Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Odporučenie jazykových vzorov pri návrhu jazyka

Diplomová práca

2022 **Bc. Katarína Šipošová**

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Odporučenie jazykových vzorov pri návrhu jazyka

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	9.2.1. Informatika
Školiace pracovisko:	Katedra počítačov a informatiky (KPI)
Školiteľ:	Ing. Sergej Chodarev, PhD.

Košice 2022 **Bc. Katarína Šipošová**

Abstrakt v SJ

Táto záverečná práca sa zaoberá skúmaním spôsobov, ako do procesu návrhu a vývoja formálnych jazykov zapojiť jazykové vzory. Jazykový vzor je spoločný znak konkrétnej syntaxe, ktorý sa opakovane vyskytuje v rôznych jazykoch. V tejto práci zisťujeme, či je možné generovať odporúčania použitia jazykových vzorov na základe výsledkov analýzy modelu jazyka. Analýza modelu je postavená na definovaných heuristikách, ktoré sa týkajú predovšetkým konceptov jazyka, ich štruktúry a typov ich vlastností.

Predstavujeme prototyp nástroja podpory jazykových vzorov Pattern. Pattern poskytuje interaktívnu analýzu modelu jazyka vyvíjaného v nástroji YAJCo a implementáciu nájdených vzorov do jazyka. Implementáciou nástroja Pattern ukazujeme, že je možné automatizovaným spôsobom na základe modelu jazyka navrhnúť jazykové vzory. Opísané sú aj obmedzenia implementovanej analýzy a potreba získavania doplňujúcich informácii od autora jazyka počas analýzy modelu.

Kľúčové slová v SJ

jazykové vzory, YAJCo, programovací jazyk, návrh jazyka

Abstrakt v AJ

This final thesis deals with the ways to involve language patterns in the process of design and development of formal languages. A language pattern is a common feature of a concrete syntax that occurs repeatedly in different languages. In the thesis we find out whether it is possible to generate recommendations for the use of language patterns based on the results of the analysis of the language model. The analysis of the model is based on defined heuristics, which mainly concern the concepts of language, their structure and types of their properties.

We present a prototype of the language patterns support tool called Pattern. Pattern provides an interactive analysis of the model of the language developed in YAJCo and the implementation of the found patterns into the language. By implementing the Pattern tool, we show that it is possible to suggest language patterns in an automated way based on a language model. The limitations of the implemented analysis and the need to obtain additional information from the author of the language during the analysis of the model are described as well.

Kľúčové slová v AJ

language patterns, YAJCo, programming language, language design

Bibliografická citácia

ŠIPOŠOVÁ, Katarína. *Odporučenie jazykových vzorov pri návrhu jazyka*. Košice: Technická univerzita v Košiciach, Fakulta elektrotechniky a informatiky, 2022. 78s. Vedúci práce: Ing. Sergej Chodarev, PhD.

64796

TECHNICKÁ UNIVERZITA V KOŠICIACH

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Katedra počítačov a informatiky

ZADANIE DIPLOMOVEJ PRÁCE

Študijný odbor: **Informatika**
Študijný program: **Informatika**

Názov práce:

Odporučenie jazykových vzorov pri návrhu jazyka

Recommendation of language patterns during language design

Študent: **Bc. Katarína Šipošová**
Školiteľ: **Ing. Sergej Chodarev, PhD.**
Školiace pracovisko: **Katedra počítačov a informatiky**
Konzultant práce:
Pracovisko konzultanta:

Pokyny na vypracovanie diplomovej práce:

1. Analyzovať problematiku jazykových vzorov a ich použitie v nástroji YAJCo.
2. Analyzovať implementáciu nástroja YAJCo.
3. Navrhnúť metódy automatického odporučenia jazykových vzorov pri návrhu jazyka.
4. Implementovať a experimentálne overiť navrhnuté metódy pri návrhu jazykov.
5. Vypracovať dokumentáciu podľa pokynov vedúceho práce.

Jazyk, v ktorom sa práca vypracuje: slovenský
Termín pre odovzdanie práce: 22.04.2022
Dátum zadania diplomovej práce: 29.10.2021

prof. Ing. Liberios Vokorokos, PhD.
dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som záverečnú prácu vypracoval(a) samostatne s použitím uvedenej odbornej literatúry.

Košice, 13.5.2022 .....

Vlastnoručný podpis

Poďakovanie

Ďakujem môjmu školiteľovi Ing. Sergejovi Chodarevovi, PhD. za všetky jeho rady, ochotnú pomoc a pripomienky počas tvorby tejto práce. Ďakujem aj mojej rodine za podporu počas celého štúdia.

Obsah

Zoznam tabuliek

Motivácia

Pri programovaní si programátor prirodzene uvedomuje podobnosť medzi programovacími jazykmi. Riešenie svojho problému vyjadruje pomocou rovnakých alebo podobných konceptov aj keď pracuje s rôznymi jazykmi. Prinajmenšom v rámci jazykov rovnakej paradigmy. Príkladom takýchto podobností môže byť to, že dokáže v jazyku vyjadriť podmienené vykonanie akcie, enumeráciu položiek, prípadne vyjadriť aritmetický výraz pomocou matematickej notácie. Autor v [1] rozoznáva tri základné koncepty, ktoré všetky programovacie jazyky zdieľajú a to existenciu základného príkazu jazyka, spôsob akým sa dokážu tieto príkazy spájať medzi sebou a spôsob akým program získava vstup pre svoju činnosť. Je teda zrejmé, že programovacie jazyky opakovane používajú nejaké vzory, ktorými poskytujú používateľovi jazyka spôsob, akým môžu vyjadriť riešenie problému.

Okrem toho, že existujú takéto vyjadrenia riešenia problému, ktoré poskytuje jazyk programátorovi, zdieľajú programovacie jazyky aj formu konkrétnej syntaxe, pomocou ktorej sa tieto koncepty vyjadrujú. Formu vyjadrenia programovacie jazyky zvyčajne prevzali z matematickej notácie alebo z anglického jazyka [2]. Príkladom by mohol byť blok kódu. V rámci zdrojového kódu je potrebné ho oddeliť od ostatných častí programu. Vo viacerých jazykoch sa využíva dvojica slov *begin* a *end*, ktoré označujú začiatok a koniec postupnosti príkazov patriacich do jedného bloku. Sú to kľúčové slová prevzaté z anglického jazyka. Ďalším príkladom opakujúcich sa vzorov zápisu by mohol byť zoznam položiek. V písanej forme prirodzeného jazyka sa používa vo vete postupnosť slov oddelených čiarkou. Podobne sa môže v programovacom jazyku zapisovať vymenovanie položiek poľa. Nie len koncept vymenovania položiek ale aj spôsob, akým sa zapíšu sa opakovane vyskytuje v konkrétnej syntaxi jazykov.

Identifikovanie opakujúcich sa prvkov v syntaxi jazyka je motiváciou pre ich zapojenie do procesu návrhu jazyka. Takýmto spôsobom by bolo možné využiť opakujúcu sa notáciu z jazykov pri vývoji nového jazyka. Existuje množstvo nástrojov podpory tvorby jazyka v rôznych fázach jeho návrhu [3]. Ak by počas

tvorby jazyka boli identifikované možnosti použitia vzorov konkrétnej syntaxe na základe modelu jazyka, bolo by možné autorovi jazyka ponúknuť spôsoby, ako ich v jazyku implementovať. Ak by sa tieto návrhy zautomatizovali, rozšíril by sa tak proces návrhu jazyka aj o nástroj podpory tvorby konrétnej syntaxe za pomoci jazykových vzorov.

Formulácia úlohy

Prvým cieľom práce je skúmať problemantiku jazykových vzorov. Opísať ich podstatu a možnosti uplatnenia v rámci tvorby jazykov.

Hlavným cieľom práce je implementovať analýzu modelu jazyka a zistiť, či je možné odporučiť na základe vykonanej analýzy návrhy jazykových vzorov automatizovaným spôsobom. Jedným zo spôsobov implementácie analýzy jazyka je zamerať sa na definovanie heuristík, pomocou ktorých je možné rozhodnúť o možnostiach návrhu niektorého zo vzorov. Heuristiky, na ktoré sa analýza zameriava sa týkajú zväčša typových vlastností konceptov jazyka a štruktúry ich abstraktnej syntaxe. Pri lexikálnych vzoroch musí analýza skúmať definované lexikálne jednotky.

Implementácia analýzy modelu bude prototypom nástroja podpory vzorov pri návrhu jazyka. Implementované metódy analýzy sa overia na vzorke jazykov vytvorených v YAJCo. V modeli jazykov budú identifikované vzory, ktoré sa následne manuálne odstránia. Následne budú jazyky analyzované implementovaným nástrojom. Snahou je overiť efektivitu odporúčaní vzorov vo vzťahu k odstráneným vzorom. Vyhodnotenie sa zameriava na presnosť a relevantnosť navrhovaných vzorov.

1 Jazykový vzor

Vzor alebo návrhový vzor je známy pojem v rámci softvérového inžinierstva. Existuje viacero definícií vzoru. Podľa [4] je vzor abstrakciou z konkrétnej opakujúcej sa formy. Je to riešenie opakujúceho sa problému preukázané ako správne, ktoré je nejakým spôsobom identifikované a zdokumentované, aby z neho mali ostatní osoh [5]. Vzory sú využívané v rámci objektovo-orientovaného programovania. V tom prípade ide o návrhové vzory. Návrhové vzory vyjadrujú opakujúce sa štruktúry programu a riešia problémy návrhu. Autor v [6] uvádza, že vzor využíva abstrakciu, pre riešenie konkrétnych problémov.

Vzory sa odlišujú od dátových štruktúr a algoritmov v tom, že to, čo vyjadrujú nie je možné jedenkrát naprogramovať a neskôr použiť ako funkciu alebo importovať ako triedu [7]. Popisujú štruktúru riešenia. Podľa [7] sa však vzor odlišuje od softvérových rámcov v tom, že neopisuje štruktúru celého systému a väčšinou je pre riešenie väčšieho systému použitých viacero príbuzných vzorov, ktoré dokážu riešiť problém zložitého systému.

Podobne aj jazykové vzory riešia nejaký problém pri návrhu jazyka. Ich využitie pri tvorbe jazyka rovnako ako pri návrhových vzoroch podporuje používanie osvedčených postupov. Samotný pojem návrhového vzoru jazyka je ale prezentovaný v inom význame u autorov článku [8] a iných autorov, ktorí sa zaoberajú vzormi návrhu jazykov ako napríklad [9], [10] alebo [7].

1.1 Definovanie jazykového vzoru

Jazyk je definovaný v troch hlavných krokoch, ktoré zahŕňajú abstraktnú syntax, konkrétnu syntax a sémantiku. Rozdiel medzi konkrétnou a abstraktnou syntaxou je možné odvodiť z chápania reálneho sveta, kde existujú koncepty bez ohľadu na ich konkrétnu reprezentáciu. Pri definovaní programovacieho jazyka je možné definovať koncepty bez ich konkrétnej reprezentácie [11] a až následne definovať ich konkrétnu syntax. Niektoré jazyky vo svojej definícií vôbec konkrétnu syntax neobsahujú [12]. Abstraktná syntax v sebe zahŕňa pojmy jazyka.

Konkrétna syntax vyjadruje notáciu týchto pojmov [11] a je alebo textová alebo grafická [13]. V rámci tejto práce sa zameriavame na textovú konkrétnu syntax jazykov.

Jazyky medzi sebou zdieľajú podobnosti [8]. V mnohých jazykoch sa nachádzajú rovnaké alebo podobné zápisy, ktoré zároveň majú rovnaký význam a reprezentujú rovnaký koncept. Ide o použitie rovnakého znaku alebo postupnosti znakov v rôznych jazykoch, s tým istým významom. Podobne ako sú opisované podobnosti v abstraktnej syntaxi, ktoré pramenia z prirodzeného jazyka, matematického zápisu alebo z iných už programátormi osvojených jazykov, aj textová syntax jazyka zdieľa podobu s inými jazykmi pre rovnaké dôvody.

Napriek tomu, že sa opakuje použitie symbolu alebo postupnosti symbolov na vyjadrenie nejakého konceptu, medzi opakovanie notácie patrí aj taký prípad, keď nie je notácia naviazaná na rovnaký koncept. Inak povedané, rovnaký zápis sa použije v rôznych jazykoch pre rôzne koncepty, ktoré môžu byť odlišné alebo len podobné. Avšak aj napriek tomu ide o použitie rovnakej konkrétnej syntaxe. Ako príklad možno uviesť slovník v Pythone a triedu v Jave. Su to rozdielne pojmy s rozdielnym významom, ale v oboch prípadoch sa v ich konkrétnej syntaxi používajú zložené zátvorky.

Podobnosti, ktoré medzi sebou jazyky zdieľajú by sa mohli označiť ako vzory návrhu jazykov. V [8] zavádzajú autori jazykový vzor ako pojem pre opakovanie zobrazenia medzi konkrétnou a abstraktnou syntaxou jazyka. Keďže jazyky medzi sebou zdieľajú pojmy, zdieľajú aj ich notáciu. Pokiaľ sa pri tvorbe jazyka autor rozhodne vyjadriť ten istý koncept podobnou alebo rovnakou notáciou, využíva tým jazykový vzor. Rovnako však aj v prípade, keď vyjadruje iný koncept rovnakou notáciou tak, ako bolo uvedené na predošlom príklade v Jave a Pythone. Vo všeobecnosti teda možno povedať, že jazykový vzor je opakujúca sa notácia naprieč rôznymi jazykmi.

Vzor využíva istú mieru abstrakcie pre riešenie konkrétnych problémov [6]. Keďže ide o abstraktnú úroveň nad konkrétnou podobou programu, nejde vždy o striktné dodržiavanie návrhového vzoru. Podobne aj pri jazykovom vzore, nejde vždy o použitie presného zápisu v nejakom význame. Môže ísť o zápis podobný alebo súvisiaci s iným zápisom. Ako príklad možno uviesť jeden z jazykových vzorov, ktorý je uvedený v [8]. Ide o návrhový vzor *Brackets* - vzor zátvoriek. V rámci tohto vzoru uvádzajú autori ako motiváciu pre použitie zátvoriek, rozlíšenie postupnosti elementov v zdrojovom kóde, ktoré spolu súvisia, od ostatných podobných elementov. Pričom musí byť jasné, kde sekvencia začína a kde končí. V rôznych jazykoch sa stretneme s ohraničením tried, funkcií alebo procedúr zát-

vorkami „{“ a „}“.

Podobne by sa mohli v zátvorkách uzavrieť položky zoznamu alebo poľa pri jeho definícií. V prípade zoznamu by bolo možné v rovnakom význame použiť zátvorky „[“ a „]“. V oboch prípadoch ide o použitie toho istého vzoru. Sú použité podobné ale nie rovnaké symboly, ktoré spolu súvisia v rovnakom význame.

Ako uvádzajú autori v [8], pri vzoroch ide v prvom rade o opakujúce sa štruktúry v rámci širokého množstva jazykov. V rámci jazykov jednej paradigmy je tiež možné sledovať isté podobnosti v zápise. Avšak pokiaľ ide o jazykové vzory, snahou je identifikovať všeobecný zápis. To sa môže týkať dátových formátov alebo programovacích jazykov [8]. Pritom ide o jazyky všeobecného použitia alebo doménovo špecifické jazyky.

V [9] je uvedené, že sa autori snažia spísať vzory pre vývoj doménovo špecifických jazykov pre pomoc autorom jazykov. Pričom vzormi majú na mysli odporúčania pre analýzu potreby tvorby a dizajn doménovo špecifických jazykov. Rozširujú prácu [7]. V [9] rozdeľujú vzory podľa fázy vývoja jazyka. Tými fázami sú rozhodovanie, analýza, návrh, implementácia a zavedenie. Pod rozhodnutím ako fázou sa myslí rozhodnutie, či je vhodné vytvárať nový jazyk.

V rámci vzorov, ktoré uvádzajú ide však skôr o odporúčania pre autorov, ktorí vyvíjajú nový jazyk. Nejde o vzory vo význame, v akom sú chápané autormi v [8]. Zatiaľ čo [8] sa snaží zmapovať vzory v konkrétnej syntaxi, teda opakujúce sa notácie v programovacích jazykoch, [9] a podobne aj [10] alebo [7] uvádzajú odporúčania pri tvorbe jazyka v zmysle rád pre tvorbu jazyka v rôznych fázach jeho vývoja. Bolo by možné povedať, že autori v [8] sa zameriavajú na fázu návrhu, konkrétne na návrh gramatiky a konkrétnej syntaxi jazyka. Ostatní spomínaní autori chápu vzor ako osvedčený princíp pri tvorbe jazyka skôr než syntaktickú konštrukciu, ktorá sa v jazykoch opakuje. V rámci tejto diplomovej práce sa zameriavame na textovú syntax jazyka a na vzory v konkrétnej syntaxi. Vzory v takom zmysle, ako sú chápané v [8]. Medzi odporúčaniami autorov [9], [10], [7] a medzi vzormi v [8] však možno nájsť prepojenia.

Ďalším rozdielom, ktorý si možno všimnúť v rámci týchto dvoch chápaní vzorov, teda vzor ako opakovane sa vyskytujúca forma zápisu alebo podoba konkrétnej textovej syntaxe a na druhej strane vzor ako návrhový vzor pri tvorbe jazyka, teda princíp alebo odporúčanie pre návrh jazyka, je zameranie sa na jazyk všeobecného použitia a doménovo špecifický jazyk. Spomínaní autori [9], [7], [10], ktorí sa zaoberajú otázkami kedy a ako navrhovať jazyk, sa zameriavajú na doménovo špecifické jazyky. Doménovo špecifické jazyky (DSL) sú oproti jazykom všeobecného použitia odlišné vo fázach návrhu a vývoja. Doménovo špecifické

jazyky majú pre svoju veľmi úzku využiteľnosť značné obmedzenia pre úsilie, ktoré je vhodné vynaložiť na ich návrh a implementáciu [7]. Z toho dôvodu sa autori zameriavajú na odporúčania, kedy je vhodné rozhodnúť sa pre vývoj nového jazyka [9] a ako vyvíjať jazyk čo najefektívnejšie využívaním iných existujúcich jazykov [10] [7].

V rámci využívania iných existujúcich jazykov možno spomenúť vzor *piggyback*, ktorý uvádza autor v [7] a spomína ho aj [9]. Ide o využitie vlastností, ktoré už v existujúcom jazyku sú implementované pre podporu týchto vlastností v doménovo špecifickom jazyku. Vzor *piggyback* môže byť použitý všade tam, kde DSL zdieľa podobné atribúty ako existujúci jazyk. Hostiteľský jazyk ponúka novému jazyku napríklad spracovanie výrazov, implementáciu procedúr a funkcií a podobne [7]. Na príklade tohto vzoru je možné vidieť, že jeho aplikovanie je v súlade s pravidlom, že úsilie vynaložené na vývoj jazyka musí byť adekvátne jeho použiteľnosti v rámci úzkej domény. Z toho dôvodu je snaha využiť už implementované vlastnosti. V súvislosti s konkrétnou syntaxou, autori píšu opäť z pohľadu čo najefektívnejšieho vývoja doménových jazykov. Napríklad [9] rozdeľuje vzory návrhu do niekoľkých kategórií. Medzi nimi je aj kategória *notácia*, ktorá sa týka návrhu konkrétnej syntaxe. V rámci tejto kategórie je uvedené, že je dôležité rozlíšiť, či je dostupná nová alebo existujúca notácia.

Na druhej strane autori v [8] sa nezameriavajú len na doménovo špecifické jazyky. Snažia sa nájsť všeobecné zápisy, ktoré možno aplikovať v jazykoch všeobecného použitia. Tak ako uvádzajú, inšpirovali sa skúsenosťami s programovacími jazykmi ako C, Java, Python alebo Haskell. Zároveň ale aj niektorými doménovo špecifickými jazykmi. Snahou je zozbierať a popísať vzory syntaxe jazykov. Obe chápania vzorov majú za cieľ byť doplňujúcim nástrojom pri vývoji jazyka.

Možno teda priblížiť, v akom význame sú popisované vzory v článku [8]. Medzi vzormi, ktoré sú zdokumentované je vzor *quoted string* [8]. Ide o vzor, ktorý rieši problém reprezentácie reťazcov symbolov v zdrojovom kóde. Je zrejmé, že takýto problém sa vyskytuje v mnohých jazykoch bez ohľadu na to, či sú doménovo špecifické alebo sú to jazyky všeobecného použitia. Riešením, ktoré sa v jazykoch často vyskytuje, je uzavrieť postupnosť znakov medzi oddeľovače. Najčastejšie používanými oddeľovačmi sú úvodzovky alebo apostrofy.

Ide teda o opakujúci sa problém, ktorý má zaužívané riešenie, ktoré sa vyskytuje vo väčšine jazykov. Riešený problém sa týka zobrazenia z abstraktnej do konkrétnej syntaxe. Ako sa autori vyjadrujú, jazykový vzor berú z pohľadu častého opakovane sa vyskytujúceho zobrazenie medzi abstraktnoou a konkrétnou syntaxou. Teda vzor je častý spôsob reprezentovania konceptu v jazyku. Koncept

reťazca sa vyskytuje vo väčšine jazykov. Reťazec je v niektorých jazykoch primitívnym typom. Je teda prirodzene súčasťou mnohých jazykov. Spôsob, akým je reprezentovaný pomocou syntaxe jazyka podobne jazyky medzi sebou zdieľajú. Preto je označený ako vzor.

1.2 Porovnanie a príklady vzorov

V rámci návrhových vzorov doménovo špecifických jazykov a vzorov konkrétnej syntaxe je možné nájsť isté prepojenia a porovnať vzory, ktoré spomínajú [8] a odporúčania zozbierané [7], [9] a [10]. Možno teda uviesť odporúčanie pre návrh jazyka a vzory konkrétnej textovej syntaxe, ktoré s nimi súvisia a implementujú tak tieto odporúčania v konkrétnej syntaxi jazyka. Ako bolo uvedené, program nie je určený len na spracovanie počítačom, ale jeho podoba by mala byť prispôsobená človeku [10]. Syntax jazyka by mala poskytovať prostriedky pre kolaboratívnu prácu ľudí [10]. Podľa [14] je jednou z úloh programovacieho jazyka podporovať dokumentovanie zdrojového kódu. Programovací jazyk by teda mal mať spôsob, akým sa dokáže zapísať postupnosť znakov, ktorá nemá mať vplyv na vykonávanie programu [8], teda sa netýka spracovania programu počítačom ale týka sa práce človeka zo zdrojovým kódom. Koncept jazyka, ktorý to umožňuje, je komentár a je zdieľaný takmer všetkými jazykmi. Potreba komentára je najvýraznejšia pri nízkoúrovňových jazykoch, ktoré sú málo prispôsobené človeku [14], pretože komentár môže vysvetľovať časť zdrojového kódu a použité riešenie [8]. Častokrát prax ukáže, že sa program viackrát číta ako píše. Preto je vhodné zameriavať sa na čitateľa zdrojového kódu nielen na jeho autora [10]. Podoba komentárov sa líši v konkrétnej syntaxi čo sa týka výberu symbolov. Avšak štruktúra a aj myšlienka použitia komentárov sa opakuje, preto ju [8] zaraďuje medzi vzory a [10] medzi odporúčania.

