Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Generování struktury instrukcí na základě záznamu webové aplikace

Kateřina Janů

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Kateřina Janů

Systémové inženýrství a informatika
Informatika

Název práce

Generování struktury instrukcí na základě záznamu webové aplikace

Název anglicky

Generating an instructions structure based on a web application log

---

Cíle práce

Cílem práce je navržení gramatiky překladače (programu), který bude schopný zpracovat obsah logu komerční webové aplikace, ze kterého vygeneruje scénář pro automatické integrační testy spouštěné v interním testovacím nástroji. Na vstupu programu bude textový soubor obsahující záznam akcí vykonaných uživatelem aplikace. Pomocí naimplementované gramatiky jazyka, program text přeloží do hierarchické struktury objektů představujících testovací scénář složený ze zaznamenaných uživatelských akcí. Výstupem programu je xml soubor obsahující serializovanou strukturu těchto objektů. Tento soubor slouží jako vstup pro zmíněný interní testovací nástroj.

Metodika

Po získání potřebných znalostí v oblasti teorie jazyků, gramatik a automatů a osvojení si základních technik syntaktické analýzy, je v praktické části navržen překladač (program).

Výstupem práce je návrh gramatiky pro všechny stupně analýzy textového řetězce – lexikální, syntaktické a sémantické. V rámci návrhu je vytvořen datový model, se kterým překladač pracuje.

- definice teoretického základu
- návrh gramatiky pro lexikální analyzátor
- návrh gramatiky pro syntaktický analyzátor
- rozšíření gramatiky o sémantickou analýzu a návrh datového modelu

Doporučený rozsah práce

35-40 stran

Klíčová slova

Automat, Gramatika jazyka, C# programovací jazyk, Testovací nástroj

---

Doporučené zdroje informací

Liberty, Jesse & MacDonald, Brian. Learning C# 3.0. O'reilly, Incorporated, 2008. 696 s. ISBN-10 0596521065

Meduna, Alexander. Automata and Languages: Theory and Applications. Londýn: Springer Verlag, 2005. 892 s. ISBN 1-85233-074-0

Výuka pro vývojáře na webu Microsoft Developer Network. Dostupné na WWW – https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/

Předběžný termín obhajoby

2019/20 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 19. 2. 2020

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 19. 2. 2020

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 21. 03. 2020

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Generování struktury instrukcí na základě záznamu webové aplikace" jsem vypracovala samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autorka uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 23.3.2020 _____

Poděkování

Ráda bych touto cestou poděkovala panu Ing. Jiřímu Brožkovi Ph.D. za poskytnuté studijní materiály, rady a ochotu při vedení mé bakalářské práce. Další velké poděkování patří mému kolegovi Ing. Jakubovi Čižinskému za jeho konzultace a pomoc pochopit zkoumanou problematiku v praxi. V neposlední řadě bych ráda poděkovala své rodině a příteli za jejich trpělivost a podporu, která mi pomohla tuto závěrečnou práci dokončit.

Generování struktury instrukcí na základě záznamu webové aplikace

Abstrakt

Cílem závěrečné práce je navržení překladače (programu), který na vstupu zpracovává textový záznam komerční webové aplikace a na jeho výstupu generuje strukturu objektů představující příkazy zapsané v logu aplikace. Výstupní hierarchická struktura objektů uložená do XML souboru je využita jako testovací scénář pro vykonání automatických testů v interním testovacím nástroji. Takto vytvořený testovací scénář je možné vykonat nebo ho importovat do testovacího nástroje a upravit.

Ke zpracování textového záznamu aplikace dochází v několika fázích. Lexikální analýza prochází vstupní textový soubor znak po znaku a na základě nadefinovaných pravidel rozpoznává klíčová slova a hodnoty. Výstupem analýzy je posloupnost takzvaných tokenů, což jsou symboly nesoucí informaci o jejich lexikálním významu neboli informaci o tom, jestli se jedná o klíčové slovo nebo hodnotu. Posloupnost tokenů je předána na vstup analýzy syntaktické, která vyhodnocuje, zdali pořadí tokenů odpovídá reálnému záznamu webové aplikace. Výstupem syntaktické analýzy je derivační strom popisující syntaktická pravidla využitá v průběhu analýzy vstupního řetězce. Derivační strom je dále podroben sémantické analýze, která dává validnímu vstupnímu řetězci význam a vytváří strukturu objektů reprezentující příkazy webové aplikace.

Výsledkem práce je návrh všech zmíněných jazykových analýz – lexikální, syntaktické a sémantické. Pro účely práce je navržen datový model v programovacím jazyce C#. Kompletní implementace překladače v reálném pracovním prostředí mého týmu výrazně urychlí a ulehčí definování testovacích scénářů pro automatické integrační testy.

Klíčová slova: překladač, log webové aplikace, analýza, lexikální, syntaktická, sémantická, datový model, testovací scénář, integrační testy

Generating an instructions structure based on a web application log

Abstract

The aim of the final thesis is to design a compiler (program), which takes in a text record of a commercial web application as the input and on its output generates a structure of objects representing commands written in the application log. The hierarchical structure of objects stored in an XML file is used as a test scenario to perform automated tests in an internal test tool. The test scenario created in this way can be executed or imported into the test tool to be edited.

Application text record is processed in several stages. Lexical analysis scans textual input file character by character and recognizes keywords and values based on defined rules. The output of the analysis is a sequence of tokens, which are symbols bearing information about their lexical meaning, in other words whether it is a keyword or value. The sequence of tokens is passed to the input of syntactic analysis, which evaluates whether the order of the tokens corresponds to the real record of the web application. The output of the parsing is a derivation tree describing the parsing rules used during input string analysis. The derivation tree is further subjected to semantic analysis, which gives meaning to the valid input string and creates a structure of objects representing the commands of the web application.

The result of this work is the design of all mentioned language analyses - lexical, syntactic and semantic Also a data model is designed in C# programming language for the purpose of this work. Complete implementation of the compiler is going to be used in real work environment of my team and will greatly speed up and facilitate the definition of test scenarios for automatic integration tests.

Keywords: compiler, log web application, analysis, lexical, syntactic, semantic, data model, test scenario, integration tests

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Tato závěrečná práce se věnuje praktickému využití teorie formálních jazyků k navržení překladače (programu), který na vstupu přijímá textový záznam webové aplikace a na výstupu generuje strukturu objektů představující příkazy zapsané v logu aplikace. Vygenerovaná struktura uložená ve formátu značkovacího jazyka XML slouží jako vstupní data pro interní testovací nástroj. Vytvořený program usnadňuje každodenní práce testera ERP softwaru komerční společnosti.

Teorie formálních jazyků slouží k zachycení a popsání pravidel zkoumaného jazyka. Jedná se o podstatnou část informatiky na jejímž základě funguje většina programovacích jazyků. Proces přeložení vstupního kódu na žádoucí výstupní kód je rozdělen do několika fází. Nejdříve je kód předložen k lexikální analýze, která prochází vstupní řetězec symbol po symbolu a vytváří z nich posloupnost tokenů. Token může představovat například identifikátor, klíčové slovo nebo hodnotu. Tato posloupnost tokenů je použita na vstupu syntaktického analyzátoru nazývaného parser. Parser řeší logickou část překladu a jeho výstupem je derivační strom, který je konstruován na základě definovaných gramatických pravidel. Sémantický analyzátor dává výstupu syntaktického analyzátoru význam a vytváří vnitřní reprezentaci kódu představující výstup celého překladače.

V teoretické části jsou definovány formální jazyky a automaty, gramatika jazyka a její dělení. Dále jsou vysvětleny jednotlivé fáze analýzy vstupního řetězce. Na závěr je představena webová aplikace, jejíž textový záznam bude v rámci práce přeložen.

Tyto teoretické znalosti jsou aplikovány v praktické části práce k realizaci překladače textového záznamu webové aplikace na strukturovaný záznam objektů představující testovací scénář.

Výsledek této práce bude využit v reálném pracovním prostředí mého týmu. Bude mít praktické využití, které by mělo uživateli urychlit a ulehčit definování testovacích scénářů. Nadefinované testovací scénáře se používají pro integrační testy spouštěné na každé vytvořené verzi produktu.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem práce je navržení gramatiky překladače (programu), který bude schopný zpracovat obsah logu komerční webové aplikace, ze kterého vygeneruje scénář pro automatické integrační testy spouštěné v interním testovacím nástroji. Na vstupu programu bude textový soubor obsahující záznam akcí vykonaných uživatelem aplikace. Pomocí naimplementované gramatiky jazyka program text přeloží do hierarchické struktury objektů představujících testovací scénář složený ze zaznamenaných uživatelských akcí. Výstupem programu je XML soubor obsahující serializovanou strukturu těchto objektů. Tento soubor slouží jako vstup pro zmíněný interní testovací nástroj.

2.2 Metodika

Po získání potřebných znalostí v oblasti teorie jazyků, gramatik a automatů a osvojení si základních technik syntaktické analýzy, je v praktické části navržen překladač (program).

Výstupem práce je návrh gramatiky pro všechny stupně analýzy textového řetězce – lexikální, syntaktické a sémantické. V rámci návrhu je vytvořen datový model, se kterým překladač pracuje.

- definice teoretického základu
- návrh gramatiky pro lexikální analyzátor
- návrh gramatiky pro syntaktický analyzátor
- rozšíření gramatiky o sémantickou analýzu a návrh datového modelu

3 Teoretická východiska

3.1 Formální jazyky

Pro pochopení teorie formálních jazyků je nezbytně nutné vysvětlení několika pojmů a termínů. Představíme si předmět zkoumání této vědní disciplíny.

3.1.1 Abeceda a jazyk

„Abecedou se rozumí libovolná konečná množina ∑, jejíž prvky nazýváme znaky (případně také písmena nebo symboly) abecedy.“ (1 str. 1) Abecedu si můžeme představit například jako množinu písmen ∑ = {a, b} nebo množinu přirozených čísel ∑ = {1, 2, 3, 4}.

Spojováním symbolů vznikají řetězce neboli slova. Slovo v je konečný řetězec symbolů nad abecedou ∑. Počet symbolů této posloupnosti |v| vyjadřuje délku slova. Slovo nemusí obsahovat žádný symbol, takovému řetězci říkáme prázdný a označujeme ho jako Ɛ.

Množinu všech slov nad abecedou ∑ značíme ∑*, množinu všech neprázdných slov ∑+. Platí např.

{a}* = {Ɛ, a, aa, aaa, aaaa,….}

{a}+ = {a}*\{Ɛ }

{0, 1}* = {Ɛ, 0, 1, 00, 01, 11, 000, 001, 010, 011,…} (1 str. 1)

Slova je možné také skládat ve větší celky, taková binární operace se nazývá zřetězení a značíme ji u.v = uv. „Operace zřetězení je asociativní, tj. u.(v.w) = (u.v).w pro libovolná slova u,v,w.“ (1 str. 1) Prázdné slovo Ɛ se v takové situaci chová jako jednotný prvek u. Ɛ = Ɛ.u = u.

Formální jazyk je podmnožina všech slov nad abecedou ∑ a značíme ho velkým písmenem L. Pravidla jazyka umožňují jednoznačně rozpoznat, zda dané slovo do jazyka patří či nikoliv. Jazyky mohou být jak konečné, tak nekonečné.

Zatímco konečné jazyky je možné definovat pouhým výčtem všech řetězců, které patří do podmnožiny slov nad abecedou jazyka, u jazyků nekonečných toto možné zjevně není. Z toho vyplývá nutnost použití sofistikovanějších mechanismů pro rozhodnutí o přijetí slova na základě nadefinovaných pravidel, jako jsou například automaty nebo formální gramatika.

3.2 Gramatika jazyka

V této práci se budeme zabývat pouze formální gramatikou, která se poněkud liší od gramatiky přirozených jazyků. Obě gramatiky se skládají ze dvou struktur. První je slovník, který definuje množinu validních slov jazyka. Druhou částí je soubor pravidel (systém), jakým způsobem je možné pracovat se slovy a jak je možné slova skládat ve větách. Hlavním rozdílem je užití jednotlivých gramatik neboli jazyků. Zatímco přirozený jazyk klade důraz na komunikaci a až poté na abstrakci systému, formální jazyk je vytvořený a řízený pomocí předem stanovených pravidel s přesně definovanou syntaxí a sémantikou definovanou pro určité praktické účely. (2 str. 1)

V roce 1956 jako první vytvořil matematický model gramatiky polský matematik Noam Chomsky. Jeho studie byla založena na výzkumu zákonitostí pravidel přirozených jazyků a velice záhy odhalila spojitost s teorií automatů. Chomsky vytvořil klasifikaci formálních jazyků a gramatik (3 str. 1), kterou si ukážeme v této kapitole.

3.2.1 Definice gramatiky formálního jazyka

Prostřednictvím gramatiky je možné zachytit a reprezentovat zkoumaný jazyk. Ke specifikaci formálního jazyka využíváme dvě konečné množiny symbolů:

1) neterminální symboly (tzv. neterminály)
2) terminální symboly (tzv. terminály)

Neterminální symbol je pomocná proměnná, která představuje logický celek. V procesu jazykové analýzy se tato proměnná používá v přepisovacích pravidlech, kde tento celek rozkládáme na jednodušší symboly a snažíme se pomocí těchto pravidel dostat k množině terminálních symbolů. Speciálním neterminálem je takzvaný začáteční symbol S, ze kterého analýza vět začíná.

