stávající pravidlo za účelem pojmenování a přiřazení vlastností rozdělované části. Tím, že uživatel označí nějakou část textu, která odpovídá nějakému pravidlu, lze takovéto pravidlo rozdělit až na 3 části – prefix, označenou část a sufix. Jak takové rozdělení vypadá, ukazuje obrázek 4.2.

Na základě rozsahu označeného textu může některá z částí zcela chybět. Ať už je ale označení textu jakékoliv, vždy alespoň jedna část zbude. V případě, že taková část chybí, tedy má nulovou délku, může být bez obav zahozena a nemusí se do výsledné množiny pravidel vůbec promítnout.



Obrázek 4.2: Ukázka, jak lze konkrétní text v jednom pravidlu rozdělit na 3 části – prefix, uživatelem označenou část a sufix.

Jakmile má aplikace k dispozici všechny části nenulové délky, může pro ně aplikace vytvořit nové pravidlo. V této fázi může přijít na řadu nějaký odhad, jaký typ by nově vytvářené pravidlo mohlo mít a podle toho může aplikace nabídnout uživateli případnou otázku ve stylu: „Vybraný text vypadá jako číslo, chcete nastavit typ pravidla na číslo?“ Aplikace mu také může případně rovnou vypsat všechny typy pravidel, kterým vybraný text odpovídá, a na uživateli by bylo pouze vybrat, co z nabízených možností se mu hodí.

Nejjednodušší a také použitou variantou je použít prosté pravidlo literálu, jehož obsahem bude přesně daná část textu, ať už se jedná o prefix, označenou část nebo sufix. Pro začátek se toto jeví jako vhodná varianta, protože díky ní bude gramatika generovat přesně stejný konkrétní jazyk a nezmění ho na obecnější variantu. Změnu na obecnější variantu ale může později učinit ručně uživatel.

Jiná situace nastává, pokud uživatel nepožaduje vytvoření nového pravidla, ale sděluje, že označená část textu je již existující pravidlo. V takovém případě je nutné přidat označený text jako alternativu k již existujícímu pravidlu. To je jednoduché, pokud typ existujícího pravidla je již alternativa – stačí pouze novou možnost výběru vytvořit jako nové pravidlo typu literál a přidat do již existujícího pravidla referenci na ni. Složitější situace nastává, pokud existující pravidlo není typu alternativa. V ten moment je nutné toto pravidlo nejprve vzít, zkopírovat a následně změnit typ originálního pravidla na alternativu, která má jako jediný prvek právě zkopírované pravidlo. Pak už je možné pokračovat standardně jako v předešlém kroku, kdy už na začátku bylo existující pravidlo typu alternativa.

Optimalizací tohoto postupu, která byla také implementována, je zapojení do procesu již existující komponentu – syntaktický analyzátor. Aby se nemusely zbytečně vytvářet další a další alternativy, je možné se nejprve zeptat syntaktického analyzátoru, zda čistě náhodou neodpovídá označený text právě tomuto existujícímu pravidlu. Pokud odpovídá, není nutné žádné alternativy ani nové pravidlo vytvářet a je možné po vygenerování případného prefixu a sufixu skočit ihned na poslední krok.

V poslední fázi už pouze stačí změnit typ aktuálního pravidla, ve kterém se nachází označený text, na operaci zřetězení. Jako operandy této operace se použijí všechny části nenulové délky, přičemž musí být zachováno pořadí, v jakém se tyto části vyskytly ve vstupním textu.

Konkrétní příklad odvození pravidla rozdělením stávajícího startovního pravidla ukazuje obrázek 4.3.

```
1 start = "{ a: 1 }"
```

```
1 start          = content-prefix content content-suffix
2 content-prefix = "{"
3 content        = " a: 1 "
4 content-suffix = "}"
```

Obrázek 4.3: Příklad odvození pravidla `content` rozdělením stávajícího pravidla `start`. Části pravidel odpovídající uživatelem označené části zdrojového textu jsou vyznačeny černými obdélníky.

4.3 Odvození seznamu položek s oddělovačem

Častou konstrukcí v programovacích a jiných podobných jazycích jsou seznamy oddělené nějakým oddělovačem. Zřejmě nejčastějšími takovými seznamy mohou být prvky pole oddělené čárkou nebo příkazy oddělené středníkem, případně znakem nového řádku. Ze syntaktického hlediska tak, jak je pravidlo seznamu navržené a vytvářené touto akcí generátoru, se jedná o neuspořádanou množinu stejných neterminálních pravidel, které mezi sebou mají povinně právě jedno oddělující neterminální pravidlo[1]. Nezáleží tak na pořadí, v jakém se pravidla ve vstupním textu vyskytnou, což znamená, že použitím této akce se z konkrétní gramatiky stává gramatika obecnější, tedy gramatika, která je schopná vygenerovat více řetězců než její původní konkrétnější varianta.

Akce pracuje s předpokladem, že uživatel označí všechny položky generovaného seznamu pomocí vícenásobné výběru a všechny tyto položky se budou nacházet v jednom stejném pravidle. Poslední zjednodušující podmínku by bylo možné odstranit, pro začátek ji ale lze ponechat platnou. Stejného efektu je totiž možné dosáhnout i tak, že se nejprve vytvoří seznam všech položek v jednom pravidle a následně se proces zopakuje i v jiném pravidle, ale tentokrát už s existujícím pravidlem seznamu.

Pokud se jedná o nově vytvářené pravidlo seznamu, nevytváří se pouze ono samotné, ale k němu se vytvoří i dvě pravidla typu alternativa – jedno pro všechny položky seznamu a druhé pro všechny oddělovače, kterých může být i více. Ať už se pak jedná o nově vytvořené pravidlo seznamu nebo i již existující, přidávají se do těchto podřízených pravidel alternativ nové položky a oddělovače, a to podle stejného algoritmu, jako se přidávají v případě odvozování pravidla rozdělením stávajícího, viz sekce 4.2.

Jak získat jednotlivé položky do seznamu je jasné – jedná se o všechny části textu, které uživatel označil. Tato informace ale stačí i k tomu, aby bylo možné určit, jaké používá daný seznam oddělovače. Ty lze získat tak, že se vezmou vždy dvě bezprostředně po sobě následující označené části textu a text, který je mezi nimi, se použije jako oddělovač. Navíc díky tomu, jak funguje přidávání nových alternativ ve spolupráci se syntaktickým analyzátorem, by se nemělo stát, že se v alternativách vyskytne stejný oddělovač vícekrát.

Jakmile je pravidlo seznamu vytvořeno a obsahuje všechny položky a oddělovače, je možné použít předchozí akci – odvození pravidla rozdělením stávajícího. Vzhledem k tomu,

[1] Toto oddělující neterminální pravidlo může být libovolné, přípustné je tedy i ϵ pravidlo

že už se bude jednat o existující pravidlo a toto pravidlo seznamu bylo vytvořeno přesně podle vybraných částí textu, akce k tomuto pravidlu seznamu přidá pouze prefix a sufix, což je přesně požadované chování.

Příklad odvození seznamu položek je možné vidět na obrázku 4.4.

```
start = "{a: 1, b: 2, c: 3}"
```

```
start             = list-prefix list list-suffix
list-prefix       = "{"
list              = list-items *(list-separators list-items)
list-items        = list-items-1 / list-items-2 / list-items-3
list-items-1      = "a: 1"
list-separators   = list-separators-1
list-separators-1 = ", "
list-items-2      = "b: 2"
list-items-3      = "c: 3"
list-suffix       = "}"
```

Obrázek 4.4: Příklad odvození seznamu `list`. Části pravidel odpovídající uživatelem označeným částem zdrojového textu jsou vyznačeny černými obdélníky.

4.4 Odvození společné části více pravidel

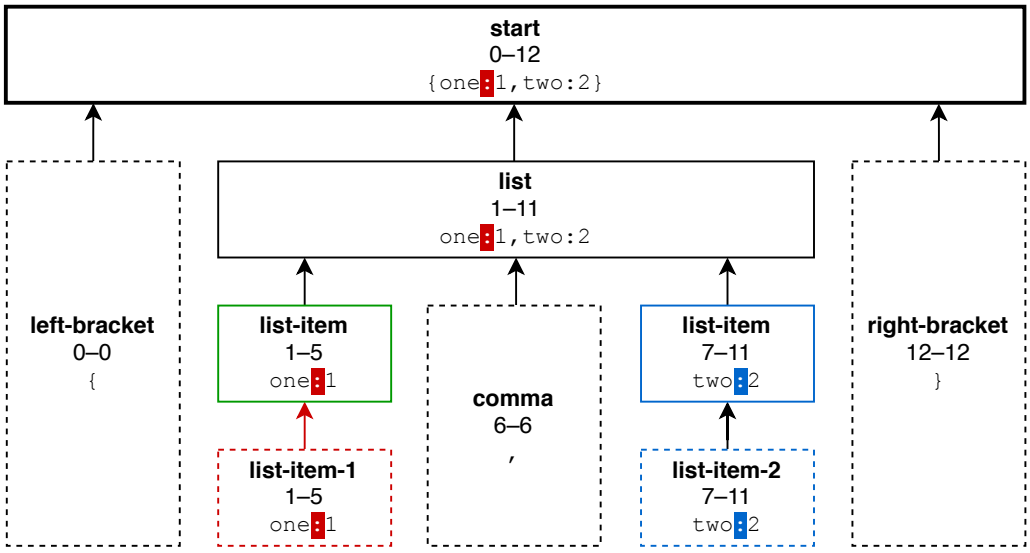
Když uživatel vytvoří pravidlo seznamu, může se stát, že jednotlivé položky budou mít podobnou strukturu a mohou tak obsahovat nějakou společnou část. Často se tak děje například u asociativních polí, kdy nějaký znak nebo skupina znaků odděluje klíč od hodnoty. Například v případě jazyka JSON to může být dvojtečka: `{"a": 1, "b": 2}`, ale používaná je i kombinace znaků rovná se a větší než `=>`, které dohromady symbolizují šipku, jako například v jazyce PHP: `["a" => 1, "b" => 2]`.

Pro usnadnění odvozování pravidel z takovýchto společných částí je uživateli k dispozici právě tato akce. Aby ji mohl uživatel úspěšně použít, stačí mu pouze vybrat jeden takovýto společný symbol, vybrat pro něj vhodný název a akci vykonat.

Pokročilejší možností by bylo využít algoritmu pro hledání nejdelšího společného podřetězce [8], kde by uživateli stačilo pouze přesunout kurzor do pravidla, ve kterém chce vyhledávání započít. Aplikace by pak sama našla nejdelší společný řetězec ve všech položkách seznamu, resp. obecně ve všech potomcích nejbližšího pravidla alternativy.

Akce si nejprve zjistí, co přesně za část textu uživatel vybral a ve které části derivačního stromu tak učinil. Následně se aplikace pokusí procházením po rodičích najít nejbližšího předka, který byl přepsán podle již zmíněného libovolného pravidla typu alternativa. Potomci takovéhoto pravidla jsou pravděpodobným zdrojem různých alternativ položek seznamu a jsou proto cílem dalšího zkoumání. Aby nebylo vyhledávání těchto potomků limitováno pouze na tu část stromu, ve které zrovna uživatel označil daný společný symbol, jsou v derivačním stromu vyhledány všechny uzly a jejich potomci, které byly přepsány

podle nalezeného pravidla typu alternativa. Jak takové hledání vypadá v nějakém konkrétním derivačním stromu, ilustruje obrázek 4.5.



Obrázek 4.5: Příklad hledání společné dvojtečky. Červené hledání dvojtečky od kořene `start` až do v tomto případě terminálního pravidla `list-item-1` je prováděno standardní metodou vyhledání pravidla podle označené části textu, která je popsána v sekci 4.1. Následně je nalezen nejbližší předek typu alternativa – zelený `list-item`. Ten je poté vyhledán v celém stromu a nalezen dále jako modrý `list-item`, přičemž jeho potomek `list-item-2` obsahuje další dvojtečku.

V potomcích všech uzlů přepsaných podle nalezeného pravidla typu alternativa je poté nutné najít uživatelem vybranou část textu. Vzhledem k tomu, že na text těchto potomků je nutné nahlížet jako na nezpracovanou část textu, nelze v případě vícenásobného výskytu uživatelem označené části textu v těchto potomcích přesně určit, který výskyt měl uživatel na mysli. V takovém případě je možné učinit nějaký odhad, který výskyt použít, a to například podle pozice výskytu v původním uzlu, ve kterém uživatel část textu označil. Nejjednodušší a také použitou možností je vzít první nalezený výskyt. Situaci ilustruje obrázek 4.6.

```
1 {
2     ":id:": 1,
3     ":name:": "Syxard: Syntax Wizard",
4     ":published_at:": "2020-05-28T23:59:59"
5 }
```

Obrázek 4.6: Ukázka rozhodování, kterou dvojtečku v jednotlivých ohraničených pravidlech vybrat, když uživatel vybral dvojtečku v černém obdélníku. Modře jsou aplikací skutečně vybrané dvojtečky, zeleně uživatelem pravděpodobně myšlené další dvojtečky a červeně dvojtečky, které uživatel pravděpodobně neměl na mysli.