Rozšíreným a štandardným spôsobom zápisu komentárov je uzavretie komentára medzi postupnosť symbolov „/* .. */“ pre skupinové komentáre a postupnosť symbolov „//“ pre jedno riadkové komentáre [10]. Možno vybrať iné príklady komentárov medzi rozšírenými programovacími jazykmi ako Python alebo Haskell. V jazyku Haskell za komentár začína postupnosťou znakov „- -“ [15]. V Pythone začína komentár znakom „#“ [16]. Konkrétna voľba symbolu sa líši, ale koncept, ktorý reprezentuje je rovnaký. Konkrétna syntax je síce odlišná ale príbuzná.

Ďalšími vzormi, ktoré sú spomenuté v [8] sú *keyword* a *concept decorator*. Vzor *concept decorator* rieši problém rozlíšenia konštruktov jazyka, ktoré sa môžu vy-

skytnúť v rovnakom kontexte, človekom ale aj počítačom pri spracovaní programu [8]. Ako príklad v [8] sú uvedené dva koncepty a to podmienený výraz a cyklus s podmienkou na začiatku. Oba majú rovnakú štruktúru, čo sa týka abstraktnej syntaxi [8]. Avšak pre ich rozdielnu sémantiku je potrebné ich rozlíšiť. Vzor *concept decorator* rieši tento problém tým, že sa zavedie kľúčové slovo alebo oddeľovač, ktorý konštrukty od seba oddelí. Kľúčové slovo, ktoré stojí na začiatku konštruktu je najčastejším riešením tohto problému, pretože je to prvá postupnosť symbolov, ktorú človek pri čítaní spracuje a na základe nej dokáže hneď rozlíšiť, o aký koncept ide [8]. To je základ vzoru *keyword*, ktorý súvisí so vzorom dekorátora konceptu *concept decorator*. *Keyword* teda kľúčové slovo sa týka dekorátora, ktorý má podobu slova. Ide o slovo, ktoré je názvom konceptu. Používa sa, pokiaľ neexistuje ustálený symbol alebo operátor pre daný koncept [8]. Vo veľkej väčšine jazykov ide o anglické slová, ktoré súvisia s konceptom, prípadne skrátené tvary týchto slov. Príkladom môže byť slovo „function“ pre označenie funkcie napríklad v jazyku JavaScript alebo „fun“ pre ten istý koncept v jazyku Kotlin. Ako príklad pre kľúčové slová v rámci vzoru *concept decorator* je možné uviesť syntax jazyka Pascal cyklov *while* a *repeat-until*.

Zdrojový kód 1.1: Gramatika cyklov *while* a *repeat-until* jazyka Pascal[1]

```
<while statement> ::= while <expression> do <statement>
<repeat statement> ::= repeat <statement> {; <statement>}
                       until <expression>
```

Tieto dva vzory je možné spojiť s odporúčaním [10], ktoré hovorí o voľbe syntaxe, ktorá je odvodená nielen z používanej notácie v iných programovacích jazykoch ale aj zo spôsobu uvažovania človeka.

Ohľad na zaužívané formy zápisu by sa mal brať aj pri výbere kľúčových slov. Mali by byť všeobecne akceptované a nie príliš odlišné od kľúčových slov v iných jazykoch [10]. Notácia by podľa autora mala byť dostatočne opisná a samovysvetľujúca. Práve preto je vhodné zvoliť pri návrhu jazyka často používané symboly a výrazy z domény v prípade doménovo špecifického jazyka. Dôležité je aj zachovanie sémantiky symbolov. Použitie známeho symbolu v inom význame by viedlo k misinterpretovaniu viet jazyka a sťažilo by nie len používanie jazyka ale aj jeho naučenie.

Vzor *concept decorator* súvisí so vzorom *brackets*, ktorý je vzorom zátvoriek v zdrojovom kóde. Rieši problém, ako oddeliť od seba inštancie konceptov, ktoré spolu súvisia a označiť, kde sekvencia končí a kde začína [8]. Pár zátvoriek, kde

[1] `https://condor.depaul.edu/ichu/csc447/notes/wk2/pascal.html`

prvá zátvorka označuje začiatok a druhá koniec, je vhodný na toto použitie. Využívané sú všetky druhy zátvoriek „{}“, „()“, „[]“, „<>“. Okrem zátvoriek zároveň do tohto vzoru spadajú aj postupnosti viacerých symbolov [8]. Príkladom môže byť podmienený výraz v jazyku Bash.

Zdrojový kód 1.2: Tvar podmieneného výrazu v jazyku Bash

```
if TEST-COMMAND
then
  STATEMENTS1
else
  STATEMENTS2
fi
```

Podmienený výraz je ohraničený kľúčovými slovami *if* a *fi*. Podobne by mohol byť ako príklad už spomínaný viacriadkový komentár „/* ... */“, ktorý ohraničuje komentár do postupnosti symbolov.

V [10] sú uvedené príklady symbolu „+“ ako symbolu pre sčítanie a symbolov čiarky „,“ a bodkočiarky „;“ ako symbolov oddeľovača. Použitie napríklad symbolu „+“ v inom význame ako pridanie, sčítanie alebo spájanie (napríklad reťazcov) by bolo veľmi neštandardné. Dôležité je teda nie len vybrať vhodnú alebo opakujúcu sa notáciu ale použiť ju aj v zaužívanom význame. S oddeľovačom „;“ súvisí opäť vzor *concept decorator*. Napríklad bodkočiarka na konci príkazu je dekorátor, ktorý označuje koniec konceptu. Nazýva sa terminátor. Keďže zároveň oddeľuje inštancie konceptu, je to alternatíva vzoru *separator* [8]. *Separator* predstavuje vzor, ktorý rieši problém ako oddeliť inštancie konceptu od seba v rámci kolekcie a to zavedením špeciálneho symbolu, ktorý prvky oddelí [8]. Typickým príkladom sú položky poľa oddelené symbolom „,“. Jednoduchým príkladom môže byť syntax pre položky množiny v jazyku Pascal. Jednotlivé položky sú od seba oddelené čiarkou.

Zdrojový kód 1.3: Gramatika množiny v jazyku Pascal[2]

```
<set> ::= [ <element list> ]
<element list> ::= <element> {, <element> } | <empty>
```

Vzor *separator*, ktorý identifikovali autori v spomínanom článku súvisí s odporúčaním pre konkrétnu syntax, ktorá by mala poskytovať jasné odlíšenie elementov. Musia byť vhodne umiestnené, aby sa ten, kto číta zdrojový kód nemusel vracať späť teda aby bolo zrejmé, aký koncept má v kóde rozoznať [10].

[2] `https://condor.depaul.edu/ichu/csc447/notes/wk2/pascal.html`

V [9] sú uvedené dva základné spôsoby vytvorenia jazyka. Prvým je vytvorenie nového jazyka a druhým je využitie existujúceho. Veľakrát zdôrazneným odporúčaním je využiť existujúce zápisy, či už z domény jazyka alebo iných existujúcich jazykov. Je vhodné prevziať formálnu notáciu z domény, pokiaľ je to možné. Podobne ale aj neformálne pojmy domény, ktoré sú používateľovi z danej domény vlastné [17]. Aj [9] odporúča, že je najjednoduchším postupom pre tvorbu jazyka, tvoriť ho na základe už existujúceho. Tento princíp je vhodné aplikovať aj pri prevzatí notácie z iného programovacieho jazyka, pokiaľ je budúci používateľ programátor a teda už má predošlú skúsenosť s nejakým programovacím jazykom. V rámci tohto odporúčania ako príklad uvádza [9] aritmetické výrazy, ktoré sa vo väčšine jazykov vyskytujú v podobe akú majú v matematike pomocou infixného operátora.

Toto odporúčanie, ktoré uvádza Mernik ako príklad je v [8] rozvinuté do samostatného vzoru *infix operator*. Ako motivácia pre použitie tohto vzoru je uvedený problém reprezentácie vnorených zápisov aritmetických výrazov, keďže binárne infixné výrazy sú typické pre aritmetické výrazy v matematike. Vnorenie do seba je potrebné zabezpečiť bez zbytočných zátvoriek a je potrebné spájať viaceré takéto výrazy dokopy [8]. Riešenie tohto problému spočíva v prevzatí zápisu z matematiky. Samotná podoba konkrétnej syntaxe takého vzoru je opísaná ako dva podvýrazy, ktoré v strede obsahujú druh výrazu reprezentovaný symbolom, ktorý je prevzatý z matematiky.

Zdrojový kód 1.4: Gramatika výrazov s infixným operátorom prevzatá z [8]

```
Expr ->  Expr '+' Expr
| Expr '-' Expr
| Expr '*' Expr
...
| '(' Expr ')'
| Value
```

To, čo je už implementované v inom jazyku je využité pre tvorbu nového. Nejde však o spomínaný vzor *piggyback*. Vzor *piggyback* využíva hostiteľský jazyk ako prostriedok pre implementovanie nového jazyka [7]. V tomto prípade je znovu použitá len syntax, ktorá už rieši istý problém, nie samotný spôsob spracovania výrazov. A zároveň prevzatím tejto notácie z matematiky, ktorá je všeobecne známa, sa zachovajú odporúčania [9] a [10], ktoér hovoria o prevzatí zápisu pre ľahšie čítanie človekom.

Pri návrhu konkrétnej syntaxe sa niekedy zvolia alternatívne zápisy konceptov jazyka, ktoré nezvyšujú expresívnosť jazyka. Teda nepridávajú žiadnu novú

funkcionalitu, ktorú môže používateľ jazyka pomocou jazyka vyjadriť. Ide o takzvaný syntaktický cukor. Slúži len na zlepšenie efektivity a to tak, že umožní programátorovi efektívnejšie vyjadrenie a zároveň zvýši čitateľnosť tým, že prinesie prehľadnejší a jednoduchší zápis [10]. V [8] je uvedený vzor *alternative notation*, ktorý súvisí s týmto odporúčaním. Motiváciou pre vzniknutie vzoru je problém, ako vyjadriť jeden koncept viacerými spôsobmi. Riešením je definovať niekoľko odvodzovacích pravidiel [8].

Vzor *alternative notation* teda umožňuje zavedenie alternatív pre syntaktické konštrukcie pri zachovaní významu. V [10] je uvedený ako príklad *for* príkaz v Jave. Zápis tohto cyklu je rozšírený o jednoduchší alternatívny zápis, rozšírený *for* príkaz - *enhanced-for* príkaz.

Zdrojový kód 1.5: *For* príkaz

```
for (int i=0; i<numbers.length; i++) {
  ...
}
```

Zdrojový kód 1.6: *Enhanced-For* príkaz

```
for (Integer num : numbers) {
  ...
}
```

Syntax takzvaného *for-each* cyklu bola predstavená vo viacerých jazykoch. V jazyku Java existujú dve osobitné gramatické pravidlá pre *for* a *enhanced-for* príkazy.

Gramatika cyklov *for* a *enhanced-for* prevzatá z [18]

```
ForStatement:
stmt -> for ([ exp {, exp}]; [exp]; [ exp {, exp }] ) stmt

EnhancedForStatement:
stmt -> for (FormalParameter : exp) stmt
```

Vo všetkých jazykoch je potrebné oddeliť frázy jazyka od seba. Najprirodzenejší spôsob oddeľovania je známy z prirodzeného jazyka a jeho písomnej podoby. Ide o medzeru. Vo všeobecnosti v jazykoch nemusí ísť len o medzeru ale aj iné biele znaky, ktoré sa pri spracovaní zdrojového kódu nevyhodnocujú. Vzor konkrétnej syntaxe, ktorý to umožňuje je vzor *whitespace* [8]. Pokiaľ nie je dostatočné jednoznačné oddelenie len použitím bieleho znaku, môžu sa na riešenie takého problému využiť už spomínané vzory *separator* alebo *brackets* [8].

V rámci rozloženia programu, do ktorého spadajú aj biele znaky, je v [10] uvedené, že rozloženie zdrojového kódu by nemalo mať vplyv na jeho význam.

Hoci vo všeobecnosti medzery a tabulátory zvyšujú čitateľnosť [8], samotné biele znaky môžu byť ťažko rozlíšiteľné človekom. Ak takéto znaky majú význam, od programátora je vyžadovaná zvýšená miera pozornosti a sústredenia, a to nielen na čítanie ale písanie programu. Prípadne potrebuje využiť podporu vývojového prostredia pri písaní zdrojového kódu [10]. Napriek tomu sú jazyky, napríklad jazyk Python, ktoré využívajú odsadenie vo význame oddeľovača fráz. Napríklad namiesto zátvoriek. To je popísané jazykovým vzorom *significant indentation* [8], v rámci ktorého sa odsadenie stáva súčasťou syntaxe a nie je ignorované pri preklade. Zmena odsadenia sa použije namiesto zátvoriek pri vnáraní blokov programu. Zároveň však stále existuje potreba používať biele znaky bez zmeny významu programu, preto nie sú vo všetkých kontextoch rozoznávané vo význame oddeľovača [8]. Pravdou je, že odsadenie s významom nepoužíva veľa programovacích jazykov. Podľa autora príspevku[3], ktorý sa odvoláva na svoju databázu jazykov, je to menej ako 2%. Hodnovernosť týchto údajov je ťažko overiteľná, minimálne však na základe praktickej skúsenosti s programovacími jazykmi, možno súhlasiť, že výskyt odsadenia ako oddeľovača nie je častý. Na druhej strane však najznámejší jazyk citlivý na odsadenie Python je populárnym jazykom [19]. Možno teda povedať, že hoci použitie tohto vzoru nie je v súlade s odporúčaním [10], nemá použitie *significant indentation* [8] negatívny vplyv na rozšírenie jazyka. Prevážiť môžu iné kvality jazyka. Princípy pre vývoj jazyka majú skôr odporúčací charakter.

Ďalším veľmi často sa vyskytujúcim vzorom je *number*. Rieši otázku reprezentácie čísel v zdrojovom kóde. Môže ísť o čísla v desatinnom tvare, prirodzené čísla alebo záporné celé čísla [8]. Spôsob zápisu čísel je prevzatý z matematiky. Tento postup je v súlade s odporúčaním v [9], podľa ktorého si v ideálnom prípade doménovo špecifický jazyk osvojuje zaužívanú notáciu. Autor jazyka by mal potlačiť tendenciu vylepšiť ich. Pravdepodobne žiadna iná notácia ako matematická nie je v prípade čísel vhodná.

Jazykové vzory uvedené v [8] teda možno charakterizovať tak, že popisujú ako má jazyk vyzerať na základe jeho vlastností, teda na základe abstraktnej syntaxe. Iní autori [9], [10], [7] sa zameriavajú skôr na to, kedy a ako jazyk navrhnúť a aké vlastnosti má mať. V prípade konkrétnej syntaxe sa najviac princípov uvádza v [10], avšak tieto odporúčania nie sú transformované do podoby zdokumentovaných vzorov konkrétnej syntaxe ako je to u [8].

[3] `https://codelani.com/posts/which-programming-languages-use-indentation.html`

2 Prítomnosť vzorov v tvorbe jazyka

Programovanie je bežne vnímané ako proces formulovania riešenia problému takým spôsobom, aby bolo vykonateľné na počítači [20]. Programovací jazyk však neslúži len na spracovanie počítačom. Program je čítaný samotným autorom programu a aj inými ľuďmi, opakovane veľakrát predtým alebo potom ako je spracovaný počítačom [14]. Preto nie je dôležité len to, ako sa dá program spracovať, ale aj to, ako ho vníma človek. Rozhraním medzi programom a človekom je syntax programovacieho jazyka. Každé dizajnové rozhodnutie pri tvorbe syntaxe ovplyvňuje myslenie programátora pri písaní programu [21]. Preto na syntaxi jazyka záleží a jej tvorba by mala prebiehať s ohľadom na notáciu prijateľnú pre človeka.

Niektoré programovacie jazyky prinášajú lepší používateľský zážitok ako iné. Programátor, tak ako hociktorý iný používateľ ľubovoľného systému, si vyberie jazyk podľa kritérií, ktoré súvisia s použiteľnosťou, efektívnosťou, efektivitou alebo s predpokladateľnou spokojnosťou s výsledným programom [22]. Snaha navrhnúť dobrý jazyk je ale zložitá, pretože ľudia majú rôzne vnímanie toho, ako by mal dobrý jazyk vyzerať. To sa týka nielen toho, kto jazyk používa, ale aj vývojára jazyka [10], ktorý môže mať iné predstavy a očakávania ako používateľ jazyka.

Syntax dobre navrhnutého jazyka by sa mala ľahko chápať a mala by efektívne reprezentovať jeho sémantiku [23]. Dobrá vlastnosť jazyka môže byť sabotovaná voľbou nevhodného zápisu [24]. Pokiaľ je pri tvorbe nového programovacieho jazyka snaha navrhnúť pre človeka pohodlnú a prijateľnú syntax, využitie existujúcej notácie iných jazykov môže byť výhodné. Opakovanie syntaktickej notácie v rôznych jazykoch sa potom stáva jazykovým vzorom. Vzory sa môžu stať súčasťou vývoja programovacích jazykov v systematickej a automatizovanej podobe.

2.1 Vzory ako princíp znovupoužitia pri tvorbe jazykov

Pri tvorbe jazyka možno využiť prístup, ktorý zakladá návrh nového jazyka na identifikovaní konceptov jazyka [25]. Koncept je to, čo jazyk používateľovi jazyka ponúka. Je zhrnutím nie len toho, čo jazyk opisuje ale aj na čo sa jazyk bude používať [10]. Koncepty sú formalizované abstraktnou syntaxou [25]. Podľa [12] je pri dobrom návrhu jazyka práve abstraktná syntax centrálnym bodom. Abstraktná syntax reprezentuje koncepty jazyka nezávisle od konkrétnej reprezentácie, ktorá je prijateľná čitateľná pre človeka [26] a je vyjadrená pomocou tried, ktoré predstavujú elementy jazyka. Na základe toho sú vyjadrené vzťahy medzi pojmami jazyka [27].

Z jednej abstraktnej syntaxe je možné vygenerovať viac ako jednu konkrétnu syntax [13]. Ak sa začne jazyk definovať od konkrétnej syntaxe smerom k abstraktnej alebo naopak, vždy existuje zobrazenie z jedného do druhého [12]. Pri tvorbe jazyka smerom od abstraktnej syntaxe existujú zobrazenia, ktoré konceptu jazyka priraďujú jeho notáciu [28]. Táto notácia môže byť vyjadrená napríklad anotáciami alebo iným doménovo-špecifickým jazykom [27]. V týchto zobrazeniach je možné využiť jazykové vzory. Keďže viacero jazykov zdieľa medzi sebou tie isté alebo podobné koncepty, existuje aj podobnosť zobrazení týchto konceptov do konkrétnej notácie. Autor v [1] rozoznáva koncepty, ktoré podľa neho všetky programovacie jazyky zdieľajú. Ide o základnú vetu jazyka - príkaz, spôsob, akým sa príkazy dokážu spájať a akým získavajú dáta na svoje vykonanie. Nie všetky jazyky sú však založené na príkazoch. V iných paradigmách môže ísť napríklad o výraz alebo funkciu namiesto príkazu.

Množstvo programovacích jazykov odvíja svoju podobu od konštrukcie počítača, teda napríklad existencia inštrukcií procesora, ich radenie za sebou, existencia pamäťových adries, registrov a podobne. Z toho vyplýva aj podobnosť medzi notáciou týchto jazykov. Podobne je to aj u jazykov iných paradigiem. Napríklad funkcionálne jazyky obsahujú nejakú podobu funkcie a ich skladanie.

Je to rozhodnutie autora jazyka, akú konkrétnu syntax priradí abstraktnému konceptu navrhovaného jazyka. Dobrým rozhodnutím pri návrhu jazyka je však vytvárať jazyk na základe už existujúceho jazyka namiesto snahy vytvárať ho od začiatku bez využitia známej notácie. Ak je snahou opätovne použiť notáciu z jedného jazyka v inom jazyku, je možné pozrieť sa na ich abstraktnú syntax a rozpoznať v nich opakujúci sa jazykový vzor, ktorý následne možno, podobne ako je to pri návrhových vzoroch pri programovaní aplikácie, využiť na implementá-

ciu vlastného jazyka [10]. Ak existujú vzory v abstraktnej syntaxi a aj zobrazenia medzi konceptmi a ich notáciou, je možné identifikovať aj vzory v konkrétnej syntaxe. Takéto vzory z iných jazykov je potom možné preniesť do vyvíjanej syntaxe.

Princíp znovu použitia je bežne prítomný vo vývoji softvéru. Aplikácie zdieľajú svoju štruktúru a spôsob, akým sú komponované na celky. Ak je problém vhodne riešený v rámci nejakého celku, celok je možné znovu použiť. Podobne možno uvažovať pri tvorbe programovacích jazykov [29]. Brať definíciu existujúceho jazyka ako východisko pri tvorbe nového jazyka, môže byť lepšie ako vytvárať jazyk od začiatku. Z toho opätovného použitia môže ťažiť nie len konkrétna ale aj abstraktná syntax. Nový jazyk si zachová vzhľad a dojem z pôvodného jazyka preto používateľ pri vyjadrovaní sa v ňom ľahko môže identifikovať známe formy zápisu [10].

Pokiaľ sú vyvíjané jazyky komplexné, ich syntax a gramatika sa pri vývoji často dopĺňa a mení, čo súvisí aj s narastajúcimi nárokmi na zachytenie doménových konceptov v jazyku v prípade doménovo špecifických jazykov. Koncepty sa tak môžu stať cieľom opätovného použitia [30]. V prípade jazykových vzorov ide o opätovné použitie syntaxe, ktorá vyjadruje ten istý koncept naprieč rôznymi jazykmi. Základom je používanú notáciu prevziať a vložiť ju do nového jazyka.

Prístup využívania komponentov pri návrhu jazyka je prirovnaný autormi v [31] k tvorbe komponentovej aplikácie. Pri špecifikácií jazyka v textovej podobe štýlom postupnej tvorby od začiatku do konca vnímame definíciu jazyka jeden celok [32]. V [31] sa autor pozerá na jazyk ako na kompozíciu modulov, ktoré predstavujú už definované časti jazyka. Tie je možné pri návrhu skladať a rozširovať. Hovorí však o moduloch jazyka pri jeho tvorbe, nie o znovupoužití modulov jedného jazyka pri tvorbe druhého. Pri využití jazykových vzorov v návrhu jazyka by bolo snahou znovu použiť syntax už existujúceho jazyka a aplikovať ju na abstraktnú syntax vyvíjaného jazyka.