Terminál je symbol, který v procesu analýzy již nenahrazujeme, jedná se o slovo jazyka. Množina terminálních symbolů je abeceda, nad kterou je definován jazyk L. (4 str. 1) Pro lepší orientaci v souboru pravidel označujeme neterminály velkými písmeny a terminály malými písmeny.

Gramatika jazyka představuje systém syntaktických pravidel, pomocí kterého je možné vygenerovat libovolný řetězec jazyka, popřípadě je možné zjistit, či daný řetězec do

jazyka patří či ne. (3 str. 2) Aplikací přepisovacích pravidel lze z libovolného neterminálu vygenerovat řetězec terminálních symbolů.

Gramatika je takzvaná čtveřice G = (N,T,P,S), kde každý prvek patří pouze do jedné z definovaných množin:

N – konečná množina neterminálních symbolů

T – konečná množina terminálních symbolů

P – konečná množina přepisovacích pravidel

S – začáteční symbol

Zjednodušeně řečeno na základě gramatiky zjišťujeme, jestli struktura analyzované věty vyhovuje přepisovacím pravidlům P. Na levé straně přepisovacího pravidla je vždy alespoň jeden neterminál, kvůli kterému se způsob náhrady definuje.

3.2.2 **Chomského hierarchie formálních gramatik**

Formální gramatiku jazyka lze klasifikovat podle tvaru jejích přepisovacích pravidel. Vymezení typu gramatiky napomáhá při přípravě analyzátoru vybrat nejvhodnější techniky a nástroje analýzy. Nejznámějším rozdělením je bezpochyby Chomského hierarchie gramatik a jazyků, která se skládá ze čtyř skupin:

- Regulární gramatika (typ 3)
- Bezkontextová gramatika (typ 2)
- Kontextová gramatika (typ 1)
- Neomezená gramatika (typ 0)

3.2.3 **Regulární gramatika a konečné automaty**

V pravidlech Chomského gramatiky typu 3 se na levé straně pravidla vždy vyskytuje pouze jeden neterminál. Jeho pravou stranu tvoří buď jeden terminál následovaný neterminálem nebo pouze jediný terminál. Jednotlivý krok derivace buď generuje terminální symbol případně změní neterminál. Derivace se uzavírá v momentě, kdy se vyskytují na pravé straně pravidla pouze terminály.

Definice:

$A \rightarrow \beta$;

$A \in N; \beta = bB$ nebo $\beta = b$; (kde $b \in T$ a $B \in N$) (3 str. 6)

Regulární gramatika má podstatný vztah s teorií konečných automatů, který si upřesníme v následujících podkapitolách.

3.2.3.1 Konečné automaty a jejich vztah s regulární gramatikou

Konečný automat je matematický model jednoduchého výpočetního zařízení s konečnou pamětí, schopný rozpoznávat určitý jazyk. (1 str. 9) Takové zařízení se skládá z takzvaných stavů a přechodů mezi těmito stavy. Stavu, ve kterém zařízení zahajuje svojí činnost, říkáme počáteční, stavy, ve kterých zařízení svojí činnost končí, označujeme jako koncové. Tento abstraktní model čte na vstupu řetězec symbolů (nad vstupní abecedu) a na základě přechodové funkce přijímá hodnoty a přechází z jednoho stavu do druhého. Automat slovo přijímá pokud se ve chvíli, kdy přečetl všechny znaky vstupního řetězce, nachází v jednom z koncových stavů. Konečné automaty přijímají pouze řetězce, které lze vygenerovat regulární gramatikou.

Definice:

Q - konečná množina stavů,

Σ - abeceda (konečná množina symbolů/písmen),

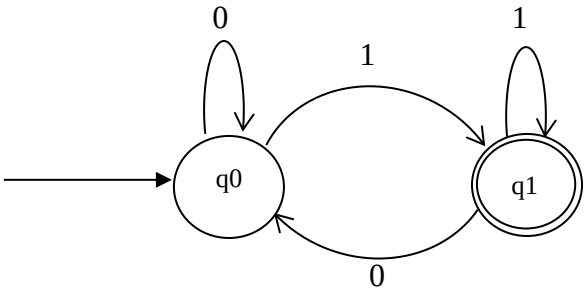
$\delta : Q \times \Sigma \longrightarrow Q$ je přechodová funkce,

$q0 \in Q$ je počáteční stav,

$F \subseteq Q$ je množina koncových (přijímajících) stavů. (1 str. 11)

Existují dva typy konečného automatu:

- Deterministické konečné automaty – každá přechodová funkce takového automatu je definována tak, že vrací pouze jeden stav.
- Nedeterministické konečné automaty – výsledek přechodové funkce nedeterministického automatu je celá množina stavů.



Obrázek 1 - Konečný automat
Zdroj – autor práce

3.2.4 **Bezkontextová gramatika**

Bezkontextová gramatika má na levé straně přepisovacího pravidla vždy pouze jeden neterminál. Rozdíl od regulární gramatiky je na pravé straně pravidla, kde se může nacházet neomezené množství terminálů a neterminálů. Jak již napovídá název, v tomto typu gramatiky se neterminál na vstupu může přepsat bez ohledu na aktuální stav okolí neboli kontextu.

Definice:

$A \rightarrow \beta$ nebo $A \rightarrow \varepsilon$

$A \in N; \beta \in (N \cup T)$ (3 str. 6)

Bezkontextová gramatika lze reprezentovat zásobníkovým automatem, podobně jako je možné regulární gramatiku reprezentovat konečným automatem.

3.2.4.1 Zásobníkové automaty

Jedná se v zásadě o konečný automat rozšířený o zásobník. Změna stavu podle daného pravidla přepíše znak v zásobníku. Přechod stavu se realizuje na základě hodnoty vstupního znaku, prvního znaku v zásobníku a aktuálního stavu automatu.

Definice:

$M = (Q, \Sigma, \Gamma, \delta, q0, Z0)$

Q - konečná neprázdná množina stavů

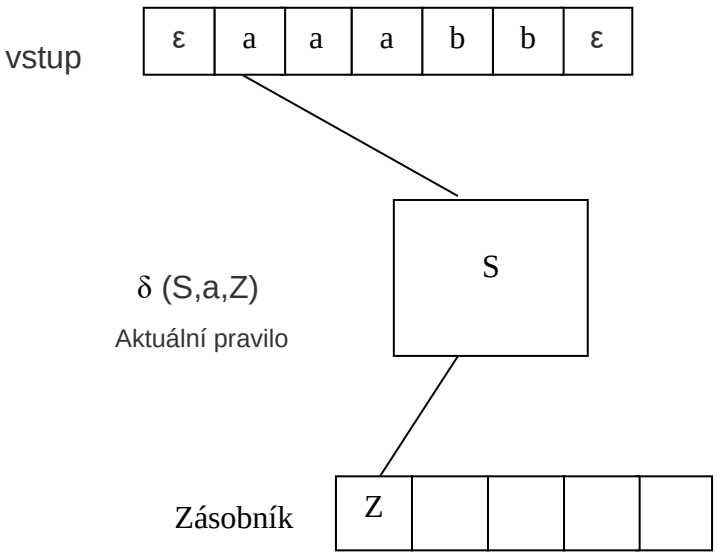
Σ - konečná neprázdná množina vstupních symbolů

Γ - konečná neprázdná množina zásobníkových symbolů

$q0 \in Q$ - počáteční stav,

$Z0 \in \Gamma$ - počáteční zásobníkový symbol

δ - zobrazení množiny $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ do množiny všech konečných podmnožin množiny $Q \times \Gamma*$. (5 str. 1)



Obrázek 2 – Zásobníkový automat
(5 str. 1)

3.2.5 **Kontextová gramatika**

V *kontextové gramatice* existuje alespoň jedno pravidlo, ve kterém se na levé straně vyskytuje víc jak jeden symbol. Podmínkou je, že řetězec symbolů levé strany předpisu obsahuje jeden neterminální prvek, ke kterému se pravidlo vztahuje. Z obou stran takového neterminálu mohou být řetězce terminálů a neterminálů, které představují kontext, zohledněný při realizaci derivace neterminálu. Na pravé straně pravidla je neprázdný řetězec, který má alespoň takový počet symbolů, jaký je definován na straně levé. Toto omezení zabraňuje zkrácení větné formy.

Definice:

$\alpha A \beta \rightarrow \alpha \gamma \beta$

$A \in N$; $\alpha, \beta \in (N \cup \Sigma)$; $\gamma \in (N \cup \Sigma)^{+}$ (3 str. 6)

Kontextová gramatika se používá ve většině vyšších programovacích jazyků.

3.2.6 **Neomezená gramatika**

Neomezená gramatika se shoduje s obecnou definicí gramatiky formálních jazyků. V této gramatice se nenachází žádná omezující pravidla pro přijetí vstupních řetězců.

Definice:

$\alpha \rightarrow \beta$

Zde se nenacházejí žádné omezující podmínky. (3 str. 6)

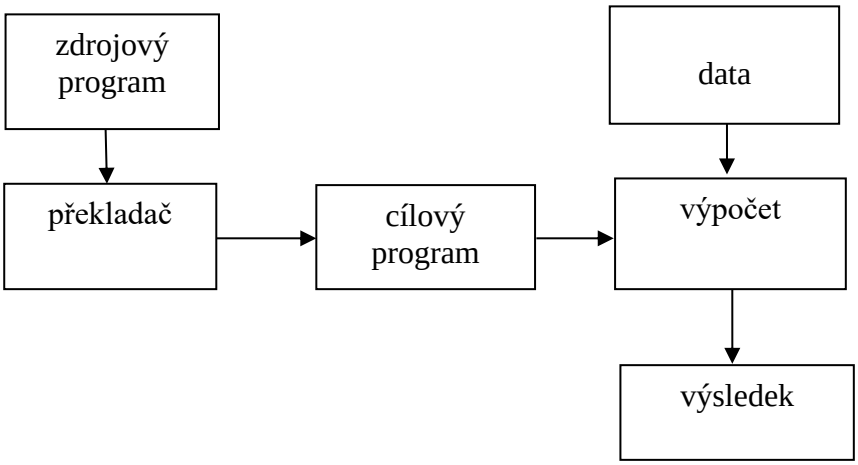
3.3 **Překladač**

V předchozích kapitolách jsme se seznámili s teoretickým základem formálních jazyků a jejich rozdělení do skupin dle různé jazykové složitosti. Popsali jsme si gramatiku jazyka představující soubor syntaktických pravidel, na jejichž základě je možné popsaný jazyk generovat. Vysvětlili jsme si fungování automatů, které dokážou jazyk rozpoznávat a na základě algoritmu rozhodují, zda testovaný prvek patří do zkoumaného jazyka či nikoliv. S těmito znalostmi si můžeme konečně vysvětlit celý proces překladu vstupního kódu na žádoucí výstupní kód, díky kterému bude realizovaná praktická část práce.

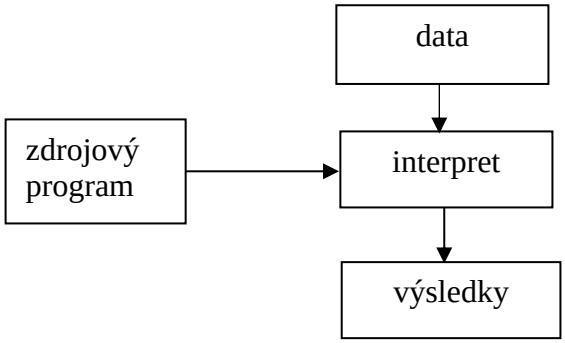
Teorie principu překladačů je jednou ze základních disciplín informatiky, která má široké využití v programovacích jazycích, přenosu dat, či překladu datových struktur. Obecně řečeno je překladač softwarový nástroj, který ze zdrojového vstupního jazyka vytváří jazyk cílový. S ohledem na různé potřeby překladu vzniklo v průběhu času hned několik typů překladačů. My si představíme činnost překladačů na jejich asi nejtypičtějším užití v dnešní době, tj. překladu programovacích jazyků.