Vyhledáním a vybráním příslušných výskytů původně označeného textu tak v podstatě vznikne množina rozsahů ve vstupním textu nebo také množina označených textů. Tato množina by mohla vzniknout i prostým označením částí textu ze strany uživatele. Množinu tak lze dále zpracovat již pomocí předchozí navržené akce – pomocí odvození pravidla rozdělením stávajícího, které je popsáno v sekci 4.2. Mimo jiné i proto by bylo možné tuto akci zaměnit také za určitou formu chytrého asistenta výběru textu, kdy by akce uživateli pouze po jejím spuštění vybrala společné části textu a další případné úpravy výběru a následného vykonání další příslušné akce by ponechala na uživateli.

Konkrétní příklad odvození společné dvojtečky pro více pravidel ukazuje obrázek 4.7.

```
1 list      = item *(separator item)
2 item      = item-1 / item-2 / item-3
3 item-1    = "a:1"
4 separator = ","
5 item-2    = "b:2"
6 item-3    = "c:3"
```

```
1  list           = item *(separator item)
2  item           = item-1 / item-2 / item-3
3  separator      = ","
4  item-1         = colon-prefix colon colon-suffix
5  item-2         = colon-prefix-1 colon colon-suffix-1
6  item-3         = colon-prefix-2 colon colon-suffix-2
7  colon-prefix   = "a"
8  colon          = ":"
9  colon-suffix   = "1"
10 colon-prefix-1 = "b"
11 colon-suffix-1 = "2"
12 colon-prefix-2 = "c"
13 colon-suffix-2 = "3"
```

Obrázek 4.7: Příklad výběru jediné dvojtečky v pravidle `item-2`, přičemž aplikace nalezne a odvodí i ostatní dvojtečky. Části pravidel odpovídající uživatelem označené části zdrojového textu jsou vyznačeny černými obdélníky.

4.5 Sloučení prefixu nebo sufixu označeného textu

Při využití akce odvození společné části více pravidel popsané v sekci 4.4, a to zvláště na asociativní pole, může vzniknout větší množství pravidel, které se od ostatních liší pouze prefixem nebo sufixem nově vzniklého pravidla společné části. Pro zjednodušené sloučení takovýchto částí je uživateli k dispozici akce sloučení prefixu nebo sufixu uživatelem označené části textu. Jak taková pravidla s rozdílnými sufixy mohou vypadat, ukazuje obrázek 4.8.

```
1 list-items-1 = id comma whitespace number
2 list-items-2 = id comma whitespace string
3 list-items-3 = id comma whitespace null
```

Obrázek 4.8: Příklad pravidel, která se liší pouze v pravidle nacházejícím se za pravidlem `whitespace`.

Uživatel pro použití akce pouze označí tu část textu, která odpovídá společnému pravidlu pro více ostatních pravidel se společným prefixem nebo sufixem, tyto společné prefixy nebo sufixy nějak nazve a zvolí provedení akce.

Ze začátku akce probíhá velmi podobně jako v případě odvození společné části více pravidel. Nejprve aplikace zjistí, v jaké části derivačního stromu uživatel text označil. Následně je nalezen nejbližší společný předek přepsaný podle pravidla typu alternativa. Poté již ale pokračuje aplikace odlišně. Aplikace si zjistí, jaké všechny alternativní pravidla obsahuje nalezené pravidlo typu alternativa. V těchto alternativních pravidlech pak vyhledá taková pravidla, která jsou typu zřetězení a obsahují právě ono uživatelem označené pravidlo.

Aplikace v této fázi vytvoří pravidlo pro společné prefixy nebo sufixy. Pokud již takové pravidlo existuje a není-li typu alternativa, je pravidlo změněno na typ alternativa a původní pravidlo je zkopírováno jako jedna z alternativ tohoto změněného pravidla. Toto se tedy děje podobně jako v případě akce odvození pravidla rozdělením stávajícího, které je blíže popsáno v sekci 4.2.

Jakmile má aplikace vše připraveno, může začít přidávat alternativní prefixy nebo sufixy do připraveného pravidla pro alternativy. To může provést tak, že si v každém nalezeném pravidlu typu zřetězení vyžádá seznam pravidel, které se mají zřetězit, a zde nalezne pozici, na které se nachází uživatelem označené pravidlo. Všechna pravidla, která jsou před, resp. za označeným pravidlem jsou poté prefixová, resp. sufixová pravidla. Pokud je takovéto pravidlo pouze jedno, lze ho ihned přidat jako alternativu do připraveného pravidla pro alternativy. Pokud je ale takových pravidel více, je nutné nejprve vytvořit nové pravidlo zřetězení, které bude ve správném pořadí obsahovat všechna tato prefixová, resp. sufixová pravidla, a až toto pravidlo je možné přidat do připraveného pravidla pro alternativy.

Poté, co jsou všechna prefixová, resp. sufixová pravidla z jednoho pravidla zřetězení přidána do připraveného pravidla pro alternativy, je možné tato pravidla nahradit v původním pravidlu zřetězení právě tímto připraveným pravidlem pro alternativy. Tímto krokem je slučování společných prefixů, resp. sufixů dokončeno.

Příklad ukazující sloučení odlišných sufixů pravidla je možné vidět na obrázku 4.9.

```
1 list-items-1 = id comma whitespace values
2 list-items-2 = id comma whitespace values
3 list-items-3 = id comma whitespace values
4 values       = number / string / null
```

Obrázek 4.9: Příklad sloučení odlišných pravidel z obrázku 4.8 do pravidla `values`. Uživatel musel označit pouze jediné pravidlo `whitespace` ve zdrojovém textu a spustit akci. Pravidla se stejnou pravou stranou, která po sloučení vznikla, dokáže optimalizovat optimalizační pravidlo popsané v sekci 5.3.

Kapitola 5

Optimalizace množiny pravidel

Ačkoliv se komponenta generátoru z kapitoly 4 může snažit generovat v určitých ohledech optimalizovanou množinu pravidel z hlediska počtu neterminálů, ne vždy to musí být pro generátor nejlepší řešení. Jednoduchost implementace nebo výhodnost neoptimalizovaného zápisu pravidel, ve kterém existuje více specifických avšak ve výsledku nepotřebných neterminálů, může občas převážit generování co nejoptimálnější množiny pravidly. Navíc díky manuálním zásahům uživatele do množiny pravidel mohou vzniknout další neoptimalizované konstrukce, které by při pouhém generování nemusely vzniknout.

V takovém případě přichází na řadu komponenta nazvaná jako optimalizátor. Úkolem této komponenty je vzít již existující množinu pravidel a tu dále transformovat pomocí optimalizačních pravidel. Tato optimalizace je přitom prováděna do té doby, dokud se nedosáhne tzv. pevného bodu, tedy stavu, kdy je výsledkem aplikace všech optimalizačních pravidel opět stejná množina optimalizovaných pravidel jako v předchozím kroku. Dosažení pevného bodu znamená, že se množina pravidel už dále nemění, tedy že už ji žádné optimalizační pravidlo nezměnilo a již není pomocí použitých optimalizačních pravidel dále co optimalizovat.

Jedním z nejjednodušších způsobů dosažení pevného bodu je opakovaná aplikace všech optimalizačních pravidel, dokud proběhla alespoň jedna změna. Ačkoliv existují i efektivnější způsoby, jak pevného bodu dosáhnout, které zahrnují například topologické uspořádání optimalizačních pravidel, pro tento účel to není vzhledem k nízkému počtu optimalizačních pravidel nezbytné.

Následující sekce popisují jednotlivá optimalizační pravidla, která jsou v komponentě optimalizátoru použita. V zásadě každé z těchto pravidel pracuje ve dvou fázích – vyhledání kandidátů na optimalizaci a optimalizace těchto kandidátů.

5.1 Odstranění aliasů pravidel

Alias pravidla je taková pravidlo, které má na pravé straně pravidla pouze jedno další pravidlo. Častým příkladem takového pravidla může být například pravidlo `list-separators = list-separator-1`, které může vzniknout například při odvození pravidla seznamu, který má pouze jeden oddělovač položek.

Alias pravidla může vzniknout v těchto 4 případech:

- pravidlo alternativy, které má jako alternativu právě jedno pravidlo;
- pravidlo zřetězení, které má řetězit právě jedno pravidlo;

- pravidlo opakování, které se má opakovat minimálně i maximálně jedenkrát, tedy právě jednou;
- pravidlo seznamu, jehož položky se mají opakovat minimálně i maximálně jedenkrát, tedy právě jednou.

Jakmile aplikace vyhledá všechna tato kandidátní pravidla, může proběhnout jejich nahrazení pravidlem, pro které tento alias existuje. To lze učinit prohledáním všech neterminálních pravidel a přepsáním reference na tento alias skutečným pravidlem, na které tento alias odkazuje. Po nahrazení všech referencí na alias může být alias bezpečně smazán.

Konkrétní příklad odstranění aliasu v gramatice, která mohla vzniknout při běžné práci uživatele, kdy uživatel pojmenoval stejnou část vstupního řetězce v podstatě dvakrát, ukazuje obrázek 5.1.

```
1 start          = content-prefix content content-suffix
2 content-prefix = "{"
3 content        = list
4 list           = list-items *(list-separators list-items)
5 content-suffix = "}"
```

```
1 start          = content-prefix list content-suffix
2 content-prefix = "{"
3 list           = list-items *(list-separators list-items)
4 content-suffix = "}"
```

Obrázek 5.1: Příklad odstranění aliasu `content`, který je aliasem pravidla `list`.

5.2 Odstranění duplicitních pravidel v pravidle alternativy

Duplicitní alternativní pravidlo v pravidle alternativy může vzniknout samotnou činností komponenty optimalizátoru, například při odstraňování aliasů nebo pravidel se stejnou pravou stranou. Mohou ale také vzniknout ručních zásahem uživatele, kdy uživatel nedopatřením vybere jako alternativu již jednou vybrané pravidlo. Tato stejná pravidla nemá smysl mít jako alternativy v jednom pravidle alternativ vícekrát – pokud nebylo pravidlo použito v prvním případě, nebude použito ani v žádném z následujících.

Aby optimalizační pravidlo odstranilo tyto duplicity, stačí mu pouze vyhledat v neterminálních pravidlech všechna pravidla alternativ a zde odstranit duplicity použitím vestavěné nebo jiné knihovní funkce pro eliminaci duplikátů.

Odstranění duplicitního pravidla v pravidle alternativy, které mohlo vzniknout sloučením prefixů některého pravidla, ukazuje na konkrétním příkladu obrázek 5.2.

```
1 list-items = item-1 / item-2 / item-1
```

```
1 list-items = item-1 / item-2
```

Obrázek 5.2: Příklad odstranění duplicitního pravidla `item-1` v pravidle alternativy `list-items`.

5.3 Odstranění pravidel se stejnou pravou stranou

Pravidla, která mají naprosto stejnou pravou stranu, lze považovat za duplicitní a je možné tak ponechat pouze jedno z nich. Výskyty těchto duplicitních pravidel v jiných pravidlech je poté možné nahradit ponechaným pravidlem a tato duplicitní pravidla je možné bezpečně odstranit.

Toto optimalizační pravidlo lze na první pohled zaměnit s optimalizačním pravidlem, které odstraňuje aliasy, resp. se může zdát, že je možné ho použít místo něj. Na rozdíl od něj ale pravidlo ponechává jedno z duplicitních pravidel, které je poté použito všude místo všech ostatních duplicitních pravidel. Nedojde tak k odstranění případného posledního aliasu, splňuje-li vůbec toto pravidlo definici aliasu, a tento alias musí být odstraněn právě pravidlem pro odstraňování aliasů.

Duplicitní pravidla mohou typicky vznikat ručním zásahem uživatele, kdy může například zvolit, že pravidlo pro identifikátory je řetězec s výchozím nastavením, a stejnou možnost zvolí i pro skutečné řetězce. Taková pravidla nemusí být problém jen pro velikost výsledné množiny pravidel, ale mohou způsobovat také problémy lexikálním analyzátorům, které nepočítají s existencí těchto pravidel.

Vyhledání kandidátních duplicitních pravidel, které je možné nahradit a následně smazat, odpovídá operaci *join*, resp. přesněji *self join*, která je známa z relačních databází. Ta spočívá ve spojení stejné tabulky sama se sebou, což v tomto případě znamená spojení množiny pravidel. Implementačně lze toto vyřešit mj. i prostým procházením pravidel, přičemž při každém průchodu jsou znovu procházena ta samá pravidla. Při takovémto procházení jsou obě pravidla porovnána a v případě, že jsou pravé strany obou pravidel shodné a kandidátní pravidlo následuje po ponechaném pravidle, je možné takovéto pravidlo nahradit a odstranit. [19]

Příklad tabulky ukazující, jak může vypadat operace *self join* pro 3 pravidla, ukazuje tabulka 5.1. Příklad odstranění pravidel přímo na gramatice je možné vidět na obrázku 5.3.