Využitie už definovanej konkrétnej syntaxe je prítomné pri vývine jazykov, hoci nie v metodologickej a štruktúrovanej podobe. Podľa [10] sú webové stránky generátorov parserov dobrým počiatočným zdrojom definícií iných jazykov. Okrem napríklad generátora syntaktických analyzátorov Antlr[1], ktoré uvádza, dokumentácia a návod na webovej stránke nástroja Tree-sitter[2] rovnako obsahuje konkrétne príklady definície jazykových pravidiel. Takéto zdroje však nie sú systematickou súčasťou vývoja. Aj v prípade všeobecne rozšírených jazykov platí, že ich syntax pripomína iný jazyk. Jazyk Java alebo napríklad TypeScript spomedzi

[1] `https://www.antlr.org/`

[2] `https://tree-sitter.github.io/tree-sitter/creating-parsers`

populárnych jazykov sú založené, čo sa týka ich syntaxe, na jazyku C.

V [33] autor hovorí o jazykoch, ktoré zdedili niektoré zlé vlastnosti vyplývajúce z voľby notácie jazyka C. Práve na tomto jazyku bolo založených mnoho iných jazykov. Podľa autora sa z tohto jazyka prenieslo niekoľko zlých javov.

Opätovné použitie existujúcej notácie patrí aj medzi uvedené odporúčania pri tvorbe doménovo-špecifického jazyka v [10]. Hovorí, že predstavenie novej syntaxe pre konkrétnu doménu pri vzniku nového doménovo špecifického jazyka nemusí byť užitočné, keďže doménoví experti už môžu byť zoznámení s nejakou formou zápisu. Softvéroví vývojári a návrhári jazykov majú množstvo skúseností z učenia sa nových jazykov. Doménoví experti však nie sú často ochotní prispôsobiť sa novému zápisu, hlavne pokiaľ sú zvyknutí dlhodobo používať jazyk, ktorý si už osvojili. Ak neexistuje pre doménu, ktorú daný jazyk rieši už existujúca notácia, je vhodné podľa autora využiť podobu zápisu iného všeobecne známeho jazyka.

V prípade doménovo špecifického jazyka, náklady na jeho vývoj a zavedenie do používania musia priniesť taký úžitok, aby tieto náklady vykompenzovali. Použitie jazykov, ktoré nie sú veľmi známe ani zdokumentované môže byť rizikom [9]. Ide hlavne o prostredie softvérového priemyslu, kedy je zavedenie nového jazyka doslova investíciou, ktorá musí byť výhodná. Keďže je oveľa menej rizikové použiť existujúci jazyk ako zavádzať nový, možno na základe toho predpokladať, že v prípade, že je rozhodnutie predsa len vytvoriť nový doménovo špecifický jazyk, je menej kritické pri jeho návrhu prevziať existujúcu syntax z iných jazykov. Samozrejme, pri doménovo špecifických jazykoch, ktoré využívajú doménoví špecialisti, ktorí by za iných okolností netvorili programy a inak ani počítačové jazyky nevyužívali, je dobré pri tvorbe využiť jazyk z ich domény. Naopak, pokiaľ je konečný používateľ jazyka programátor, je vhodné inšpirovať sa existujúcimi rozšírenými programovacími jazykmi, s ktorými sa s veľkou pravdepodobnosťou už vo svojej praxi stretol.

Ďalším príkladom znovu použitia zápisu je využitie notácie prevzatej z matematiky v programovacích jazykoch pre aritmetické operácie ako je napríklad sčítanie [2]. To už bolo spomínané v súvislosti so vzorom *infix operator*. Koncept aritmetického výrazu sa nachádza vo veľkej množine jazykov. Len zriedka programovací jazyk uvádza inú notáciu ako tú, ktorá je prevzatá z matematiky. I keď matematika samotná nie je programovacím jazykom, jej notáciu programovacie jazyky využívajú.

Zápis výrazu sčítania je potom opätovne využitý v rôznych programovacích jazykoch. Podobne tiež programovacie jazyky prevzali kľúčové slová a notáciu

z anglického jazyka [2]. Ako príklad možno uviesť syntax podmieneného výrazu. V tomto prípade sa opätovne používajú slová *if* a *else* ako dekorátory konceptov výrazu v podmienke a vetve podmieneného výrazu. Ďalším príkladom je slovo „import" [10], používané pri odkazovaní sa na iný menný priestor. Podobné využitia prirodzeného jazyka môžu uľahčiť učenie sa nového programovacieho jazyka. Ak sa výraz, ktorého význam človek pozná na základe prirodzeného jazyka využíva opakovane v rôznych jazykoch, môže to uľahčiť osvojenie si syntaxe jazyka.

Príklad podmieneného výrazu ukazuje aj existenciu vzoru v abstraktnej syntaxe medzi jazykmi. Jazyky zdieľajú to, že v ich modeli sa vyskytuje koncept pre podmienený výraz, koncept pre podmienku. Výraz vo vetve podmienky je samostatným konceptom, ktorý je len využitý v podmienenom výraze. Pri tomto vzore zdieľajú programovacie jazyky aj zobrazenie do konkrétnej syntaxe. Obsahujú notáciu, ktorá využíva anglické slová *if* a *else* alebo ich variácie ako *elseif* a *fi*, prípadne v kombinácii so zátvorkami. V definícií väčšiny jazykov všeobecného použitia nájdeme pravdepodobne koncept s názvom podobným kombinácii týchto slov *if* a *else*. Napríklad koncept s názvom *IfElse*. To vyjadruje fakt, že vzor tejto notácie je znovu použitý vo viacerých jazykoch natoľko, že je použitý aj na označenie abstraktného konceptu, ktorý sa tiež v jazykoch opakuje.

Ak teda existuje notácia pre vyjadrenie konceptu jazyka, ktorý je prítomný v rôznych používaných jazykoch, je vhodné ju využiť pri implementovaní nového jazyka. Princíp znovupoužitia sa takto môže dostať aj do procesu návrhu programovacích jazykov. Keďže je programovací jazyk určený, okrem spracovania počítačom, pre chápanie a čítanie človekom, je užitočné využiť známu, prijatú notáciu. Využitie vzoru môže zvýšiť čitateľnosť a pochopiteľnosť jazyka.

2.2 Prepojenie konkrétnej a abstraktnej syntaxe

Prepojenie medzi textovou syntaxou a konceptmi jazyka môže byť v mnohých prípadoch pre človeka intuitívne. Na základe skúseností s inými jazykmi alebo odvodením je možné rozpoznať koncepty jazyka zo zápisu zdrojového kódu zapísaného v tomto jazyku. Autor v [34] rozoberá spôsoby získavania abstraktnej syntaxe na základe uvedenej konkrétnej syntaxe. Jeho snahou je vylepšiť spôsob, ako odvodiť pojmy jazyka z notácie v prípade, keď konkrétna syntax už existuje ale neexistuje reprezentácia abstraktnej syntaxe jazyka.

Na nasledujúcom príklade, ktorý uvádza [34], sa ukazuje, ako je možné dedukovať pojmy jazyka zo zápisu časti zdrojového kódu. Pritom ani nemusí byť

známe, v akom jazkyu je program zapísaný.

Zdrojový kód 2.1: Príklad prevzatý z [34]

```
begin
  declare x,y: integer;
  x := read();
  y := f(x,time-of-day());
  return g(x,y);
end
```

Z častí zápisu **begin** a **end** možno dedukovať, že v jazyku existuje koncept, ktorý možno nazvať blokom. Je to teda symbolmi **begin** a **end** ohraničená pomenovaná konštrukcia, ktorá je zložená z menších častí. Ďalej, že existujú deklarácie, ktoré je potrebné uviesť pred postupnosťou príkazov v bloku [34]. Takéto odvodenie je pravdepodobne ovplyvnené praktickou skúsenosťou z iných existujúcich jazykov. Ukazuje sa takto, že existuje intuitívne prepojenie založené na predošlej skúsenosti s programovacími jazykmi. V tomto prípade ide o prepojenie slov **begin** a **end** s blokom príkazov. To, že sa práve takéto prepojenie abstraktnej a konkrétnej syntaxe, teda konceptu bloku programu a kľúčovými slovami **begin** a **end**, opakuje v rôznych jazykoch možno nazvať vzorom. Ide nielen o opakovanie vzoru abstraktnej syntaxe, teda existencia bloku, ale aj o opakovanie spôsobom akým sa blok v zdrojovom kóde vyjadrí.

V [10] sú uvedené tri princípy prepojenia konkrétnej a abstraktnej syntaxe. Jedným z princípov je, že elementy, ktoré sa odlišujú v konkrétnej syntaxe by mali mať odlišnú abstraktnú syntax. To vyjadruje význam zobrazenia medzi abstraktnou a konkrétnou syntaxou. Ďalším odporúčaním je nenaväzovať konkrétnu syntax na kontext. Ako príklad je uvedené použitie symbolu „=“ vo význame priradenia alebo ako znak porovnania podľa kontextu, kde sa nachádza. Takéto princípy sú však len odporúčaniami a nie sú priamo zapojené do vývoja jazyka. Autor jazyka musí sám zvládnuť prepojenie medzi abstraktnou a konkrétnou syntaxou.

Pri prístupoch k tvorbe jazyka založených na konkrétnej syntaxi musí autor už v prvých fázach návrhu myslieť na konkrétnu syntax. Tento prístup je pri návrhu jazykov najrozšírenejší [35]. Pri tomto tradičnom prístupe je pri vývoji jazyka spojený návrh jeho syntaxe spolu so spôsobom akým sa spracuje. Špecifikácia jazyka založená na modeli sa snaží tento problém riešiť tak, že oddelí definíciu konceptov jazyka od spôsobu, akým sa jazyk spracuje.

K jednej abstraktnej syntaxi však môže prislúchať viac zápisov [8]. Pri návrhu založenom na modeli môže definovať návrhár jazyka viacero modelov konkrétnej

syntaxe k jednému modelu abstraktnej syntaxe. Jazyky môžu mať a často majú viacero konkrétnych syntaxí. Môžu mať napríklad textovú a vizuálnu syntax [27].

Pri postupe návrhu jazyka od konkrétnej syntaxe je podľa použitého generátora syntaktického analyzátora potrebné pre autora napísať alebo vytvoriť špecifikáciu jazyka založenú na gramatike použitím špecifickej syntaxe vlastnej danému procesoru. V prípade, že sa jazyk zmení, úpravu musí vývojár jazyka vykonať manuálne v celej reťazi, ktorá tvorí špecifikáciu jazyka. Takéto úpravy sú časovo náročné, zdĺhavé a náchylné na chyby. Môže sa pri nich zaviesť chyba, ktorú je následne potrebné opraviť. Častokrát je pre rôzne potreby, ako je implementácia podpory jazyka v rámci IDE alebo vytvorenie debuggera potrebné uchovávať viacero foriem definície jazyka v rôznej syntaxi [36].

Návrh jazyka založený na modeli má za úlohu umožniť definovanie viacerých konkrétnych syntaxí pre ten istý model jazyka [36]. Tento prístup pre vývoj jazyka využíva aj generátor prekladačov YAJCo, ktorý je použitý v rámci tejto práce. Vývoj jazyka pri práci s týmto nástrojom definuje jazyk od abstraktnej syntaxi, pričom notáciu konceptov definuje priamo počas jej návrhu. Koncepty jazyka sú definované ako triedy a vzťahy medzi nimi. Konkrétna syntax je definovaná pomocou anotácií v zdrojovom kóde [25]. Problém, ktorý je spomenutý v [36], ktorý sa týka potreby zápisu gramatiky jazyka v syntaxi použitého nástroja pre generovanie syntaktického analyzátora, rieši YAJCo tak, že generuje na základe definovaného modelu špecifikáciu pre použitý generátor [8].

Na veľmi podobných princípoch a technikách je postavený nástroj ModelCC. ModelCC je generátor parserov založený na modeli jazyka, ktorý umožňuje definovať abstraktnú syntax jazyka pomocou tried. Umožňuje definovať zobrazenie z abstraktnej syntaxe do konkrétnej syntaxe pomocou anotácií v modeli jazyka alebo pomocou doménovo špecifického jazyka [27]. Abstraktný model jazyka je tvorený základnými elementmi jazyka, ktoré možno prirovnať k neterminálnym symobolom pri tvorbe jazyka riadenom konkrétnou syntaxou. Pojmy jazyka sú spájané do väčších celkov, čo je príbuzné spájaniu a opakovaniu neterminálnych symbolov pri špecifikácií jazyka založenej na konkrétnej syntaxi.

Model založený na vzoroch je podľa [27] často preceňovaný a autori zabúdajú na potrebu konkrétnej syntaxe pri návrhu jazyka. Zároveň ale však uvádza, že pri dobrom návrhu jazyka je abstraktná syntax základným prvkom. Podľa neho by však mala byť hlavným bodom, odkiaľ sa definuje konkrétna syntax a sémantika, ktoré potrebujú formálnu špecifikáciu. Hovorí, že sa často argumentuje, že pri dnešných možnostiach vývoja, ktoré poskytujú jazykové prostredia, kde je možné graficky navrhnúť jazyk už nie je potrebné formálne definovať gramatiku tradič-

ným spôsobom pomocou odvodzovacích pravidiel pre generátor prekladača.

V [27] je predostretý spôsob návrhu, ktorý využíva zároveň BNF špecifikáciu a metamodel jazyka. Ako uvádza, podobne je to implementované v nástroji Xtext. Konkrétna syntax je čiastočne opísaná v metamodeli, ktorý nazýva *parse model*. Z neho je vygenerovaná BNF špecifikácia jazyka. Podobne funguje aj generátor parserov YAJCo, ktorý sústreďuje návrh jazyka okolo abstraktnej syntaxe, ale definuje súčasne aj konkrétnu syntax pomocou anotácií, ktoré reprezentujú notáciu elementov jazyka.

Vývoj jazykov pri ktorom je oddelená definícia modelu jazyka a konkrétnej syntaxe je podporovaný aj nástrojmi na uľahčenie vývoja doménovo špecifických jazykov ako je napríklad jazykové prostredie *Eclipse Graphical Modeling Framework GMF*. Takéto rámce oddeľujú abstraktnú a konkrétnu syntax a poskytujú spôsob, ako ich prepojiť pomocou zobrazení. Tieto zobrazenia sa nazývajú modely zobrazenia. Modely sa môžu rozdeliť do dvoch kategórií. Prvou je interné prepojenie, kde je konkrétna syntax priamo prepojená na pojmy z abstraktnej syntaxe. Druhou kategóriou je externé prepojenie, kde sú konkrétna syntax a pojmy jazyka prepojené cez externý model zobrazenia [37].

V rámci tejto práce je využívaný spomínaný generátor YAJCo, ktorý pre definovanie modelu jazyka využíva Java triedy anotované metadátami pojmov jazyka. Pomocou týchto anotácií je možné definovať konkrétnu syntax. Takto je teda zabezpečené zobrazenie medzi abstraktnou a konkrétnou syntaxou. Prepojenie medzi modelom jazyka a konkrétnou syntaxou je dôležité z pohľadu tejto práce z toho dôvodu, že jazykové vzory sa týkajú konkrétnej syntaxe. Návrhy na ich implementovanie počas tvorby jazyka sa však majú generovať vo fáze tvorby konkrétnej syntaxe, o ktorú je doplnený model jazyka pomocou anotácií. Spôsob, akým by to malo prebiehať je načrtnutý v [8]. Ponúkané návrhy vzorov musia teda vychádzať z modelu jazyka. Prípadne byť doplnené informáciami od autora jazyka počas jeho návrhu .

2.3 Podpora vzorov pri tvorbe jazyka

Vývoj nového jazyka zahŕňa rôzne úlohy, od definície syntaxe jazyka, generovanie syntaktického analyzátora a vytvorenie prostredia pre vývoj v danom jazyku. Jazykové inžinierstvo je súčasťou všetkých týchto fáz [38]. Vývoj doménovo špecifického jazyka je náročný, pretože je potrebné mať nielen zručnosť pre tvorbu jazykov ale aj znalosti z domény jazyka. Návrh jazyka môže byť podporovaný nástrojmi alebo sadami nástrojov [9]. Jazykové prostredia podporujú nielen vývoj

jazyka ale aj tvorbu nástrojov a prostredí pre prácu v tomto jazyku [38].

V [9] je uvedená tabuľka systémov a nástrojov, ktoré slúžia na tvorbu jazykov spolu s ich autormi, miestom a rokom ich vytvorenia. V tabuľke 2.1 sú vybrané niektoré z nich spolu so stručným popisom ich funkcie. Keďže existujú rôzne prístupy k tvorbe jazykov, nástroje sa líšia tým, ako podporujú tvorbu jazyka.

Autor v [9] uvádza informácie o tom, aké fázy vývoja doménovo špecifického jazyka sú aktuálne v čase písania jeho článku podporované. Vývoj jazyka delí do fáz. Prvou je rozhodnutie vytvoriť nový doménovo špecifický jazyk. Ďalším je analýza, čo by mal budúci jazyk obsahovať. Potom jeho samotný návrh, teda jeho abstraktná, konkrétna syntax a sémantika a nakoniec samotná implementácia jazyka [9]. Pre všetky štyri fázy uvádza, aká je ich podpora v rámci nástrojov, ktoré majú pomáhať tvorbe jazyka. Žiadnu podporu nemá rozhodovanie sa pre vyvinutie nového domového špecifického jazyka. Pre fázu analýzy uvádza, že existujúce nástroje nie sú zatiaľ celkom integrované do procesu tvorby. Návrh jazyka má slabú podporu. Najsilnejšiu podporu v nástrojoch má implementácia [9]. To je pochopiteľné vzhľadom na veľké množstvo napríklad generátorov prekladačov a vývojových prostredí. S týmto stavom môže súvisieť to, akú podobu má väčšinou definícia jazyka. Ako uvádzajú autori v [8], väčšinou návrh jazyka začína definíciou konkrétnej syntaxe v textovej podobe. V definícii syntaxe je najrozšírenejšou a najakceptovateľnejšou formou formálnej definície jazyka BNF [35] alebo jej rozšírená forma EBNF [8].

Nástroj, ktorý by podporoval vzory v návrhu jazyka by spadal do fázy návrhu jazyka. Všetky nástroje, ktoré sú spomínané a ktoré pomáhajú pri vývoji jazykov potrebujú vstup, ktorý opisuje nejaký aspekt jazyka [9]. Pokiaľ by bola snaha zamerať sa na nástroj, ktorý by dokázal poskytnúť podporu jazykových vzorov pri návrhu jazyka, bolo by potrebné získať ako vstup syntax jazyka ešte pred tým, než sa navrhne samotná jej konkrétna podoba. V takom prípade tradičný postup, ktorý začína jej definíciou nebude stačiť.

YAJCo generátor však zavádza prístup, ktorý definíciu začína abstraktnou syntaxou [25]. Ako je uvedené v [8], pri vzoroch je dôležité zobrazenie z abstraktnej do konkrétnej syntaxe. Ak by teda nástroj na podporu vývoja jazyka mal k dispozícií model jazyka obsahujúci jeho koncepty, mohol by na základe nich dokázať navrhnúť reprezentáciu kontextov implementovaním niektorého zo spomínaných vzorov. Autori [8] teda navrhujú, že by bolo možné zapojiť vzory do tvorby jazyka automatizovaným alebo poloautomatizovaným interaktívnym spôsobom pomocou nástroja.

Aktuálne sú pravdepodobne najviac nepriamo podporované jazykové vzory

whitespace a *comment*. Ako príklad možno uviesť spôsob, akým je možné priamo pri definovaní lexikálneho analyzátora špecifikovať biele znaky, ktoré sa nemajú spracovávať. Napríklad v rámci nástroja Antlr je k dispozícií príkaz *skip* [39], pomocou ktorého sa zadajú analyzátoru tieto biele znaky.

Zdrojový kód 2.2: Príkaz *skip* v Antlr [39]

```
WS: [\s\t\r]+ -> skip
```

Tento príkaz možno považovať za podporu vzoru *whitespace*, keďže tým, že je k dispozícií nie len umožňuje jeho implementáciu v jazyku ale aj indikuje, že je v rámci jazyka potrebné riešiť tento problém. Tento príkaz možno použiť aj pre implementovanie komentárov v zdrojovom kóde, ktoré sa nemajú spracovávať pri syntaktickej analýze a teda sa preskakujú.

Pri používaní sady nástrojov *Grammar-kit* pre definovanie gramatiky a syntaktického analyzátora pre vytvorenie podpory vlastného jazyka vo vývojovom prostredí IntelliJ IDEA, sa v rámci definovania triedy syntaktického analyzátora vyžaduje implementácia metód, ktoré ošetrujú komentáre a biele znaky[3]. Takýmto spôsobom tiež nástroj podporuje zavedie komentárov a preskakovanie bielych znakov oddeľovačov fráz pre lepšiu čitateľnosť v jazyku. I keď v rámci vývoja doplnku pre jazyk je už zvyčajne jazyk definovaný, a teda nemožno hovoriť o vývoji doplnku IDE ako o úplnej súčasti návrhu jazyka.

Väčšina generátorov syntaktických analyzátorov ponúka aj možnosť definovať priority a asociatívnosť operácií. Týmto spôsobom podporujú vzor *infix operator* [8].

[3] `https://plugins.jetbrains.com/docs/intellij/lexer-and-parser-definition.html`

Názov	Stručný opis
ASF+SDF	Interaktívna podpora formálnych definícií jazykov vo formalizme ASF+SDF. Generovanie interaktívnych prostredí pre jazyk [40].
Gem-Mex	Vytváranie grafických špecifikácií. Generovanie typových kontrolérov, interpreterov a debuggerov zo špecifikácií [41].
JTS	Sada nástrojov pre rozšírenie industriálnych programovacích jazykov o doménovo špecifické konštrukty pre vytvorenie DSL na hostiteľskom jazyku [42].
Khepera	Nástroj na vývoj DSJ princípom *source-to-source tranformation* [43].
LISA	Generátor prekladača a interpreta z formálnej definície jazyka. Sémantika je opísana atributovou gramatikou [35].
SmartTools	Generátor vývojových prostredí [44].
smgn	Nástroj na tvorbu prekladačov DSJ do textovej podoby [45].
SPARK	Nástroj pre definovanie doménovo špecifických jazykov v Pythone [46].
TXL	Nástroj pre rýchle prototypovanie nových rozšírení jazyka. Využíva definovanú syntax jazyka a prekladá ju pomocou *source-to-source tranformation* do pôvodného jazyka. Slúži na prezentovanie a skúšanie novej syntaxi jazyka [47].