Pro příklad si ukážeme dvě kategorie takových zařízení, kompilační překladač a interpretační překladač:

- Kompilační překladač – překladač tvoří ze zdrojového programu ve vyšším programovacím jazyce výstupní program ve strojovém jazyce (ve formě spustitelného souboru)
- Interpretační překladač – interpret naproti tomu nevytváří strojový kód, ale přímo interpretuje příkazy zdrojového jazyka a provádí příslušné akce (6 str. 11)



Obrázek 3 – Kompilační překladač (6 str. 11)



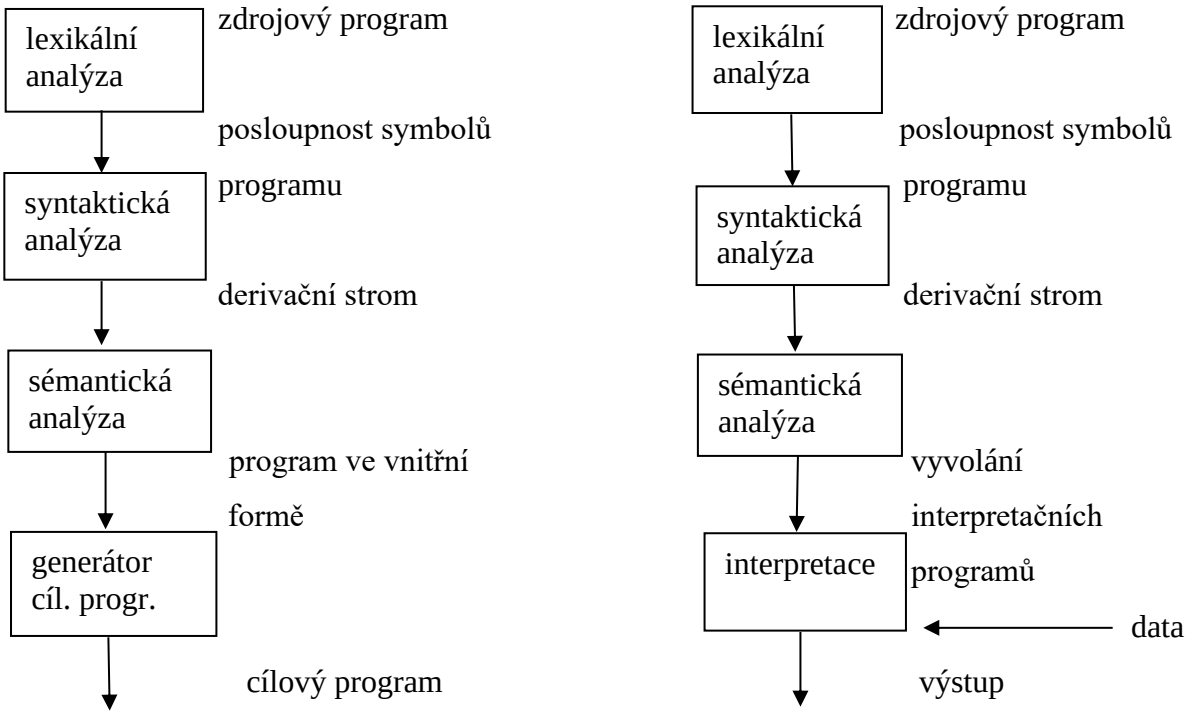
Obrázek 4 – Interpretační překladač
(6 str. 11)

Každý z těchto typů překladače má své podstatné výhody a využití. Zásadním rozdílem je rychlost zpracování vstupních dat. Kompilátor provádí analýzu a překlad vstupního programu pouze jednou, a to nezávisle na spuštění samotného programu. Výstupem kompilace je strojový kód, jehož generování je méně náročné v porovnání s interpretováním vstupního kódu na kód spustitelný u interpretačního zařízení. Interpret provádí analýzu a překlad vstupního program při každém jeho spuštění. Z toho vyplývá, že vykonání strojového kódu kompilátorem je v praxi výrazně rychlejší než vykonání interpretovaného kódu interpretační překladačem.

Nabízí se zde otázka, proč volit pomalejší řešení. Interpret má však naprosto nenahraditelnou a velmi zásadní vlastnost - je přenositelný mezi platformami. Kompilátor toto nedokáže, protože produkuje přímo strojový kód, který je specifický každému zařízení. Další neodmyslitelnou výhodou interpretu je snadnější odhalování chyb, jelikož odpadá mezikrok, kdy kód kompilujeme do spustitelného souboru.

Celý proces překladu se skládá z několika částí, které postupně analyzují a přetvářejí vstupní program:

- Lexikální analyzátor
- Syntaktický analyzátor
- Sémantický analyzátor
- Generování cílového programu



Obrázek 5 – Ukázka procesu překladu kompilačního a interpretačního překladače
(6 str. 13)

3.3.1 **Lexikální analýza**

Jako první je zdrojový kód předložen lexikální analýze, která zpracovává vstupní řetězec symbol po symbolu a vytváří z nich posloupnost tokenů. Token představuje lexikální jednotku (lexém) neboli abstraktní výraz, který nese význam. Pro lexikální analýzu definujeme jazyk popisující, jaká slova či druhy slov se mohou ve vstupním řetězci vyskytovat. Ve většině případů si v této části překladu bohatě vystačíme s gramatikou regulárního jazyka, která jako výpočetní model využívá konečný automat. Jedním z úkolů analyzátoru je také odstranění nepotřebných znaků vstupního kódu jako jsou bílé znaky (mezery, tabulátory) či komentáře. Výsledná posloupnost tokenů se předkládá na vstup syntaktického analyzátoru.

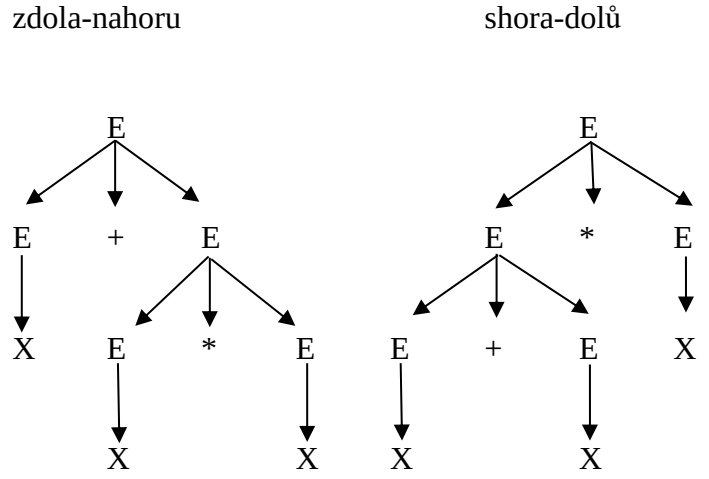
3.3.2 **Syntaktická analýza**

Syntaktická analýza je proces, který zkoumá správnost pořadí vstupních symbolů řetězce a jejich vzájemných vztahů. Na vstupu syntaktického analyzátoru je předložen řetězec lexémů, jednotek nesoucích význam, vygenerovaných ve fázi lexikální analýzy. Syntaxe se zabývá vztahy mezi jednotlivými symboly neboli zkoumá, jakým způsobem se z tokenů vytvářejí větší celky. Tyto celky v lingvistice představují věty, v informatice si je můžeme představit jako výrazy, deklarace, funkce atd. Syntaktický analyzátor je typicky definovaný bezkontextovou gramatikou a jeho cílem je zjistit, zda vstupní řetězec tokenů je validní větou nadefinovaného jazyka.

K ověření, zda přijatý řetězec odpovídá nadefinovaným pravidlům gramatiky jazyka, se využívá konstrukce hierarchické struktury derivačního stromu. Jedná se o znázornění konstrukce vět pomocí pravidel gramatiky. Kořenem stromu je počáteční symbol S, každý jeho uzel tvoří neterminální prvek a jeho listy jsou terminálními prvky abecedy zkoumaného jazyka.

Konstrukce stromu je realizovaná levou či pravou derivací a je sestavena na základě gramatických pravidel. Levou derivací přepisujeme nejlevější neterminál a používá se v syntaktické analýze shora-dolu. Pravou derivací naopak přepisujeme nejpravější neterminál a používá se pro syntaktickou analýzu zdola-nahoru. (7 str. 1)

Př.: X + X * X



Obrázek 6 – Derivační strom syntaktického analyzátoru zdola-nahoru a shora-dolu
Zdroj – autor práce

V případě, že k vstupnímu řetězci nalezneme derivační strom, je posloupnost symbolů správná a vyhovuje pravidlům dané gramatiky. Tento derivační strom je výstupem analýzy a znázorňuje posloupnost pravidel, podle kterých byla odvozena slova vstupního řetězce. Jestliže derivační strom nalezen není, program vyhodí chybu a překlad vstupních symbolů je přerušen.

Na druhou stranu v případě, že existuje více derivačních stromů pro jeden řetězec, jedná se o víceznačnou gramatiku. Analýza pomocí takové gramatiky je nedeterministická a vzhledem k nutnosti ověření všech variant, je proces analýzy složitější.

3.3.2.1 Deterministická syntaktická analýza shora-dolů

Syntaktická analýza shora-dolů spočívá v konstruování derivačního stromu od kořenového uzlu představující počáteční neterminální symbol S, směrem dolů k jeho listům představující terminální symboly, které patří do daného jazyka. Derivační strom prochází z leva doprava, podle levé derivace. Analýza se realizuje pomocí zásobníkového automatu. Takový výpočetní model nevyužívá stavy, ale pracuje pouze se zásobníkem. K analýze vstupního řetězce se používají pouze dvě základní operace:

- Expanze – na vrcholu zásobníku je neterminální symbol. Takový symbol nahrazujeme pravou stranou přepisovacího pravidla.
- Srovnání – na vrcholu zásobníku je terminální symbol. Symbol na vrcholu zásobníku se porovná se symbolem na vstupu a pokud se shodují, oba symboly jsou odstraněny.

Pokud se na konci analýzy nenachází žádný symbol na vstupu ani v zásobníku automatu, vstupní řetězec byl rozpoznán a je slovem daného jazyka. V případě, že se neterminální symbol levé strany pravidla může expandovat podle více pravidel, je nutné sestrojení takzvané překladové tabulky. Tato tabulka nám pomáhá k realizaci algoritmu, který rozhoduje o tom, jaké pravidlo bude využito při konstrukci každé větve derivačního stromu, pokud je na vstupu konkrétní symbol nebo více symbolů. Pro sestavení takové tabulky je nutný vypočet množin FIRST a FOLLOW. Tyto množiny prvků nám pomáhají vybrat správnou variantu pravé strany pravidla. V následující sekci si představíme konstrukci překladové tabulky pro LL(1) syntaktický analyzátor, to jest analyzátor shora-dolu, který analyzuje vstup zleva doprava (Left) a konstruuje nejlevější derivaci věty (Leftmost) s pomocí znalosti jednoho symbolu na vstupu (1).

3.3.2.1.1 Výpočet množiny FIRST

Množinu FIRST vypočítáváme, když pro daný neterminál existuje více variant pravidel, podle kterého se expanduje. Množina obsahuje všechny terminály, kterými pravá strana pravidla může začínat. Umožnuje algoritmu rozhodnout, které pravidlo vybrat pro expandovaný neterminál na vrcholu zásobníku s přihlédnutím k aktuálnímu symbolu na vstupu. Jednoduše řečeno, je vybrané takové pravidlo, jehož první terminální symbol se shoduje se symbolem na vstupu. Množinu určujeme pro každé přepisovací pravidlo dané gramatiky. Algoritmus pro výpočet množiny FIRST lze popsat takto:

1. Pokud $\alpha = \varepsilon$, pak First(α) = {ε}.
2. Pokud $\alpha = a\beta$, pak First(α) = {a}.
3. Pokud $\alpha = A\beta$, $\varepsilon \notin$ First(A), pak First(α) = First(A).
4. Pokud $\alpha = A\beta$, $\varepsilon \in$ First(A), pak First(α) = (First(A) \ {ε}) ∪ First(β). (8 str. 1)

3.3.2.1.2 Výpočet množiny FOLLOW

Množinu FOLLOW potřebujeme vypočítat v případě, že pro neterminální symbol existuje na pravé straně pravidlo obsahující pouze prázdné slovo epsilon. V takovém případě tento neterminál hledáme na pravé straně všech pravidel a zjišťujeme jakým symbolem může začínat řetězec, který ho bezprostředně následuje. Tyto symboly jsou prvky FOLLOW množiny. Formálně lze FOLLOW vypočítat:

1. Pro startovní symbol S přiřaďme Follow(S) = {ε}.
2. Pro všechny výskyty neterminálních symbolů na pravých stranách:
   - Pokud je pravidlo ve tvaru $X \rightarrow \alpha Y\beta$, pak Follow(Y) = Follow(Y) ∪ (First(β) \ {ε})
   - Pokud je pravidlo ve tvaru $X \rightarrow \alpha Y\beta$ a platí-li $\varepsilon \in$ First(β), pak Follow(Y) = Follow(Y) ∪ Follow(X)
3. Opakujme krok 2, dokud se změnila aspoň jedna z množin Follow(X).

(8 str. 1)

3.3.2.1.3 Překladová tabulka

Překladovou tabulku využívá algoritmus syntaktické analýzy pro určení gramatického pravidla, kterým expandovat neterminál na vrcholu zásobníku s přihlédnutím na symbol na vstupu. V řádcích tabulky jsou všechny neterminální symboly. Ve sloupcích

se nachází všechny terminální symboly. Do tabulky se zapisují pravidla, či jejich číselné označení, podle kterých je možné daný neterminál v řádku přepsat. Pravidlo se zapíše do souřadnic tam, kde se neterminální prvek v řádku potkává se symboly z jeho FIRST a FOLLOW množiny ve sloupci.

Při vytváření rozkladové tabulky můžeme zjistit, že námi navržená gramatika nevyhovuje pravidlům LL(1) gramatiky. Takové situace nazýváme kolize a mohou nastat ve dvou podobách. V této práci si představíme obě.

Kolize FIRST-FIRST nastává v případě, že je možné přepsat neterminální symbol podle více pravidel a zároveň existuje víc jak jedno takové pravidlo, které začíná stejným symbolem, tj. několik pravidel má shodné prvky v množině FIRST. Pro odstranění této dvouznačnosti existuje metoda levé faktorizace.