5.4 Spojení vnořených pravidel alternativy a zřetězení

Existují-li dvě pravidla, která mají stejný typ alternativa nebo zřetězení, a existuje-li mezi nimi vztah rodič–potomek, lze takového potomka vložit přímo do rodiče. Jelikož jsou obě pravidla stejného typu, není většinou nutné mít tato pravidla rozdělená do dvou. Významným případem, kdy rozdělení do dvou pravidel dává smysl, je skutečnost, kdy je pravidlo vystupující jako potomek použito nejen ve svém rodiči, ale ještě také v jakémkoliv jiném dalším pravidle. I kdyby bylo možné v takovém případě provést vložení pravidla do všech

#	1. pravidlo	2. pravidlo
1.	`start = item-1 / item-2`	`start = item-1 / item-2`
2.	`start = item-1 / item-2`	`item-1 = id comma values`
3.	`start = item-1 / item-2`	`item-2 = id comma values`
4.	`item-1 = id comma values`	`start = item-1 / item-2`
5.	`item-1 = id comma values`	`item-1 = id comma values`
6.	**`item-1 = id comma values`**	**`item-2 = id comma values`**
7.	`item-2 = id comma values`	`start = item-1 / item-2`
8.	`item-2 = id comma values`	`item-1 = id comma values`
9.	`item-2 = id comma values`	`item-2 = id comma values`

Tabulka 5.1: Příklad operace *self join* pro 3 pravidla. Hledána je přitom zvýrazněná kombinace na základě podmínky, kdy se pravidla liší v názvu, ale mají stejnou pravou stranu a první pravidlo předchází druhému pravidlu.

```
1 start  = item-1 / item-2 / item-3
2 item-1 = id comma whitespace values
3 item-2 = id comma whitespace values
4 item-3 = id comma whitespace values
5 values = number / string / null
```

```
1 start  = item-1 / item-1 / item-1
2 item-1 = id comma whitespace values
3 values = number / string / null
```

Obrázek 5.3: Příklad odstranění pravidel `item-2` a `item-3`, které mají stejnou pravou stranu jako pravidlo `item-1`.

odkazujících pravidel, znamenalo by takové vložení spíše nárůst velikosti výsledné množiny pravidel než její snížení.

I toto optimalizační pravidlo může na první pohled připomínat vylepšenou variantu pravidla pro odstraňování aliasů – vždyť také dokáže v případě nalezení něčeho, co odpovídá definici aliasu, tento alias vzít a vložit jeho obsah na místo výskytu toho aliasu. Potíž je ale v tom, že toto optimalizační pravidlo vzhledem k faktu, že dokáže takto vkládat více pravidel, umožňuje provést vložení pouze do rodičů stejného typu. Takoví rodiče totiž předpokládají, že mohou obsahovat jako potomky více pravidel, což většina ostatních pravidel neumožňuje a vložení by tak nebylo možné. Navíc toto optimalizační pravidlo provádí vkládání pouze pokud existuje jediný odkazující rodič, což v případě pravidla pro odstraňování aliasů neplatí.

Pravidla, která je možné takto optimalizovat, vznikají poměrně jednoduše činností komponenty generátoru. Když totiž generátor odvozuje pravidlo rozdělením stávajícího pravidla, vznikají na sebe navázaná pravidla zřetězení, které obsahují vždy maximálně 3 další pravidla – prefix, uživatelem označenou část a sufix. Generátor by sice pravděpodobně mohl

generovat již optimalizovanou variantu, ale na takový přístup by mohl doplatit uživatel. Pokud totiž udělá chybu někde na začátku procesu vytváření gramatiky a nyní potřebuje označením přesáhnout více už existujících pravidel, v případě přesažení potomků jednoho dlouhého pravidla zřetězení by přišel při současném nastavení o podstatě větší kus práce, než kdyby tato pravidla byla jednodušší, tzn. měla by maximálně 3 části.

Nalezení kandidátních pravidel pro vložení do rodiče spočívá nejprve ve vyhledání všech referencí na pravidla typu alternativa nebo zřetězení. Pokud jsou taková pravidla referencována právě jednou a zároveň mají jako rodiče pravidlo stejného typu, je možné provést vložení a následné odstranění dotyčného pravidla. Operace vložení přitom odpovídá operaci známé jako *splice*, která spočívá v odstranění části pole a nahrazení této části jinými elementy [13]. V tomto případě je tedy pravidlo vystupující jako potomek nahrazeno všemi potomky tohoto potomka.

Příklad vnoření pravidel typu zřetězení, které vznikají při činnosti generátoru, ukazuje obrázek 5.4.

```
1 start          = content-prefix content content-suffix
2 content-prefix = "{"
3 content        = item-prefix item item-suffix
4 item-prefix    = "a"
5 item           = ": "
6 item-suffix    = "1"
7 content-suffix = "}"
```

```
1 start          = content-prefix item-prefix item item-suffix content-suffix
2 content-prefix = "{"
3 item-prefix    = "a"
4 item           = ": "
5 item-suffix    = "1"
6 content-suffix = "}"
```

Obrázek 5.4: Příklad vnoření pravidel pravé strany pravidla `content` do pravidla `start`.

5.5 Odstranění nepoužívaných pravidel

Nepoužívaná či mrtvá pravidla mohou vzniknout například ručním zásahem uživatele do množiny pravidel. Typicky se tak může dít u terminálních symbolů, kdy uživatel v textu označí několik terminálních symbolů, které jsou z hlediska doslovného porovnání odlišné, přestože reprezentují jeden terminální symbol. Příkladem mohou být čísla. Uživatel vybere čísla 1 a 2 a oznámí aplikaci, že se jedná o stejný terminální symbol – číslo. Jelikož ještě aplikace nemá k dispozici žádné informace o těchto hodnotách, při porovnání zjistí, že se jedná o dvě různé hodnoty a vytvoří pro ně tak pravidlo alternativy. Později přichází uživatel, který ale určí, že toto pravidlo není typu alternativa, nýbrž typu číslo. V ten

moment vznikají dvě nepoužitá pravidla typu literál – jedno pro číslo 1 a druhé pro číslo 2. Odstraněním těchto pravidel se zabývá právě toto optimalizační pravidlo.

Funkční podstata tohoto optimalizačního pravidla by se dala přirovnat k tzv. *garbage collection*, což je způsob automatické správy paměti [20]. Stejně jako u něj vznikají v průběhu práce s aplikací nepotřebná pravidla, která je možné bez problému odstranit. Pro nalezení takových kandidátních pravidel k odstranění je přitom možné využít stejných algoritmů jako v případě *garbage collection*. Jedním z nejjednodušších algoritmů je tzv. počítání referencí. Jelikož ale tento algoritmus nedokáže, na rozdíl od jiných algoritmů, najít a odstranit cyklické reference, které se v bezkontextových jazycích mohou vyskytnout, je pro tento účel lepším řešením algoritmus *mark and sweep*.

Následující odstavce představují stručné shrnutí podstaty a vlastností algoritmu *mark and sweep* tak, jak je uvádí publikace [9].

Algoritmus *mark and sweep* využívá pro svůj chod některého z algoritmu pro průchod grafem, například prohledávání do hloubky. V každém průchodu cyklu má algoritmus k dispozici jeden uzel, resp. pravidlo, do kterého se bylo možné nějakým způsobem dostat z kořene stromu, resp. ze startovního pravidla. Tento uzel poté označí jako navštívený, a tedy nějakým způsobem dostupný. Jakmile má algoritmus takto označené všechny dostupné uzly, jako neoznačené mu zbudou ty uzly, ke kterým se nelze nijak dostat. Tyto uzly je pak možné bezpečně odstranit.

Výhodou algoritmu *mark and sweep* je jeho přirozená schopnost poradit si s cyklickými referencemi a odstranit takováto cyklická pravidla. Nevýhodou je, že program musí být pozastaven, aby mohl být takovýto úklid proveden. Tato skutečnost ale v tomto případě prakticky nehraje roli, jelikož program už je sám o sobě pozastaven jen kvůli provedení ostatních optimalizačních pravidel, a to navíc na žádost samotného uživatele.

Konkrétní příklad odstranění nepoužívaných pravidel, které mohly vzniknout ručním zásahem uživatele do gramatiky skrze panel pro úpravu pravidel, ukazuje obrázek 5.5.

```
1 number = *DIGIT
2 number-1 = "1"
3 number-2 = "2"
```

```
1 number = *DIGIT
```

Obrázek 5.5: Příklad odstranění nikde nepoužívaných pravidel `number-1` a `number-2`. Startovní pravidlo – v tomto případě `number` – odstraněno není.

Kapitola 6

Generování výstupní gramatiky a kódu

Složitost generování výstupní gramatiky do některé varianty Backus–Naurovy formy popsané v sekci 2.3 se liší podle vyjadřovacích schopností zvolené varianty. Pro začátek se jako nejjednodušší jeví varianta ABNF, jejíž generování popisuje sekce 6.1, a to zejména díky pokročilé podpoře zápisu terminálních symbolů. Aplikace ale počítá s rozšířením také o další případné výstupní formáty, čehož bylo využito i pro generování pseudokódu popsaného v sekci 6.2, který by měl uživateli pomoci se v generované gramatice lépe orientovat.

Základem pro generování výstupní gramatiky je množina pravidel, jejichž vnitřní reprezentace odpovídá popisu datového modelu tak, jak je popsán v sekci 3.2 a vyobrazen na obrázku 3.2. Každé pravidlo se pak zpracovává individuálně podle jeho typu. Aby se však uživateli co nejvíce usnadnilo čtení takto generované gramatiky, používají se pokud možno symboly, které by uživatel mohl znát, než ty, které tak známé být nemusejí, ale mohou být jednodušší na vygenerování. Příkladem může být známější symbol `*` značící opakování 0 až ∞, který by ale mohlo být jednodušší vygenerovat jako `{0,}`.

Všechny generované výstupní gramatiky používají pro samotné znaky i znakové kódy kódování UTF-8, které bylo zvoleno pro svou rozšířenost a také proto, že dokáže reprezentovat velkou množinu různých znaků včetně znaků češtiny.

6.1 Specifika generování ABNF

Pravidla typu alternativa a zřetězení lze v ABNF z vnitřní reprezentace vygenerovat poměrně jednoduše a přímočaře prostým poskládáním neterminálů za sebe a proložením takového poskládání příslušnými operátory. Podobně jednoduchým je i pravidlo volitelnosti, jehož volitelné pravidlo stačí umístit do hranatých závorek.

Zajímavější jsou pravidla, která obsahují konfigurovatelné opakování: bílé znaky, opakování, seznam a seznam znaků. Pro opakování totiž v ABNF existuje několik možností zápisu, které by měly usnadnit čtení. Proto využívají tato pravidla mapovací tabulku podobnou tabulce 6.1.

6.1.1 Specifika opakování položek seznamu

U seznamu je potřeba si dát pozor na skutečnost, že položky jsou oddělené nějakým oddělovačem. Pro výpis opakování části `separator item` je tak nutné od celkového počtu opakování položek jeden výskyt, který před sebou nemá tento oddělovač, odečíst. Tuto část

Min	Max	Zápis
0	1	`[ rule ]`
0	∞	`*rule`
0	<m>	`*<m>rule`
1	1	`rule`
<n>	∞	`<n>*rule`
<n>	<m>	`<n>*<m>rule`

Tabulka 6.1: Tabulka pro podporu rozhodování, jaký styl zápisu bude použit pro opakování pravidla v ABNF na základě minimálního a maximálního počtu opakování.

s oddělovačem tak není třeba vůbec nevypisovat v případě, kdy je maximální počet položek menší než 2, což by pro tuto část znamenalo, že by se neopakovala za žádných okolností. Zároveň je celý seznam potřeba obalit hranatými závorky v případě, že jde o seznam volitelný a nemusí tak obsahovat ani jednu položku.

6.1.2 Doplněk seznamu znaků a řetězce

Seznamy znaků představují pro generování zajímavou výzvu. Kromě zápisu jednotlivých znaků totiž podporují také zápis rozsahů znaků ve formátu `a-z` a tyto znaky mohou představovat jak povolené znaky v případě režimu bílé listiny, tak i znaky zakázané v případě režimu černé listiny. Generování znaků ani rozsahů není samo o sobě díky ABNF žádný problém. Nejinak tomu je i v případě režimu bílé listiny, avšak složitější část přichází u režimu černé listiny.

V režimu černé listiny je nutné najít tzv. doplněk zadané množiny znaků. Regulární výrazy mívají pro tuto operaci obvykle konstrukci s hranatými závorkami a znakem stříšky, tedy například vyloučení rozsahu a–z by se zapsalo jako `[^a-z]`. ABNF bohužel ale takovouto konstrukcí nedisponuje, a proto je nutné vytvořit tento doplněk ručně a zapsat ho pomocí jiných dostupných konstrukcí.

Vzhledem k tomu, že je pro kódování znaků využito UTF-8, doplněk je nutné vytvářet k množině znaků definované právě tímto kódováním. A právě pro vytváření doplňku byl vytvořen programátorsky zajímavý algoritmus. Kromě toho, že v první fázi musí sloučit všechny rozsahy, které se nějakým způsobem překrývají a tyto rozsahy vrátit seřazené, zde existuje i druhá zajímavější fáze. Ta spočívá v postupném procházení sloučených rozsahů a vytváření nových doplňkových rozsahů pouze pokud se počáteční číslo právě procházeného rozsahu nerovná dalšímu číslu, které bezprostředně následovalo předcházejícímu rozsahu. Takto vygenerované doplňkové rozsahy už lze poté přímo vygenerovat v podobně ABNF rozsahů.

Tohoto principu je využito také pro generování pravidla typu řetězec. To pracuje na principu přečtení počátečního znaku uvozujícího řetězec a poté libovolně dlouhé posloupnosti jakýchkoliv znaků, kromě znaku značícího konec řetězce a *escape* znaku.

6.1.3 Náhrada číslice předdefinovaným pravidlem

V případě generování popisu čísla stačí poskládat jeho nakonfigurované části – znaménka, desetinné tečky a podobně – správně za sebe. Pro lepší čitelnost ale existuje i pro číslo

mapovací tabulka, která se týká použití jednoho z předdefinovaných pravidel pro různé základy čísla a je podobná tabulce 6.2.