Tabuľka 2.1: Nástroje a systémy pre vývoj doménovo špecifických jazykov

3 YAJCo

V rámci tejto práce je využívaný nástroj YAJCo. YAJCo je generátor syntaktických analyzátorov založený na anotáciach. Tento nástroj slúžiaci na návrh jazykov je orientovaný na abstraktnú syntax jazyka [25] a využíva anotovaný model jazyka [48].

Hoci nie je nutné pri definovaní jazyka pomocou YAJCa mať k dispozícií vopred jeho gramatiku vo forme BNF alebo EBNF, pre mnohých autorov jazykov je prirodzené rozmýšľať pri návrhu jazyka v pojmoch BNF. Dôvodom je, že tento spôsob definície jazyka je používaný vo veľkom množstve generátorov syntaktických analyzátorov, s ktorými majú predošlú skúsenosť. Preto je možné vysvetliť princípy definície jazyka pomocou nástroja YAJCo práve na pojmoch známych z tohto tradičného prístupu.

Definícia jazyka má v prostredí nástroja YAJCo podobu tried, ktoré vyjadrujú abstraktnú syntax jazyka. Každá trieda prislúcha konceptu jazyka a je ňou vyjadrený neterminálny symbol v gramatike jazyka. Vzťahy medzi triedami ako dedičnosť a kompozícia, sú použité na konštrukciu pravej strany odvodzovacích pravidiel [48]. Pri definovaní abstraktnej syntaxe autor definuje aj konkrétnu syntax a to pomocou anotácií [25]. Detailné informácie o konkrétnej syntaxi by nebolo možné vygenerovať na základe modelu [48], preto sa využívajú spomínané anotácie.

V rámci celého procesu generácie parsera sa na základe modelu jazyka generuje súbor obsahujúci gramatiku. V počiatkoch nástroja sa generovala JavaCC gramatika, keďže ale spôsob, akým je definovaná konkrétna syntax nie je závislý na konkrétnej technológii, je možné doplniť aj iné generátory podľa potreby konkrétnej techniky parsovania [25]. Aktuálne je podporovaný aj Beaver generátor[1].

Spomínané vzťahy medzi triedami použité pre definovanie abstraktnej syntaxe - dedičnosť a kompozícia sa označujú aj ako vzťah „je“ a vzťah „má“. Kompozícia znamená, že koncept obsahuje členské premenné reprezentujúce inštancie iných konceptov, teda neterminálne symboly na pravej strane pravidla. Kom-

[1] `https://github.com/kpi-tuke/yajco/wiki/`

pozícia má isté limity pre použitie na definovanie konkrétnej syntaxe pomocou anotácií. V prípade, že by sa použili anotácie na členských premenných triedy, bolo by potrebné riešiť problémy s faktom, že je možné použiť viacero notácií pre jeden koncept, ďalej s tým, že nie je možné určiť poradie symbolov v pravidle inak ako z poradia v zdrojovom kóde definovanej triedy a tiež problém s konverziou dátových typov medzi konkrétnou a abstraktnou syntaxou. Z týchto dôvodov sú namiesto toho použité anotácie konštruktora [25]. Pri definovaní jazyka je potrebné uviesť aj symboly, ktoré sa spracovávať nemajú. Spravidla sú to biele znaky a komentáre.

Lexikálne jednotky - tokeny, sa uvádzajú pomocou anotácií *@Parser*, *@TokenDef* a *@Skip*. Biele znaky a komentáre sa definujú anotáciou *@Skip*, iné lexikálne jednotky anotáciou *@TokenDef*.

V rámci tejto práce sa zameriavame na definovanie konkrétnej syntaxe pomocou návrhu vzorov a ich následné doplnenie do definície jazyka. Práve pre tento účel sú spomínané anotácie pre definovanie konkrétnej syntaxe a lexikálnej gramatiky dôležité. Zoznam anotácií, ktoré sú využívané počas návrhu a implementované v rámci nástroja YAJCo, spolu s ich stručným opisom, ktorý uvádza [25], sú uvedené v tabuľke 3.1.

3.1 Podpora vzorov pomocou anotácií

Autor [49] v svojej práci rozšíril YAJCo o niekoľko možností, ktoré umožňujú v modeli jazyka aplikovať niektoré vzory. Autor pridal podporu pre typ *Optional*, ktorý je možné využiť, pokiaľ je potrebné vyjadriť to, že výskyt nejakej vlastnosti konceptu v jeho zápise je voliteľný. Tento typ sa dá využiť v rámci implementácie vzoru *flag* a to tak, že sa tento typ priradí parametru konštruktora, ktorý reprezentuje vlastnosť (angl. property) konceptu, ktorá má úlohu príznaku. Pri aplikovaní tohto vzoru sa nahradí typ parametra v modeli YAJCo z typu *Boolean* typom *Optional<String>*, pričom tento reťazec je samotným príznakom.

Ďalším rozšírením bolo pridanie anotácie *@UnorderedParameters* [49] označujúcej taký konštruktor, ktorého parametre sa môžu vyskytovať v ľubovoľnom poradí. Použitie tejto anotácie má význam v rámci vzorov *unordered group* a *alternative notations*. Pri vzore *unordered group* je dôležité to, aby bolo možné zoradiť rôznym spôsobom niekoľko vlastností konceptu. Preto je potrebné v jeho konkrétnej syntaxi umožniť rôzne formy zápisov. Podobne pri vzore *alternative notations* je potrebné využiť rôzne spôsoby zápisu toho istého konceptu. V prípade tohto vzoru nemusí ísť len o umožnenie alternatívneho poradia ale pokiaľ je to vhodné,

je možné v rámci tohto vzoru využiť aj túto anotáciu.

Vo svojej práci tiež autor [49] implementoval možnosť zdieľaného konceptu. To súvisí so vzorom *shared notation*. V rámci tohto vzora je v [8] uvedený príklad opakovanej deklaracie premenných toho istého typu. Tento zápis je možné vo väčšine jazykov skrátiť tak, že sa typ uvedie len raz a za ním nasledujú premenné tohto typu. Implementácia tohto vzoru v modeli je umožnená anotáciou *@Shared*. Parameter tejto anotácie špecifikuje názov parametru konceptu, ktorý bude zdieľaný pre položky poľa v konštruktore konceptu.

Podpora pre vzor *quoted string* je implementovaná v [49] pomocou anotácie *@StringToken*. Implementovanie tejto anotácie zabezpečilo, aby bolo možné v jazyku využívať reťazce uzavreté v úvodzovkách tak, aby v rámci nich neboli rozpoznávané kľúčové slová jazyka.

Impelemntácia vzoru prebieha tak, že sa touto anotáciou označí parameter konštruktora prislúchajúci vlastnosti konceptu, ktorá má typ *String* a je žiadané ju využiť nie ako identifikátor ale ako reťazec uzavretý v zátvorkách. Anotácia svojim parametrom umožňuje špecifikovať, medzi aké symboly sa má reťazec uzavrieť. Napríklad úvodzovky alebo apostrof.

3.2 Definícia jazyka simpleHCL pomocou YAJCo

Definícia jazyka pomocou YAJCa je demonštrovaná na príklade tvorby modelu jazyka simpleHCL. Jazyk simpleHCL bol vytvorený za účelom ukážky vytvárania jazykov v nástroji YAJCo. Je odvodený z jazyka HCL. HCL je metajazykom, ktorý slúži podľa jeho autorov ako systém na tvorbu konfiguračných jazykov. HCL definuje konštrukty, ktoré môže aplikácia využiť na vytvorenie konfiguračného jazyka[2]. K dispozícií je natívna syntax HCL[3], ktorú možno použiť. Na základe tohto jazyka bol zjednodušením - definovaním len niektorých jeho konceptov - vytvorený jazyk simpleHLC. Gramatika simpleHCL je uvedená v 3.1.

Zdrojový kód 3.1: Gramatika jazyka simpleHCL

```
Body = Attribute*;
Attribute = Identifier "=" Expression;
Expression = ExprTerm;
ExprTerm = (LiteralValue |CollectionValue);
LiteralValue = (NumericLit | Boolean| Null)
Boolean = ("true" | "false" );
```

[2] https://github.com/hashicorp/hcl

[3] https://github.com/hashicorp/hcl/blob/main/hclsyntax/spec.md

```
Null = "null";
CollectionValue = Tuple
Tuple = "[" ( (ExprTerm ("," ExprTerm)* ",")?)?) "]";
```

Štartovacím symbolom gramatiky je neterminál Body. To je potrebné uviesť v anotácii *@Parser* zadaním hlavného konceptu jazyka.

Zdrojový kód 3.2: Definícia hlavného konceptu jazyka simpleHCL

```
@Parser(
  mainNode = "sk.tuke.yajco.simpleHCL.model.Body",
  ...
)
```

V jazykoch sa tradične preskakujú pri spracovaní biele znaky. To je definované rovnako v rámci anotácie *@Parser*.

Zdrojový kód 3.3: Definícia preskakovaných znakov v simpleHCL

```
@Parser(
    mainNode = "sk.tuke.yajco.simpleHCL.model.Body",
    skips = {@Skip("\\s")},
    ...
)
```

Samotný štartovací neterminálny symbol *Body* je definovaný triedou **Body**. Konfiguračný súbor sa skladá z postupnosti atribútov. To je vyjadrené vzťahom medzi triedou **Body** a triedou **Attributes** tak, že **Body** obsahuje inštanciu **Attributes**.

Zdrojový kód 3.4: Koncept **Body** jazyka simpleHCL

```
public class Body {
  private Attributes attributes;

  public Body(Attributes attributes) {
      this.attributes = attributes;
  }
  ...
}
```

Trieda **Attributes** vyjadruje koncept postupnosti neterminálnych symbolov *Attribute*, ktoré tvoria konfiguračný súbor. To je vyjadrené tak, že trieda **Attributes** dedí od triedy **ArrayList**. Tento prístup je použitý aj v rámci návrhu jazyka Oberon-0 v [50]. V tomto prípade sú už použité aj anotácie, ktoré špecifikujú konkrétnu syntax, ktorá je priradená konceptu. Je potrebné vyjadriť, že atribúty

sú uzavreté v zátvorkách „{}“ a sú oddelené symbolom „,“. Na to sú využité anotácie *@Separator*, *@Before* a *@After*.

Zdrojový kód 3.5: Koncept **Attributes** jazyka simpleHCL

```
public class Attributes extends ArrayList<Attribute> {

  public static Attributes of(Attribute attribute) {
      return new Attributes(Collections.singletonList(
         attribute));
  }

  public Attributes(@Separator(",") @Before("{") @After(
     "}") List<Attribute> attributes) {
      addAll(attributes);
  }

  public List<Attribute> getAttributes() {
      return this;
  }
}
```

Samotný koncept *Attribute* má nasledovnú syntax.

```
Attribute = Identifier "=" Expression;
```

Je potrebné teda definovať ďalší koncept *Expression*. Pre určenie, akú podobu má mať identifikátor je potrebné doplniť lexikálnu gramatiku. To je definované opäť v rámci anotácie *@Parser* a to tak, že je definovaná lexikálna jednotka *name* spolu s regulárnym výrazom, ktorý popisuje jej gramatiku. Aby bolo možné priradiť tento token príslušnému parametru konceptu *Attribute*, využije sa spomínaná anotácia *@Token*.

Zdrojový kód 3.6: Definícia lexikálnej jednotky *name*

```
@Parser(
  ...
  tokens = {
    @TokenDef(name = "name", regexp = "[a-z]+[A-Za-z0
       -9]*"),
    ...
  }
  ...
)
```

Zdrojový kód 3.7: Priradenie tokenu *name* parametru *name*

```
public class Attribute {
  ...
  public Attribute(@Token("name") String name, @Before("
     =") Expression expression) {
    this.name = name;
    this.expression = expression;
  }
  ...
}
```

Podľa gramatiky je neterminálny symbol *Expression* odvodený do neterminálneho symbola *ExprTerm*. V pôvodnej gramatike HCL je týchto možností viac.

Zdrojový kód 3.8: Časť gramatiky jazyka HCL

```
Expression = ExprTerm;
ExprTerm = (LiteralValue | CollectionValue);
LiteralValue = (NumericLit | Boolean| Null)
```

To je možné vyjadriť pomocou dedičnosti. Keďže samotný neterminál *Expression* nemôže byť vyjadrený v konkrétnej syntaxi, je možné ho definovať v rámci modelu ako abstraktnú triedu **Expression**, od ktorej následne dedia ďalšie triedy. V tomto príklade je to trieda **ExprTerm**.

Trieda **ExprTerm** je v tomto prípade podobná a rovnako je možné ju definovať ako abstraktnú. Od nej následne dedia triedy **LiteralValue** a **CollectionValue**.

LiteralValue je koncept, ktorý je buď reprezentovaný číslom alebo reťazcami "true", "false“ alebo "null". Trieda **LiteralValue** je teda rovnako abstraktná. Od nej dedia triedy **NumericLit**, **Boolean** a **Null**. Všetky tri triedy sú komponované podobným princípom a to tak, že parameter konštruktora je označený anotáciou *@Token*, ktorá uvádza, ktorá lexikálna jednotka mu je priradená.

Zdrojový kód 3.9: Koncept **Boolean** jazyka simpleHCL

```
public class Boolean extends LiteralValue {
  ...
  public Boolean(@Token("bool") boolean value) {
    this.value = value;
  }
  ...
}
```

Ďalším typom konceptu *ExprTerm* je *CollectionValue*. V pôvodnej gramatike existujú koncepty *Tuple* a *Object*. SimpleHCL má implementovaný len koncept *Tuple*. Trieda **Tuple** dedí od triedy **CollectionValue**, ktorá dedí od triedy **ExprTerm**. *Tuple* je koncept vyjadrujúci n-ticu inštancií konceptu **Expression**, ktorá je uzavretá v zátvorkách „[]“ a jednotlivé položky sú oddelené čiarkou. Terminálne symboly zátvoriek a čiarky sú vyjadrené už spomínanými anotáciami *@Separator*, *@Before* a *@After*. Na tejto triede je možné ukázať aj implementáciu vzoru *separator* pomocou anotácie *@Separator* a tiež vzor *brackets* pomocou anotácií *@Before* a *@After*.

Zdrojový kód 3.10: Koncept **Tuple** jazyka simpleHCL

```
public class Tuple extends CollectionValue {
  ...
  public Tuple(@Before("[") @After("]") @Separator(",")
     List<Expression> items) {
    this.items = items;
  }
  ...
}
```

Tento príklad jazyka simpleHCL je stručný a ukazuje len základnú štruktúru návrhu jazyka s YAJCo. Celý zdrojový kód príkladu je dostupný v GitHub repozitári[4].

[4] https://github.com/katarina-siposova/simpleHCL

Anotácia	Stručný opis
@After	Symbol v anotácii má stáť za inštanciou neterminálneho symbolu.
@Before	Symbol v anotácii má stáť pred inštanciou neterminálneho symbolu.
@Identifier	Počas spracovania sa bude kontrolovať jedinečnosť členských premenných označených s touto anotáciou [25].
@Operator	Implementuje jazykový vzor *operator* [25]. Umožňuje zadať prioritu a asociatívnosť operátora operácie, ktorú koncept reprezentuje.
@Optional	Označuje koncept ako voliteľný [25]. V súčasnej verzií je označená ako *@Deprecated*
@Parentheses	Implementuje jazykový vzor *brackets* v koncete operátora [25].
@Parser	Obsahuje konfiguráciu parsera a lexikálnu gramatiku jazyka [25]. Môže sa ňou označiť trieda alebo deklarácia balíka v súbore *package-info.java*. V celom modeli jazyka je len jedna anotácia @Parser.
@Range	Označuje minimum a maximum výskytov inštancií konceptu [25].
@References	Označuje prepojenie na identifikátor.
@Separator	Umožňuje definovať symbol oddeľovača zoznamu prvkov a tak implementovať vzor *separator*. U
@Shared	Umožňuje implementovať vzor *shared notation* [49].
@Skip	Definuje postupnosť symbolov, ktoré sa nespracujú pri syntaktickej analýze.
@StringToken	Umožňuje implementovať vzor *quoted string* [49].
@Token	Priraďuje lexikálnu jednotku konceptu [25]. Lexikálna jednotka v anotácií priraďuje parametru konštruktora neterminálneho symbolu token definovaný v anotácii @Parser.
@TokenDef	Definuje sa pomocou nej lexikálna jednotka.
@UnorderedParameters	Umožňuje v notácii ľubovoľné poradie vlastností konceptu.

Tabuľka 3.1: Anotácie konkrétnej syntaxe v YAJCo

4 Proces analýzy jazyka

Cieľom tejto práce je vytvoriť prototyp nástroja automatizovanej podpory návrhov vzorov počas tvorby jazyka. Tento nástroj analyzuje jazyky navrhnuté v nástroji YAJCo a využíva jeho aplikačné rozhranie aj jeho formu reprezentácie modelu jazyka.

Pre nájdenie možného použitia vzoru je potrebné analyzovať model jazyka. Analýza je založená na definovaných heuristikách. Heuristiky sa týkajú informácií dostupných v modeli. Dôležité sú informácie o konceptoch, ich vztahoch, o typoch ich vlastností a tiež informácie o použitej notácii konceptov. Definované podmienky sú vychádzajú z opisu jednotlivých vzorov, ktoré sú zozbierané v článku [8]. Na základe výsledkov analýzy je potom možné do istej miery predpokladať, že je v danom koncepte vhodné aplikovať nejaký jazykový vzor.

YAJCo sústreďuje definíciu jazyka okolo jeho abstraktnej syntaxe. Koncepty jazyka sú reprezentované triedami. Teda pri návrhu jazyka autor pracuje s Java triedami. Vnútorne, je v rámci projektu model jazyka serializovaný a uložený v súbore XML. Tento súbor obsahuje informácie o jazyku potrebné pre jeho analýzu. Model je preto potrebné pred analýzou deserializovať a následne z neho pomocou aplikačného rozhrania YAJCo získavať informácie o syntaxi jazyka potrebné pre generovanie návrhov vzorov.

V tejto kapitole sú bližšie opísané spôsoby ako sa informácie o jazyku ukladajú a ako je možné ich využiť. Opisované sú spôsoby, akým sa využívajú pre analýzu jednotlivých vzorov.

4.1 Analýza modelu jazyka

Ako bolo načrtnuté v úvode tejto kapitoly, analýza potrebuje získavať informácie dostupné v modeli jazyka. Tieto informácie sa týkajú abstraktnej syntaxe, konkrétnej syntaxe a aj lexikálnej gramatiky jazyka. V YAJCo definuje autor jazyka abstraktnú syntax pomocou Java tried. V nich využíva anotácie na definovanie konkrétnej syntaxe. Lexikálnu gramatiku definuje v rámci anotácie *@Parser*. Spô-

sob, akým si YAJCo uchováva informácie o koncepte a aké informácie je možné získať je opísaný v nasledujúcej časti.

Na príklade konceptu Attribute jazyka simpleHCL sú uvedené informácie uchovávané v rámci modelu ohľadom konceptov jazyka. Zvýraznené sú časti modelu v triede, ktorá koncept definuje. Následne sú zvýraznené uložené informácie o definovanom koncepte v súbore *yajco-model.xml*, ktorý obsahuje serializovaný model.

Zdrojový kód 4.1: Trieda konceptu **Attribute**

```
public class Attribute {
    private String name;
    private Expression expression;

    public Attribute(@Token("name") String name, @Before
       ("=") Expression expression) {
        this.name = name;
        this.expression = expression;
    }
    ...
}
```

Zdrojový kód 4.2: Koncept **Attribute** v XML reprezentácii modelu jazyka

```
<concept>
<name>Attribute</name>
<abstractSyntax>členské premenné konceptu - properties
  <yajco.model.Property>
    <name>name</name>
    <type class="yajco.model.type.PrimitiveType">
      <primitiveTypeConst>STRING</primitiveTypeConst>
    </type>
    <patterns/>
  </yajco.model.Property>
  <yajco.model.Property>
    <name>expression</name>
    <type class="yajco.model.type.ReferenceType">
      <concept>
        <name>Expression</name>
        <abstractSyntax/>
```

```
            <concreteSyntax/>
            <patterns/>
          </concept>Expression je abstraktná trieda. Abstraktná aj konkrétna syntax sú prázdne.
        </type>
        <patterns/>
      </yajco.model.Property>
    </abstractSyntax>
    <concreteSyntax>Vzťahuje sa na konštruktor triedy, ktorý obsahuje notáciu.
      <yajco.model.Notation>
        <parts>
          <yajco.model.PropertyReferencePart>
           <property reference="../../../../../
              abstractSyntax/yajco.model.Property"/>
            <patterns>
              <yajco.model.pattern.impl.Token>
                <name>name</name>
              </yajco.model.pattern.impl.Token>
            </patterns>
          </yajco.model.PropertyReferencePart>
          <yajco.model.TokenPart>
            <token>=</token>
          </yajco.model.TokenPart>
          <yajco.model.PropertyReferencePart>
            <property reference="../../../../../
               abstractSyntax/yajco.model.Property[2]"/>
            <patterns/>
          </yajco.model.PropertyReferencePart>
        </parts>
        <patterns/>
      </yajco.model.Notation>
    </concreteSyntax>
    <patterns/>
  </concept>
```

Dostupné sú aj informácie o lexikálnej gramatike. Tie sú uvedené v anotácii *@Parser*. Tá obsahuje okrem lexikálnych jednotkách spolu s regulárnym výrazom, ktorý ich definuje aj symboly, ktoré sa pri spracovaní programu preskakujú. Napríklad biele znaky, ktoré sa ignorujú.

Zdrojový kód 4.3: Lexikálna gramatika a jej XML reprezentácia

```
@Parser(
  ...
  skips = {@Skip("\\s")},
  tokens = {
    ...
    @TokenDef(name = "number", regexp = "[0-9]+")
    ...
  }
  ...
)

<tokens>
  ...
  <yajco.model.TokenDef>
    <name>number</name>
    <regexp>[0-9]+</regexp>
  </yajco.model.TokenDef>
</tokens>

<skips>
  ...
  <yajco.model.SkipDef>
    <regexp>\s</regexp>
  </yajco.model.SkipDef>
</skips>
```

4.2 Lexikálne vzory

Pri hľadaní niektorých vzorov nie je nutné skúmať koncepty jazyka. Sú to vzory, ktoré sa týkajú len lexikálnych jednotiek jazyka. Možno povedať, že analýza lexikálnych vzorov je spomedzi všetkých jazykových vzorov najjednoduchšia. Nie je pri nich potrebné overovať veľké množstvo podmienok ani do hĺbky analyzovať model. Medzi takéto vzory patria vzor *Number*, *Whitespace* a *Comment*.

Podstatou pri hľadaní lexikálnych vzorov je skúmanie lexikálnej gramatiky a hľadanie lexikálnej jednotky, ktorá so vzorom súvisí. Napríklad pri vzore *Number* je potrebné zistiť, či už lexikálna gramatika neobsahuje lexikálnu jednotku

definujúcu číslo.