Ta spočívá ve vytknutí symbolu, který se vyskytuje na začátku více jak jednoho pravidla. Tento symbol zůstává jako jediný nadále na pravé straně pravidla a k němu je vytvořen nový neterminální symbol. Nově vytvořený neterminál značíme A` a představuje zbytek kolizního pravidla po vytýkání. Nový neterminál A` později rozkládáme tak, že na jeho pravé straně pravidla zapíšeme část kolizních pravidel, které zbylo po vytknutí:

A => aB | aC | aD | ae

Transformace:
A => aA`
A` => B | C | D | e
(9 str. 1)

Druhou častou kolizí vyskytující se v gramatice může být kolize FIRST-FOLLOW. Pokud je v gramatice nalezen neterminál, pro který existuje pravá strana pravidla obsahující pouze symbol epsilon (prázdné slovo), vypočítáváme množinu FOLLOW. Kolize nastává, pokud neterminál na pravé straně pravidla následuje stejný symbol, kterým začíná jedna z variant pravé strany původního pravidla. V takovém případě využijeme metody substituce a za neterminál vyskytující se na pravé straně pravidla dosadíme všechna pravidla, podle kterých je možné tento neterminál přepisovat. Tímto krokem vznikne zákonitě kolize FIRST-FIRST, kterou jsme již schopni odstranit metodou faktorizace popsanou o pár řádků výše.

S => Aab

A => a | Ɛ

First(A => a) = a

First(A => Ɛ) => Ɛ

First(A) = {a, Ɛ}

Follow(A) = a

Po substituci:

S => aab | ab

Po faktorizaci:

S => aS`

S` => ab | b

(9 str. 1)

3.3.2.2 Deterministická syntaktická analýza zdola-nahoru

Analýza zdola-nahoru začíná odspoda derivačního stromu neboli od jeho listů, představující terminály, směrem k jeho kořenovému uzlu. Konstrukce stromu je realizovaná podle pravé derivace. Na základě pravidel gramatiky v průběhu analýzy nahrazujeme podřetězec neterminálem, na který lze daný řetězec přepsat. V praktické části se touto analýzou zabývat nebudeme, proto zde není potřeba zabíhat do větších detailů.

3.3.3 **Sémantická analýza**

Sémantická analýza formálních jazyků zkoumá správnost významu syntakticky platných řetězců na základě předem stanovených pravidel. Ke každému přepisovacímu pravidlu se definuje jedno nebo více sémantických pravidel. Tato fáze analýzy se neprovádí v případě, že vstupní řetězec nebyl rozpoznán gramatikou syntaktické analýzy neboli nebyl vyhodnocen jako validní věta daného jazyka. Sémantická analýza je často realizovaná atributovou překladovou gramatikou. Výstupem analýzy je validní žádoucí kód, který je abstraktní reprezentací překladu vstupního kódu dle nadefinovaných pravidel. Takový

výstupní kód již není problém například serializovat do formátu XML, kterého se budeme snažit docílit v praktické části práce.

3.3.3.1 Atributová gramatika

Jedná se o rozšíření bezkontextové gramatiky o sémantické vlastnosti pomocí parametrů (atributů), které popisují význam jednotlivých symbolů. Atributy jsou přiřazeny každému symbolu, který je nadefinovaný v pravidlech syntaktické analýzy. Symboly s jejich atributy jsou použity k sestavení pravidel gramatiky sémantické analýzy. Podle způsobu výpočtu jejich hodnoty je rozdělujeme do dvou skupin:

$$A \rightarrow BnCD \{A.val = B.val + D.val, C.des = (A.m + B.val) * 10 + n.lex\}$$

- A.val je syntetizovaný, závisí na atributech symbolů-potomků
- C.des je dědičný, závisí na atributech předka a „bratrů“ vlevo (7 str. 1)

Z definic výše vyplývá, jaké atributy mohou být přiřazeny jakým symbolům. Vstupní terminální symboly mají pouze syntetizované atributy. „Neterminální symboly mohou mít jak syntetizované, tak dědičné atributy.“ (10 str. 1)

3.4 Webová aplikace

Aplikace, jejíž záznam provedených instrukcí bude v rámci práce zpracován, je součástí platformy podnikového informačního systému (ERP). Jedná se o aplikaci poskytující analýzu podnikových dat jako je rozpočet, plánování nebo vytváření predikcí nadcházejících období.

Platforma poskytuje komplexní řešení od řízení uživatelských oprávnění, přes správu a modelování databází, po zobrazení aplikací ve webovém prostředí. Skládá se z několika oddělených modulů (SOA architecture), které vzájemně komunikují. Uživatelské rozhraní, reporty a formuláře pro zadávání jsou vytvářeny v takzvaném Application Studiu modulu (service).

Report je tabulka, která obsahuje seznam prvků a jejich přidružených hodnot, získaných ze zdrojů dat prostřednictvím datových připojení. Může však obsahovat i seznamy položek nebo skupiny příkazových tlačítek. Na reportu je možné vytvářet akce, grafy, používat formule k výpočtům nebo různé typy proměnných, které drží hodnoty v rámci

jednoho reportu, ale i mezi nimi. Kombinací a propojením těchto reportů je možné vytvořit interaktivní aplikaci, která poskytuje konkrétní komerční využití, jako je třeba finanční plánování.

Aplikace je možné prohlížet na webu ale i v mobilních zařízení prostřednictvím Dashboards modulu (service). Je to nástroj, který umožnuje vytvořit každému uživateli vlastní prostředí, analýzy a úkoly v rámci nadefinovaných možností aplikace.

Veškerá interakce uživatele s aplikací otevřenou v Dashboards modulu odesílá příkazy Application studiu a čeká na jeho odezvu s obsahem, který se má uživateli zobrazit nazpět. Tuto komunikaci simuluje náš testovací nástroj, ve kterém se definují příkazy, které jsou při spuštění testu odesílány Application studio modulu, a kontroluje, jestli jsou jeho odezvy správné. Nadefinované testovací scénáře jsou uložené v XML formátu, který je při běhu testu zpracováván.

Všechny příkazy, které si tyto moduly vymění mezi sebou, se zapisují do logu. Této skutečnosti jsem využila k zjednodušení zdlouhavého definování testovacích scénářů. Textový záznam příkazů je zpracován navrženou gramatikou a přeložen na výstupní XML kód, který je možné buď spustit jako nový test, nebo naimportovat do testovacího nástroje a upravovat.

4 Vlastní práce

4.1 Vytvoření záznamu zpráv webové aplikace

Zapsání akce do logu vyvolá interakce uživatele s aplikací prostřednictvím Dashboards modulu v okně webového prohlížeče. V tu chvíli je odeslán příkaz k vykonání akce Application Studio modulu, který ho zpracuje a vrátí odezvu určenou k zobrazení uživateli. Příkazy jsou v kódu aplikace volány jako metody s určitým počtem parametrů a jejich signatura je spolu s aktuálními hodnotami parametrů v okamžik volání zapsána do logu webové aplikace. Vzhledem ke stále probíhajícímu vývoji aplikace hrozí, že se metoda a její parametry mohou v čase měnit. Použití takových automaticky generovaných záznamů pro účely této práce tudíž není vhodné, jelikož by každá jejich změna znamenala nutné úpravy navrhovaného překladače. Proto je pro stabilizaci záznamu logu vhodné vložit nové nezávislé záznamy, které nebudou ovlivněny změnami kódu.

Na základě komunikace mezi jednotlivými moduly jsem navrhla strukturu záznamů, které se budou zapisovat při zavolání konkrétních příkazů. Jedná se o sadu klíčových slov reprezentující názvy příkazů a určitého počtu parametrů ohraničených uvozovkami. Parametry jsou vždy stejného typu “value” a představují konkrétní hodnoty atributů zaznamenávaných příkazů. Volitelné parametry zapisujeme do složených závorek {}. Symbol * představuje opakování 0 – n a symbol + představuje opakování 1 – n.

applicationname "value"
loadreport "value" "value" {"value" "value" "value" "value"}*
opendialog
closedialog
commanddatacellclick "value" "value" "value"
commanddatadragdrop "value" "value" "value" "value"
commanddataexpandcollapse "value" "value"
commanddatabuttonclick "value" "value"
commanddatachartclick "value" "value" "value" "value"
commanddatavisualizationclick "value" "value" "value" "value"
commanddatawriteback {"value" "value" "value"}+
commanddatalookupclick "value" "value"
commanddatalvselectionchanged "value" "value"

Posloupnost takto navržených záznamů je vstupní řetězec, který je předložen ke zpracování překladači (programu). Jak již bylo vysvětleno v teoretické části, jako první je vstupní text v programu zpracováván lexikální analýzou, která ho prochází symbol po symbolu a na základě navržených pravidel generuje seznam tokenů. Jakým způsobem postupovat při návrhu lexikálního analyzátoru bude vysvětleno v následující kapitole.

4.2 Lexikální analýza

V první řadě je nutné si uvědomit jakou má vstupní řetězec strukturu a jaká slova je nutné v řetězci rozpoznávat. Jedná se o posloupnost názvů příkazů a jejich parametrů. Parametry příkazů jsou vždy ohraničeny uvozovkami a jednotlivá slova v záznamu jsou oddělena mezerou. Účelem lexikální analýzy je nalezení lexémů neboli jednotek, které nesou nějaký význam.

V případě záznamu aplikace je poměrně snadné si uvědomit, že je nutné rozpoznat dva typy lexémů. Jedná se o klíčová slova představující názvy příkazů a hodnoty jejich parametrů. Na základě této úvahy jsem vytvořila seznam lexémů, které jsou pro zkoumaný záznam validní. S tímto seznamem budou srovnána všechna slova rozpoznaná v lexikální analýze. V případě shody přijatého slova s některou z položek seznamu lexemů, je z daného slova vytvořen takzvaný token, představující symbol výstupního kódu analýzy. Tento token již nese určitý význam pro další zkoumání v procesu překladu.

TOKENY:
applicationName
loadReport
openDialog
closeDialog
commandDataCellClick
commandDataDragDrop
commandDataExpandCollapse
commandDataButtonClick
commandDataChartClick
commandDataVisualizationClick
commandDataWriteback
commandDataLookupClick

commandDataLVSelectionChanged
value

Pro lexikální analýzu vstupního řetězce je naprosto dostačující využít gramatiku regulární, kde se na levé straně pravidla vyskytuje vždy jeden neterminální symbol a jeho pravou stranu tvoří terminální a neterminální symbol, nebo pouze jeden terminální.

Při návrhu lexikálního analyzátoru, který zpracovává navržený záznam logu, je nutné si uvědomit, jaké všechny symboly se mohou v textu objevit. V tomto konkrétním případě platí, že klíčová slova jsou tvořena pouze malými písmeny. Další důležitý poznatek je, že hodnoty parametru jsou vždy ohraničeny uvozovkami. Tyto uvedené skutečnosti napomáhají k rozpoznání typu jednotlivých slov ve vstupním řetězci.

Po analýze vstupního textu se můžeme přesunout ke konkrétnímu řešení návrhu gramatických pravidel. Jak jsme již řekli, na vstup analyzátoru mohou přijít dva typy slov. Prvním je název příkazu, který vždy začíná písmenem, a druhým je hodnota parametru, která vždy začíná uvozovkami. V obou případech je tento první symbol následován neterminálem, který představuje množinu zbývajících symbolů tvořící slovo. První pravidlo gramatiky vypadá následovně (“a-z“ představuje symbol z množiny malých písmen od a do z):

S -> a-z Keyword | “ Value

Dále je nutné nadefinovat, jakým způsobem je možné přepsat jednotlivé neterminální symboly. První možností přepsání neterminálního symbolu Keyword je písmeno s rekurzivním voláním stejného neterminálu. Tento rekurzivní cyklus končí, až když na vstup analyzátoru přijde symbol mezera. Mezera je druhou možností přepsání neterminálu a v takovém případě jsou všechny předchozí symboly přijaty a tím algoritmus končí. Tímto způsobem přijímáme klíčová slova.

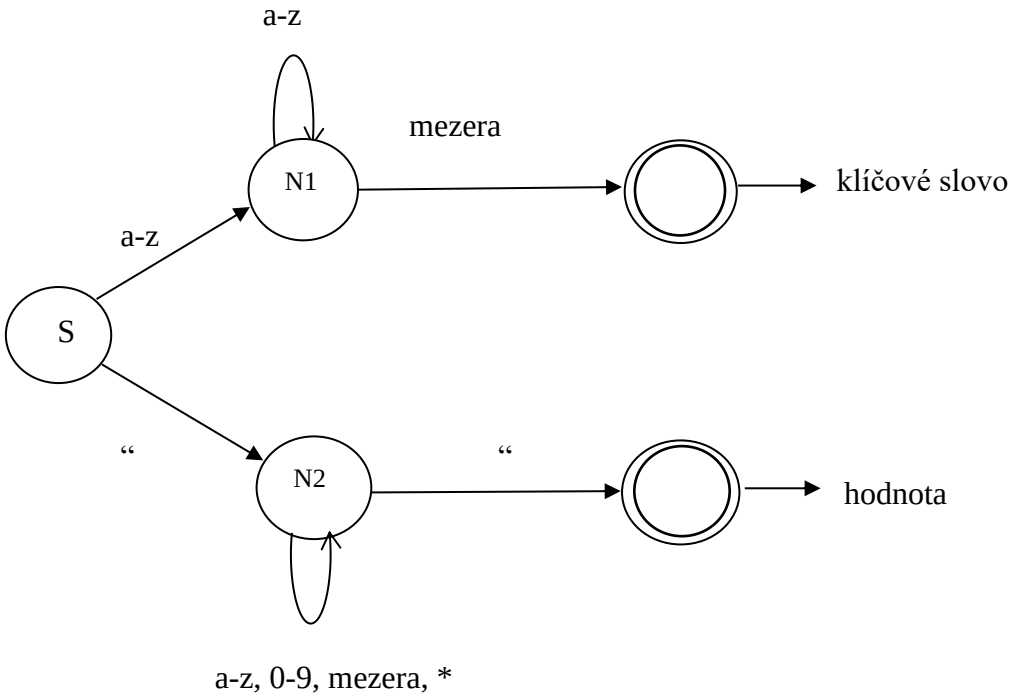
Keyword -> mezera | a-z Keyword

Neterminální symbol Value můžeme přepsat jakýmkoliv symbolem s rekurzivním voláním stejného neterminálu. Tento rekursivní cyklus končí až v momentu, kdy na vstup automatu přijde symbol uvozovky. V takovém případě jsou všechny předchozí symboly přijaty a algoritmus končí. Uvozovky jsou druhou možností přepisu neterminálu Value. Tímto způsobem přijímáme slovo typu hodnota parametru.