Základ	**Zápis**
2	`BIT`
10	`DIGIT`
16	`HEXDIG`
<n>	`("0" / "1" / ... / "<n - 1>")`

Tabulka 6.2: Tabulka pro mapování základů čísla na předdefinovaná pravidla číslic, pokud existují.

6.1.4 Převod řetězců do čitelnější podoby

Zajímavým problémem a zároveň motivací pro vznik pseudokódu se ukázal v případě pravidla typu literál převod těchto literálů a dalších řetězců do čitelnější podoby. ABNF sice disponuje konstrukcí pro zápis řetězců nerozlišujících a od RFC 7405 i rozlišujících velikost písmen, ale tento zápis podporuje jen omezenou množinu znaků, přičemž zde chybí podpora například pro velmi používané dvojité uvozovky. Znaky, které nelze pomocí této konstrukce zapsat, je nutné zapisovat pomocí číselných kódů, které ale uživatel dost pravděpodobně znát nebude.

I za těchto podmínek se aplikace při generování snaží zachovat co největší čitelnost. Toho dosahuje tak, že v textu, který má být vypsán jako literál do výstupní gramatiky, hledá pomocí regulárního výrazu co nejdelší nepřerušované posloupnosti znaků, které lze zapsat pomocí ABNF řetězce, a naopak nejdelší nepřerušovanou posloupnost znaků, které takto zapsat nelze. Tyto posloupnosti jsou pak následně generovány příslušnou konstrukcí.

6.1.5 Ošetření názvů pravidel

Názvy pravidel není nutné při generování nijak ošetřovat, jelikož jsou pravidla pro vytváření názvů pravidel v aplikaci nastavena tak, aby odpovídala platným názvům pravidel podle RFC 5234. Aplikace tedy nedovolí uživateli zadat jiné názvy než ty, které začínají malým nebo velkým písmenem anglické abecedy a volitelně pokračují libovolnou kombinací malých a velkých písmen anglické abecedy, čísel a spojovníku.

6.2 Pseudokód a jeho specifika při generování

Pseudokód vznikl jako čitelnější náhrada za generovanou ABNF. Zejména jde o čitelnější terminály – literály a rozsahy znaků. Původní syntaxe vycházela z velké části ze syntaxe ABNF, později byla ale postupným vývojem transformována na styl jazyka JavaScript, resp. z velké části i na styl jazyka C, ze kterého JavaScript vychází.

Pro terminály využívá pseudokód výhradně regulární výrazy, a to všude kromě literálů. Ty jsou zapsány pomocí nejvhodnější varianty řetězce jazyka JavaScript, tedy té varianty, která obsahuje nejméně uvozovek, které by musely být ošetřeny. Na výběr jsou přitom 3 varianty řetězců: `"string"`, `'string'` nebo `` `string` `` a vybírány jsou právě v tomto pořadí. Znaky, které jsou v těchto literálech ošetřeny, zahrnují zpětné uvozovky, návrat vozíku, nový řádek a uvozovky sloužící pro uvození a ukončení řetězce.

Neterminální symboly využívají pro zřetězení stejně jako v případě ABNF pouhou mezeru. Pro alternativy je využit operátor známý jako „nebo“ v podobě dvou svislých čar `||`. Pro opakování jsou využity stejné symboly jako u regulárních výrazů, a tedy i u terminálů. Výčet těchto symbolů a jejich použití v závislosti na počtu opakování ukazuje mapovací tabulka 6.3.

Min	**Max**	**Zápis**
0	1	`rule?`
0	∞	`rule*`
1	1	`rule`
1	∞	`rule+`
<n>	∞	`rule{<n>,}`
<n>	<m>	`rule{<n>,<m>}`

Tabulka 6.3: Mapovací tabulka sloužící pro rozhodování, který znak opakování bude při generování regulárních výrazů využit na základě minimálního a maximálního počtu opakování.

Vzhledem k tomu, že navržený pseudokód je poměrně volný a umožňuje zapsat všechna pravidla poměrně jednoduchý způsobem, neexistují při jeho generování větší problémy. Zpravidla vše lze jednoduše poskládat správným způsobem za sebe, případně pro to využít nějakou konstrukci, kterou nabízejí regulární výrazy. Menším problémem by mohlo být ošetření speciálních znaků, které se vyskytují v regulárních výrazech, ale tento problém poměrně zdařile řeší již existující knihovny. I díky tomu lze poté výsledné vygenerované výrazy využít nejen pro zobrazení v podobně pseudokódu uživateli, ale i pro samotný syntaktický analyzátor.

6.3 Generování kódu pro syntaktický analyzátor

Pro usnadnění integrace vygenerované gramatiky do uživatelova kódu existují tzv. kousky kódu. Ty může uživatel zkopírovat a s minimálním úsilím zahrnujícím případnou instalaci potřebné knihovny ho použít ve svém kódu.

V současné době existují v aplikaci 2 generované kousky kódu: jeden pro jazyk JavaScript s využitím knihovny `apg-js2-exp` a druhý experimentální pro jazyk Python s využitím knihovny `PLY`. Aplikace ale opět počítá s případným rozšířením o další kousky kódu pro další programovací jazyky a knihovny.

6.3.1 Generování pro JavaScript

Generování kousku kódu pro knihovnu `apg-js2-exp` jazyka JavaScript je velmi přímočaré a jednoduché. Knihovna se svým aplikačním rozhraním snaží napodobit práci s nativními regulárními výrazy pro jazyk JavaScript, ale na rozdíl od vstupní gramatiky ve formě regulárního výrazu přijímá na vstupu nadmnožinu ABNF [18]. Stačí si tedy nechat vygenerovat ABNF určené pro zobrazení uživateli a použít ho jako vstupní gramatiku této knihovny. Jediným problémem tak zůstává správné ošetření případných speciálních znaků, pro což ale opět existují knihovny v čele s nativní knihovnou jazyka JavaScript.

Příklad vygenerovaného kousku kódu, který byl vygenerován pro zjednodušenou gramatiku, která generuje pole podobné tomu ve formát JSON, ukazuje obrázek 6.1.

```
1  import ApgExp from "apg-exp";
2
3  const apgExp = new ApgExp
4  (
5    "start         = left-bracket array right-bracket\r\n" +
6    "array         = number *(comma number)\r\n" +
7    "comma         = %s\",\"\r\n" +
8    "number        = [\"+\" / \"-\"] 1*DIGIT [\".\" 1*DIGIT]\r\n" +
9    "left-bracket  = %s\"[\"\r\n" +
10   "right-bracket = %s\"]\"\r\n" +
11   "DIGIT = %x30-39\r\n"
12 );
13
14 const isValid = apgExp.test("=== YOUR INPUT STRING GOES HERE ===");
```

Obrázek 6.1: Příklad kousku kódu v jazyce JavaScript pro gramatiku jednoduchého pole podobného tomu v JSONu, který byl vygenerován z řetězce `[1,2,3]`.

6.3.2 Generování pro Python

Trochu složitějším úkolem se ale ukázalo generování kousků kódu pro knihovnu `PLY` jazyka Python. Knihovna totiž nepoužívá pro specifikaci neterminálů ABNF, ale pouze jistou odnož BNF, které ale chybí lepší podpora pro různá opakování, jak uvádí mimo jiné i sekce 2.3.1. To je mj. i důsledkem toho, že knihovna `PLY` používá pro syntaktickou analýzu `LALR(1)`, při kterém se opakování musí přepsat na levě nebo pravě rekurzivní a ukončující pravidlo, jak ukazuje například pravidlo `p_array` na obrázku 6.2. V současné době jsou tak v aplikaci podporována pouze jednoduchá opakování 1 až nekonečno. [2]

Nejmenší komplikací, a dokonce i možnou výhodou, se například oproti generování ABNF ukázaly být terminály. Ty jsou v knihovně totiž vyjadřovány pomocí regulárních výrazů a je proto možné využít regulární výrazy generované v rámci pseudokódu. Vzhledem k využití lexikálního analyzátoru v této knihovně ale nelze vytvořit regulární výraz terminálu, který by mohl odpovídat i prázdnému řetězci. Terminály s možnou nulovou délkou proto nejsou v současné době při generování kousků kódu pro tuto knihovnu podporovány.

Při generování je také nutné si dávat pozor na názvy identifikátorů generovaných funkcí. Názvy funkcí reprezentují v knihovně `PLY` názvy terminálů a neterminálů, a proto je potřeba takto generované názvy patřičně ošetřit. Naštěstí jediným znakem pro ošetření je znak spojovníku, který se v jazyce Python i jiných jazycích používá jako znaménko mínus. Jediné, co tak musí aplikace udělat, je nahradit tento symbol podtržítkem a tím je ošetřené hotové.

Příklad vygenerovaného kousku kódu, který byl vygenerován pro zjednodušenou gramatiku, která generuje pole podobné tomu ve formát JSON, ukazuje obrázek 6.2.

```
tokens = ("comma", "number", "left_bracket", "right_bracket", )

def t_comma(t):
  r"(?u),"
  return t

def t_number(t):
  r"(?iu)[+-]?(?:(?:[0-9][0-9]*|0)(?:\.[0-9]+)?)"
  return t

def t_left_bracket(t):
  r"(?u)\["
  return t

def t_right_bracket(t):
  r"(?u)\]"
  return t

def t_error(t):
  pass

import ply.lex as lex
lex.lex()

def p_start(p):
  """start : left_bracket array right_bracket"""
  pass

def p_array(p):
  """array : number
  | array comma number"""
  pass

def p_error(p):
  pass

import ply.yacc as yacc
yacc.yacc()
yacc.parse("=== YOUR INPUT STRING GOES HERE ===")
```

Obrázek 6.2: Příklad kousku kódu v jazyce Python pro gramatiku jednoduchého pole podobného tomu v JSONu, který byl vygenerován z řetězce `[1,2,3]`.

Kapitola 7

Technologie pro tvorbu webových aplikací

Pro tvorbu webových aplikací existuje v dnešní době velká spousta různých knihoven, frameworků a dalších technologií. Jejich chytrým využitím si lze ušetřit spoustu práce a není tak nutné pokaždé znovu vynalézat kolo. Vzhledem k tomu, že aplikace nepotřebuje nijak komunikovat se serverem a veškerá činnost tak může být prováděna u klienta, využívá aplikace pouze technologií na straně klienta v podobě jednostránkové webová aplikace.

7.1 Jednostránkové webové aplikace

Jednostránkové webové aplikace, známé také pod anglickým názvem *single page applications*, jsou takové aplikace, která provádí většinu aplikační logiky na straně klienta, tedy v tomto případě ve webovém prohlížeči uživatele. Pokud to aplikace vyžaduje, dotazuje se serveru především pomocí webových aplikačních rozhraní. [16]

Základem pro tvorbu nejen jednostránkových webových aplikací jsou jazyky:

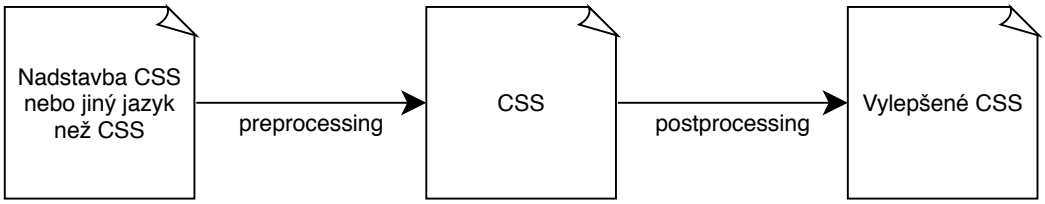
- HTML pro popis struktury stránky,
- CSS pro popis vzhledu stránky a
- JavaScript pro zajištění funkčního chodu aplikace.

V závislosti na architektuře aplikace a možnostech použitých knihoven může server vrátit pouze velmi jednoduchou až prakticky prázdnou stránku, jako to dělá i webová aplikace popisovaná v této práci. Taková stránka obsahuje pouze odkazy na kaskádové styly a skripty jazyka JavaScript. Veškerou kontrolu a vykreslování dalších částí HTML dokumentu poté přebírají příslušné skripty a knihovny jazyka JavaScript.

7.2 CSS preprocesory a postprocesory

CSS preprocesory a postprocesory představují ve světe nedokonalého jazyka CSS způsob, jak si usnadnit vývoj a nechat udělat část práce těmito procesory. Preprocessing v tomto případě představuje proces, kdy se z nějaké nadstavby jazyka CSS nebo i zcela odlišného jiného jazyka vytvoří jazyk CSS, kterému rozumí samy o sobě webové prohlížeče. Postprocessing naopak vylepšuje již hotové a pro prohlížeče srozumitelné CSS o další užitečné konstrukce, jako jsou například prefixy novějších vlastností. Rozdíl mezi preprocesingem

a postprocesingem ilustruje obrázek 7.1, který byl inspirován zdrojem [15], ze kterého vychází i značná část tohoto odstavce.



Obrázek 7.1: Ilustrace rozdílu mezi preprocessingem a postprocessingem v kontextu webových technologií a jazyka CSS.