Pokiaľ by vzor *Number* v jazyku už implementovaný bol, niekde v anotácii *@Parser* by bola anotácia *@TokenDef*, ktorá by uvádzala názov a regulárny výraz pre číslo. Táto lexikálna jednotka by bola v modeli uložená v elemente *tokens*.

Prostredníctvom aplikačného rozhrania po deserializovaní XML YAJCo modelu, je možné získať zoznam všetkých definovaných lexikálnych jednotiek.

Cieľom je zistiť, či niektorý z definovaných tokenov obsahuje reprezentáciu čísla. Číslo môže mať rôznu notáciu, závislú aj od toho, či ide o celé číslo alebo desatinné číslo. Možno spomenúť niekoľko príkladov. Celé číslo, ktorý sa môže vyjadriť regulárnym výrazom '[0-9]+' alebo číslo s desatinnou čiarkou "([0-9]+[.][0-9]+)".

Lexikálne jednotky sú definované pomocou regulárneho výrazu. Regulárny výraz popisuje gramatiku regulárneho jazyka, ktorý je ním definovaný. Rôzne gramatiky môžu popisovať ten istý alebo podobný jazyk. Problémom je, že je viacero možných regulárnych výrazov, ktorými možno definovať tú istú reprezentáciu čísla.

Je teda potrebné porovnať, či dva regulárne výrazy zachytávajú rovnaké reťazce. Ide v podstate o porovnanie dvoch regulárnych jazykov. Riešenie takéhoto problému môže byť v zistení prieniku medzi množinami reťazcov, ktoré tieto regulárne výrazy produkujú. Navrhovaným postupom je definovať regulárny výraz, ktorý danú notáciu zachytáva a získať prienik tohto výrazu s jazykom definovaným regulárnym výrazom v lexikálnych jednotkách. Pokiaľ je prienikom neprázdna množina reťazcov, tak je lexikálna jednotka schopná zachytiť danú reprezentáciu čísla.

Výpočet prieniku dvoch regulárnych jazykov nie je triviálny. Zároveň implementácia tejto operácie na regulárnych výrazoch nie je ani cieľom tejto práce. Pri praktickej realizácii bolo využité riešenie pomocou Java knižnice *automaton*[1], ktorá operáciu prieniku regulárnych výrazov implementuje. Okrem iného táto knižnica ponúka aj možnosť generovania minimálneho príkladu prieniku. V prípade, že by sme porovnávali regulárne výrazy $[0-9]$ a $[0-9]+$, je generovaný príklad reťazca 0.

Ďalším vzorom, pre ktorý je potrebné analyzovať lexikálnu gramatiku je vzor *whitespace*. V prípade tohto vzoru je potrebné sledovať použité anotácie *@Skip*. Opäť je možné použiť spomínaný postup hľadania prieniku medzi regulárnymi výrazmi. Implementácia YAJCo však implicitne vždy pridáva do definície preskakovaných symbolov biele znaky. Preto v praktických príkladoch je vzor *whitespace*

[1] `https://www.brics.dk/automaton/`

v jazyku vždy prítomný.

Tento spôsob porovnávania regulárnych výrazov je využitý aj pri hľadaní ďalšieho vzoru *comment*. Komentár patrí medzi postupnosti symbolov, ktoré sa pri spracovaní ignorujú. Nachádza sa preto, rovnako ako biele znaky, v elemente *skips*. Pri hľadaní vzoru *comment* je postačujúce zistiť, či sa medzi preskakovanými symbolmi nachádza aj iný symbol ako biely znak. Lexikálne jednotky sú porovnávané s regulárnym výrazom bieleho znaku. Pokiaľ je prienik prázdny, znamená to, že sa našiel komentár.

4.3 Vzor *concept decorator*

Vzor *concept decorator* rieši problém odlíšenia konceptov, ktoré sa vyskytujú v rovnakom kontexte a majú rovnakú štruktúru. Rieši ho tak, že označí koncepty dekorátormi. Tieto dekorátori môžu byť kľúčové slová, oddeľovače alebo iné symboly [8].

Pri analýze tohto vzoru je potrebné sledovať a porovnať štruktúru konceptov modelu jazyka. Analýza sa v tomto prípade zameriava na porovnanie abstraktnej syntaxe konceptov. Indikáciou použitia tohoto vzoru je nájdenie dvoch konceptov, ktoré majú rovnaký počet vlastností. Pričom sa musia rovnať aj typy týchto vlastností. V pojmoch EBNF by pravidlá takých konceptov mali mať rovnakú podobu, čo sa týka poradia a typu neterminálnych symbolov.

Dôležité je tiež zistiť, či tieto koncepty už nemajú vzor *concept decorator* implementovaný. Je potrebné teda kontrolovať aj konkrétnu syntax konceptov a sledovať, či niektorý z nich nemá svoje vlastnosti označené kľúčovými slovami. Dva koncepty, pri ktorých je navrhovaný vzor *concept decorator* by nemali obsahovať žiadne terminálne symboly.

Dobrým príkladom možnosti navrhnúť tento vzor sú koncepty **While** a **If** [8]. Tieto koncepty by mohli byť implementované tak, že by sa skladali z vlastností, ktoré predstavujú podmienku a telo podmienky alebo cyklu.

Zdrojový kód 4.4: Triedy konceptov **While** a **If**

```
public class While {
    public While(ExprTerm condition, List <ExprTerm> body) {
      ...
    }
}
```

```
public class If {
    public If(ExprTerm condition, List <ExprTerm> body) {
      ...
    }
}
```

Pri aplikovaní vzoru je však potrebné dbať aj na to, že koncept môže mať viacero notácií. Pokiaľ teda majú koncepty spoločnú štruktúru abstraktnej syntaxe, je potrebné zistiť, v ktorej forme notácie naozaj konflikt nastane. Analyzuje sa teda aj konkrétna syntax konceptu.

V implementácií modelu jazyka v YAJCo je pri analyzovaní konkrétnej syntaxe dostupná abstraktná syntax jednotlivých častí notácie. Opačne to však neplatí. Teda vlastnosť konceptu nemá informácie o svojej konkrétnej syntaxi.

V tomto prípade je potrebné analyzovať konkrétnu syntax konceptov a nájsť tú notáciu, ktorá je konfliktná. Nájdená notácia je v zmysle YAJCo modelu, ktorý má podobu Java tried, konštruktor triedy konceptu.

Ak by napríklad trieda **While** obsahovala dva konštruktory, jeden by reprezentoval cyklus *while* a druhý *do-while*, pričom by sa tieto konštruktory líšili len poradím parametrov, pre implementovanie vzoru *concept decorator* je dôležitý len ten konštruktor, ktorý má také parametre ako konštruktor konceptu **If**. Podobne ak by aj koncept **If** mal viacero notácií, dôležitá pre vzor *concept decorator* je tá, ktorá má rovnakú štruktúru ako niektorá iná z konceptu **While**.

Zdrojový kód 4.5: Trieda konceptu **While**

```
public class While {
  public While(ExprTerm condition, List <ExprTerm> body) {
    ...
  }
  public While(List <ExprTerm> body, ExprTerm condition) {
    ...
  }
}
```

Aplikácia vzoru *concept decorator* spočíva vo využití YAJCo anotácie *@Before*, ktorou sa označí konfliktný konštruktor. V parametri anotácie sa zadá kľúčové slovo, ktoré predstavuje samotný dekorátor konceptu.

Zdrojový kód 4.6: Trieda konceptu **While** po úprave

```
public class While {
  @Before("while")
  public While(ExprTerm condition, List<ExprTerm> body) {
    ...
  }

  public While(List<ExprTerm> body, ExprTerm condition) {
    ...
  }
}
```

4.4 Vzor *brackets*

Vzor *brackets* sa zaoberá tým, ako je možné vyjadriť kde začína a kde končí nejaká postupnosť symbolov. Tento problém rieši pomocou zátvoriek, do ktorých sa sekvencia uzavrie.

Pri analýze modelu pri hľadaní použitia vzoru *brackets* je dôležité sledovať typ vlastnosti konceptu. Vyhľadávané sú vlastnosti konceptov, ktoré sú kolekciami. Musia buď polia alebo zoznamy.

Pokiaľ by aj koncept mal viacero foriem zápisu, je potrebné nájsť konštruktory len na základe vlastnosti konceptu. Týchto konštruktorov môže byť niekoľko. Je potrebné získať všetky konštruktory, ktoré danú vlastnosť obsahujú. Tiež je nutné overiť, či už vlastnosti zátvorky neobsahujú. Preto sa tiež musí analyzovať aj konkrétna syntax daného konceptu.

Tento vzor sa do modelu jazyka v YAJCo aplikuje pomocou anotácií *@Before* a *@After*. Parametre týchto anotácii obsahujú symboly, ktoré sú použité ako zátvorky, do ktorých sa vlastnosť konceptu uzavrie. Označí sa nimi parameter konštruktora, ktorý danú vlastnosť reprezentuje.

4.5 Vzor *flag*

Vzor *flag* rieši problém, ako reprezentovať vlastnosti konceptu, ktoré nadobúdajú hodnoty *true* alebo *false*. Umožňuje reprezentovať nie len hodnotu, ktorú vlastnosti majú, ale aj názov vlastnosti, ktorá tieto hodnoty nadobúda [8]. Príkladom použitia tohto vzoru v jazyku môže byť kľúčové slovo *static* v Jave. Jeho prítomnosť znamená hodnotu *true* vlastnosti, ktorá určuje, či je metóda statická. Jeho

absencia hodnotu *false*.

Bez použitia vzoru *flag* by notácia konceptu metódy mohla obsahovať kľúčové slovo *static* a dvojbodku, za ktorými by stála hodnota *true* alebo *false* podľa toho, či ide o statickú metódu alebo nie.

Indikáciou pre navrhnutie vzoru *flag* je vlastnosť konceptu, ktorá je typu *boolean*. Vzor *flag* by upravil koncept tak, že pravdivostná hodnota vlastnosti by závisela od prítomnosti alebo neprítomnosti kľúčového slova.

Zdrojový kód 4.7: Trieda konceptu **Method**

```
public class Method {
  private boolean staticFlag;
  Method(..., @Before({"static",":"}) boolean staticFlag){
    this.staticFlag = staticFlag;
    ...
  }
}
```

Zdrojový kód 4.8: Trieda konceptu **Method** po úprave

```
public class Method {
  private Boolean staticFlag;
  public Method(..., Optional<String> staticFlag, ...) {
      this.staticFlag = staticFlag.isPresent();
  }
  ...
}
```

Volanie metódy *isPresent*() zabezpečí nastavenie príznaku na správnu hodnotu. Parameter *staticFlag* by mohol byť označený anotáciou *@Token*, ktorá by špecifikovala kľúčové slovo. Pokiaľ sa však definuje lexikálna jednotka *staticFlag* s týmto menom, nie je to potrebné. Takto sa vzor *flag* prepája zároveň so vzorom *keyword*, ktorý využíva. Token *staticFlag* je kľúčovým slovom.

4.6 Vzory *identifier* a *quoted string*

Identifier sa týka reprezentácie mien v kóde a vzor *quoted string* sa týka reprezentácie ľubovoľných reťazcov, uzavretých medzi symbolmi. Zvyčajne medzi úvodzovkami alebo apostrofmi. Keďže sa pri oboch vzoroch vyhľadávajú vlastnosti konceptu typu reťazec, počas analýzy spolu súvisia.

Pokiaľ je vlastnosť typu reťazec, môže ísť buď o ľubovoľný reťazec alebo o identifikátor ako napríklad názov premennej. V tomto prípade neexistuje spoľahlivá

heuristika ako zistiť, ktorá možnosť je žiadaná. To je prípad, kedy je analýza poloautomatizovaná a vyžaduje si doplňujúci vstup od autora jazyka.

Ak sa autor rozhodne aplikovať vzor *identifier*, je potrebné implementovať lexikálnu jednotku identifikátora. Autor zadá regulárny výraz a názov lexikálnej jednotky identifikátora. Pokiaľ už v lexikálnej gramatike je token so zadaným regulárnym výrazom, je možné ho využiť. Parameter konštruktora prislúchajúci identifikátoru sa označí anotáciou *@Token* s menom lexikálnej jednotky, ktorá bola nájdená. Pokiaľ má koncept viac konštruktorov, označia sa vlastnosti vo všetkých konštruktoroch, ktoré ich obsahujú.

Pokiaľ takýto token ešte v jazyku nie je, vytvorí sa nová položka lexikálnej gramatiky. Pridá sa nová anotácia *@TokenDef* do definície lexikálnej gramatiky v anotácii *@Parser*. Anotácia *@TokenDef* bude obsahovať identifikátor so zadaným menom a regulárnym výrazom.

Ak autor chce umožňiť zadávanie ľubovoľného reťazca na mieste parametra, aplikuje vzor *quoted string*. Tento vzor má priamu podporu v YAJCo pomocou anotácie *@StringToken*. Parameter *delimiter* tejto anotácie určuje, do akých symbolov sa uzavrie reťazec. Implicitne nastavená hodnota je symbol úvodzoviek.

4.7 Vzor *infix operator*

Tento vzor rieši problém reprezentácie matematických výrazov v konkrétnej syntaxi jazyka. Pravidlá matematických operácií ako napríklad operácia sčítania sú v EBNF rekurzívne. Identifikácia použitia tohto vzoru spočíva v nájdení rekurzívneho konceptu, ktorý má práve dve vlastnosti.

Pre nájdenie rekurzívneho konceptu je potrebné sledovať typ jeho vlastností a typ samotného konceptu. Koncept je rekurzívny, ak sa jeho typ alebo typ ktorého je podtypom rovná typom jeho vlastností.

Príkladom takéhoto rekurzívneho konceptu by mohla byť nasledujúca trieda **Add**.

Zdrojový kód 4.9: Trieda konceptu **Add**

```
public class Add extends Expression {
  public Add(Expression left, Expression right) {
      ...
  }
}
```

Trieda **Add** má dve vlastnosti typu **Expression**. Zároveň je podtriedou triedy **Expression**.

Avšak trieda reprezentujúca operáciu sčítania môže mať aj inú podobu. V jazykoch sa často stretávame s definíciou matematických operácií v nasledovnom tvare.

Zdrojový kód 4.10: Triedy **Add**, **BinaryExpression**, **Expression**

```
public class Expression {
  ...
}

public class BinaryExpression extends Expression {
  ...
}

public class Add extends BinaryExpression {
  public Add(Expression left, Expression right){
    ...
  }
}
```

Koncept **Add** má dve vlastnosti typu **Expression** a dedí od konceptu **BinaryExpression**. **BinaryExpression** je podtypom **Expression** a teda koncept je rekurzívny.

Preto je potrebné sledovať aj triedu, od ktorej dedí trieda **BinaryExpression** a porovnať ju s typom vlastností konceptu. V príklade ide o zistenie, či **BinaryExpression** je podtypom **Expression**. Avšak dedičnosť nemusí byť vždy priama. Preto je potrebné pozerať sa na rodičovský typ triedy rekurzívne v hierarchii typov konceptov v jazyku.

Realizácia vzoru *infix operator* je podporovaná pomocou anotácie *@Operator*, ktorá umožňuje definovať asociatívnosť a prioritu operátora, ktorý koncept implementuje. Samotný symbol operátora sa môže definovať pomocou YAJCo anotácie *@Before*, ktorou sa označí druhý parameter v konštruktore.

Zdrojový kód 4.11: Koncept **Add** po implementovaní vzoru *infix operator*

```
public class Add extends BinaryExpression {
  @Operator(associativity = Associativity.LEFT, priority = 2)
  public Class(Expression left, Expression right) {
    ...
  }
  ...
}
```

4.8 Vzor *mixed repetition*

Podstatou vzoru *mixed repetition* je umožniť zapísať niekoľko inštancií rôznych konceptov v prelínajúcom sa poradí. Autori v [8] uvádzajú v príklade použitia tohto vzoru jazyk pre konečnostavový automat. Tento jazyk má koncepty stav - **State**, príkaz - **Command** a udalosť - **Event**. Kolekcie týchto konceptov sa môžu v zápise vyskytovať viackrát za sebou v rôznom poradí. Zdrojový kód by mohol napríklad obsahovať niekoľko stavov a následne jeden príkaz a opäť niekoľko stavov a napokon udalosť. Ich radenie za sebou nie je dôležité pre sémantiku programu. Preto syntax jazyka umožňuje prelínanie konceptov.

Indikáciou pre použitie tohto vzoru je koncept, ktorý obsahuje len vlastnosti, ktoré sú kolekciami. Tieto kolekcie reprezentujú opakovanie konceptov, ktoré je možné ľubovoľne prelínať. Ak také vlastnosti koncept obsahuje, analýza potrebuje získať od autora vstup, či si žiada, aby bolo možné prelínať jednotlivé sekvencie. Pokiaľ áno, aplikuje sa vzor *mixed repetition*.

Podpora tohto vzoru v nástroji YAJCo však aktuálne implementovaná nie je. V rámci implementácie aplikácie tohto vzoru do modelu jazyka bola vytvorená nová anotácia *@MixedRepetition*. Touto anotáciou sa označí konštruktor konceptu, ktorého vlastnosti sa majú dať prelínať.

Implementácia tejto anotácie však do YAJCo pridaná nie je. Jej použitie preto nemá pri generovaní syntaktického analyzátora žiaden účinok. Cieľom tejto práce nie je rozšíriť YAJCo o podporu tohto vzoru. Preto je zatiaľ implementácia ponechaná ako budúce rozšírenie nástroja YAJCo. Vytvorená anotácia *@MixedRepetition* je dostupná[2] a možno ňou označiť konštruktor konceptu.

Riešením problému prelínania bez použitia anotácie by si vyžadovalo vytvorenie novej triedy v modeli jazyka, od ktorej by jednotlivé koncepty dedili. Táto operácia by v pojmoch EBNF znamenala vytvorenie nového neterminálneho symbolu [8]. Pravá strana odvodzovacieho pravidla jeho odvodzovacieho pravidla by obsahovala pravidlo alternatívy konceptov *State*, *Command*, *Event*. To by si vyžadovalo pomerne rozsiahle generovanie zdrojového kódu. Bolo by potrebné vygenerovať novú triedu a upraviť koncepty tak, aby od nej dedili. V novej triede by sa museli vytvoriť členské premenné. Tie by boli typu tried konceptov, ktorých kolekcie sa majú prelínať. Následne by bolo potrebné generovať aj metódy a konštruktory pre novú triedu.

[2] `https://github.com/katarina-siposova/yajco/tree/mixed_repetition`

4.9 Vzor *separator*

Vzor *separator* rieši jednoznačné oddelenie položiek rovnakého typu pomocou oddeľovača. Analýza sa pri tomto vzore zameriava na vlastnosti konceptov, ktoré sú kolekcie. Zisťuje sa, či sú vlastnosti konceptu zoznamom alebo poľom.

Pre aplikovanie tohto vzoru je v nástroji YAJCo implementovaná anotácia *@Separator*. Ňou je možné označiť parameter konštruktora triedy konceptu. Anotácia sa aplikuje na parameter vlastnosti vo všetkých konštruktoroch, ktoré túto vlastnosť obsahujú.

Samotný symbol oddeľovača je, nezávislý od modelu a je úplne na vôli autora, preto je opäť nutné ho zadať ako vstup od užívateľa.

4.10 Vzory *sequence, unordered group, keyword*

Implementácia vzorov *sequence* a *unordered group* spolu súvisí. Oba vzory sa týkajú skupiny konceptov nasledujúcich za sebou. Ak sú koncepty od seba jasne odlíšiteľné, je možné riešiť ich kompozíciu stanovením pevného poradia v rámci zápis konceptu. To je podstatou vzoru *sequence*. Keďže vzor *sequence* vyžaduje stále poradie inštancií konceptu v zápise, je nutné si ho pamätať. To však nemusí byť zložitejších konceptoch jednoduché. Preto sa pri viac ako dvoch položkách odporúča označiť ich dekorátorom. Tento dekorátor môže byť kľúčové slovo, ktoré vyjadruje koncept, ktorý má nasledovať. Takto teda vzor *sequence* súvisí aj so vzorom *keyword*. Pri vzore *unordered group* je naopak poradie vlastností konceptu nepodstatné a je potrebné umožniť ho použiť ľubovoľne.

Počas analýzy sú vyhľadávané koncepty, ktoré sa skladajú z niekoľkých inštancií rôznych konceptov. V rámci rozhodovania pri analýze je minimálny počet vlastností stanovený na tri. Teda pokiaľ má koncept tri a viac vlastností, zistí sa, či neobsahuje nejaké terminálne symboly. Teda či už nejaký jeho parameter nie je označený kľúčovým slovom. Pokiaľ nie je, je to považované za vhodný prípad pre navhrnutie vzoru *sequence*. Je však stále nutné opýtať sa autora jazyka, či na poradí konceptov záleží. Pokiaľ na poradí záleží, získa implementácia kľúčové slová, ktorými sa jednotlivé položky v postupnosti označia, ako vstup od autora. V konštruktore sa príslušná vlastnosť označí anotáciou *@Before* so zadaným kľúčovým slovom.

Pokiaľ na poradí nezáleží, implementuje sa vzor *unordered group*. V tomto prípade je potrebné umožniť radenie prvkov sekvencie v ľubovoľnom poradí. Tento problém spočíva v umožnení rôznych foriem notácie konceptu. V modeli YAJCo

by to znamenalo vytvorenie rôznych konštruktorov pre daný koncept, ktoré by obsahovali rôzne kombinácie jeho vlastností. To však nie je potrebné. Pre aplikovanie tohto vzoru v jazyku je v YAJCo implementovaná anotácia *@UnorderedParameters*, ktorou sa označí konštruktor. Takto sa zabezpečí, že analyzátor rozpozná koncept aj pri rôznych formách jeho zápisu.

Opäť je potrebné zistiť, ktorý konštruktor sa má anotáciou označiť a to na základe typov vlastností konceptu. V prípade, ak by koncept obsahoval viac konštruktorov spĺňajúcich úvodnú podmienku, každý z nich bude osobitným prípadom vzoru *sequence*.

Samotný vzor *keyword* nie je dostatočne špecifický na to, aby ho bolo možné navrhnúť autorovi počas analýzy samostatne. Jeho analýza je preto spojená so vzorom *sequence*. Je spojená ale aj so vzorom *concept decorator*, kde kľúčové slovo predstavuje samotný dekorátor.