Value -> “ | * Value (* se rozumí jakýkoliv symbol kromě uvozovek)

Tato jednoduchá sada pravidel nám stačí ke zpracování textu záznamu aplikace. V této fázi je nutné navrhnout algoritmus, který bude na základě nadefinovaných pravidel symboly na vstupu rozpoznávat. Regulární gramatiku je možné zpracovávat pomocí konečných automatů. Pro navrženou gramatiku bude konečný automat vypadat takto:

S -> a-z Keyword | “ Value
Keyword -> “ | * Keyword
Value -> mezera | a-z Value



Obrázek 7 - Lexikální analyzátor (N1 – Keyword, N2 – value)
Zdroj – autor práce

Mějme počateční stav, který přijímá písmeno nebo uvozovky, podle toho jaký typ slova je automatem právě zpracováván. Pokud příjme jako první písmeno, může tento symbol následovat pouze sada dalších písmen nebo mezera, která algoritmus ukončí. Výstupem algoritmu je klíčové slovo, které bude srovnáno s položkami seznamu validních slov jazyka. Pokud se s některou z položek shoduje, je z tohoto přijatého slova vytvořen token, který nese význam - název příkazu.

Druhou možností je, že počáteční stav přijme jako první uvozovky. V takovém případě může být přijatý symbol nasledován jakýmkoliv symbolem včetně mezery, dokud se na vstupu automatu neobjeví uvozovky, které algoritmus ukončí. Výstupem algoritmu je hodnota parametru. Z přijatého slova je vytvořen token, který nese význam – hodnota parametru. Součástí vytvořeného tokenu je i konkrétní hodnota přijatého parametru.

4.3 Syntaktická analýza

Na vstupu syntaktické analýzy je předložen řetězec tokenů vygenerovaných v lexikální analýze. Syntaktická analýza zkoumá vztahy mezi tokeny, které již nesou informaci o svém typu. Zkoumá, jestli pořadí tokenů v posloupnosti odpovídá navrženému záznamu aplikace. Vzhledem ke složitosti záznamu aplikace je vhodné využít k návrhu syntaktického analyzátoru bezkontextovou gramatiku. Tato gramatika se vyznačuje tím, že na levé straně přepisovacího pravidla je vždy jeden neterminál a jeho pravou stranu tvoří neomezené množství terminálů a neterminálů. Možnost většího počtu symbolů na pravé straně pravidla je důležitá například pro popis akce otevření dialogu, kdy je potřeba v gramatice vyjádřit, že počet klíčových slov označujících otevření dialogu (terminál openDialog) musí být stejný jako počet klíčových slov označujících zavření dialogu (terminál closeDialog). Taková podmínka v regulární gramatice vyjádřit nelze. Pro vyhodnocení bezkontextové gramatiky se využívá algoritmus zásobníkového automatu. Volba konkrétního typu syntaktického analyzátoru je vcelku jednoduchá. Analýza shora-dolů je pro manuální návrh mnohem intuitivnější ve srovnání s analýzou zdola-nahoru. Gramatiku tedy navrhneme tak, aby splňovala pravidla nejběžnějšího analyzátoru shora-dolů LL(1).

4.3.1 Návrh gramatiky syntaktického analyzátoru

4.3.1.1 První fáze návrhu

Každý záznam logu začíná informací o tom, do jaké webové aplikace se uživatel přihlásil. Tento záznam bezprostředně následuje příkaz načtení reportu nebo několika reportů, bez kterých by nebylo možné provádět další akce. Zbytek příkazů, které se mohou v záznamu aplikace vyskytovat, jsou vázané ke konkrétnímu reportu. Z toho vyplývá, že počáteční neterminální symbol S je možné přepsat neterminály ApplicationName a Reports.

1. S → ApplicationName Reports

V dalších krocích je nutné definovat, jakým způsobem je možné expandovat nově vytvořené neterminály. Neterminál ApplicationName představuje záznam příkazu otevírání aplikace s jedním parametrem reprezentujícím jméno aplikace. Na pravé straně přepisovacího pravidla se bude vyskytovat terminální symbol applicationName, reprezentující název příkazu, a neterminální symbol Name, reprezentující hodnotu parametru. Parametry jsou speciální skupinou neterminálů, které jsou vždy přepsány na terminální symbol value.

2. ApplicationName → applicationName Name
3. Name → value

Druhý neterminální symbol Reports představuje příkaz či příkazy pro načtení jednoho nebo více reportů. Neterminál je možné přepsat dvěma způsoby. Jeho první varianta obsahuje pouze jeden neterminální symbol Report. Tuto variantu vyjadřuje v případě, kdy vstupní řetězec obsahuje pouze jeden report. Jeho druhá varianta obsahuje neterminální symbol Report s rekurzivním voláním původního neterminálu Reports. Tato varianta se uplatňuje v případě, že vstupní řetězec obsahuje více jak jeden report. Takto navržená pravidla vyjadřují podmínku, že validní záznam obsahuje vždy alespoň jeden report.

4. Reports → Report | Report Reports

Další pravidla definujeme pro neterminál Report. Tento neterminál představuje příkazy spojené s načtením nového reportu a s akcemi prováděnými na tomto reportu. Rozkládá se na dva neterminální symboly LoadReport a Commands. LoadReport se

rozkládá na jeden terminál loadReport představující příkaz načtení reportu a dva neterminály ReportValue a ReportParameters, které reprezentují jeho vlastnosti. ReportValue drží dva neterminály ReportName a ReportFlag, které jsou hodnoty parametru. Jak jsme si již řekli, tato skupina neterminálů se vždy přepisuje na terminál value. Parametry reportu (neterminál ReportParameters) nejsou povinné, proto jedna z variant pravé strany pravidla obsahuje symbol epsilon neboli prázdné slovo. Jeho druhou variantu tvoří dva neterminály. První neterminál je ReportParameter, který opět patří do skupiny hodnota parametru a je v následujícím kroku přepsán, v tomto případě hned čtyřmi terminály value, jelikož jsou parametry reportu v aplikaci popsány čtveřicí hodnot. Druhý neterminál je ReportParameters, neboli rekursivní volání sebe samého vyjadřující skutečnost, že report může mít více parametrů reportu.

5. Report → LoadReport Commands
6. LoadReport → loadReport ReportValue ReportParameters
7. ReportValue → ReportName ReportFlag
8. ReportName → value
9. ReportFlag → value
10. ReportParameters → ε | ReportParameter ReportParameters
11. ReportParameter → value value value value

Neterminál Commands je možné přepsat dokonce třemi různými způsoby. Může nastat situace, kdy byl report zobrazen ve webovém prohlížeči, ale nebyly na něm provedeny žádné uživatelské akce. Proto jedna z variant pravých stran obsahuje symbol epsilon neboli prázdné slovo. Druhou variantu tvoří dva neterminály Command a Commands. Neterminální symbol Command je možné přepsat sadou pravidel, začínajících vždy terminálem, který představuje příkaz odpovídající konkrétní uživatelské akci, následovaný sadou neterminálů, představujících hodnoty jeho parametrů. Neterminál Commands vytváří rekursivní volání sebe sama pro případ, kdy bylo na reportu provedeno hned několik akcí. Speciálním případem je varianta třetí, která popisuje situace, kdy je v reakci na uživatelskou akci z jednoho reportu otevřený další report v podobě dialogu. V dialogu mohou být vykonány příkazy, které mohou eventuálně otevřít další report. Po zavření dialogu mohou následovat další akce prováděné na původním reportu, proto je pravidlo zakončené opět rekursí.

12. Commands → ε | Command Commands | Command openDialog Reports closeDialog Commands

V následující části jsou popsány skupiny parametrů jednotlivých příkazů reprezentující uživatelské akce. Pro větší srozumitelnost některé z nich dále rozšiřujeme na neterminální symboly, které následně přepisujeme na terminální symbol představující hodnotu parametru value.

13. Command → commandDataCellClick CellCoordinate MouseButton
14. Command → commandDataDragDrop DragCellCoordinate DropCellCoordinate
15. Command → commandDataExpandCollapse CellCoordinate
16. Command → commandDataButtinClick ObjectUniqueName MouseButton
17. Command → commandDataChartClick ObjectUniqueName MouseButton CellCoordinate
18. Command → commandDataVisualizationClick ObjectUniqueName CellCoordinate ActionData
19. Command → commandDataWriteback WritebackValueList
20. Command → commandDataLookupClick ObjectUniqueName MouseButton
21. Command → commandDataLVSelectionChanged ObjectUniqueName MouseButton
22. CellCoordinate → Row Column
23. Row → value
24. Column → value
25. MousButton → value
26. DragCellCoordinate → CellCoordinate
27. DropCellCoordinate → CellCoordinate
28. ObjectUniqueName → value
29. ActionData → value

Poslední bod, který je nutné okomentovat je příkaz, který představuje akci zapsání hodnoty do buňky tabulky. Tento příkaz se jmenuje commandDataWriteback a jeho provedením je možné zapsat hodnoty do jedné nebo více buněk najednou. Proto existují pro parametr příkazu WritebackValueList dvě varianty pravé strany pravidla. První obsahuje neterminál CellCoordinate představující lokaci buňky a neterminál WritebackValue

představující zapisovanou hodnotu. Druhá varianta pravidla kromě těchto dvou neterminálů obsahuje ještě neterminál WritebackValueList, který vytváří rekursivní volání sebe sama, pro případ, kdy příkaz zapisuje více hodnot najednou.

30. WritebackValueList → CellCoordinate WritebackValue | CellCoordinate WritebackValue WritebackValueList
31. WritebackValue → value

Zásobníkový automat při rozhodování o tom, podle kterého pravidla se neterminál na vrcholu zásobníku přepíše vzhledem k aktuálnímu symbolu na vstupu, používá překladovou tabulku. V následující kapitole si představíme, jakým způsobem automat s tabulkou pracuje a jak se taková tabulka sestavuje.

4.3.1.2 Odladění kolizí a vytvoření překladové tabulky

Pro snadnější pochopení sestavování tabulky si představíme, jaký význam pro činnost automatu taková tabulka má. Jak už bylo uvedeno výše, tabulka pomáhá automatu v rozhodování, podle jakého pravidla expandovat neterminál na vrcholu zásobníku.

V řádcích tabulky se nachází neterminální symboly a její sloupce tvoří terminální symboly. K sestavení tabulky je nutné nejdříve vypočítat FIRST a FOLLOW množinu. FIRST množina obsahuje terminální symboly, kterými můžou začínat všechna pravidla podle kterých se přepisuje. Množina FOLLOW se vypočítává, pokud existuje alespoň jedno pravidlo, která obsahuje pouze symbol epsilon neboli prázdné slovo. V takovém případě hledáme neterminál na pravé straně všech ostatních pravidel a vypočítáváme FIRST pro symbol napravo od zkoumaného neterminálu. Prvky těchto množin udávají sloupce a tvoří spolu s neterminály v řádcích souřadnice, do kterých se zapisují odpovídající pravidla.

Při výpočtu množin pro navrhovaný analyzátor s použitím pravidel gramatiky definované v předchozí sekci se objevilo hned několik kolizí, které vedou k nejednoznačnosti při rozhodování o výběru přepisovacích pravidel. První kolize se vyskytuje ve čtvrtém pravidle navržené gramatiky. Jedná se o kolizi FIRST-FIRST, kdy obě z variant přepisovacích pravidel začínají stejným terminálních symbolem. K odstranění takové nejednoznačnosti existuje metoda faktorizace, kde tento symbol vytkneme a přiřadíme k němu nově vytvořený neterminální symbol, který představuje zbytek pravidla

po vytknutí. Tento nový neterminál tedy přidáme do gramatiky s přepisovacími pravidly odpovídajícími zbytkům pravidel původního neterminálu.
Původní návrh:

4. Reports → Report | Report Reports

FIRST(Reports → Report) = {loadReport}
FIRST(Reports → Reprort Reports) = {loadReport}

Překladová tabulka by tedy na souřadnicích Reports, loadReport obsahovala dvě pravidla.