Zřejmě nejznámějšími CSS preprocesory jsou Less a Sass. Oba preprocesory dokážou určitým způsobem zapsat prakticky to stejné, výběr tedy závisí především na programátorovi a dalších okolnostech. Jelikož aplikace využívá pro tvorbu vzhledu knihovnu Bootstrap 4 s tmavým tématem Darkly z knihovny Bootswatch a všechny tyto součásti využívají preprocesoru Sass, jehož prostřednictvím nabízejí širokou škálu možností přizpůsobení, je volba preprocesoru Sass jasná.

CSS postprocesorů dnes již existuje celá řada, ovšem zřejmě jedním z nejznámějších je postprocesor PostCSS. V aplikaci je využit v kombinaci s modulem Autoprefixer, který se stará o doplňování prefixů pro novější vlastnosti jazyka CSS.

7.3 Knihovny pro tvorbu uživatelského rozhraní

Části následujících odstavců vychází ze článku [14], který shrnuje rozdíly mezi knihovnami React a Vue.js, a slouží tak jako stručný výtah pro uvedení do dané problematiky.

Ačkoliv existuje pro JavaScript spousta knihoven zaměřujících se na tvorbu uživatelského rozhraní, nabraly v současné době na popularitě zejména knihovny React a Vue.js. Knihovny jako takové jsou si velmi podobné – obě například využívají virtuálního DOMu a vytvářejí rozhraní pomocí znovupoužitelných komponent. Aplikace však využívá pro svou činnost starší knihovnu React, která ale na druhou stranu nabízí záštitu v podobě firmy Facebook, jenž tuto knihovnu aktivně vyvíjí, a také bohatý ekosystém s knihovnami třetích stran.

Pro pohodlnější psaní komponent využívá React jazyku JSX, což je nadstavba jazyka JavaScript umožňující psát HTML značky přímo v kódu podobně jako by se jednalo o čistý HTML dokument. Toho dosahuje prostřednictvím knihovny Babel, která nejenže dokáže přeložit jazyk JSX do čistého JavaScriptu, ale nabízí i možnost využívat funkce z novější verze JavaScriptu. Tyto novější funkce poté překládá do starší verze JavaScriptu, který dokáže zpracovat většina současných prohlížečů.

Dalšími použitými knihovnami pro tvorbu uživatelského rozhraní jsou:

- `react-bootstrap` pro snadnější integraci interaktivních komponent knihovny Bootstrap do Reactu,
- `react-dropzone` pro jednoduché nahrávání souborů stylem „táhni a pusť“ a
- `react-hook-form` pro zjednodušení vytváření a správu formulářů.

U poslední zmíněné knihovny pro vytváření formulářů je nutné zmínit její poněkud nestandardní použití. Ukázalo se, že je poměrně problematické zajistit synchronizaci mezi stavem aplikace a stavem formuláře, který si knihovna sama ukládá. Pro tento účel muselo být využito porovnání mezi stavem aplikace a stavem formuláře a v případě rozdílu se musel tento stav ve formuláři vynuceně aktualizovat. To se liší od běžného použití Reactu a jeho komponent, kdy by měl tyto rozdíly zjišťovat a aktualizovat React sám o sobě.

Jako poměrně nestandardní použití této knihovny se ukázala i okamžitá aktualizace stavu aplikace už v době, kdy uživatel formulář vyplňuje. Díky tomu uživatel okamžitě vidí všechny změny přímo v generovaném kódu. Jestli se opravdu jedná o nestandardní použití této knihovny nebo jde o její limitaci, případně co vlastně způsobovalo tyto problémy, se bohužel nepodařilo dohledat.

7.4 Knihovny pro správu stavu aplikace

Pro správu stavu aplikace opět existuje spousta knihoven, z nichž zřejmě nejznámějšími a v poslední době nejpopulárnější je knihovna Redux. Ta navazuje na architekturu Flux zavedením několika omezení, která mají za cíl učinit změny stavu předvídatelnými.

Následující odstavce krátce shrnují podstatu a principy knihovny Redux tak, jak je uvedeno v její dokumentaci [1], a slouží tak jako stručný úvod do této problematiky.

Celý stav aplikace je v knihovně Redux ukládán do jediného anonymního objektu jazyka JavaScript nazývaného *store* nebo také *jediný zdroj pravdy*. Tento *store* může být dále členěn na další podobjekty obsahující skalární hodnoty, pole, další objekty apod. Nad tímto *store* se pomocí dispečera vykonávají akce definované jejím typem a parametry.

Akce zpracovává jeden hlavní *reducer*, což je z definice tzv. čistá funkce, tedy funkce, která pro stejné vstupní parametry vrací stejnou výstupní hodnotu a která při spuštění nemá žádný vedlejší efekt. Tato funkce přijímá jako jediné dva parametry aktuální stav aplikace nebo jeho část a akci, kterou má vykonat, a vrací následující stav aplikace. Hlavní *reducer* může uvnitř volat libovolný počet dalších *reducerů* a těm předávat zodpovědnost třeba jen za část stavu aplikace. Žádný z *reducerů* ale nesmí změnit původní stav. Místo změny musí vracet změněnou kopii stavu nebo v případě, že stav nemění vůbec, musí vracet původní stav.

Díky těmto omezením bylo možné implementovat funkci „zpět“ a „vpřed“ prostým nainstalováním a nakonfigurováním knihovny `redux-undo`. Ta pracuje na principu ukládání snímků stavu po provedení zvolených akcí. Nevýhodou tohoto přístupu je vyšší paměťová náročnost oproti přístupům zahrnující například návrhový vzor *Command*, což ale u takto malé aplikace nemusí být problém, případně je možné omezit počet ukládaných snímků stavu.

Další výhodou plynoucí z těchto omezení je jednoduchá serializace stavu. Celý stav či *store* stačí vzít, převést například do formátu JSON a uložit například do lokálního úložiště prohlížeče. Jedním z důvodů, proč tato serializace není v aplikaci implementována, je i skutečnost, že je v *reducerech* vytvářejících entity datového modelu použita funkce pro generování unikátního identifikátoru entity, která ale ukládá svůj stav mimo funkci, což porušuje princip čisté funkce a znemožňuje tak tuto serializaci. Serializaci znemožňuje také použití literálů `Infinity`, které nelze serializovat do formátu JSON.

7.5 Editor zdrojových kódů

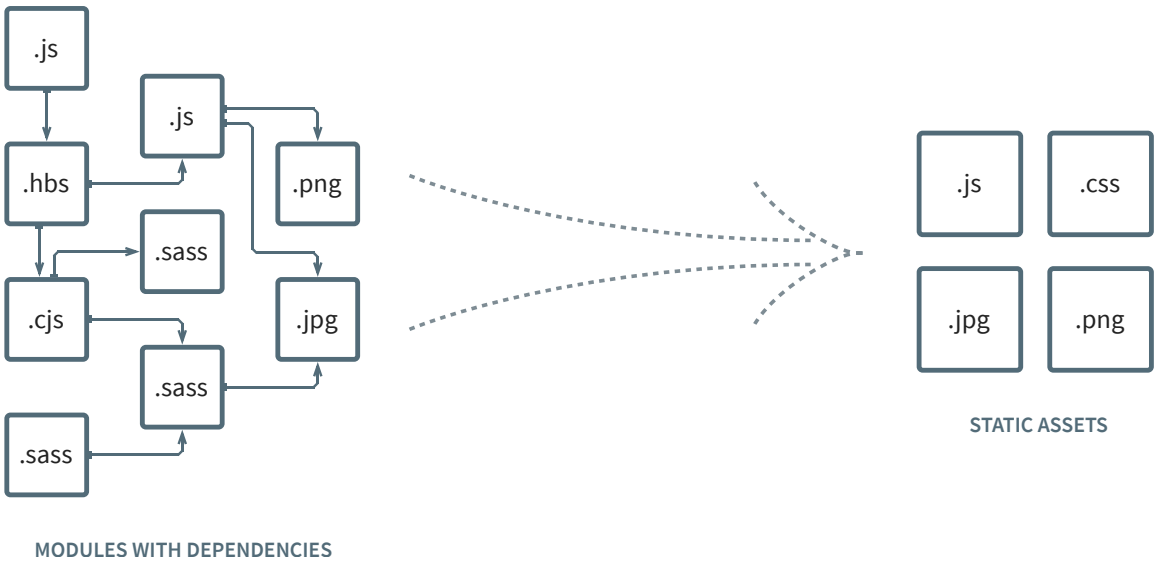
Pro editovaní zdrojového kódu byl z velkého množství editorů zvolen editor Ace, který lze celkem snadno integrovat do aplikace prostřednictvím knihovny třetí strany `react-ace` jako komponentu Reactu. Pro výběr editoru pro tuto aplikace je důležité, že editor podporuje vybírání více částí textu zároveň a že umožňuje zapnout zvýrazňování syntaxe, případně vytvořit vlastní zvýrazňovač. Důležité jsou i tzv. *markery*, které umožňují libovolně zvýrazňovat určité části textu. Vzhledem k tomu, že aplikace je laděná do tmavých barev, nahrává editoru i přítomnost tmavého tématu.

Ukázku editoru v aplikaci je možné vidět v sekcích následující kapitoly 8 a mj. i na obrázku 8.7.

7.6 Knihovny pro sestavení výsledné aplikace

Vedle nástrojů jako Grunt nebo Gulp, případně NPM skriptů, existuje i nástroj webpack použitý v této aplikaci. Na rozdíl od jiných nástrojů je webpack určen zejména pro moderní JavaScriptové aplikace a slouží jako balíčkovač statických modulů. Uvnitř si buduje graf závislostí, na základě kterého potom vytváří jeden nebo více balíčků. Přitom využívá koncept tzv. *loaderů*, které mu dovolují zpracovávat více formátů souborů než pouze JavaScript a JSON. [11]

V aplikaci slouží webpack pro sestavení a kompilaci modulů JavaScriptu, CSS preprocessing a postprocessing a slouží také jako podpora při vývoji, kdy při jakékoliv změně dokáže pomocí knihovny `webpack-dev-server` ihned provést kompilaci a obnovit aplikaci v prohlížeči. Pro kompilaci JavaScriptu využívá již dříve zmíněný `babel-loader` a pro CSS zase `css-loader`, `postcss-loader`, `sass-loader` a `style-loader`, které plní úlohy preprocessingu, postprocessingu a vůbec možnosti načítání stylů z webpacku.



Obrázek 7.2: Ilustrace principu, na kterém pracuje webpack – moduly se závislostmi jsou webpackem převedeny na statické zdroje. S úpravami převzato z [https://webpack.js.org/].

Kapitola 8

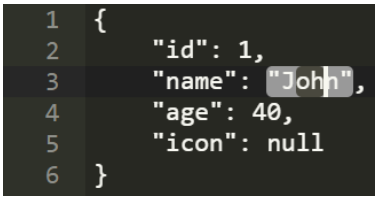
Prvky grafického uživatelského rozhraní aplikace

Podle způsobu interakce uživatele s aplikací existující v zásadě dvě kategorie aplikací: aplikace s textovým nebo grafickým uživatelským rozhraním. Protože se jedná o webovou aplikaci, která má uživateli co nejvíce usnadnit práci pomocí grafických prvků, spadá aplikace právě do druhé zmíněné kategorie. Tyto prvky usnadňující orientaci uživatele v aplikaci jsou popsány právě v následujících sekcích. Jednotlivé sekce přitom vychází z důležitých komponent grafického uživatelského rozhraní, které je možné vidět na obrázku 8.7.

V sekcích 8.1, 8.2, 8.4 a 8.5 jsou popsány úpravy editorů pro nahraný text a výslednou gramatiku, které mají usnadnit uživateli orientaci v pravidlech a pochopit souvislosti mezi nimi. Sekce 8.3 a 8.6 zase popisují implementaci a účel postranních panelů vpravo od editorů zdrojových textů.

8.1 Zvýrazňování symbolů gramatiky v editoru kódu

Pro usnadnění orientace uživatele ve vstupním textu při vybírání jeho částí, ze kterých následně bude uživatel odvozovat nová pravidla, existuje v aplikaci zvýrazňování vybraných symbolů gramatiky. Díky němu uživatel hned vidí, v rámci jakého většího celku se svým výběrem pohybuje a případně ho zvýrazňování může i upozornit, že přesáhl rozsah nějakého symbolu a pohybuje se nyní v jeho rodiči. Ukázku takovéhoto zvýraznění je možné vidět na obrázku 8.1.



Obrázek 8.1: Ukázka zvýraznění symbolu gramatiky při výběru menší podčásti tohoto zvýrazněného symbolu. Na řádku 3 je vybrána podčást `oh` a zvýrazněn je symbol `"John"`.

Ace editor popsaný v sekci 7.5 nabízí pro zvýrazňování částí textu něco, čemu se říká *markery*. Jedním z typů těchto *markerů* je i typ `text`, který umožňuje označovat pouze určené části textu podobně, jako to dělá například HTML element `<span>` nebo CSS vlast-

nost `display: inline` při nastaveném pozadí. Právě tento typ se hodí pro označený symbol gramatiky, protože nelze předem určit, jestli se bude jednat o celý řádek nebo vícero celých řádků.