4.11 Vzor *shared notation*

Tento vzor spočíva v tom, že viacnásobné výskyty konceptu v texte programu možno spojiť pomocou spoločnej vlastnosti. Podstatou je vytvorenie nového konceptu, ktorý špecifikuje, čo medzi sebou zdieľajú jeho položky. Tento vzor však nie je jednoduché odhaliť na základe modelu jazyka. To, že je žiadané aby bolo možné zapísať stručnejšie a prehľadnejšie istú opakujúcu sa notáciu je zrejmé až zo zdrojového kódu zapísaného v danom jazyku. Preto tento vzor v rámci tejto práce implementovaný nie je. Jeho podpora je však opísaná v predošlej kapitole 3.1.

4.12 Vzor *alternative notation*

Vzor *alternative notation* rieši problém, ako umožniť rôzne formy zápisu toho istého konceptu. Tento vzor nie je v modeli jazyka celkom jednoznačné identifikovateľný. Nie je na základe podmienok možné zistiť, ktoré koncepty by bolo vhodné vedieť reprezentovať aj alternatívnym zápisom. Preto nie sú implementované heuristiky, ktoré by overovali, či nie je možné takýto vzor autorovi jazyka ponúknuť.

Samotná implementácia vzora spočíva v umožnení alternatívnych konštruktorov konceptu. To je implementované v rámci vzoru *unordered group*. Tento vzor *alternative notation* však nemusí spočívať len v zmene poradia vlastností konceptu, ale môže ísť aj o celkom inú notáciu konceptu. Napríklad čo sa týka použitia kľúčových slov.

Implementácia tohto vzoru by mohla využiť anotáciu *@FactoryMethod*, ktorou by sa označili metódy, ktoré by mali funkciu konštruktora. Takto by sa zapezpečilo, aby nevznikli rôzne konfliktné konštruktory z dôvodu napríklad mazania typov (angl. type erasure) pri preklade.

4.13 Vzory *heredoc* a *string interpolation*

Oba tieto vzory súvisia s notáciou, aká sa používa pre reťazce. Vzor *string interpolation* umožňuje do reťazcov zakomponovať aj iné prvky jazyka ako výrazy alebo identifikátory. *Heredoc* umožňuje zadávanie dlhých reťazcov bez takzvaného rušenia. Tieto vzory nie je možné nájsť v modeli jazyka. Je možné ich navrhnúť len na základe typu vlastnosti konceptu. Pokiaľ je vlastnosť typu reťazec a nie je žiadané využiť ho ako identifikátor, je možné navrhnúť autorovi, aby implementoval aj možnosť zadávať reťazec zložený z viacerých riadkov tak ako je to vo vzore *heredoc* alebo aby dokázal využívať vlastnosti vzoru *string interpolation*. Priamu podporu žiaden z týchto vzorov v YAJCo zatiaľ nemá.

5 Nástroj Pattern

V rámci tejto záverečnej práce je implementovaný prototyp nástroja, ktorý poskytuje podporu pre jazykové vzory počas tvorby jazyka. Vytvorený je nástroj Pattern. Pattern implementuje metódy hľadania jazykových vzorov, ktoré boli opísané v predošlej kapitole. Nástroj poskytuje analýzu modelu jazyka definovaného v YAJCo. Na základe výsledkov analýzy generuje odporúčania jazykových vzorov. Interaktívnym spôsobom umožňuje získavať doplňujúce informácie od autora jazyka. Umožňuje aj implementáciu vzorov do jazyka a úpravu jeho modelu.

Aplikácia Pattern funguje je nástrojom príkazového riadka. Využíva rámec Picocli[1], ktorý umožňuje vytvárať aplikácie príkazového riadka v Jave.

Pattern môže fungovať v dvoch rôznych režimoch. Jeden režim je interaktívny. V ňom aplikácia počas analýzy nie len ukazuje autorovi, kde v jeho jazyku je vhodné použiť nejaký vzor, ale mu aj kladie doplňujúce otázky. Aplikácia sa pýta na informácie, ktoré nie je možné zistiť z modelu a čaká na odpovede autora jazyka. Na základe interakcie s používateľom sú vzory do jazyka implementované a model jazyka sa aktualizuje.

Druhým režimom je beh aplikácie bez interakcie. V ňom aplikácia len generuje návrhy vzorov a zobrazuje ich pomocou textových výpisov. Ak chce autor navrhnutý vzor aplikovať, potrebuje využiť príkaz, ktorý daný vzor implementuje.

Hľadanie možných odporúčaní jazykových vzorov spočíva v analýze modelu jazyka. Model sa v YAJCo definuje pomocou Java tried označených anotáciami. Pri procese generovania syntaktického analyzátora pomocou YAJCo je model serializovaný a uložený v súbore *yajco-lang.xml*. Model obsahuje všetky informácie, ktoré sú definované v rámci tried reprezentujúcich koncepty jazyka, ich vzťahy a naviac aj konkrétnu syntax jazyka - notáciu jednotlivých konceptov a tiež lexikálnu gramatiku jazyka. Všetky tieto informácie sú využívané počas analýzy hľadania možných použití vzorov v jazyku. Pre účely nájdenia vzorov je tento

[1] `https://picocli.info/`

model deserializovaný. Aplikácia využíva na získanie všetkých týchto informácii aplikačné rozhranie YAJCo.

Okrem analýzy poskytuje Pattern aj aplikovanie navrhnutých vzorov do jazyka. Implementácia vzoru do modelu spočíva v úprave tried v Java projekte vyvíjaného jazyka. Na manipuláciu s modelom je potrebné získať syntaktický strom zdrojového kódu. Na to treba využiť syntaktický analyzátor jazyka Java. Pattern využíva knižnicu JavaParser[2]. Tá umožňuje okrem spracovania zdrojového kódu v Jave aj jeho transformáciu.

Nasledujúca kapitola bližšie opisuje implementáciu analýzy jazyka a transformáciu jeho modelu.

5.1 Analýza a hľadanie vzorov

Analýza modelu spočíva v prechádzaní konceptov jazyka, jeho lexikálnych jednotiek a konkrétnej syntaxe jednotlivých konceptov. V aplikačnom rozhraní sú informácie o syntaxi dostupné cez objekt triedy **yajco.model.Language**. Hľadajú sa vlastnosti jazyka, ktoré by naznačovali možnosť ponúknuť nejaký vzor. Analýza je implementovaná v triede **LanguageModel**. Kľúčovým bodom je metóda **visit** z ktorej celá analýza vychádza.

Trieda **yajco.model.Language** obsahuje metódu **getConcepts**, ktorá poskytuje zoznam objektov typu **yajco.model.Concept** všetkých konceptov jazyka. Postupne sa získaným zoznamom prechádza a koncepty sa bližšie analyzujú. Je niekoľko podmienok, ktoré sa pri konceptoch overujú. Každá súvisí s niektorým so vzorov. To, na čo sa analýza pri jednotlivých vzoroch zameriava, bolo už opísané v predošlej časti práce.

Ako prvá podmienka sa overuje, či koncept obsahuje len také vlastnosti, ktoré sú kolekcie. To je potrebné pre vzor *mixed repetition*. Proces zisťovania vyžaduje prechádzanie všetkými vlastnosťami konceptu, ktoré sú dostupné prostredníctvom metódy **getAbstractSyntax** triedy **yajco.model.Concept**. Táto metóda poskytuje zoznam prvkov typu **yajco.model.Property**. Trieda **Property** obsahuje metódu **getType**, ktorá poskytuje informáciu o type vlastnosti objektu. Typ je reprezentovaný triedou **yajco.model.type.Type**.

Veľkú časť analýzy modelu tvorí zisťovanie typov konceptov a ich vlastností. Vyhľadávané sú vlastnosti, ktoré sú typu **ListType** alebo **ArrayType**. Triedy **ListType** a **ArrayType** dedia od triedy **Type**. To je dôležité pre vzory *brackets* a *separator*. Pre vzor *flag* je dôležité, či je vlastnosť typu **Boolean**.

[2] `https://javaparser.org/`

Ďalšou vlastnosťou, ktorá sa zisťuje je to, či je koncept rekurzívny. Opäť ide o analyzovanie typov vlastností. Proces zisťovania, či je koncept rekurzívny bol načrtnutý v predošlej kapitole pri vzore *infix operator*, pre ktorý je táto informácia dôležitá. V prvom rade je dôležité zistiť počet vlastností konceptu, teda zistiť veľkosť zoznamu položiek získaných metódou **getAbstractSyntax**. Ak má koncept dve vlastnosti, získa sa typ prvej aj druhej z nich.

Ak sa tieto typy nerovnajú, koncept nie je rekurzívny. Ak sa rovnajú, je potrebné pozrieť sa na rodičovský typ konceptu. Rodičovský koncept je dostupný pomocou metódy **getParent** triedy **Concept**. Rekurzívne sa zisťuje, či je typ rodiča aktuálneho konceptu rovný typu jeho vlastností. Rekurzívne hľadanie rodiča sa ukončí, pokiaľ sa typ rodiča a typ vlastnosti rovná alebo ak už koncept rodiča nemá, teda je na vrchole hierarchie konceptov. Pokiaľ sa po skončení procesu typy rovnajú, koncept je rekurzívny.

Pre vzory je dôležitá aj konkrétna syntax jazyka. Model ju môže a nemusí obsahovať do potrebnej miery. Závisí to od toho, v akom štádiu je návrh jazyka. Konkrétna syntax sa netýka len kľúčových slov, zátvoriek a podobne ale aj notácií konceptov. Pod týmito notáciami sa myslí konkrétna forma zápisu konceptu v zmysle poradia ich vlastností v zápise. Je to informácia, ktorá je v YAJCo modeli reprezentovaná definíciou konštruktora konceptu.

Informáciu o konkrétnej syntaxi získame pomocou metódy **getConcreteSyntax** triedy **Concept**. Táto metóda poskytuje zoznam notácií, ktoré sú reprezentované objektmi triedy **yajco.model.Notation**. Notácia sa skladá z častí, ktoré sú reprezentované triedou **NotationPart**. Zoznam získaný metódou **getConcreteSyntax** môže obsahovať viacero objektov notácií. Pokiaľ trieda konceptu obsahuje viac konštruktorov, prípadne *factory* metód, získaný zoznam bude obsahovať viac objektov notácií.

Časti notácie, teda objekty typu **NotationPart**, sú buď terminálne alebo neterminálne symboly. Sú detailnejšie špecifikované tým, či záleží na ich poradí alebo či sú v rámci zápisu voliteľné. To je určené ich typom, ktorý je niektorá z tried dediacich od triedy **NotationPart**. Získať časti notácie možno metódou **getParts** triedy **NotationPart**.

Ďalej je možné získať informáciu o type vlastnosti konceptu, ktorú *notation part* v notácii zastupuje. Táto informácia sa získava pomocou referencie na *property* konceptu, ktorú *notation part* má.

Pri vzoroch *identifier* a *quoted string* sa táto informácia o type skombinuje s informáciou o tom, či je *notation part* spojená s nejakou lexikálnou jednotkou. Pokiaľ je časť notácie typu *String* a nemá prislúchajúcu žiadnu lexikálnu jednotku, je spl-

nená základná podmienka pre navrhnutie týchto vzorov.

Ak je počet častí, z ktorých sa skladá notácia väčší ako tri a zároveň medzi týmito časťami nie je žiadna inštancia triedy **TokenPart**, teda koncept neobsahuje kľúčové slová, ktorými by boli označené časti jeho zápisu, je splnená kľúčová podmienka pre vzor *sequence*.

Analyzovanie notácií je dôležité aj pre vzor *concept decorator*. Je nutné nájsť notáciu, ktorá je konfliktá. Ak má koncept niekoľko notácií, je potrebné zistiť, ktorá z nich je rovnaká ako niektorá iná notácia iného konceptu. Preto je potrebné kombinovať informáciou o konkrétnej syntaxi spolu s informáciou o vlastnostiach konceptov ako ich počet a typ.

Analýza pri vzoroch *comment*, *number* a *whitespace* spočíva v porovnaní regulárnych výrazov lexikálnych jednotiek v lexikálnej gramatike. Pri vzore *whitespace* sa porovnávajú regulárne výrazy, ktoré definujé preskakované sekvencie symbolov. Tie sú dostupné z modelu jazyka pomocou metódy *getSkips* triedy **yajco.model.Language**. Táto metóda vracia zoznam objektov typu *SkipDef*. Jednotlivé regulárne výrazy v objektoch triedy **SkipDef** sú porovnávané s definovaným regulárnym výrazom bieleho znaku. Porovnanie je realizované s využitím knižnice *automaton*[3], ktorá implementuje operáciu vypočítania prieniku dvoch regulárnych výrazov. Porovnanie má nasledujúcu podobu.

```
new RegExp( (WHITESPACE)&(SKIP) , RegExp.INTERSECTION).
   toAutomaton().isEmpty()
```

WHITESPACE je regulárny výraz bieleho znaku a SKIP je regulárny výraz nachádzajúci sa v objekte triedy **SkipDef**. Knižnica Automaton vykoná výpočet prieniku dvoch regulárnych jazykov (RegExp.INTERSECTION). Následne sa overí, či je množina reťazcov v prieniku prázda. Pokiaľ nie je, znamená to, že vzor *whitespace* je už implementovaný.

Pri vzore *comment* sa tiež vykonáva rovnaké porovnanie. Ak je množina prieniku prázda, znamená to, že medzi preskakovanými symbolmi v zozname vrátenom *getSkips* metódou sa nachádza aj iná sekvencia ako biely znak. To znamená, že v jazyku je komentár už implementovaný.

Pri vzore *number* sa rovnakým spôsobom porovnávajú definované lexikálne jednotky s regulárnym výrazom pre notáciu čísla. Lexikálne jednotky je možné získať metódou *getTokens* triedy **Language**. Táto metóda vráti zoznam objektov typu **TokenDef**. Regulárny výraz z objektov **TokenDef** sa porovná s definovaným regulárnym výrazom čísla. Ak je výsledný prienik prázdny, jazyk neobsahuje vzor *number*.

[3] `https://www.brics.dk/automaton/`

5.2 Publikovanie udalostí

Informácie o modeli jazyka potrebné pre jednotlivé vzory sú získavané v metóde *visit*, ktorá je implementovaná v triede **LanguageModel**. Táto metóda vykonáva prvotnú analýzu. Získané informácie sú poslané jednotlivým vzorom na ďalšiu analýzu a spracovanie.

Každý vzor je reprezentovaný rovnomennou triedou. Jednotlivé objekty vzorov sú upozornené na publikovanie udalostí, ktoré odoberajú. Takto sú notifikované o zistených informáciách modelu, ktoré s nimi nejako súvisia a sú pre dané vzory dôležité. Napríklad vzor *separator* je notifikovaný, ak sa nájde v koncepte kolekcia. Vzory informácie zachytia a ďalej bližšie analyzujú potrebný koncept alebo vlastnosť a zisťujú, či je vhodné navrhnúť daný vzor. Ak je to vhodné, generujú odporúčanie vzoru. Prípadne si žiadajú ďalší vstup od používateľa, pokiaľ je aplikácia spustená v interaktívnom režime. Na základe vstupu potom aplikujú navhrnutý vzor do jazyka.

Publikovanie informácií je realizované pomocou zbernice, do ktorej sa publikujú udalosti o rôznych vlastnostiach modelu získaných počas analýzy. Zbernicu implementuje trieda **EventBus**. Dáta, ktoré sa po zbernici posielajú, obsahujú informácie o koncepte a vlastnostiach konceptu, ich konkrétnej syntaxi alebo informácie týkajúce sa lexikálnych jednotiek jazyka.

Vzory implementujú rozhranie **Pattern**, ktoré obsahuje metódu *update*- Táto metóda sa zavolá pri publikovaní správy. Koncepty sa prihlásujú na odber pri svojom vzniku. Pri vytvorení inštancie vzoru sa vzor prihlási na odber žiadaných udalostí. Ak sa následne pri analýze publikuje udalosť nejakého typu, napríklad udalosť, že sa v jazyku vyskytuje rekurzívny koncept, zavolá sa metóda na všetkých objektoch, ktoré sú prihlásené na odber informácií o rekurzívnych konceptoch. V tomto prípade odoberá túto informáciu len vzor *infix operator*. Objekt vzoru obdrží spolu s udalosťou aj dáta, ktoré potrebuje bližšie analyzovať alebo spracovať.

Každý vzor potrebuje zistiť, či už v jazyku implementovaný nie je. Na to jednotlivé vzory analyzujú notáciu a použité lexikálne jednotky.

Niektoré vzory sa nezaobídu bez získania vstupu od používateľa. Pokiaľ je aplikácia spustená v interaktívnom móde, je možné získavať doplňujúce informácie počas jej behu. Ak nie je, tak sú vygenerované návrhy vzorov spolu s inofrmáciou o prípade, kedy je vhodné zvážiť použitie iného vzora.

5.3 Implementácia príkazov

Ak je aplikácia spustená bez interaktívneho režimu, generované návrhy vzorom len zobrazí rostredníctvom textových výpisov. Ak chce autor vzor aplikovať, potrebuje využiť príkazy, ktoré umožňujú aplikovanie jednotlivých vzorov.

Každá trieda vzoru reprezentuje podpríkaz hlavného príkazu aplikácie, ktorý je spúšťaný z príkazového riadka. Hlavným príkazom je príkaz *pattern*. Aplikácia obsahuje aj ďalší príkaz, ktorý nie je príkazom vzoru. Týmto príkazom je možné nastaviť cestu ku projektu s jazykom.

Každý z príkazov vzorov vyžaduje rôzne parametre. Počas interaktívneho behu nie je nutné ich zadávať, väčšina z nich vyplýva z analýzy. Ide napríklad o názov konceptu, ktorý sa má doplniť o žiadaný vzor alebo konkrétna vlastnosť, ktorú je napríklad potrebné označiť nejakým dekorátorom. Pokiaľ nie je aplikácia spustená v interaktívnom móde, používateľ zadáva parametre vzorom pomocou prepínačov. Rámec Picocli poskytuje možnosti jednoduchej implementácie prepínačov a parametrov príkazov. Trieda implementujúca príkaz sa označí anotáciami, ktoré tento rámec využije pre vytvorenie rozhrania príkazového riadka.

Triedy vzorov implementujú rozhranie **Callable<Integer>**. Tým sa zabezpečí, aby ich mohol rámec Picocli využiť na realizovanie príkazov. Vzory implementujú metódu **call** rozhrania **Callable**. Implementácia tejto metódy realizuje logiku aplikovania daného vzoru do modelu jazyka.

Ako príklad možno uviesť triedu vzoru *separator*.

Zdrojový kód 5.1: Trieda, ktorá implementuje vzor **Separator**

```
@CommandLine.Command(
  name = "separator",
  mixinStandardHelpOptions = true,
  description = "This command marks the concept parameter with
      a @Separator annotation."
)
public class Separator extends Mutable implements Pattern,
   PatternCommand, Callable<Integer> {

  @CommandLine.Option(
    names = {"-c", "--concept"},
    description = "concept",
    required = true
  )
  String concept;
```

```
    @CommandLine.Option(
      names = {"-p", "--property"},
      description = "property",
      required = true
    )
    String property;

    @CommandLine.Option(
      names = {"-s", "--separator"},
      description = "separator",
      required = true
    )
    String separator;

    @CommandLine.Option(
      names = {"separator"},
      description = "Hit Enter to dismiss.",
      interactive = true,
      echo = true, hidden = true
    )
    String interactiveSeparator;

    @Override
    public void update(EventData data) {
      ...
    }

    @Override
    public Integer call(){
      ...
    }

}
```

Trieda je označená anotáciou *@Command*, ktorou sa označujú príkazy v rámci Picocli. Členské premenné **concept**, **property** a **separator** sú označené anotáciami *@Option*, ktoré označujú prepínače, ktoré nastavujú možnosti spustenia príkazu. Premenná *interactiveSeparator* je používaná počas interaktívneho beho aplikácie. Parameter jej anotácie *@Option interactive* je nastavený na hodnotu **true**, čo umož-

ňuje zadávať symbol oddeľovača interaktívne a čakať na vstup od používateľa.

5.4 Úprava modelu jazyka

Aplikovanie vzoru do jazyka spočíva v úprave YAJCo modelu jazyka. Zväčša ide o malé úpravy ako pridanie anotácie, ale aj o zmeny typov parametra konštruktora alebo úprava definície lexikálnej gramatiky, ktorá je definovaná v anotácii *@Parser*.

Aplikovanie vzoru do modelu prebieha v niekoľkých krokoch. Prvým je získanie syntaktického stromu triedy, ktorá sa má upraviť. Následne je potrebné nájsť konkrétnu časť syntaktického stromu, ktorú chceme modifikovať. Napríklad konštruktor. Hľadanie konštruktora sa deje väčšinou na základe typov jeho parametrov, ktoré sú získané z abstraktnej syntaxe jazyka počas analýzy. Prípadne na základe mena parametra, ktorý reprezentuje vlastnosť konceptu, ktorú chceme upraviť.

Na modifikáciu modelu jazyka je využitá knižnica JavaParser, ktorá implementuje spracovanie zdrojového kódu v Jave, jeho transformáciu a generovanie. Po vykonaní úprav sa syntaktický strom prevedie do podoby reťazca a zapíše sa do príslušného súboru.

Pri lexikálnych vzoroch, ale aj pri vzoroch ako *identifier*, *quoted string* alebo *flag* je potrebné pracovať s lexikálnou gramatikou. Lexikálne jednotky sú definované v anotácii *@Parser*. Je potrebné spracovať súbor, ktorý túto anotáciu obsahuje.

Anotácia *@Parser* sa väčšinou nachádza v súbore *package-info.java*. Anotáciou *@Parser* je možné označiť aj niektorú triedu jazyka. Napríklad v prípade malého jazyka je anotáciou označená hlavná trieda jazyka. Hlavná trieda je trieda, ktorá je v pojmoch EBNF štartovacím symbolom gramatiky. V takom prípade sa spracuje namiesto súboru *package-info.java* trieda, ktorá anotáciu *@Parser* obsahuje.

V projekte jazyka sa vždy nachádza len jedna anotácia *@Parser*. Pri zadávaní cesty k projektu jazyka sa zadá aj meno súboru, ktorý túto anotáciu obsahuje. Pokiaľ sa nezadá, implicitne ide o súbor *package-info.java*. Zadávanie cesty k projektu jazyka implementuje príkaz *init*. Jeho použitie je opísané v systémovej príručke.

Po úspešnej úprave jazyka je potrebné dosiahnuť aj aktualizovanie serializovaného modelu. Tento model je používaný počas analýzy vzorov. V YAJCo vzniká počas generovania syntaktického analyzátora po spustení príkazu **maven clean package**. Aplikácia Pattern tento príkaz spúšťa vzdialene v projekte jazyka. Využíva na to metódu *exec* triedy **java.lang.Runtime**.

```
Runtime.getRuntime().exec("mvn clean package", null,
   languageProject);
```

Proces tvorby jazyka však nie je vždy priamočiary. Autor môže navrhnúť napríklad konfliktné pravidlá, ktoré spôsobia, že gramatika je nejednoznačná. Rovnako môže do modelu zaviesť iné chyby, ako napríklad používanie tokenu, ktorý nie je definovaný a podobne. Všetky tieto chyby končia neúspešným vygenerovaním syntaktického analyzátora. Z toho dôvodu sa ani neserializuje model jazyka, ktorý je pre analýzu vzorov potrebný. Dostupnosť modelu by nemala závisieť od úspešnosti generovania parsera, keďže ide len o serializovanie modelu, ktorý je už v podobe Java tried definovaný.