Faktorizace:

4. Reports → Report Reports`
5. Reports` → ε
6. Reports` → Reports

FIRST(Reports → Report Reports') = {loadReport}
FIRST(Reports` → Reports) = {loadReport}
FIRST(Reports` → ε) = {ε}

V případě, že jedno z pravidel obsahuje pouze symbol epsilon, je nutné počítat FOLLOW množinu, která nám určuje, pro jaké symboly na vstupu expandovat daný neterminál na prázdný řetězec. V tomto případě budeme hledat, kde se na pravé straně pravidel vyskytuje neterminální symbol Reports`. Neterminál se vyskytuje na pravé straně pouze v pravidle číslo 4. V tomto pravidle však není neterminál následován žádným symbolem. Tzv. „zabalíme“ hledaný neterminál do neterminálu na levé straně tohoto pravidla a dále zjišťujeme FOLLOW množinu tohoto neterminálu, to jest hledáme na pravé straně pravidel neterminální symbol Reports. Neterminál Reports se na pravé straně nachází hned ve dvou pravidlech, 1 a 18. V pravidle 1 neterminál také nenásleduje žádný symbol, takže pravou stranu pravidla opět zabalíme a dále hledáme na pravé straně neterminální symbol S. Symbol S se však na pravé straně pravidel nikde nevyskytuje, proto vložíme do námi hledané množiny FOLLOW(Reports') symbol epsilon.

V pravidle 18 je neterminální prvek následován terminálním symbolem closeReport, který přidáme do FOLLOW množiny.

FOLLOW(Reports`) → {ε, closeReport}

Překladová tabulka bude tedy obsahovat na následujících souřadnicích tyto pravidla:

Reports, loadReport: Reports = Report Reports‘

Reports’, loadReport: Reports` = Reports

Reports’, closeReport: Reports` = ε

Reports’, ε : Reports` = ε

Ke kolizi FIRST-FIRST dochází v navržené gramatice také v pravidle číslo 12, kde je vytvořen dodatečný neterminál FollowingCommands a v pravidle 30, kde je vytvořen neterminál WritebackValueList`. Tyto nedostatky odstraníme stejným způsobem jako jsme popsali výše. Kolize může nastat i při výpočtu množiny FOLLOW, kdy dochází k takové nejednoznačnosti, že se prvek FOLLOW množiny rovná některému z prvků množiny FIRST pro daný neterminál. K takové kolizi ale v našem návrhu nedochází. Po odladění nejednoznačností dostáváme finální verzi gramatických pravidel pro navrhovaný syntaktický analyzátor.

Finální verze gramatiky syntaktického analyzátoru:

1. S → AppName Reports
2. AppName → applicationName Name
3. Name → value
4. Reports → Report Reports‘
5. Reports‘ → ε
6. Reports‘ → Reports
7. Report → LoadReport Commands
8. LoadReport → loadReport ReportValue ReportParameters
9. ReportValue → ReportName ReportFlag
10. ReportName → value
11. ReportFlag → value
12. ReportParameters → ε
13. ReportParameters → ReportParameter ReportParameters

14. ReportParameter → value value value value
15. Commands → ε
16. Commands → Command FollowingCommands
17. FollowingCommands → Commands
18. FollowingCommands → openDialog Reports closeDialog Commands
19. Command → commandDataCellClick CellCoordinat MouseButton
20. Command → commandDataDragDrop DragCellCoordinate DropCellCoordinate
21. Command → commandDataExpandCollapse CellCoordinate
22. Command → commandDataButtinClick ObjectUniqueName MouseButton
23. Command → commandDataChartClick ObjectUniqueName MouseButton CellCoordinate
24. Command → commandDataVisualizationClick ObjectUniqueName CellCoordinate ActionData
25. Command → commandDataWriteback WritebackValueList
26. Command → commandDataLookupClick ObjectUniqueName MouseButton
27. Command → commandDataLVSelectionChanged ObjectUniqueName MouseButton
28. CellCoordinate → Row Column
29. Row → value
30. Column → value
31. MousButton → value
32. DragCellCoordinate → CellCoordinate
33. DropCellCoordinate → CellCoordinate
34. ObjectUniqueName → value
35. ActionData → value
36. WritebackValueList → CellCoordinate WritebackValue WritebackValueList‘
37. WritebackValueList‘ → ε
38. WritebackValueList‘ → WritebackValueList
39. WritebackValue → value

V tuhle chvíli je ten správný čas pro sestavení překladové tabulky. Vypočítáme si FIRST množinu pro všechny neterminály navržené gramatiky a v případě potřeby i jejich FOLLOW množinu. Do řádků tabulky napíšeme neterminální symboly a do sloupců napíšeme terminální symboly. Do souřadnic, kde se neterminální prvek potkává s prvky jeho FIRST a FOLLOW množiny, zapisujeme číslo pravidla, ze kterého se daný prvek množiny vypočítal.

Tabulka 1 - Překladová tabulka

	applicationName	value	loadReport	commandData CellClick	commandData DragDrop
S	1				
AppName	2				
Name		3			
Reports			4		
Reports'			6		
Report			7		
LoadReport			8		
ReportValue		9			
ReportParameters		13	12	12	12
ReportParameter		14			
ReportName		10			
ReportFlag		11			
Commands			15	16	16
FollowingCommands			17	17	17
Command				19	20
CellCoordinate		28			
Row		29			
Column		30			
MouseButton		31			
DragCellCoordinate		32			
DropCellCoordinate		33			
ObjectUniqueName		34			
ActionData		35			
WritebackValueList		36			
WritebackValueList'		38	37	37	37
WritebackValue		39			

	commandData ExpandCollapse	commandData ButtonClick	commandData ChartClick	commandData VisualizationClick	commandData Writeback
S					
AppName					
Name					
Reports					

Reports'					
Report					
LoadReport					
ReportValue					
ReportParameters	12	12	12	12	12
ReportParameter					
ReportName					
ReportFlag					
Commands	16	16	16	16	16
FollowingCommands	17	17	17	17	17
Command	21	22	23	24	25
CellCoordinate					
Row					
Column					
MouseButton					
DragCellCoordinate					
DropCellCoordinate					
ObjectUniqueName					
ActionData					
WritebackValueList					
WritebackValueList'	37	37	37	37	37
WritebackValue					
	commandData LookupClick	**commandData LVSelectionChanged**	**openDialog**	**closeDialog**	**ε**
S					
AppName					
Name					
Reports					
Reports'				5	5
Report					
LoadReport					
ReportValue					
ReportParameters	12	12		12	12
ReportParameter					
ReportName					
ReportFlag					
Commands	16	16		15	15
FollowingCommands	17	17	18	17	17
Command	26	27			
CellCoordinate					
Row					
Column					
MouseButton					

DragCellCoordinate					
DropCellCoordinate					
ObjectUniqueName					
ActionData					
WritebackValueList					
WritebackValueList'	37	37		37	37
WritebackValue					

4.4 Sémantická analýza

Sémantická pravidla vyjadřují význam řetězců, které jsou syntaktickou analýzou vyhodnoceny jako validní pro danou gramatiku. Na vstup sémantického analyzátoru je předložen derivační strom zkonstruovaný při analýze syntaktické. V této práci jsem pro definování významu jednotlivých položek derivačního stromu zvolila atributovou gramatiku. Každému symbolu syntaktického pravidla je podle potřeby přiřazen jeden nebo více atributů. Symboly rozšířené o atributy používáme pro sestavení sémantických pravidel. V průběhu analýzy se vypočítává hodnota atributů od nejspodnějších listů derivačního stromu směrem nahoru a s jejich konkrétními hodnotami jsou na základě nadefinovaných pravidel tvořeny instance objektů datového modelu. Tyto objekty jsou spojovány do složitějších datových struktur, které na konci analýzy tvoří abstraktní model představující výstup překladače. Jinými slovy, sémantická pravidla určují vztahy mezi symboly gramatiky a definují, jakým způsobem je výstupní kód vytvářen.

Jelikož cílem práce je vytvořit datovou strukturu reprezentující příkazy zaznamenané v logu aplikace, která bude následně serializovaná do XML formátu, je nutné vytvoření datového modelu. Obecně datový model popisuje formát a strukturu dat, včetně vztahů mezi jednotlivými prvky.

Ještě před tím, než se pustíme do definování sémantického analyzátoru pro řešený návrh překladače, je nutné okomentovat, jak gramatická pravidla souvisí s datovým modelem. Přístup k realizaci sémantické fáze překladače není vždy stejný, spíše se jedná o subjektivní řešení vývojáře. Datový model vytvoříme za pomoci programovacího jazyka, což nabízí možnost definovat část sémantických pravidel v kódu jeho implementace. Vzhledem k tomu, že v rámci mé pracovní pozice čas od času pracuji v C# programovacím jazyce, bylo pro mě naprosto přirozené této skutečnosti využít. Datový model a sémantická

pravidla proto nejsou oddělené části řešení, ale jedná se o velmi úzce související aparáty, jejichž vývoj probíhá paralelně. V následujících kapitolách bude vysvětleno vytvoření výstupní datové struktury překladače za předpokladu, že byl ke zpracování předložen validní řetězec.

4.4.1 Vytvoření datové struktury

Jak již bylo řečeno v předchozí části kapitoly Sémantická pravidla, pro vytvoření datového modelu jsme si zvolili programovací jazyk C#. Pro všechny druhy atributů sémantických pravidel vytváříme třídy, ve kterých definujeme, jaká data mají instance těchto tříd držet. Data, která jsou do těchto tříd ukládána, přímo vyplývají z pravých stran sémantických pravidel, kde se pro konkrétní pravidla vyskytují různé druhy atributů.

Pro snadnější orientaci v pseudokódu sémantických pravidel, kde definujeme vytvoření instancí konkrétních tříd, jsme zvolili unikátní názvosloví. Každá třída přejímá název neterminálního symbolu, jemuž je přiřazen atribut, do kterého chceme uložit instanci této třídy. Tento název je doplněný příponou „Data“. V případě, že je daný druh atributu přiřazen k několika různým neterminálům, volíme název neterminálu, který bude nejlépe „reprezentovat“, jaká data daná třída bude ukládat.

Při kompletní implementaci překladače by byly instance tříd datového modelu vytvářeny s konkrétními hodnotami parametrů ve spuštěném programu. Celá implementace datového modelu je odevzdána jako příloha závěrečné práce. V této písemné části si ukážeme a popíšeme pouze jednu třídu pro atribut REPORTSLIST neterminálního symbolu Reports.

```
    public class ReportsData
    {
        public List<ReportData> ReportList { get; set; }

        public ReportsData(ReportData report, ReportsData reportsList, ReportsData
dialogReportsList)
        {
            ReportList = new List<ReportData>();

            ReportList.Add(report);
            List<ReportData> list = reportsList.ReportList;
            List<ReportData> dialogList = dialogReportsList.ReportList;
            ReportList.AddRange(list);
            ReportList.AddRange(dialogList);
        }

        public ReportsData(ReportsData reportsList, ReportsData dialogReportsList)
        {
            ReportList = new List<ReportData>();

            List<ReportData> list = reportsList.ReportList;
            List<ReportData> dialogList = dialogReportsList.ReportList;
            ReportList.AddRange(list);
            ReportList.AddRange(dialogList);
        }

        public ReportsData(ReportsData reportsList)
        {
            ReportList = new List<ReportData>();

            List<ReportData> list = reportsList.ReportList;

            ReportList.AddRange(list);
        }

        public ReportsData()
        {
            ReportList = new List<ReportData>();
        }
    }
```

Jako první je ve třídě nadefinován data member ReportsList. Data member představuje list objektů typu ReportData, který drží seznam reportů. Zbytek kódu definuje čtyři konstruktory s unikátní sadou parametrů. Definice konstruktorů byla stanovena na základě vytvořených sémantických pravidel, která jsou k nahlédnutí v následující kapitole.

První konstruktor je volán ze sémantického pravidla číslo 4. V konstruktoru je vytvořen nový list, do kterého se vloží instance typu ReportData uložená v atributu Report.REPORT a obsah listů ze dvou instancí typu ReportsData uložených v atributech Reports‘.LIST, Report.REPORTSLIST. Při zpracování konkrétního webového záznamu jsou hodnoty atributů použitých v parametrech konstruktoru vypočteny níže v derivačním

stromu a dále předávány směrem nahoru uzlům, které je potřebují k vytvoření dalšího objektu výstupního kódu.

4) Reports → Report Reports‘;

Reports.REPORTSLIST = new ReportsData(Report.REPORT, Reports‘.REPORTSLIST, Report.REPORTSLIST)

```
public ReportsData(ReportData report, ReportsData reportsList, ReportsData dialogReportsList)
        {
            ReportList = new List<ReportData>();

            ReportList.Add(report);
            List<ReportData> list = reportsList.ReportList;
            List<ReportData> dialogList = dialogReportsList.ReportList;
            ReportList.AddRange(list);
            ReportList.AddRange(dialogList);
        }
```

Druhý konstruktor je volaný ze sémantického pravidla číslo 18, kde vypočítává hodnotu atributu FollowingCommands.REPORTSLIST. Opět je v konstruktoru vytvořen nový list, do kterého jsou předány hodnoty atributů předaných jako parametry. Jinými slovy do nově vytvořeného listu přidáváme obsahy listů z instancí Reports.LIST a Commands.REPORTSLIST.