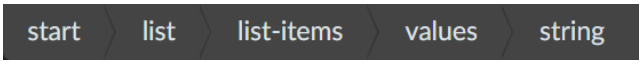
Zařídit implementačně zobrazení zvýrazněné části už poté není nic složitého. Využít k tomu je totiž možné již existující funkci pro vyhledání uzlu derivačního stromu na základě vybrané části textu, která je popsána v sekci 4.1. Menší potíž spočívá v tom, že jak Ace editor, tak i aplikace používají rozdílný způsob reprezentace rozsahů označeného textu. Zatímco Ace editor používá pro reprezentaci rozsahu dvojici dvojice (řádek, sloupec), aplikace používá dvojici jednoduchých hodnot – počtu znaků od začátku textu. Pro převod tak slouží dva převodníky – jeden pro převod z reprezentace používané v Ace editoru do reprezentace používané v aplikaci a druhý provádějící tento převod naopak.

Při převodu z reprezentace Ace editoru je sestavena ze vstupního textu pomocná znovupoužitelná mapa, která mapuje řádek vstupního textu na počet znaků od začátku textu. Z této mapy je poté vybrán příslušný počet znaků od začátku textu pomocí příslušného řádku z rozsahu Ace editoru a k němu je přičten sloupec tohoto rozsahu. Toto se udělá jak pro počáteční, tak i pro koncovou pozici v textu a výsledné dvě hodnoty jsou vráceny jako výsledek převodu.

Při převodu z reprezentace používané v aplikaci je nutné postupovat mírně odlišně. Zde je nutné spočítat počet znaků nového řádku, a to až do místa, kde se nachází hledaná pozice – ať už se jedná o pozici počáteční nebo koncovou. Přitom je nutné si také zapamatovat pozici posledního znaku nového řádku, který se nachází před hledanou pozicí. Spočítáním nových řádků je nalezeno číslo řádku pro Ace reprezentaci a odečtením hledané pozice od pozice posledního nového řádku je nalezen také sloupec pro Ace reprezentaci.

8.2 Drobečková navigace pravidel

Drobečková navigace slouží jako další pomůcka uživatele pro orientaci ve struktuře pravidel. Zobrazena je přímo pod editorem vstupního textu a zobrazuje pravidla postupně od startovního až po nejkonkrétnější pravidlo, jak ukazuje obrázek 8.2.



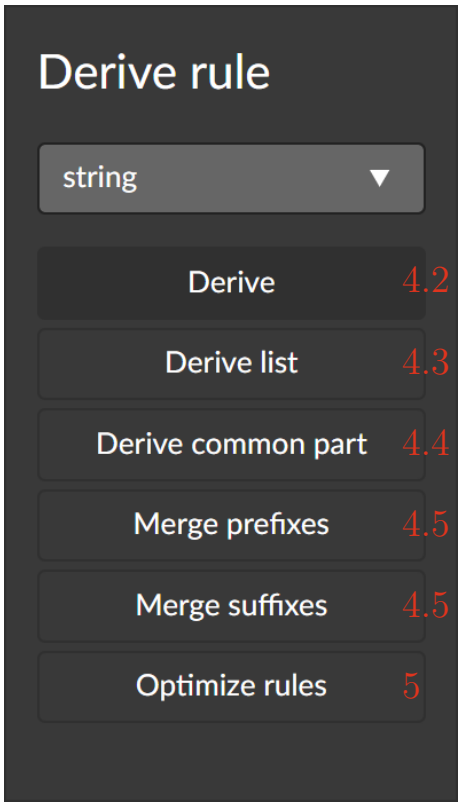
Obrázek 8.2: Drobečková navigace struktury pravidel pro výběr provedený v symbolu přepsaném podle pravidla s názvem `string` stejnojmenného typu.

Implementačně se jedná v podstatě o stejnou záležitost jako v případě zvýrazňování symbolů gramatiky popsaném v předchozí sekci 8.1 – pro vybranou část textu se použije vyhledávání uzlu derivačního, které je popsané v sekci 4.1. Tato funkce nalezne nejkonkrétnější uzel derivačního stromu, přičemž pak už jen stačí na zásobník kromě samotného konkrétního uzlu ukládat jeho předky a následně je odstraňováním ze zásobníku vypsat. Stejné efektu, tedy obráceného a pro uživatele správného pořadí předků, lze dosáhnout i za použití pole jeho reverzací nebo případným procházením pole od konce.

8.3 Panel pro odvozování pravidel

Aby mohl uživatel provádět akce popsané v předchozích kapitolách, potřebuje nějaké rozhraní, přes které toto může učinit. V současné době mu k tomu slouží vyskakovací panel po pravé straně editoru vstupního textu. Panel může uživatel zobrazovat a skrývat dle libosti například pro získání více místa pro text v editoru nebo při editaci v menším okně či na menším zařízení. Skrývání a zobrazování může provádět pomocí šedého proužku, který je umístěn po levé straně panelu.

Hlavní částí panelu je textové pole kombinované s výběrovým polem, do kterého uživatel zapisuje název nového pravidla, případně může pomocí výběrového pole provést výběr již existujícího pravidla. Následně klikne na jedno z tlačítek pod textovým polem, jejichž funkcionalitu ukazuje obrázek 8.3.



Obrázek 8.3: Panel s akcemi pro odvozování nových pravidel a případnou optimalizaci. Čísla odkazují do sekcí či kapitol, které jednotlivé akce popisují.

8.4 Zvýrazňování syntaxe ABNF a pseudokódu

Ace editor bohužel nepodporuje zvýrazňování syntaxe ABNF a pochopitelně ani vymyšleného pseudokódu. Vytvoření vlastního zvýrazňovače není ale nic složitého. Stačí vytvořit odvozenou třídu od základní třídy pro zvýrazňovače, v ní nadefinovat seznam terminálů používaných v daném jazyce a přiřadit jim nějakou klasifikaci, tedy že se jedná například o identifikátor proměnné, klíčové slovo, číslo apod. Terminály se přitom definují standard-

ními regulárními výrazy, které se používají v jazyce JavaScript, což ještě dále usnadňuje práci.

V případě ABNF znamenalo vytvoření zvýrazňovače manuální převod terminálů ze syntaxe ABNF, která je specifikovaná v příslušných RFC dokumentech, do regulárních výrazů, kde pro účely aplikace vzniklo 6 terminálů: identifikátor proměnné, řetězcový literál, 5 operátorů, číslo, prefix čísla a vestavěné názvy pravidel. Výsledné zvýraznění syntaxe ukazuje obrázek 8.4.

```
1  start             = left-bracket whitespace list whitespace right
       -bracket whitespace
2  list              = list-items *(list-separators-1 list-items)
3  list-items        = string colon whitespace values
4  list-separators-1 = comma whitespace
5  colon             = %s":"
6  left-bracket      = %s"{"
7  right-bracket     = %s"}"
8  whitespace        = *(CR / LF / %d12 / SP / HTAB / %d11)
9  number            = ["+" / "-"] (1*DIGIT ["." 1*DIGIT] / "." 1*DIGIT)
       ["e" ["+" / "-"] 1*DIGIT]
10 null              = %s"null"
11 string            = %d34 *(%d0-33 / %d35-91 / %d93-69631 / %s"\" %d0
       -69631) %d34
12 comma             = %s","
13 values            = number / string / null
```

Obrázek 8.4: Výsledné zvýraznění syntaxe ukázkové gramatiky v ABNF v tématu Monokai.

Pro pseudokód byla situace o něco složitější kvůli neexistenci přesné specifikace. Jelikož ale vychází pseudokód zejména z jazyka JavaScript, mohlo být pro spoustu terminálů využito stejných regulárních výrazů. Zejména se to projevilo v případě samotných regulární výrazů, jejichž zvýrazňování mohlo být kompletně přenecháno na již existujícím zvýrazňovači jazyka JavaScript. Vytvořeny tak byly pouze regulární výrazy pro řetězcové literály, komentáře, identifikátory proměnných a nové operátory. Výsledné zvýraznění syntaxe ukazuje obrázek 8.5.

8.5 Teplotní mapa vzdálenosti pravidel

Když uživatel označí část vstupního textu, která odpovídá nějakému pravidlu, zvýrazní se v editoru s generovanou množinou pravidel toto pravidlo a všichni jeho předci z derivačního stromu. Metoda grafické reprezentace dat, kdy jsou pravidla zvýrazněna postupně slábnoucí barvou, která slábne tím víc, čím výše v derivačním stromu se pravidlo nachází, se nazývá teplotní či tepelná mapa [17].

Teplotní mapa jakožto grafická reprezentace dat slouží uživateli pro lepší orientaci v generované množině pravidel. Barva slábne postupným zvyšováním průhlednosti této barvy, přičemž teoreticky by bylo možné průhlednost snižovat rovnoměrně od maximální neprůhlednosti po maximální průhlednost, případně po nějakou dolní hranici průhlednosti, ale s tímto postupem nastávají 2 problémy:

```
start             = left-bracket whitespace list whitespace right
    -bracket whitespace // concat
list              = list-items (list-separators-1 list-items)* // list
list-items        = string colon whitespace values // concat
list-separators-1 = comma whitespace // concat
colon             = ":" // literal
left-bracket      = "{" // literal
right-bracket     = "}" // literal
whitespace        = /\s*/ // whitespace
number            = /[+-]?(?:(?:[0-9][0-9]*|0)(?:\.[0-9]+)?|(?:\.[0-9]
    +))(?:e[+-]?[0-9]+)?/iu // number
null              = "null" // literal
string            = /"([^"\\]|\\.)*"/ // string
comma             = "," // literal
values            = number || string || null // alteration
```

Obrázek 8.5: Výsledné zvýraznění syntaxe ukázkové gramatiky v pseudokódu v tématu Monokai.

- se zvyšujícím se množstvím pravidel začíná být obtížné od sebe odlišit dvě sousední barvy a porovnat tak, které pravidlo předchází jinému pravidlu nebo po něm naopak následuje, a
- použitá knihovna pro snadnější práci s Ace editorem prakticky neumožňuje tohoto výsledku dosáhnout.

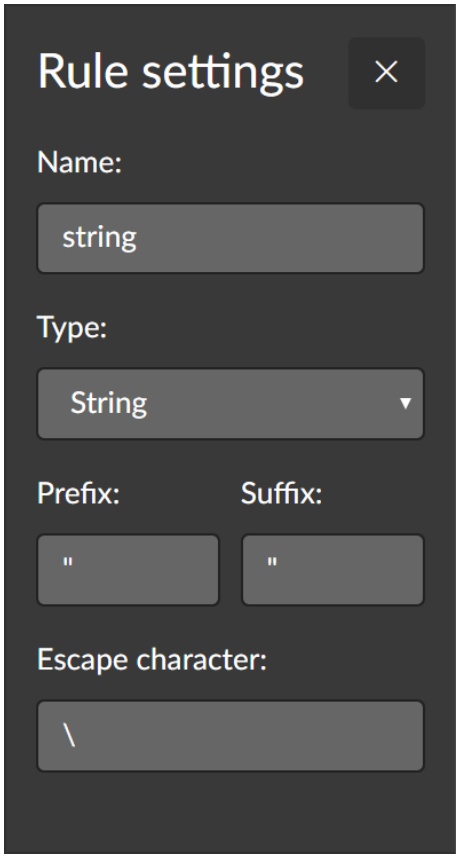
Z těchto důvodů aplikace používá 4 úrovně intenzity barvy. Nejkonkrétnější pravidlo má intenzitu 100 % a každé další pravidlo nižší a nižší intenzitu, dokud intenzita nedosáhne ve 4. úrovni na 30 %, kde zůstane i pro všechna obecnější pravidla. Barva přitom zůstává stejně intenzivní i pro menší počet pravidel, než kolik je celkem úrovní. Ukázku teplotní mapy pro pravidlo s názvem `string` je možné vidět mimo jiné i na obrázku 8.5 nebo 8.4.

Implementačně se jedná v podstatě o stejnou záležitost jako v případě drobečkové navigace, kterou popisuje sekce 8.2. Navíc je nutné si při průchodu stromem uchovávat informaci o vzdálenosti od konkrétního pravidla a tuto vzdálenost poté přenést do názvu CSS pravidla, které je použito při generování příslušných *markerů* pro Ace editor.