Jednou možnosťou ako riešiť problém nedostupnosti modelu by bolo jednoducho čakať, kým budú z modelu odstránené všetky chyby. Teda nespustiť analýzu pokým sa úspešne nevygeneruje parser a model sa neserializuje. Druhou možnosťou by bolo v prípade, ak model nie je dostupný, analyzovať samotné triedy modelu. To by však vyžadovalo implementáciu statickej analýzy Java tried. Takáto implementácia navyše by nebola efektívna.

Zvolené je riešenie, pri ktorom sa úpraví samotný nástroj YAJCo. Je potrebné zabezpečiť, aby sa generovanie modelu uskutočnilo aj vtedy, keď nastane nejaká výnimka počas vytvárania syntaktického analyzátora. Po zachytení výnimky je pomocou triedy *XMLSerializer* vygenerovaný súbor *yajco-lang.xml*. Po tejto úprave je YAJCo model k dispozícií takmer za každých okolností. Nie je samozrejme dostupný, pokiaľ triedy modelu neprejdú kompiláciou. Upravená verzia YAJCo je dostupná v GitHib repozitári[4].

Pri modifikácii modelu môže nastať situácia, keď sa chyba do modelu zavedie po aplikovaní nejakého vzoru. V takom prípade je vhodné umožniť používateľovi vrátiť späť vykonané zmeny. Autor jazyka si nemusí byť vedomý všetkých zmien. Pri manuálnej úprave by ich nemusel všetky vrátiť späť. Preto si pred aktualizáciou modelu aplikácia uloží zmenené súbory spolu s ich predošlým obsahom. V prípade výskytu chýb aplikácia vyzve používateľa, či chce vrátiť vykonané zmeny späť. Ak výzvu príjme, všetky zmenené súbory sa nahradia ich predošlým obsahom.

Ukladanie histórie modelu implementuje trieda **History**. Táto trieda obsahuje mapu súborov spolu s ich obsahom. Do tejto mapy sa ukladajú súbory ešte predtým, než sú modifikované. V prípade, že je žiadané zvrátiť vykonané zmeny, prechádza sa mapou a obsah všetkých súborov v mape sa nahradí ich predošlým obsahom. Trieda obsahuje aj metódu pre vymazenie histórie verzií súborov, ktorá

[4] https://github.com/katarina-siposova/yajco/tree/xml_generation

vymaže všetky položky mapy.

5.5 Ukážka použitia nástroja Pattern

V tejto časti sa nachádza ukážka fungovania nástroja Pattern. Ako príklad použitia nástroja je zvolený vzor *identifier*. V príklade sa analyzuje model jazyka simpleHCL. SimpleHCL bol predstavený v predošlej časti práce3.2. Bol navrhnutý za účelom ukážky tvorby jazyka v YAJCo.

Gramatika jazyka simpleHCL je dostupná v jeho GitHub repozitári[5]. Jazyk okrem iných obsahuje koncept **Attribute**. Atribúty sú základným prvkom jazyka simpleHCL. Atribút sa skladá z mena a hodnoty. Má vlastnosti **name** a **expression**.

Zdrojový kód 5.2: Koncept **Attribute**

```
public class Attribute {
  private String name;
  private Expression expression;

  public Attribute(String name, @Before("=") Expression
     expression) {
    this.name = name;
    this.expression = expression;
  }
}
```

Aplikácia sa spustí pomocou príkazu *pattern*. Interaktívny režim sa zapne pomocou prepínača **-i**. Pattern poskytne autorovi textový výpis o priebežnom výsledku analýzy.

Zdrojový kód 5.3: Výpis analýzy vzoru *identifier*

```
$ ./pattern -i

The property name in the concept Attribute is of type
   String.
If it is an identifier, you should create a token in the
    parser definition and specify what this identifier
   should look like.
Do you want to assign a token to the property name ?
```

[5] https://github.com/katarina-siposova/simpleHCL

```
[y/n] (Hit Enter to dismiss):
```

Parameter konceptu *name* je typu *String*. To znamená, že je buď identifikátorom alebo reťazcom. Analýza nevie na základe modelu zistiť, ktorá možnosť je žiadaná. Preto je potrebné získať doplňujúce informácie od používateľa. Pri spustení v interaktívnom móde tak aplikácia kladie autorovi otázky.

Ak autor odpovie kladne, aplikácia ho žiada aby zadal regulárny výraz, ktorý bude definovať novú lexikálnu jednotku a meno lexikálnej jednotky.

Zdrojový kód 5.4: Interaktívne zadávanie lexikálnej jednotky

```
Enter value for regexp (Hit Enter to dismiss.): [a-z]+\w*
Enter value for identifier (Hit Enter to dismiss.):
   attributeName
```

Ak v lexikálnej gramatike existuje token so zadaným regulárnym výrazom, je možné ho využiť a označiť parameter anotáciou *@Token* s menom tohto tokenu.

Zdrojový kód 5.5: Výpis pri nájdení zadanej lexikálnej jednotky

```
There already is a token with the same regexp you entered
   defined in the model.
Do you want to use the token to mark the property name?
[y/n] (Hit Enter to dismiss):
```

Pokiaľ používateľ odpovie áno, parameter sa označí anotáciou *@Token* s existujúcim identifikátorom. Ak nie, autor je vyzvaný zadať nový regulárny výraz. Napokon keď sa autor rozhodne jedným z týchto spôsobov aplikovať vzor *identifier*, je model jazyka modifikovaný. Ak upravený model obsahuje nejaké chyby, aplikácia ponúkne autorovi možnosť vrátiť zmeny späť. Aplikácia si udržiava záznamy o histórii zmenených súborov. Tie sa potom môžu vrátiť do pôvodnej verzie pred úpravami. Následne analýza pokračuje ďalej.

Zdrojový kód 5.6: Výpis aplikácie Pattern pri chybnom modeli jazyka

```
[ERROR] Failed to execute goal org.apache.maven.plugins:
   maven-compiler-plugin:3.1:compile (default-compile) on
    project simpleHCL: Fatal error compiling: java.lang.
   RuntimeException: yajco.generator.GeneratorException:
   Cannot generate Beaver parser (beaver.spec.
   Grammar$Exception: grammar has conflicts) -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-
   run Maven with the -e switch.
```

```
[ERROR] Re-run Maven using the -X switch to enable full
   debug logging.
[ERROR]
[ERROR] For more information about the errors and
   possible solutions, please read the following articles
   :
[ERROR] [Help 1] http://cwiki.apache.org/confluence/
   display/MAVEN/MojoExecutionException
The model contains errors.
Do you want to revert the changes?
[y/n] (Hit Enter to dismiss):y
```

Na začiatku kládla aplikácia autorovi otázku, či chce priradiť parametru konceptu lexikálnu jednotku. Ak by autor nechcel, môže to znamenať, že parameter má byť ľubovoľným reťazcom v zdrojovom kóde. V takom prípade sa analýza zameria na vzor *quoted string*. Ak sa autor rozhodne implementovať vzor *identifier*, je potrebné notifikovať vzor *quoted string* o tom, že nemá už byť navrhnutý.

Pozastavenie navrhovania vzoru implementuje trieda *Mutable*, ktorú triedy vzorov rozširujú. Táto trieda si ukladá dáta typu *MuteData*, ktoré obsahujú koncept, vlastnosť a vzor, ktorého sa týkajú. Ak je do zbernice publikovaná udalosť o stíšení nejakej správy, sú notifikované všetky vzory. Ak sa trieda vzoru v objekte *MuteData* zhoduje s tiedou notifikovaného vzoru, uloží si tento objekt medzi stíšené návrhy vzorov. Pokiaľ je neskôr notifikovaný o nejakej udalosti, porovná prijaté dáta so stíšenými. Ak zistí, že táto správa je stíšená, nevygeneruje návrh vzoru.

Ak si autor neželá implementovať vzor *identifier*, je navrhnutý vzor *quoted string*. Vzor *quoted string* sa aplikuje pomocou anotácie *@StringToken*. Ak autor zadal symbol oddeľovača reťazca, je uvedený v parametri anotácie. Ak nie, anotácia ostane prázdna a používa sa prednastavená hodnota - úvodzovky.

Zdrojový kód 5.7: Dialóg vzoru *quoted string*

```
If the property name of the concept Attribute should be
   a string literal, implement a quoted string pattern.
You may want to allow the language to construct a string
    by concatenating text fragments with other language
   elements (e.g., variables).
Also allow multiline text that require escaping without
   the need to escape for better readability.
```

```
Known form:
<<<HEREDOC
long
multiline
text.
HEREDOC
Do you want the property name to be a string literal?
[y/n] (Hit Enter to dismiss):y
Enter value for delimiter (Default value is "."):
```

V rámci vzoru *quoted string* je zároveň zobrazená informácia o možnej podpore pre dlhé reťazce a reťazce v ktorých sú zakomponované výrazy jazyka. Dlhé reťazce, ktoré je potrebné ošetriť takzvaným rušením (z angl. escaping), je možné implementovať v jazyku aj bez rušenia pomocou vzoru *heredoc*. Reťazce, ktoré majú v sebe vložené výrazy jazyka implementuje vzor *string interpolation*. Výrazy sú v reťazci zakomponované pomocou použitia špeciálnych symbolov, do ktorých sa uzavrú zástupné symboly (z angl. placeholders), na ktorých miesto neskôr majú prísť samotné výrazy. Oba tieto vzory však nemajú priamu podporu v YAJCo a ich aplikácia do modelu teda nie je v rámci práce implementovaná.

V pasívnom režime aplikácia len generuje návrhy vzorov a informuje o možnostiach, ako ich aplikovať pomocou príkazov.

Zdrojový kód 5.8: Výpis aplikácie Pattern v pasívnom režime

```
The property name in the concept Attribute is of type
   String.
If it is an identifier, you should create a token in the
    parser definition and specify what this identifier
   should look like.
You can use command identifier
Run pattern identifier -h to view the usage.

If the property name of the concept Attribute should be
   a string literal, implement a quoted string pattern.
You may want to allow the language to construct a string
    by concatenating text fragments with other language
   elements (e.g., variables).
Also allow multiline text that require escaping without
   the need to escape for better readability.
```

```
Known form:
<<<HEREDOC
long
multiline
text.
HEREDOC
You can use command quoted-string
Run pattern quoted-string -h to view the usage.
```

Ak chce autor aplikovať nejaký vzor, využije na to príkazy jednotlivých vzorov. Vzory sú implementované ako podpríkazy hlavného príkazu. Prepínačmi s parametrami sa im nastavujú jednotlivé hodnoty ako meno konceptu, vlastnosti konceptu a ostatné potrebné parametre, ktoré sú pre každý vzor odlišné. Pri vzore *identifier* je nutné zadať meno a regulárny výraz lexikálnej jednotky identifikátora.

Zdrojový kód 5.9: Príklad aplikovania vzoru v pasívnom režime

```
$ ./pattern identifier -c "Attribute" -p "name" -i "name
   " -r "[a-z]+\w*"
```

Analýza vlastnosti konceptu typu *String* prešla v tejto ukážke od vzoru *identifier* k vzoru *quoted string*.

6 Vyhodnotenie

Vyhodnotenie implementácie prototypu nástroja podpory vzorov bolo vykonané na analýze vybraných jazykov. Bolo zvolených päť jazykov. Jazyky boli vybraté z existujúcich jazykov navrhnutých pomocou YAJCo. Tieto jazyky sú dostupné v GitHub repozitári[1]. Spomedzi týchto jazykov boli vybrané jazyky JSON, MathExpression, simpleRobot, StateMachine a DeskNielsen.

V každom jazyku sa najprv manuálne identifikovali jazykové vzory. Pre každý nález sa zaznamenal koncept, vlastnosť a prislúchajúci vzor. Pri lexikálnych vzoroch sa koncept ani vlastnosť nezaznamenali. Pri niektorých vzoroch sa zaznamenal len koncept bez vlastnosti. Následne sa identifikované vzory z jazyka odstránili.

Po odstránení vzorov sa spustila analýza modelu jazyka. Výsledkom analýzy boli nálezy možných použití vzorov v jazykoch. Medzi navrhovanými vzormi boli označené výskyty vzorov, ktoré boli pred analýzou odstránené. Označené boli aj tie, ktoré analýza navrhla, ale predtým neboli súčasťou modelu.

Boli zaznamenané aj hodnoty, ktoré určovali, či je nájdený vzor relevantný alebo nie. Všetky vzory, ktoré boli odstránené z jazyka boli označené ako relevantné. Nie všetky návrhy vzorov, ktoré predtým neboli súčasťou modelu, boli nerelevantné. Napríklad navrhovaný vzor *comment*, ktorý nebol medzi odstránenými vzormi v jazyku DeskNielsen.

[1] `https://github.com/kpi-tuke/yajco-examples`

Jazyk	T_P	F_P	F_N	P	R
JSON	7	3	3	0,7	0,7
StateMachine	6	3	4	0,67	0,6
DeskNielsen	3	1	4	0,75	0,43
MathExpression	10	0	0	1	1
simpleRobot	2	2	2	0,5	0,5

Tabuľka 6.1: Súhrnné výsledky analýzy

Je potrebné poznamenať, že v použitej verzii YAJCo je implementácia vzoru *whitespace* implicitná. Pokiaľ nie sú definované sekvencie, ktoré sa majú preskakovať, implicitne sa do vygenerovaného modelu pridáva preskakovanie bielych znakov. Aj keď sa na začiatku z modelu odstránili všetky jazykové vzory, vzor *whitespace* bol vo výslednom serializovanom modeli, ktorý sa analyzuje, vždy prítomný. Preto tento vzor nie je vôbec súčasťou vyhodnotenia.

Medzi vygenerovanými návrhmi vzorov boli spočítané výskyty skutočných pozitív (T_P), falošných pozitív (F_P) a falošných negatív (F_N). Z týchto boli následne určené hodnoty presnosti (P) (precision) a výťažnosti (R) (recall). Hodnoty pre jednotlivé jazyky sú uvedené v tabuľke 6.1.

Skutočné pozitíva sú nálezy vzorov, ktoré boli navrhované a zároveň relevantné. Nálezy, ktoré boli ponúkané ale sú nerelevantné, sú falošné pozitíva. Falošné negatíva sú relevantné návrhy vzorov, ktoré analýza nedokázala odhaliť.

Analýza, ktorá má vysoké hodnoty výťažnosti a malú presnosť identifikuje veľa vzorov ale málo z nich je relevantných. Naopak analýza, ktorej výstup ukazuje vysokú hodnotu presnosti ale malú výťažnosť dokázala identifikovať len málo vzorov ktoré mala. Avšak tie, ktoré identifikovala, identifikovala správne.

Hodnoty z uvedenej tabuľky 6.1 sú podrobnejšie rozoberané v nasledujúcej časti. Osobitne pre jednotlivé jazyky spolu s detailnejšími záznamami o nálezoch. Nález obsahuje koncept, vlastnosť a vzor, ktorý bol identifikovaný analýzou. Položka O - odstránený predstavuje to, či bol vzor predtým v modeli a bol pred analýzou odstránený. Položka N - navrhnutý, predstavuje to, či bol vzor analýzou navrhnutý. Položka R - relevantnosť, reprezentuje to, či bol vzor relevantný, teda či ho analýza mala alebo nemala navrhnúť. Hodnoty týchto položiek sú buď symbol ✓ak nález spĺňa danú vlastnosť alebo prázdne políčko, ak nespĺňa danú vlastnosť.

6.1 JSON

Ako prvý bol analyzovaný model jazyka JSON. Nálezy, tohto jazyka sú uvedené v tabuľke 6.2.

Analýza pre jazyk JSON mala vysokú hodnotu presnosti aj výťažnosti - 0,7 6.1. Ako vidno v tabuľke 6.2, jazyk JSON obsahoval veľa výskytov vzorov *brackets*, *separator* a *identifier*. Identifikácia týchto vzorov je relatívne priamočiará.

Návrhy vzorov *identifier* a návrh vzoru *quoted string* v koncepte *Members* a *NodeString* spolu súvisia. Kým nález vzoru *identifier* je falošne pozitívny, nález *quoted string* je správny. Pri oboch vzoroch sa analýza zameriava na vlastnosti typu reťa-

Koncept	Vlastnosť	Vzor	O	N	R
Array	node	brackets	✓	✓	✓
Array	-	brackets	✓		✓
Array	node	separator	✓	✓	✓
Members	string	quoted string	✓	✓	✓
Members	value	concept decorator	✓		✓
NodeString	retazec	quoted string	✓	✓	✓
Object	men	brackets	✓	✓	✓
Object	-	brackets	✓		✓
Object	men	separator	✓	✓	✓
-	-	comment	✓	✓	✓
Members	string	identifier		✓	
NodeString	retazec	identifier		✓	
Boolean	bo	flag		✓	

Tabuľka 6.2: Nálezy vzorov v jazyku JSON

zec. To ukazuje, že analýza nedokáže správne odlíšiť, kedy je vhodné aký reťazec v jazyku implementovať. To je v súlade s opisom analýzy týchto dvoch vzorov v predošlej časti práce.

Pri konceptoch *Array* a *Object* nebol vzor *brackets* navrhnutý v prípadoch, keď išlo o prázdne pole alebo objekt. Notácia týchto konceptov bola v týchto dvoch prípadoch nasledovná.

Zdrojový kód 6.1: Notácia konceptov *Array* a *Object*

```
@Before("ZACPOLE")
@After("KONPOLE")
public Array() {

}

@Before("ZACOBJ")
@After("KONOBJ")
public Object() {

}
```

Vzor *brackets* sa zameriava na vlastnosti, ktoré sú kolekcie. Triedy *Array* a *Object* obsahujú konštruktory s kolekciou ale aj prázdne konštruktory. Teda sú defi-

nované notácie pre prázdne aj neprázdne pole a objekty. Analýza našla vlastnosti typu kolekcie a navrhla vzor *brackets*. Následne sa ale nepozerala na to, či v triede neexistujú aj konštruktory, ktoré vyjadrujú prázdnu kolekciu. Preto neponúkla vzor *brackets* v prípade prázneho poľa a objektu.

v koncepte *Members* nebol vzor *concept decorator* návrhnutý, pretože analýza tohto vzoru sa zameriava na vzory, ktoré majú rovnakú štruktúru. V tomto prípade má dekorátor v podobe symbolu dvojbodky význam pre zlepšenie čitateľnosti a prehľadnosti. Definícia konceptu *Members* je nasledovná.

Zdrojový kód 6.2: Notácia konceptu *Members*

```
public Members(@Token("VALUE") String string, @Before("
   DVOJBODKA") Node value) {
 ...
}
```

Identifikovaný vzor *concept decorator* sa tu vyskytuje v jednej zo svojich podôb, ako vzor *keyword*. Ako bolo spomínané v opise riešenia, vzor kľúčového slova je pre svoju všeobecnosť ťažko identifikovateľný. Preto ani v tomto prípade nebol v koncepte *Members* vzor *concept decorator* navrhnutý.

6.2 StateMachine

Hodnoty nálezov vzorov v jazyku StateMachine sú uvedené v tabuľke 6.3. Analýza v tomto jazyku dosiahla presnosť 0,67 6.1 a výťažnosť 0,6 6.1.

Uvedená je notácia konceptu *State*. Inštancia stavu sa začína slovom *state* a je ukončená bodkočiarkou. Podobne aj koncept *Trasition*, ktorý začína slovom *transition*. Tento prípad vzoru *concept decorator* nie je jednoduché odhaliť, keďže pre neho v týchto konceptoch nie sú žiadne indície. Ako bolo spomenuté pri jazyku JSON, vzor *concept decorator* sa navrhuje v prípade, ak je nájdená zhoda medzi štruktúrou dvoch konceptov. V tomto prípade ide skôr o sprehľadnenie notácie *State* a *Transition*. Toto použitie vzoru *concept decorator* nie je pomocou definovaných podmienok možné odhaliť. Závisí skôr na preferencii autora ako na modeli jazyka.

Zdrojový kód 6.3: Notácia konceptu *State*

```
@Before("state") @After(";") @Token("ID")
public State( String id) {
  ...
}
```

Koncept	Vlastnosť	Vzor	O	N	R
State	-	concept decorator	✓		✓
State	id	identifier	✓	✓	✓
Transition	-	concept decorator	✓		✓
Transition	label	identifier	✓	✓	✓
Transition	sourceLabel	identifier	✓		✓
Transition	sourceLabel	concept decorator	✓	✓	✓
Transition	targetLabel	identifier	✓		✓
Transition	targetLabel	concept decorator	✓	✓	✓
-	-	number		✓	
StateMachine	declarations	brackets		✓	✓
Transition	label	quoted string		✓	
Transition	-	sequence	✓	✓	✓
Transition	-	unordered group		✓	

Tabuľka 6.3: Nálezy vzorov v jazyku StateMachine

Nájdený vzor *sequence* pri koncepte *Transition* smeruje ku vzoru *keyword*, ktorý je odvodený od vzoru *concept decorator*. Návrh vzoru *sequence* je relevantný, keďže v modeli sú naozaj použité kľúčové slová na označenie poradia prvkov *Transition*. Pri koncepte je ponúknutý aj vzor *unordered group*. Analýza nevie autonómne rozhodnúť, či na poradí vlastností konceptu v notácii záleží alebo nie. Preto sú navrhnuté oba vzory *sequence* aj *unordered group*.

Zdrojový kód 6.4: Notácia konceptu *Transition*

```
@Before("trans")
@After(";")
public Transition(
  @Token("ID") String label,
  @Before(":") @Token("ID")
  References(value = State.class, field = "source") String
     sourceLabel,
  @Before("->") @Token("ID")
  @References(value = State.class, field = "target") String
     targetLabel) {

}
```

Vzor *number* bol navrhnutý ale v skutočnosti nie je v jazyku potrebný. Ďalej

v koncepte *StateMachine* je ponúknutý vzor *brackets*. *StateMachine* je hlavným konceptom jazyka. Ak by bol uzavretý v zátvorkách, znamenalo by to, že celý kód v jazyka StateMachine je uzavretý v nejakých symboloch. Ponúknuť tento vzor v tomto prípade je nerelevantné. Autor jazyka nechcel, aby sa všetky deklarácie museli uzavrieť medzi zátvorky či nejaké kľúčové slová. Pokiaľ by podmienka v analýze kontrolovala, či koncept nie je hlavným konceptom a vylúčila by takéto koncepty zo vzoru *brackets*, nemuselo by to byť vždy správne. Napríklad v prípade konfiguračných jazykov ako JSON alebo pri jazyku LISP je žiadané uzatváranie celého programu do zátvoriek.