18) FollowingCommands → openDialog Reports closeDialog Commands;
FollowingCommands.REPORTSLIST = new ReportsData(Reports.REPORTSLIST, Commands.REPORTSLIST)

```
public ReportsData(ReportsData reportsList, ReportsData dialogReportsList)
        {
            ReportList = new List<ReportData>();

            List<ReportData> list = reportsList.ReportList;
            List<ReportData> dialogList = dialogReportsList.ReportList;
            ReportList.AddRange(list);
            ReportList.AddRange(dialogList);
        }
```

Třetí konstruktor je volán například ze sémantického pravidla číslo 6, kde vypočítavá hodnotu atributu Reports‘.LIST. V konstruktoru je vytvořen nový list, do kterého je přidán list z instance atributu Reports.LIST.

6) Reports‘ → Reports

Reports‘.REPORTSLIST = new ReportsData(Reports.REPORTSLIST)

```
public ReportsData(ReportsData reportsList)
       {
           ReportList = new List<ReportData>();

           List<ReportData> list = reportsList.ReportList;

           ReportList.AddRange(list);
       }
```

Poslední konstruktor je volán ze sémantického pravidla číslo 5, kde definuje hodnotu atributu Reports‘.REPORTSLIST. V konstruktoru je vytvořen pouze prázdný list.

5) Reports‘ → ε

Reports‘.REPORTSLIST = new ReportsData()

```
public ReportsData()
       {
           ReportList = new List<ReportData>();
       }
```

4.4.2 Návrh sémantických pravidel pomocí atributové gramatiky

Pojďme se podívat na konstrukci sémantických pravidel pro náš konkrétní návrh. Pravidla jsou zapsána v takzvaném pseudokódu odkazujícího se na datový model. Pseudokód je neformální zápis algoritmu, který dodržuje konvence programovacího jazyka, ale neobsahuje detailní implementaci. Sémantická pravidla se vytváří pro všechna pravidla syntaktického analyzátoru. Tato pravidla určují, jaké konkrétní hodnoty jsou potřeba pro vytvoření objektů výstupního kódu.

První sémantické pravidlo říká, že pro vytvoření objektu reprezentujícího testovací scénář je nutné, aby mu ze spodu derivačního stromu byly předány instance typu ApplicationNameData a ReportsData. Tyto dvě instance byly vytvořeny podle jiných sémantických pravidel, kde se jejich hodnota uložila do odpovídajících atributů, které jsou

pak použity jako parametry konstruktoru vytvářejícího objekt v prvním pravidle. Detaily pro vytvoření instance objektu test scénáře se nachází v datovém modelu. Vytvořený objekt reprezentující testovací scénář tak dává význam počátečnímu neterminálu S. Stejným způsobem pokračujeme ve zbytku návrhu sémantického analyzátoru.

1. S → AppName Reports;
   S.START = new ScenarioData(AppName.APPNAME, Reports.REPORTSLIST)
2. AppName → applicationName Name;
   AppName.APPNAME = new ApplicationNameData(Name.NAME)
3. Name → value;
   Name.NAME = new NameValueData(value.VALUE)

Komplikovanější situace nastává například v pravidlech číslo 4, 5, 6, kde byla použita metoda faktorizace pro odstranění FIRST-FIRST kolize. Tato pravidla definují několik možností pro tvoření atributu typu ReportsData. Pravidlo 4 vyjadřuje přidání reportu do seznamu reportů. Pravidlo číslo 5 vyjadřuje vytvoření prázdného listu reportů (jedná se o ukončení rekurze konstruující seznam reportů). A pravidlo číslo 6 vyjadřuje překopírování již existujícího listu reportů. V datovém modelu je definovaná třída pro atributy typu ReportsData, kterou jsme si podrobněji popsali v předchozí části. Podobná situace se vyskytuje pro výpočet atributů typu ReportParamatersData (pravidla 12, 13) a WritebackValueListData (pravidla 35, 37, 38).

4. Reports → Report Reports‘;
   Reports.REPORTSLIST = new ReportsData(Report.REPORT,
   Reports‘.REPORTSLIST, Report.REPORTSLIST)
5. Reports‘ → ε
   Reports‘.REPORTSLIST = new ReportsData()
6. Reports‘ → Reports
   Reports‘.REPORTSLIST = new ReportsData(Reports.REPORTSLIST)
7. Report → LoadReport Commands;
   Report.REPORT = new ReportData(LoadReport.LOADREPORT, Commands.
   COMMANDSLIST)
   Report.REPORTSLIST = new ReportsData(Commands.REPORTSLIST)

8. LoadReport → loadReport ReportValue ReportParameters;

   LoadReport.LOADREPORT = new

   LoadReportData(ReportValue.REPORTVALUE, ReportParameters.LIST)

9. ReportValue → ReportName ReportFlag

   ReportValue.REPORTVALUE = new

   ReportValueData(ReportName.REPORTNAME, ReportFlag.REPORTFLAG)

10. ReportName → value;

    ReportName.REPORTNAME = new ReportNameData(value.VALUE)

11. ReportFlag → value;

    ReportFlag.REPORTFLAG = new ReportFlagData(value.VALUE)

12. ReportParameters → ε;

    ReportParameters.REPORTPARAMETERS = new ReportParametersData()

13. ReportParameters → ReportParameter ReportParameters

    ReportParameters.LIST = new

    ReportParametersData(ReportParameter.REPORTPARAMETER,

    ReportParameters.LIST)

14. ReportParameter → $value_1$ $value_2$ $value_3$ $value_4$;

    ReportParameter.REPOPRTPARAMETER = new

    ReportParameterData($value_1$.VALUE, $value_2$.VALUE, $value_3$.VALUE,

    $value_4$.VALUE)

15. Commands → ε;

    Commands.COMMANDSLIST = new CommandsData()

    Commands.REPORTSLIST = new ReportsData()

16. Commands → Command FollowingCommands;

    Commands.COMMANDSLIST = new CommandsData(Command.COMAND,

    FollowingCommands.COMMANDSLIST)

    Commands.REPORTSLIST = new ReportsData(FollowingCommands.

    REPORTSLIST)

17. FollowingCommands → Commands;

```
FollowingCommands.COMMANDSLIST = new
CommandsData(Commands.LIST)
FollowingCommands.REPORTSLIST = new
ReportsData(Commands.REPORTSLIST)
```

Poslední bod, který je nutné okomentovat je pravidlo číslo 18, ve kterém definujeme otevření dialogu. Dialog není nic jiného než načtení dalšího reportu či reportů, ze kterých je možné dokonce otevírat další dialogy. Načtení takového reportu či reportů musí být vyvoláno uživatelskou akcí, proto je nutné definovat toto extra pravidlo. Získaný seznam reportů je nutné předat nahoru až do pravidla číslo 4, kde se všechny existující instance listů reportů spojí do jedné. Proto bylo nutné pro uložení listu nadefinovat další atribut FollowingCommands.REPORTSLIST, do kterého se instance ReportsData uloží. Atribut musel být následně přidán do několika dalších neterminálů, aby bylo umožněno předání hodnoty atributů do vyšších pater derivačního stromu.

Jako první hledáme, kde se na pravé straně syntaktického pravidla vyskytuje neterminalní symbol FollowingCommands. V korespondujícím sémantickém pravidle je přidán nově vytvořený atribut, abychom zaručili předání hodnoty atributu vypočítané níže. Taková situace nastává v pravidle číslo 16, kde definujeme, jakým způsobem se přepisuje neterminální symbol Commands. Po přidání nového atributu analogicky hledáme syntaktická pravidla, na jejichž pravých stranách se vyskytuje neterminál Commands. Je tedy nutné přidat nový atribut také do pravidel 17 a 7. Vytvořené instance listu reportů uložené v atributech využijeme při volání konstruktoru v pravidle 4, kde se všechny instance listu reportů sloučí a uloží do přiřazeného atributu.

18. FollowingCommands → openDialog Reports closeDialog Commands;

```
FollowingCommands.COMMANDSLIST = new
CommandsData(Commands.LIST)
FollowingCommands.REPORTSLIST = new
ReportsData(Reports.REPORTSLIST, Commands.REPORTSLIST)
```

19. Command → commandDataCellClick CellCoordinate MouseButton;
    Command.COMMAND = new
    CommandDataCellClickData(CellCoordinate.CELLCOORDINATE,
    mouseButton.VALUE)
20. Command → commandDataDragDrop DragCellCoordinate DropCellCoordinate;
    Command.COMMAND = new
    CommandDataDragDropData(DragCellCoordinate.DRAGCELLCOORDINATE,
    DropCellCoordinate.DROPCELLCOORDINATE)
21. Command → commandDataExpandCollapse CellCoordinate;
    Command.COMMAND = new
    CommandDataExpandCollapseData(CellCoordinate.CELLCOORDINATE)
22. Command → commandDataButtonClick ObjectUniqueName MouseButton;
    Command.COMMAND = new
    CommandDataButtonClickData(ObjectUniqueName.OBJECTUNIQUENAME,
    MouseButton.MOUSEBUTTON)
23. Command → commandDataChartClick ObjectUniqueName MouseButton
    CellCoordinate;
    Command.COMMAND = new
    CommanDataChartClickData(ObjectUniqueName.OBJECTUNIQUENAME,
    MouseButton.MOUSEBUTTON, CellCoordinate.CELLCOORDINATE)
24. Command → commandDataVisualizationClick ObjectUniqueName
    CellCoordinate ActionData;
    Command.COMMAND = new
    CommandDataVisualizationClickData(ObjectUniqueName.OBJECTUNIQUENA
    ME, CellCoordinate.CELLCOORDINATE, ActionData.ACTIONDATA)
25. Command → commandDataWriteback WritebackValueList;
    Command.COMMAND = new
    CommandDataWritebackData(WritebackValueList.LIST)
26. Command → commandDataLookupClick ObjectUniqueName MouseButton;
    Commend.COMMAND = new
    CommandDataLookupClickData(ObjectUniqueName.OBJECTUNIQUENAME,
    MouseButton.MOUSEBUTTON)

27. Command → commandDataLVSelectionChanged ObjectUniqueName MouseButton;
    Command.COMMAND = new CommandDataLVSelectionChanged(ObjectUniqueName.OBJECTUNIQUENAME, MouseButton.MOUSEBUTTON)
28. CellCoordinate → Row Column;
    CellCoordinate.CELLCOORDINATE = new CellCoordinate(Row.ROW, Column.COLUMN)
29. Row → value;
    Row.ROW = new RowData(value.VALUE)
30. Column → value;
    Column.COLUMN = new ColumnData(value.VALUE)
31. MouseButton → value;
    MouseButton.MOUSEBUTTON = new MouseButtonData(value.VALUE)
32. DragCellCoordinate → CellCoordinate;
    DragCellCoordinate.DRAGCELLCOORDINATE = new CellCoordinateData(CellCoordiante.CELLCOORDNATE)
33. DropCellCoordinate → CellCoordinate;
    DropCellCoordinate.DROPCELLCOORDINATE = new CellCoordinateData(CellCoordinate.CELLCOORDINATE)
34. ObjectUniqueName → value;
    ObjectUniqueName.OBJECTUNIQUENAME -> new ObjectUniqueNameData(value.VALUE)
35. ActionData → value
    ActionData.ACTIONDATA = new ActionDataData(value.VALUE)
36. WritebackValueList → CellCoordinate WritebackValue WritebackValueList‘;
    WritebackValueList.LIST = WritebackValuelListData(CellCoordinate.CELLCOORDINATE, WritebackValue.WRITEBACKVALUE, WritebackValueList‘.LIST)
37. WritebackValueList‘ → ε;
    WritebackValueList‘.LIST = new WritebackValueListData()

38. WritebackValueList‘ → WritebackValueList;
    WritebackValueList‘.LIST = new
    WritebackValueListData(WritebackValueList.LIST)
39. WritebackValue → value;
    WritebackValue.WRITEBACKVALUE = new
    WritebackValueData(value.VALUE)

Návrh sémantických pravidel by se tedy dal obecně shrnout tak, že každému neterminálu je nadefinován jeden nebo více syntetizovaných atributů, do nichž se přiřazují instance tříd datového modelu. To nám umožňuje vytvořit pravidla jako prostá volání konstruktorů tříd a logika popisující skládání dat do výsledných struktur je definována v konstruktorech tříd datového modelu.

5 Výsledky a diskuse

Jako výsledek práce bude v této kapitole zpracován konkrétní záznam webové aplikace. Popíšeme si zpracování vstupního kódu lexikální a syntaktickou analýzou. Sémantická analýza v kombinaci s datovým modelem je již poměrně komplikovaný proces, který by bylo obtížné a velmi pracné rozepisovat písemně.

Záznam popisuje pár základních funkcionalit aplikace. Pojďme si popsat akce, které ve webovém prostředí náš konkrétní záznam aplikace vytvoří. Jako první otevřeme aplikaci, ve které následně otevřeme report. Na reportu zmáčkneme tlačítko, které otevře dialog. V nově otevřeném dialogu zapíšeme hodnotu a tím se dialog zavře tak, že se objevíme na původním reportu. V původním reportu následně zmáčkneme jiný druh tlačítka, který nám otevře dialog pouze k nahlédnutí. Po zavření dialogu záznam aplikace končí.