8.6 Panel pro úpravu pravidel

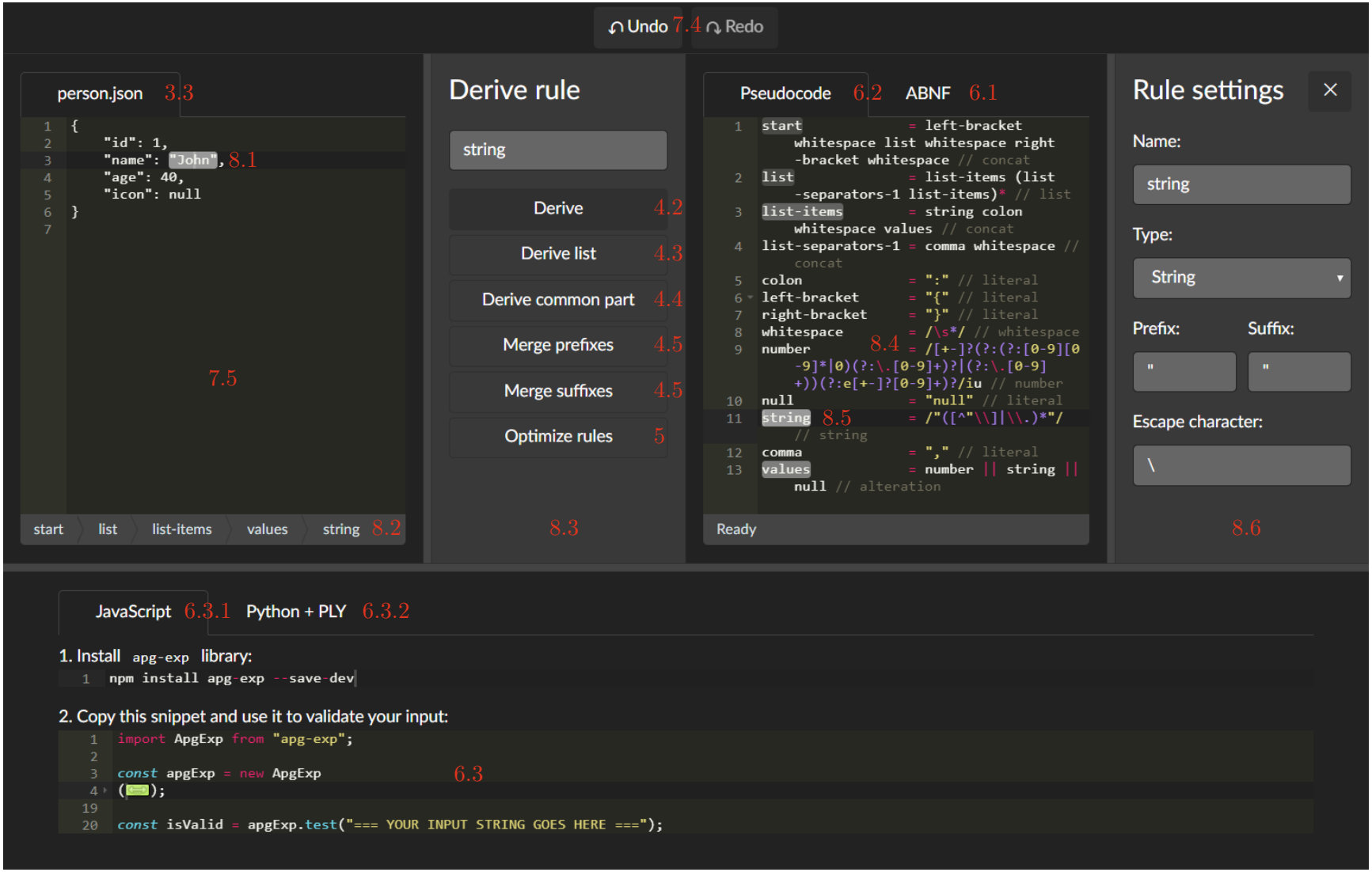
Některé vlastnosti pravidel nelze přímo odvodit ze vstupního textu nebo by jejich odvození bylo náročné či nepřesné. Pro tyto účely existuje vyskakovací panel pro úpravu vlastností pravidel, který se nachází na pravé straně od editoru s generovanými pravidly. Panel by teoreticky bylo možné nahradit povolením úprav v editoru, ale to by znamenalo nutnost použití dalšího syntaktického analyzátoru a bylo by potřeba detekovat typ pravidel, resp. umožnit jejich přidávání a úpravu jiným jednoduchým způsobem, a s tím související převod do dalších forem. Panel sice nenabízí takové možnosti, ale usnadňuje implementační záležitosti a také umožňuje uživateli upravovat pravidla více za použití myši. Ukázku panelu pro pravidlo typu řetězec je možné vidět na obrázku 8.6.

Implementačně jde o pouhou aktualizaci pravidel v *redux store*, a to prostřednictvím knihovny `react-hook-form`, která byla popsána v sekci 7.3. V případě validního formuláře



Obrázek 8.6: Ukázka postranního panelu pro editaci vlastností pravidla – zde pro pravidlo typu řetězec. Křížek vpravo nahoře slouží ke smazání pravidla.

jsou změny okamžitě promítány do všech částí aplikace zcela automaticky díky využití knihovny *React*. Pokud uživatel udělá chybu, může standardně využít tlačítko „zpět“ či „vpřed“. Změny ve formuláři jsou navíc shlukovány prostřednictvím konfigurace knihovny `redux-undo`, takže nedojde k zahlcení historie s každým napsaným písmenem.



Obrázek 8.7: Náhled do všech důležitých částí grafického uživatelského rozhraní aplikace včetně kapitol a sekcí je popisující.

Kapitola 9

Testování na bezkontextových jazycích

Pro ověření funkčnosti a zda dokáže aplikace obstát při tvorbě obecné gramatiky je nutné ji nějakým způsobem otestovat. Následující sekce shrnují základní podmínky, předpoklady a průběh samotného testování. To se přitom zaměřuje především na celkovou použitelnost aplikace uživatelem.

Mimo to aplikace obsahuje také automatizované testy napsané v knihovně Jest[1], které ověřují základní správnost vybraných důležitých částí. Jmenovitě jde o testy pro komponentu generátoru, syntaktického analyzátoru, optimalizátoru, okrajově i pro matematickou část funkcí pro rozsahy a také testy generování výsledných kousků kódu a jejich spuštění v cílových programovacích jazycích.

9.1 Podmínky testování

Testování pracuje s následujícími podmínkami:

1. Každý testovaný jazyk musí obsahovat alespoň jeden bezkontextový prvek, který není regulární.
2. Uživatel se musí pokusit vytvořit gramatiku, která bude generovat co největší podmnožinu validních řetězců testovaného jazyka.
3. Největší část činnosti uživatele musí být soustředěna na panel pro odvozování pravidel.
4. Změny provedené pomocí panelu pro úpravy pravidel musí být zdokumentovány.
5. Testovacím uživatelem je autor této práce.

9.2 Předpoklady testování

Podmínky pro testování vychází z následujících předpokladů:

1. Cílem je ukázat, že lze v aplikaci pracovat i s bezkontextovými prvky, které nejsou regulární.

[1] https://jestjs.io/

2. Pokud lze z konkrétního řetězce daného jazyka vytvořit jeho co možná nejobecnější popis, který stále odpovídá podmnožině testovaného jazyka, může uživatel některý krok zobecnění vynechat a vytvořit tak konkrétnější gramatiku, která může být vhodnější pro účely validace.

3. Použití panelu pro odvozování pravidel znamená, že pravidla jsou generována přímo ze vstupního řetězce. Případné použití panelu pro úpravu pravidel by znamenalo v podstatě ruční vytváření gramatiky a nebralo by příliš v potaz vstupní řetězec.

4. Pokud už je nutné použít panel pro úpravu pravidel, znamená to, že některý aspekt nemohl být vyjádřen jinak než ruční úpravou. Navazující práce se mohou na tento aspekt zaměřit a pokusit se ho eliminovat.

5. Pokud by ani autor práce, který má o fungování aplikace nejlepší povědomí, nebyl schopen dosáhnout obecné gramatiky, je někde pravděpodobně chyba.

9.3 Testování na vzorcích JSON a XML

Pro testování aplikace byly vybrány ukázky řetězců formátů JSON a XML. Tyto formáty byly vybrány jakožto jazyky, pro které existují schémata, která byla popsána v sekcích 2.4.2 a 2.4.3. Přesné znění řetězců těchto jazyků, které byly použity pro testování, uvádí příloha B. Vygenerované gramatiky v ABNF, které vznikly testováním řetězců příslušných jazyků, jsou dostupné v příloze C. Následující sekce uvádějí postup při testování jednotlivých řetězců a dokumentují problémy, které se při testování objevily. Tučně jsou přitom označeny kroky, které vyžadovali použití panelu pro úpravu pravidel.

9.3.1 JSON

Při práci s řetězcem jazyka JSON bylo postupováno následovně:

1. Celé pole bylo odvozen jako `array`.
2. Vytvořen seznam `array-list` označením obou prvků pole.
3. Objekty `John` a `Eduard` byly odvozeny jako `object`.
4. Vytvořen seznam `object-list` označením všech prvků položek objektu `John`.
5. Do seznamu z předchozího kroku byly přidány i prvky objektu `Eduard`.
6. Označena první dvojtečka a pomocí odvození společné části byla odvozena jako `colon`.
7. První čárka byla odvozena jako `comma`.
8. Odvozena všechna čísla kromě čísla v objektu `eduard.png` jako `number`.
9. Předchozí krok byl postupně proveden i pro řetězce `string` a klíčové slovo `null`.
10. Objekt `eduard.png` byl odvozen jako `object`.
11. Čárka oddělující objekty `John` a `Eduard` byla odvozena jako `comma`.
12. Závorky pole byly odvozeny jako `left-array-bracket` a `right-array-bracket`.

13. Závorky objektu `John` byly odvozeny jako `left-object-bracket` a `right-object-bracket`.
14. Všechny klíče v objektu `John` byly odvozeny jako `id`.
15. Klíč `"icon"` v objektu `Eduard` byl odvozen jako `id`.
16. Bílé znaky mezi `[` a `{` byly odvozeny jako `whitespace`.
17. **Typ pravidla `whitespace` byl změněn na `Whitespace`.**
18. Všechny bílé znaky, které ještě nebyly odvozeny a které se nenacházejí v objektu `eduard.png`, byly postupně po jednom odvozeny jako `whitespace`.
19. **Změněny typy pravidel `id`, `number` a `string` na typy `String`, `Number` a `String` v tomto pořadí.**
20. Množina pravidel byla optimalizována.
21. Sloučeny sufixy jako `values` označením první mezery mezi `:` a `1`.
22. Množina pravidel byla optimalizována.
23. **Z pravidel `object` a `object-list-items` byla odebrána všechna pravidla kromě `object-1` a `object-list-items-1`.**
24. Množina pravidel byla optimalizována.

Vytvoření gramatiky šlo podle těchto kroků poměrně snadno, jediným problémem byly bílé znaky. U nich musel být typ pravidla změněn ihned po vytvoření pravidla, jinak hrozilo, že syntaktický analyzátor nebude schopen text analyzovat.

9.3.2 XML

Při práci s řetězcem jazyka XML bylo postupováno následovně:

1. Z prvního řádku obsahující hlavičku bylo odvozeno pravidlo `header`.
2. Celá značka `<persons>` byla odvozena až po ukončovací značku `</persons>` jako `tag`.
3. Celá otevírací značka `<persons>` a celá ukončující značka `</persons>` byly odvozeny jako `opening-tag` a `closing-tag`.
4. Jména značek `persons` byla odvozena jako `tag-name`.
5. **Typ pravidla `tag-name` byl změněn na `Characters`.**
6. Otevírací závorky, ukončující závorky, lomítko a otazník značek `<persons>` a `<?xml?>` byly odvozeny jako `left-bracket`, `right-bracket`, `slash` a `question-mark`.
7. Jméno značky `<?xml?>` bylo odvozeno jako `tag-name`.
8. Vytvořen seznam `attribute-list` označením všech celých atributů značky `<?xml?>`.
9. Označen první znak rovná se a pomocí odvození společné části byl odvozen jako `equals`.

10. Jméno prvního atributu bylo odvozeno jako `attribute-name`.
11. **Typ pravidla `attribute-name` změněn na `Characters`.**
12. Hodnota prvního atributu byla odvozena jako `attribute-value`.
13. **Typ pravidla `attribute-value` byl změněn na `String`.**
14. Vytvořen seznam `tag-list` označením všech značek `<person>`.
15. Celá první značka `<person>` byla odvozena jako `tag`.
16. Jméno první značky `<person>` bylo odvozeno jako `tag-name`.
17. Všechny atributy první značky `<person>` byly jediným označením odvozeny jako `attribute-list`.
18. Otevírací závorka, ukončující závorka a lomítko první značky `<person>` byly odvozeny jako `left-bracket`, `right-bracket` a `slash`.
19. Celá druhá značka `<person>` byla odvozena jako `tag`.
20. Celá druhá otevírací značka `<person>` a celá druhá ukončující značka `</persons>` byly odvozeny jako `opening-tag` a `closing-tag`.
21. Jméno druhé značky `<person>` bylo odvozeno jako `tag-name`.
22. Všechny atributy druhé značky `<person>` byly jediným označením odvozeny jako `attribute-list`.
23. Otevírací závorka a ukončující závorka druhé značky `<person>` byly odvozeny jako `left-bracket` a `right-bracket`.
24. Celá značka `<icon>` byla odvozena jako `tag`.
25. První bílý znak byl odvozen jako `whitespace`.
26. **Typ pravidla `whitespace` byl změněn na `Whitespace`.**
27. Všechny bílé znaky, které ještě nebyly odvozeny a které se nenacházejí mezi značkami `</person>` a `<person>`, byly postupně po jednom odvozeny jako `whitespace`.
28. Množina pravidel byla optimalizována.
29. **Z pravidla `attribute-list-items` bylo odebráno pravidlo `attribute-list-items-2`.**
30. Množina pravidel byla optimalizována.

Při vytváření této gramatiky nastaly problémy zejména s mezerami na koncích atributů za uvozovkami, tedy např. v `"?>` nebo v `"/>`. Při pokusu o vytvoření seznamu, který má na konci tuto mezeru, se seznam vytvořil špatně. Za to není zodpovědná komponenta generátoru, ale komponenta syntaktického analyzátoru, která čte mylně poslední mezeru jako oddělovač seznamu. Mezera musela být odstraněna, s její přítomností nebylo možné gramatiku vytvořit.

Dalším problémem jsou bílé znaky mezi značkami `<person>`. Ty nesmí být převedeny na pravidlo `whitespace`, jinak syntaktický analyzátor skončí neúspěchem. Neúspěchem skončí také při pokusu o odvození značky `<icon>` jako `tag-list`.

9.4 Vyhodnocení testů

Pomocí navržených testů se v aplikaci podařila s některými ústupky vytvořit určitá podmnožina testovaných jazyků. V případě formátu JSON vypadá gramatika podobně jako kompletní gramatika, pokud je bráno v potaz, že se jedná o podmnožinu. Gramatika vygenerovaná pro formát XML některé společné prvky, jako například definici atributů, sdílí s kompletní gramatikou, ovšem existuje spousta míst, kde by bylo možné gramatiku ještě nadále optimalizovat nebo přetransformovat do podoby shodnější s kompletní gramatikou. Zejména se jedná části, jakou může být například volitelný seznam atributů. Pro účely validace ale toto může postačovat.

Jako slabší část řešení se ukázal syntaktický analyzátor, který také působí největší potíže. Vzhledem k tomu, že v něm není zcela implementován zpětný návrat, stává se, že konstrukce, které by byly jinak v pořádků, syntaktický analyzátor nedokáže analyzovat. Na něm přitom staví celý generátor, který bez něj nelze používat. Pro reálné nasazení se jeví jako nutnost tento vylepšený syntaktický analyzátor implementovat.

V poslední řadě si lze také všimnout, že po optimalizaci zůstala v generované gramatice některá pravidla, přestože nebyla pro syntaktickou analýzu dále potřeba. Pro to, aby byl optimalizátor schopen odstranit i takováto prakticky mrtvá pravidla, by musel mít ještě navíc přístup k derivačnímu stromu, díky kterému by mohl prohlásit některé větve gramatiky za nepoužívané a tyto větve by mohl případně i odstranit. Bez přístupu k derivačnímu stromu je tak nutné mu v současné době pomoci odstraněním odkazu z příslušného pravidla alternativy, přičemž zbytek už optimalizátor zvládne optimalizovat sám.

Kapitola 10

Závěr

Cílem této práce bylo ukázat, jak lze z konkrétního příkladu řetězce gramatiky vytvořit jeho obecný popis v podobě gramatiky v některé z variant Backusovy–Naurovy formy, které mohou být dále využity pro účely validace vstupního textu. Vznikla tak jednostránková webová aplikace vycházející z poznatků ze schémat pro formáty JSON a XML, která mimo jiné obsahuje i editory dokumentů podporující kromě zvýraznění syntaxe také další prvky usnadňující orientaci uživatele. Zdrojové texty těchto editorů lze přitom pomocí interaktivních tlačítek a formulářů upravovat tak, že uživatel ihned vidí výsledek. Výstupem aplikace je poté gramatika v rozšířené Backusově–Naurově formě a kousky kódů pro jazyk JavaScript a experimentálně i pro Python, které slouží jako validátory vstupních řetězců.

Díky testování aplikace na příkladech jazyků JSON a XML se podařilo s několika ústupky demonstrovat, že i konkrétní příklad může poměrně dobře posloužit jako základní stavební kámen obecnější gramatiky. Tímto se také podařilo představit lehce odlišný způsob přemýšlení od příkladu k obecnosti, který může být pro člověka přirozenější. Přístup jako tento může sloužit jako základ dalšího zkoumání, co vše lze ještě dále z konkrétního příkladu vyčíst a jak lze uživateli proces vytváření gramatiky ještě více usnadnit.

Do budoucna se ukázalo jako nutnost použít lepší či silnější syntaktický analyzátor, tedy nedeterministický syntaktický analyzátor využívající zpětného návratu. Díky němu bude možné předejít spoustě situací, kdy si analyzátor nedokáže poradit se vstupním textem nebo kdy kvůli tomu poskytne generátoru špatné informace.

Pěkné by nejspíše bylo i zakomponování informovaného odhadu datových typů, které označená část textu může reprezentovat, aby uživatel nemusel příliš používat panel pro úpravu pravidel a aby se zabránilo případnému bobtnání gramatiky konkrétními příklady. Rozšířit by si zasloužil také seznam generovaných typů gramatik a kousků kódu pro více jazyků a knihoven za účelem ještě lepší využitelnosti výstupů aplikace.

Literatura

[1] Abramov, D. *Redux: A predictable state container for JavaScript apps* [online]. 2015. Aktualizováno 18. 4. 2020 [cit. 2020-04-23]. Dostupné z: https://redux.js.org/.

[2] Beazley, D. M. dabeaz. *PLY (Python Lex-Yacc)* [online]. 2020. Aktualizováno 16. 4. 2020 [cit. 2020-04-18]. Dostupné z: https://www.dabeaz.com/ply/ply.html.

[3] Beech, D., Mendelsohn, N., Thompson, H., Sperberg McQueen, M., Maloney, M. et al. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. W3C Recommendation. W3C, duben 2012. Dostupné z: https://www.w3.org/TR/xmlschema11-1/.

[4] Câmpeanu, C., Salomaa, K. a Yu, S. A Formal Study of Practical Regular Expressions. *International Journal of Foundations of Computer Science*. 2003, sv. 14, č. 06, s. 1007–1018. Dostupné z: http://137.149.157.5/Articles/index.php?aid=1.

[5] Crocker, D. a Overell, P. *Augmented BNF for Syntax Specifications: ABNF*. RFC 5234. RFC Editor, leden 2008. Dostupné z: https://www.rfc-editor.org/rfc/rfc5234.txt.

[6] Dennis, G. et al. *JSON Schema: A Media Type for Describing JSON Documents*. Draft 2. Internet Engineering Task Force, září 2020. Dostupné z: https://tools.ietf.org/html/draft-handrews-json-schema-02.

[7] Češka, M. a Rábová, Z. *Gramatiky a jazyky*. skriptum. Vysoké učení technické v Brně, červen 1992. Dostupné z: http://kifri.fri.uniza.sk/~bene/vyuka/kompilatory/pomocne-materialy/Gramatiky%20a%20jazyky-Ceska.3.pdf.

[8] GeeksForGeeks. GeeksforGeeks. *Longest Common Substring | DP-29* [online]. Květen 2013. Aktualizováno 24. 7. 2019 [cit. 2020-04-09]. Dostupné z: https://www.geeksforgeeks.org/longest-common-substring-dp-29/.

[9] GeeksforGeeks. GeeksforGeeks. *Mark-and-Sweep: Garbage Collection Algorithm* [online]. Květen 2016. Aktualizováno 4. 6. 2018 [cit. 2020-04-12]. Dostupné z: https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/.

[10] ISO/IEC. *Information technology – Syntactic metalanguage – Extended BNF*. Standard ISO/IEC 14977:1996(E). International Organization for Standardization, prosinec 1996. Dostupné z: http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip.

[11] Larkin, S., Ewald, J. et al. webpack. *Concepts* [online]. 2017. Aktualizováno 23. 2. 2020 [cit. 2020-04-23]. Dostupné z: https://webpack.js.org/concepts/.

[12] Lee, X. Xah Code. *What's the Difference Between BNF, EBNF, ABNF?* [online]. Listopad 2014. Aktualizováno 9. 2. 2018 [cit. 2020-03-22]. Dostupné z: http://xahlee.info/parser/bnf_ebnf_abnf.html.

[13] Mozilla. MDN Web Docs. *Array.prototype.splice()* [online]. Aktualizováno 14. 3. 2020 [cit. 2020-04-12]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice.

[14] Nowak, M. Monterail. *Vue vs React in 2020: Which Framework to Choose and When* [online]. Březen 2019. Aktualizováno 1. 4. 2020 [cit. 2020-04-19]. Dostupné z: https://www.monterail.com/blog/vue-vs-react-2020.

[15] Rocheleau, J. Hongkiat. *CSS Post-Processors For Beginners: Tips and Resources* [online]. Listopad 2016. Aktualizováno 1. 11. 2018 [cit. 2020-04-19]. Dostupné z: https://www.hongkiat.com/blog/css-post-processors-tips-resources/.

[16] Smith, S. a olprod. Microsoft Docs. *Vyberte si mezi tradičními webovými aplikacemi a jednostránkovými aplikacemi (SPA)* [online]. Prosinec 2019. Aktualizováno 18. 3. 2020 [cit. 2020-04-19]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps.

[17] STEM/MARK. Marketingový výzkum a analýza dat - STEM/MARK. *Heat mapa; Heatmap* [online]. Duben 2018. Aktualizováno 9. 4. 2018 [cit. 2020-04-26]. Dostupné z: https://www.stemmark.cz/encyklopedie-heat-mapa-heatmap/.

[18] Thomas, L. D. GitHub. *Apg-exp - APG Expressions* [online]. Březen 2016. Aktualizováno 15. 4. 2017 [cit. 2020-04-18]. Dostupné z: https://github.com/ldthomas/apg-js2-exp/blob/2d8c8e5b64844fe4a1972d508f8b3f0f48ede83b/README.md.

[19] w3resource. w3resource. *SQL Self Join* [online]. Aktualizováno 26. 2. 2020 [cit. 2020-04-11]. Dostupné z: https://www.w3resource.com/sql/joins/perform-a-self-join.php.

[20] Warren, G. a olprod. Microsoft Docs. *Základy uvolňování paměti* [online]. Listopad 2019. Aktualizováno 18. 3. 2020 [cit. 2020-04-12]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/standard/garbage-collection/fundamentals.

Příloha A

Obsah paměťového média

- `dist/` – zkompilované JavaScriptové soubory.
- `doc/` – zdrojové kódy LaTeXu a zkompilovaná PDF verze této práce.
- `scss/` – zdrojové kódy stylů stránky pro preprocesor Sass.
- `src/` – zdrojové kódy JavaScriptových modulů včetně vstupního bodu `index.js`.
- `tests/` – testy pro knihovnu Jest a demonstrační řetězce použité pro testování v této práci.
- `.babelrc` – konfigurace knihovny Babel.
- `index.html` – vstupní HTML stránka aplikace připravená pro otevření v prohlížeči.
- `package.json` – informace o aplikaci a seznam závislých knihoven JavaScriptu.
- `package-lock.json` – přesné informace o verzích knihoven JavaScriptu.
- `postcss.config.js` – konfigurace knihovny PostCSS.
- `README.md` – základní informace o případné instalaci a spuštění aplikace.
- `webpack.config.js` – konfigurace knihovny webpack.

Příloha B

Řetězce použité při testování

B.1 JSON

```
[
  {
    "id": 1,
    "name": "John",
    "age": 40,
    "icon": null
  },
  {
    "id": 2,
    "name": "Eduard",
    "age": 30,
    "icon":
    {
      "id": 1,
      "location": "eduard.png"
    }
  }
]
```

B.2 XML

```
<?xml version="1.0" encoding="UTF-8"?>
<persons>
  <person id="1" name="John" age="40"/>
  <person id="2" name="Eduard" age="30">
    <icon id="1" location="eduard.png"/>
  </person>
</persons>
```

Příloha C

Gramatiky vytvořené při testování

C.1 JSON

```
1  start                    = left-array-bracket whitespace array-list
       whitespace right-array-bracket whitespace
2  array-list               = object-1 *(array-list-separators-1 object-1)
3  array-list-separators-1  = comma whitespace
4  object-1                 = left-object-bracket whitespace object-list
       whitespace right-object-bracket
5  object-list              = object-list-items-1 *(array-list-separators-1
       object-list-items-1)
6  object-list-items-1      = string colon whitespace values
7  colon                    = %s":"
8  comma                    = %s","
9  number                   = ["+" / "-"] (1*DIGIT ["." 1*DIGIT] / "." 1*DIGIT)
       ["e" ["+" / "-"] 1*DIGIT]
10 string                   = %d34 *(%d0-33 / %d35-91 / %d93-1114111 / %s"\" %
       d0-1114111) %d34
11 null                     = %s"null"
12 left-array-bracket       = %s"["
13 right-array-bracket      = %s"]"
14 left-object-bracket      = %s"{"
15 right-object-bracket     = %s"}"
16 whitespace               = 1*(CR / LF / %d12 / SP / HTAB / %d11)
17 values                   = number / string / null / object-1
```

C.2 XML

```
1  start                   = left-bracket question-mark tag-name whitespace
       attribute-list question-mark right-bracket whitespace tag whitespace
2  tag                     = tag-1 / tag-2 / tag-3
3  opening-tag             = opening-tag-1 / opening-tag-2
4  closing-tag             = left-bracket slash tag-name right-bracket
5  tag-name                = 1*(%d48-57 / %d65-90 / %d97-122)
```

```
6  left-bracket           = %s"<"
7  right-bracket          = %s">"
8  slash                  = %s"/"
9  question-mark          = %s"?"
10 attribute-list         = attribute-list-items-1 *(whitespace attribute-list
       -items-1)
11 attribute-list-items-1 = tag-name equals attribute-value
12 equals                 = %s"="
13 attribute-value        = %d34 *(%d0-33 / %d35-91 / %d93-1114111 / %s"\" %d0
       -1114111) %d34
14 tag-list               = tag *(tag-list-separators-1 tag)
15 tag-list-separators-1  = %d10.9
16 tag-1                  = opening-tag whitespace tag-list whitespace closing
       -tag
17 tag-2                  = left-bracket tag-name whitespace attribute-list
       slash right-bracket
18 tag-3                  = opening-tag whitespace tag whitespace closing-tag
19 opening-tag-1          = left-bracket tag-name right-bracket
20 opening-tag-2          = left-bracket tag-name whitespace attribute-list
       right-bracket
21 whitespace             = 1*(CR / LF / %d12 / SP / HTAB / %d11)
```