Záznam webové aplikace:

```
applicationname "value"
loadreport "value" "value"
commanddatabuttonclick "value" "value"
opendialog
loadreport "value" "value"
commanddatawriteback "value" "value" "value"
closedialog
commanddatalookupclick "value" "value"
opendialog
loadreport "value" "value"
closedialog
```

Tento text záznamu webové aplikace předkládáme jako první ke zpracování lexikální analýze, která ho prochází znak po znaku a vytváří posloupnost tokenů.

Posloupnost tokenů:

applicationName value(..) loadreport value(..) value(..) commanddatabuttonclick value(..) value(..) opendialog loadreport value(..) value(..) commanddatawriteback value(..) value(..) value(..) closedialog commanddatalookupclick value(..) value(..) opendialog loadreport value(..) value(..) closedialog

Tato posloupnost tokenů je předložena na vstup zásobníkového automatu syntaktického analyzátoru, jehož fungování si vzápětí ukážeme. Rozbor syntaktické analýzy rozepíšeme do tří sloupců – stav zásobníku, dva znaky na vstupu automatu a činnost, kterou automat v daném kroce vykonává. Při rozhodování, jaké pravidlo využijeme k přepsání

neterminálu na vrcholu zásobníku, používáme překladovou tabulku. Podíváme se v překladové tabulce jaké číslo pravidla je zapsané na souřadnicích tvořených počátečním neteminálním symbolem S a prvním nezpracovaným terminálním symbolem applicationName na vstupu automatu. Zjistíme tedy, že symbol S expandujeme podle prvního pravidla a výsledek zapíšeme do dalšího řádku tak, že neterminál S na zásobníku nahradíme pravou stranou prvního pravidla.

Na vrcholu zásobníku se momentálně nachází neterminální symbol AppName. V překladové tabulce hledáme číslo pravidla na souřadnicích s terminálním symbolem applicationName, který je na vstupu automatu. Z toho plyne, že v tomto kroce automatu přepisujeme aktuální neterminál podle pravidla 2. Na vrchol zásobníku se tak dostává terminální symbol applicationName. Ve chvíli, kdy se na vrcholu zásobníku nachází terminální symbol, dochází ke srovnání symbolu se symbolem na vstupu automatu. Pokud se symboly shodují, jsou oba dva z analýzy odstraněny. Stejným způsobem pokračujeme v analýze dál.

Stav zásobníku	2 znaky vstupu	činnost
S	applicationName value(..)	expanze 1.
AppName Reports	applicationName value(..)	expanze 2.
applicationName Name Reports	applicationName value(..)	srovnání
Name Reports	value(..) loadreport	expanze 3.
value Reports	value(..) loadreport	srovnání
Reports	loadreport value(..)	expanze 4.
Report Reports‘	loadreport value(..)	expanze 7.
LoadReport Commands Reports‘	loadreport value(..)	expanze 8.
loadReport ReportValue ReportParameters Commands Reports‘	loadreport value(..)	srovnání
ReportValue ReportParameters Commands Reports‘	value(..) value(..)	expanze 9.
ReportName ReportFlag ReportParameters Commands Reports‘	value(..) value(..)	expanze 10.
value ReportFlag ReportParameters Commands Reports‘	value(..) value(..)	srovnání
ReportFlag ReportParameters Commands Reports‘	value(..) commanddatabuttonclick	expanze 11.
value ReportParameters Commands Reports‘	value(..) commanddatabuttonclick	srovnání
ReportParameters Commands Reports‘	commanddatabuttonclick value(..)	expanze 12.

V tomto bodě analýzy nastává speciální případ, kdy je neterminál (ReportParameters) na vrcholu zásobníku přepisován symbolem epsilon neboli prázdným slovem. V takovém případě se neterminální prvek z vrcholu zásobníku odstraní a analýza pokračuje dál standartním způsobem.

Commands Reports‘	commanddatabuttonclick value(..)	expanze 16.
Command FollowingCommands Reports‘	commanddatabuttonclick value(..)	expanze 22.
commandDataButtonClick ObjectUniqueName MouseButton FollowingCommands Reports‘	commanddatabuttonclick value(..)	srovnání
ObjectUniqueName MouseButton FollowingCommands Reports‘	value (..) value(..)	expanze 34.
value (..) MouseButton FollowingCommands Reports‘	value (..) value(..)	srovnání
MouseButton FollowingCommands Reports‘	value(..) opendialog	expanze 31.
value(..) FollowingCommands Reports‘	value(..) opendialog	srovnání
FollowingCommands Reports‘	opendialog loadreport	expanze 18.
openDialog Reports closeDialog Commands Reports‘	opendialog loadreport	srovnání
Reports closeDialog Commands Reports‘	loadreport value(..)	expanze 4.

Report Reports‘ closeDialog Commands Reports‘	loadreport value(..)	expanze 7.
LoadReport Commands closeDialog Commands Reports‘	loadreport value(..)	expanze 8.
loadReport ReportValue ReportParameters Commands closeDialog Commands Reports‘	loadreport value(..)	srovnaní
ReportValue ReportParameters Commands closeDialog Commands Reports‘	value(..) value(..)	expanze 9.
value value ReportParameters Commands closeDialog Commands Reports‘	value(..) value(..)	srovnání
value ReportParameters Commands closeDialog Commands Reports‘	value(..) commanddatawriteback	srovnání
ReportParameters Commands closeDialog Commands Reports‘	commanddatawriteback value(..)	expanze 12.
Commands closeDialog Commands Reports‘	commanddatawriteback value(..)	expanze 16.
Command FollowingCommands closeDialog Commands Reports‘	commanddatawriteback value(..)	expanze 25.
commandDataWriteback WritebackValueList FollowingCommands closeDialog Commands Reports‘	commanddatawriteback value(..)	srovnání
WritebackValueList FollowingCommands closeDialog Commands Reports‘	value(..) value(..)	expanze 36.
CellCoordinate WritebackValue WritebackValueList‘ FollowingCommands closeDialog Commands	value(..) value(..)	expanze 28.
Row Column WritebackValue WritebackValueList‘ FollowingCommands closeDialog Commands	value(..) value(..)	expenze 29.
value Column WritebackValue WritebackValueList‘ FollowingCommands closeDialog Commands	value(..) value(..)	srovnání
Column WritebackValue WritebackValueList‘ FollowingCommands closeDialog Commands	value(..) value(..)	expanze 30.
value WritebackValue WritebackValueList‘ FollowingCommands closeDialog Commands	value(..) value(..)	srovnání
WritebackValue WritebackValueList‘ FollowingCommands closeDialog Commands	value(..) closedialog	expanze 39.
value WritebackValueList‘ FollowingCommands closeDialog Commands	value(..) closedialog	srovnání
WritebackValueList‘ FollowingCommands closeDialog Commands	closedialog commanddatalookupclick	expanze 37.
FollowingCommands closeDialog Commands	closedialog commanddatalookupclick	expanze 17.
closeDialog Commands	closedialog commanddatalookupclick	srovnání
Commands	commanddatalookupclick value(..)	expanze 16.
Command FollowingCommands	commanddatalookupclick value(..)	expanze 26.
commandDataLookupClick ObjectUniqueName MouseButton FollowingCommands	commanddatalookupclick value(..)	srovnání
ObjectUniqueName MouseButton FollowingCommands	value(..) value(..)	expand 34.
value MouseButton FollowingCommands	value(..) value(..)	srovnání
MouseButton FollowingCommands	value(..) opendialog	expanze 31.
value FollowingCommands	value(..) opendialog	srovnání
FollowingCommands	opendialog loadreport	expanze 18.
openDialog Reports closeDialog Commands	opendialog loadreport	srovnání
Reports closeDialog Commands	loadreport value(..)	expanze 4.
Report Reports‘ closeDialog Commands	loadreport value(..)	expanze 7.
LoadReport Reports‘ closeDialog Commands	loadreport value(..)	expanze 8.
loadReport ReportValue ReportParameters Reports‘ closeDialog Commands	loadreport value(..)	srovnání
ReportValue ReportParameters Reports‘ closeDialog Commands	value(..) value(..)	expanze 9.
ReportName ReportFlag ReportParameters Reports‘ closeDialog Commands	value(..) value(..)	expanze 10.
value ReportFlag ReportParameters Reports‘ closeDialog Commands	value(..) value(..)	srovnání
ReportFlag ReportParameters Reports‘ closeDialog Commands	value(..) closedialog	expanze 11.
value ReportParameters Reports‘ closeDialog Commands	value(..) closedialog	srovnání
ReportParameters Reports‘ closeDialog Commands	closedialog	expanze 12.
Reports‘ closeDialog Commands	closedialog	expanze 5.
closeDialog Commands	closedialog	srovnání
Commands	ε	expanze 15.
ε	ε	konec

Vstupní řetězec je vyhodnocen jako validní, pokud na konci analýzy není žádný symbol na vstupu ani na zásobníku. Výstupem analýzy je derivační strom, který následně předáváme na vstup analýzy sémantické.

6 Závěr

Cílem práce bylo seznámit se s teorií formálních jazyků, gramatik a automatů a osvojit si základní jazykové analýzy využívané v teorii překladačů. Tyto znalosti následně využít pro návrh programu, který na vstupu zpracovává textový záznam komerční webové aplikace a na jeho výstupu generuje strukturu objektů představující příkazy zapsané v logu aplikace. Návrh takového překladače se skládá z několika částí – z analýzy lexikální, syntaktické a sémantické.

V rámci práce se podařilo zhotovit funkční návrh překladače, který najde využití v reálném pracovním prostředí mého týmu. Byla navržena lexikální analýza, která dokáže rozpoznat jednotlivá slova ve vstupním řetězci. Dále syntaktická analýza, která zkoumá logickou správnost vstupního řetězce. A jako poslední sémantická analýza, která dává vstupnímu řetězci význam a vytváří jeho vnitřní formu generovanou na výstupu programu. V programovacím jazyce C# byl pro účely této práce vytvořen datový model, se kterým překladač pracuje.

Určité časti práce by bylo možné v budoucnu vylepšit. Existují speciální příkazy v komunikaci aplikace s webovým rozhraním, které byly z časových důvodu z návrhu vynechány. Pro kompletní implementaci programu bude nutné tyto příkazy doplnit. Další prostor pro zlepšení se nachází v lexikální analýze, kde by bylo vhodné umožnit výskyt uvozovek uvnitř slov nesoucí lexikální význam hodnota. Stěžejním rozšířením práce by byla kompletní implementace programu. Sémantická část návrhu překladač je popsán pomocí pseudokódu a datového modelu, který do značné míry popisuje, jakým způsobem by byla tvořena výstupní data programu.

Téma bakalářské práce a návrh fungujícího překladače pro mě byla velká výzva. Jsem vděčná za nově získané znalosti, které výrazně přispěly k mému komplexnímu vnímání IT odvětví. Překladač bude kompletně implementován v rámci mého pracovního projektu. Výsledný program výrazně urychlí a zjednoduší definování testovacích scénářů pro integrační automatické testy vyvíjeného produktu. Očekávám, že tato optimalizace přispěje k většímu pokrytí funkcionalit webové aplikace v těchto testech.

7 Bibliografie

1. **ČERNÁ, Ivana, KŘETÍNSKÝ,Mojmír a KUČERA Antonín.** *Automaty a formální jazyk I.* Brno: Masarykova univerzita : Elportál, 2006. ISSN 1802-128X.
2. **Rožnovský, Lukáš.** http://wiki.knihovna.cz/. *Přirozené vs. umělé jazyky.* [Online] 20. Únor 2012. [Citace: 3. Leden 2020.] http://wiki.knihovna.cz/index.php/P%C5%99irozen%C3%A9_vs._um%C4%9Bl%C3%A9_jazyky.
3. **Barták, Roman.** http://kifri.fri.uniza.sk/. *Formální jazyky a gramatiky.* [Online] 20.3.2012. Březen 2012. [Citace: 8.2.2020. Únor 2020.] https://ktiml.mff.cuni.cz/~bartak/automaty/lectures/lecture06.pdf.
4. **Doc. Ing. Bořivoj Melichar, DrSc.** *Jazyky a překlady.* Praha : ČVUT, 1999. 2236.
5. **Žoltá, Lucie.** http://lucie.zolta.cz/. *Zásobníkové automaty.* [Online] [Citace: 4. Leden 2020.] http://lucie.zolta.cz/index.php/zaklady-teoreticke-informatiky/14-bezkontextove-gramatiky/11-zasobnikove-automaty.
6. **Dr. Hashim Habiballa, PhD.** *Překladače.* Ostrava : Ostravská univerzita v Ostravě, 2005.
7. **Vavrečková, Šárka.** http://vavreckova.zam.slu.cz/. *Atributovaný překlad.* [Online] 5. Prosinec 2008. [Citace: 11. Únor 2020.] http://vavreckova.zam.slu.cz/obsahy/prekl/prezentace/prekl_11_atrib.pdf.
8. **Mička, Pavel.** algoritmy.net. *LL1-gramatika.* [Online] https://www.algoritmy.net/article/69/LL1-gramatika.
9. —. algoritmy.net. *Transformace-na-LL1.* [Online] https://www.algoritmy.net/article/71/Transformace-na-LL1.
10. **Janoušek, Jan.** https://adoc.tips/. *Adoc.* [Online] 2011. [Citace: 3. Březen 2020.] https://adoc.tips/programovaci-jazyky-a-pekladae-formalismy-pro-syntaxi-izeny-.html. BI-P JP.

Přílohy

Soubor: “Translator_bc“