Riešenie by mohlo spočívať v implementovaní podmienky analýzy tak, aby sa vzor *brackets* neponúkal, ak je koncept hlavným konceptom a zároveň sa v jazyku nevyskytuje rekurzívne.

6.3 DeskNielsen

Výsledky analýzy pre jazyk DeskNielsen sú uvedené v tabuľke 6.4. Na základe nízkej hodnoty výťažnosti - 0,4 6.1 možno konštantovať, že bolo identifikovaných málo vzorov spomedzi tých, ktoré mali byť navrhnuté. Avšak tie, ktoré boli navrhnuté boli relevantné. To ukazuje vysoká hodnota presnosti - 0,75 6.1.

Fakt, že bolo identifikovaných málo vzorov môže súvisieť s vysokým výskytom vzoru *concept decorator*, ktorý je vo všeobecnosti ťažko identifikovateľný. Častokrát závisí len na vôli autora, ktorý sa ním rozhodne označiť niektorý prvok.

K vysokej hodnote presnosti prispela identifikácia vzoru *comment*. Tento vzor sa vyhľadáva pomocou porovnávania regulárnych výrazov. Analýza lexikálnych vzorov je najjednoduchšia. Lexikálny vzor *number*, hoci je ľahko identifikovateľný, nemusí byť stále relevantný. Naopak vzor *comment* je relevantný pravdepodobne v každom jazyku.

6.4 MathExpression

Jazyk MathExpression je veľmi obmedzený. Možno ho dokonca nazvať doménovo špecifickým, keďže sa zaoberá len matematickými výrazmi. Z toho vyplýva aj malá rozmanitosť vzorov, ktoré sú v ňom prítomné. Prevahu má vzor *infix operator*.

Binárne matematické operácie obsahujú vo svojom zápise symbol operátora. Tento operátor je v notácii reprezentovaný vzorom *concept decorator*.

Koncept	Vlastnosť	Vzor	O	N	R
Add	-	infix operator	✓	✓	✓
Constant	number	condept decorator	✓		✓
Program	expression	condept decorator	✓		✓
Program	constants	condept decorator	✓		✓
Program	constants	separator	✓		✓
-	-	number	✓	✓	✓
-	-	comment		✓	✓
Program	constants	brackets		✓	

Tabuľka 6.4: Nálezy vzorov v jazyku DeskNielsen

Koncept	Vlastnosť	Vzor	O	N	R
Add	-	infix operator	✓	✓	✓
Div	-	infix operator	✓	✓	✓
Mul	-	infix operator	✓	✓	✓
Sub	-	infix operator	✓	✓	✓
-	-	number	✓	✓	✓
-	-	comment	✓	✓	✓
Sub	-	concept decorator	✓	✓	✓
Mul	-	concept decorator	✓	✓	✓
Add	-	concept decorator	✓	✓	✓
Div	-	concept decorator	✓	✓	✓

Tabuľka 6.5: Nálezy vzorov v jazyku MathExpression

Výstup z analýzy je uvedenýv tabuľke 6.5. Presnosť a výťažnosť majú hodnotu 1 6.1. Vzor *infix operator* je pomerne priamočiary, presnosť jeho identifikácie je veľmi vysoká. Zároveň nie je potrebné hlbšie rekurzívne analyzovanie typov operandov, keďže všetky operácie dedia od triedy *Expression* a všetky operandy sú tiež typu *Expression*. Nízka rozmanitosť vzorov, ktoré sú v tomto jazyku je externou hrozbou validity testu.

6.5 simpleRobot

Výsledky analýzy pre jazyk simpleRobot sú uvedené v tabuľke 6.6. Hodnoty presnosti a výťažnosti sú relatívne nízke - 0,5 6.1. Opäť to možno odvôvodniť častým výskytom vzoru *concept decorator*. Ten nebol ani v jednom prípade identifikovaný

správne. Oba koncepty, kde ho bolo potrebné identifikovať obsahovali prázdnu notáciu. V takom prípade sa nehľadal v jazyku koncept s podobnou štruktúrou a preto nebol vzor *concept decorator* ponúknutý.

Zdrojový kód 6.5: Notácia konceptov *Move* a *TurnLeft*

```
@Before("move")
public Move() {

}

@Before("turn-left")
public TurnLeft() {

}
```

Vzor *separator* je odporúčený v koncepte *Robot*. Notácia konceptu *Robot* ho však neobsahuje.

Zdrojový kód 6.6: Notácia konceptu *Robot*

```
@Before("begin")
@After("end")
public Robot(List<Command> commands) {

}
```

Koncept *Command* je definovaný ako rozhranie, ktoré implementujú jednoslovné koncepty *Move* a *TurnLeft*. Koncept *Robot* obsahuje zoznam položiek typu *Command*. Bol v ňom navrhnutý vzor *separator*. Počas analýzy sú jednoslovné koncepty vylúčené zo vzoru *separator*. Avšak keďže vyhodnotenie prebiehalo tak, že boli dostránené všetky vzory, odstránili sa aj anotácie *@Before* z konceptov *Move* a *TurnLeft*. To spôsobilo, že tieto koncepty už neboli jednoslovné.

Návrh vzoru *number* nie je relevantný vzhľadom na to, že jazyk neobsahuje žiadne výrazy, kde by bolo možné použiť číslo.

6.6 Ohrozenia validity

Validitu vyhodnotenie ovplyvňujú faktory ako výber jazykov, miera rozmanitosti vzorov v jazykoch a tiež samotná metóda, ako je vykonané vyhodnotenie.

Ako najväčšiu internú hrozbu validity možno označiť podmienky, za akých prebehla analýza. Z jazyka sa pred spustením analýzy odstránili všetky vzory.

Koncept	**Vlastnosť**	**Vzor**	**O**	**N**	**R**
Move	-	concept decorator	✓		✓
Robot	commands	brackets	✓	✓	✓
TurnLeft	-	concept decorator	✓		✓
-	-	comment		✓	✓
-	-	number		✓	
Robot	commands	separator		✓	

Tabuľka 6.6: Nálezy vzorov v jazyku simpleRobot

To však nemusí byť v súlade s reaálnym použitím nástroja Pattern. Autor jazyka, ktorý by sa snažil automatizovaným spôsobom získavať návrhy vzorov, by jazyk priebežne upravoval a dopĺňal. Model jazyka by sa v iteráciách menil. Obsahoval by nejaké vzory a konkrétnu syntax by mal implementovanú do väčšej miery. To by poskytlo analýze viac zdrojov informácií.

Externými ohrozeniami validity sú vzory, ktoré sa v jazykoch vyskytovali. Nahromadenie vzoru, ktorý sa veľmi ťažko identifikuje ovplyvňuje presnosť aj výťažnosť. Pri veľmi všeobecných vzoroch je analýza málo presná, keďže ich navrhuje aj tam, kde nie sú potrebné. Pri lexikálnych vzoroch naopak môže byť presnosť veľmi vysoká, keďže sa hľadajú najjednoduchšie a okrem vzoru *number* sú takmer vždy relevantné.

7 Záver

Práca ukázala, že je možné odporučiť jazykové vzory na základe analýzy modelu jazyka. Z implementácie analýzy modelu vyplynulo, že najdôležitejším zdrojom informácií pre návrh vzorov sú typové informácie. Mnoho jazykových vzorov súvisí s typom vlastností konceptu. Ako príklad možno uviesť vlastnosť konceptu typu *boolean* ako indikáciu vzoru *flag*. Podobne typ reťazec pri vzoroch *quoted string* alebo *identifier*. Pri vzoroch *separator* alebo *brackets* sú dôležité vlastnosti konceptu, ktoré sú kolekcie.

Model jazyka obsahuje informácie, na základej ktorých je možné automatizovaným spôsobom ponúknuť použitie vzoru. Avšak v niektorých prípadoch analýza nie je schopná rozhodnúť o relevantnosti odporúčania. Informácie dostupné z modelu jazyka nie sú postačujúce na to, aby analýza mohla rozhodnúť o použití niektorých vzorov. V takom prípade je potrebné získať vstup od používateľa, ktorý dodá analýze doplňujúce informácie. Napríklad pri vzoroch *unordered-group* a *sequence* si analýza žiada informáciu o tom, či na poradí vlastností konceptov záleží.

V práci je predstavený nástroj Pattern ako proptotyp nástroja automatizovanej podpory vzorov pri návrhu jazyka. Tento nástroj analyzuje model jazyka definovaného v nástroji YAJCo. Analýza je založená na implementácii heuristík, ktoré sa týkajú konceptov jazyka, ich štruktúry a typov a štruktúry vlastností konceptov. Na základe výsledkov analýzy sú generované odporúčania aplikovania vzorov do jazyka.

Nástroj Pattern analyzoval niekoľko vybraných projektov jazykov definovaných v YAJCo. V týchto jazykoch boli manuálne identifikované jazykové vzory. Následne boli tieto vzory odstránené a jazyk bol nástrojom Pattern analyzovaný. Analýza bola schopná navrhnúť chýbajúce vzory, avšak v niektorých prípadoch vykazovala falošné pozitíva. Bolo tomu tak hlavne v prípadoch, keď prichádzali do úvahy dva rôzne súvisiace vzory. Napríklad vzory *quoted string* a *identifier*. V prípade týchto vzorov nie je možné na základe definovaných podmienok zistiť, či nájdená vlastnosť má byť reťazcom alebo identifikátorom.

Pattern implementuje okrem samotných odporúčaní aj interaktívnu analýzy a implementáciu vzorov. Možno povedať, že automatizovaný návrh vzorov je možný len do istej miery a v niektorých prípadoch potrebuje podporu autora. Toto zistenie je v súlade s tvrdením autorov v článku, ktorý opisuje jazykové vzory [8]. V tomto článku autori predostreli myšlienku automatizovenej podpory vzorov a hovoria, že by si takáto podpora žiadala doplňujúce informácie od autora jazyka.

Vzor *concept decorator* a jeho špecifické prípady ako *keyword* sa ukázali ako najťažšie identifikovateľné. Dôvodom je ich všeobecnosť. Stanovené heurisiky sa zamerali na porovnanie štruktúry konceptov. Pokiaľ sa zhodovala, bolo navhrnuté označenie konceptu dekorátorom. Avšak v niektorých prípadoch boli koncepty označované dekotrátormi aj z iných dôvodov. Nie vždy to súvislo s podobou modelu a niekedy išlo len o vôľu autora, ktorý sa rozhodol kľúčovým slovom alebo dekorátorom zlepšiť čitateľnosť alebo porozumenie jazyka. Takéto prípady analýza nedokázala vždy odhaliť.

Naopak najjednoduchšími z pohľadu analýzy sa javia lexikálne vzory. Pri nich je dôležité sledovať prítomnosť alebo neprítomnosť konkrétnej lexikálnej jednotky v lexikálnej gramatike jazyka. Pri týchto vzoroch bola analýza najpresnejšia.

Implementácia vzorov v modeli jazyka vytváranom v nástroji YAJCo predstavuje možnosť pre ďalší vývoj. Niektoré vzory majú v YAJCo priamu podporu pomocou anotácií. Ide napríklad o vzory *unordered group*, ktorého podpora je zabezpečená anotáciou *@UnorderedParameters* alebo vzor *infix operator* s anotáciou *@Operator*. Podpora niektorých vzorov je zabezpečená pomocou všeobecnejšie použiteľných anotácii ako *@Before* alebo *@After*. Tie možno využiť v rozličných vzoroch, napríklad vo vzore *brackets* ale aj vzore *keyword* alebo *concept decorator*. Podpora vzoru *mixed repetition* aktuálne v YAJCo neexistuje. Práca opisuje, prečo by bolo vhodné ju zaviesť a čo by obnášalo implementovanie tohto vzoru v jazyku bez podpory v YAJCo. Prezentovaná je nová anotácia *@MixedRepetition* pre podporu tohto vzoru, avšak bez jej implementácie.

Práca v analytickej časti opisuje aj formy podpory tvorby jazyka v jednotlivých fázach jeho vývoja. Podpora tvorby jazyka vo fáze návrhu jeho konkrétnej syntaxe má slabé zastúpenie medzi nástrojmi v rámci celého systému podpory tvorby jazykov. Jazykové vzory a ich automatizovaná alebo poloautomatizovaná podpora môžu zaujať chýbajúce miesto v rámci procesu návrhu a tvorby programovacích jazykov.

Spôsob, akým sa hľadajú možnosti odporúčaní vzorov v jazyku ponúka priestor pre ďalší vývoj. V rámci tejto práce sú implementované heuristiky analýzy

modelu. Iné spôsoby analýzy, ako napríklad využitie strojového učenia pre identifikovanie vzoru, sú stále nepreskúmané.

Literatúra

1. RAPHAEL, Bertram. The structure of programming languages. *Communications of the ACM*. 1966, roč. 9, č. 2, s. 67–71.
2. HERRIOT, Robert G. Towards the ideal programming language. In: *Proceedings of an ACM conference on Language design for reliable software*. 1977, s. 56–62. Dostupné tiež z: `https://dl.acm.org/doi/pdf/10.1145/800022.808311?casa_token=XbHsCDeFgXEAAAAA:uIIQI_semAjLWG2BwOzpJHqRt_4AswNLaW0ulhNkubhVerY1XQy931-hWBzCphwplnDpaMnlrZgKOg`.
3. TAHA, Walid. Domain-specific languages. In: *Proc. Intl Conf. Computer Engineering and Systems (ICCES)*. 2008. Dostupné tiež z: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.206.3969&rep=rep1&type=pdf`.
4. APPLETON, Brad. Patterns and software: Essential concepts and terminology. *Object Magazine Online*. 1997, roč. 3, č. 5, s. 20–25.
5. WELSH, Noel. The Death of Patterns. 2003.
6. MIKKONEN, Tommi. Formalizing design patterns. In: *Proceedings of the 20th international conference on Software engineering*. 1998, s. 115–124.
7. SPINELLIS, Diomidis. Notable design patterns for domain-specific languages. *Journal of systems and software*. 2001, roč. 56, č. 1, s. 91–99.
8. CHODAREV, Sergej; PORUBÄN, Jaroslav; JUHÁR, Ján; SULÍR, Matúš; BAČÍKOVÁ, Michaela. *Language syntax design patterns*. 2021. unpublished.
9. MERNIK, Marjan; HEERING, Jan; SLOANE, Anthony M. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*. 2005, roč. 37, č. 4, s. 316–344.
10. KARSAI, Gabor; KRAHN, Holger; PINKERNELL, Claas; RUMPE, Bernhard; SCHINDLER, Martin; VÖLKEL, Steven. Design guidelines for domain specific languages. *arXiv preprint arXiv:1409.2378*. 2014. Dostupné tiež z: `https://arxiv.org/pdf/1409.2378.pdf`.

11. FONDEMENT, Frédéric; BAAR, Thomas. Making metamodels aware of concrete syntax. In: *European Conference on Model Driven Architecture-Foundations and Applications*. 2005, s. 190–204.

12. KLEPPE, Anneke. A language description is more than a metamodel. In: *Fourth international workshop on software language engineering*. 2007, zv. 1, s. 1–4.

13. KRAHN, Holger; RUMPE, Bernhard; VÖLKEL, Steven. Integrated definition of abstract and concrete syntax for textual languages. In: *International Conference on Model Driven Engineering Languages and Systems*. 2007, s. 286–300.

14. HOARE, Charles A. *Hints on programming language design*. 1973. Dostupné tiež z: `https://apps.dtic.mil/sti/pdfs/AD0773391.pdf`. Technická správa. STANFORD UNIV CA DEPT OF COMPUTER SCIENCE.

15. HUDAK, Paul; FASEL, Joseph H. A gentle introduction to Haskell. *ACM Sigplan Notices*. 1992, roč. 27, č. 5, s. 1–52.

16. VAN ROSSUM, Guido; DRAKE JR, Fred L. The python language reference. *Python software foundation*. 2014.

17. WILE, David. Lessons learned from real DSL experiments. *Science of Computer Programming*. 2004, roč. 51, č. 3, s. 265–290.

18. QIU, Dong; LI, Bixin; BARR, Earl T.; SU, Zhendong. Understanding the syntactic rule usage in java. *Journal of Systems and Software*. 2017, roč. 123, s. 160–172. ISSN 0164-1212. Dostupné z DOI: `https://doi.org/10.1016/j.jss.2016.10.017`.

19. SAABITH, AL Sayeth; FAREEZ, MMM; VINOTHRAJ, T. Python current trend applications-an overview. *International Journal of Advance Engineering and Research Development*. 2019, roč. 6, č. 10.

20. HERMANS, Felienne; ALDEWERELD, Marlies. Programming is writing is programming. In: *Companion to the first International Conference on the Art, Science and Engineering of Programming*. 2017, s. 1–8.

21. ZAYTSEV, Vadim. Language design with intent. In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2017, s. 45–52. Dostupné tiež z: `https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8101245&casa_token=oOKCXNToG8kAAAAA:mJ4Nfz-7vRILAxvz-_yR8K6UeUl-VZv3gIbz198A3hu92EXgv_TWVJujupw1ykGF4CeNWkXu9qZBCg&tag=1`.

22. MORALES DÍAZ, Leonel Vinicio. Programming languages as user interfaces. In: *Proceedings of the 3rd Mexican Workshop on Human Computer Interaction*. 2010, s. 68–76.

23. HEIDENREICH, Florian; JOHANNES, Jendrik; KAROL, Sven; SEIFERT, Mirko; WENDE, Christian. Derivation and refinement of textual syntax for models. In: *European Conference on Model Driven Architecture-Foundations and Applications*. 2009, s. 114–129.

24. MCIVER, Linda; CONWAY, Damian. Seven deadly sins of introductory programming language design. In: *Proceedings 1996 International Conference Software Engineering: Education and Practice*. 1996, s. 309–316. Dostupné tiež z: `https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=534015&casa_token=vus0qxbH_HQAAAAA:LNtkRswFMsNJixiTw-kvYw4tErr2smVpzDBYCIl47QplpRzKLd4-6uWYbM6_yxSyuTCshyywrEGITw&tag=1`.

25. PORUBÄN, Jaroslav; FORGÁČ, Michal; SABO, Miroslav. Annotation based parser generator. In: *2009 International Multiconference on Computer Science and Information Technology*. 2009, s. 707–714.

26. MULLER, Pierre-Alain; FONDEMENT, Frédéric; FLEUREY, Franck; HASSENFORDER, Michel; SCHNEKENBURGER, Rémi; GÉRARD, Sébastien; JÉZÉQUEL, Jean-Marc. Model-driven analysis and synthesis of textual concrete syntax. *Software & Systems Modeling*. 2008, roč. 7, č. 4, s. 423–441.

27. QUESADA, Luis; BERZAL, Fernando; CUBERO, Juan-Carlos. A domain-specific language for abstract syntax model to concrete syntax model mappings. In: *2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 2014, s. 158–165.

28. FONDEMENT, Frédéric; SCHNEKENBURGER, Rémi; GÉRARD, Sébastien; MULLER, Pierre-Alain. *Metamodel-aware textual concrete syntax specification*. 2006. Technická správa.

29. KRISHNAMURTHI, Shriram. *Linguistic reuse*. Rice University, 2001.

30. KRAHN, Holger; RUMPE, Bernhard; VÖLKEL, Steven. MontiCore: a framework for compositional development of domain specific languages. *International journal on software tools for technology transfer*. 2010, roč. 12, č. 5, s. 353–372.

31. MERNIK, Marjan; ŽUMER, Viljem. Incremental programming language development. *Computer Languages, Systems & Structures*. 2005, roč. 31, č. 1, s. 1–16.

32. VACCHI, Edoardo; CAZZOLA, Walter; PILLAY, Suresh; COMBEMALE, Benoît. Variability support in domain-specific language development. In: *International Conference on Software Language Engineering*. 2013, s. 76–95.

33. MANDRIOLI, Dino; PRADELLA, Matteo. Programming Languages Shouldn't Be "Too Natural". *SIGSOFT Softw. Eng. Notes*. 2015, roč. 40, č. 1, s. 1–4. ISSN 0163-5948. Dostupné z DOI: 10.1145/2693208.2693232.

34. WILE, David S. Transforming Concrete Syntax to Abstract Syntax.

35. MERNIK, Marjan; KORBAR, Nikolaj; ŽUMER, Viljem. LISA: A tool for automatic language implementation. *ACM Sigplan Notices*. 1995, roč. 30, č. 4, s. 71–79.

36. QUESADA, Luis; BERZAL, Fernando; CUBERO, Juan-Carlos. A DSL for Mapping Abstract Syntax Models to Concrete Syntax Models in ModelCC. *arXiv preprint arXiv:1301.4858*. 2013.

37. RÁTH, István; ÖKRÖS, András; VARRÓ, Dániel. Synchronization of abstract and concrete syntax in domain-specific modeling languages. *Software & Systems Modeling*. 2010, roč. 9, č. 4, s. 453–471.

38. VÖLTER, Markus; VISSER, Eelco. Language extension and composition with language workbenches. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. 2010, s. 301–304.

39. PARR, Terence. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

40. BRAND, Mark GJ van den et al. The ASF+ SDF meta-environment: A component-based language development environment. *Electronic Notes in Theoretical Computer Science*. 2001, roč. 44, č. 2, s. 3–8.

41. ANLAUFF, Matthias; KUTTER, Philipp W; PIERANTONIO, Alfonso. Montages/Gem-Mex: a meta visual programming generator. In: *Proceedings. 1998 IEEE Symposium on Visual Languages (Cat. No. 98TB100254)*. 1998, s. 304–305.

42. BATORY, Don; LOFASO, Bernie; SMARAGDAKIS, Yannis. JTS: Tools for implementing domain-specific languages. In: *Proceedings. Fifth International Conference on Software Reuse (Cat. No. 98TB100203)*. 1998, s. 143–153.

43. FAITH, Rickard E; NYLAND, Lars S; PRINS, Jan F. KHEPERA: A System for Rapid Implementation of Domain Specific Languages. In: *DSL*. 1997, zv. 97, s. 19–19.

44. ATTALI, Isabelle; COURBIS, Carine; DEGENNE, Pascal; FAU, Alexandre; FILLON, Joël; PARIGOT, Didier; PASQUIER, Claude; COEN, Claudio Sacerdoti. SmartTools: a development environment generator based on XML technologies. *XML Technologies and Software Engineering*. 2001.

45. KIENLE, Holger M; MOORE, David L. smgn: Rapid prototyping of small domain-specific languages. *Journal of computing and information technology*. 2002, roč. 10, č. 1, s. 37–53.

46. AYCOCK, John. The design and implementation of SPARK, a toolkit for implementing domain-specific languages. *Journal of computing and information technology*. 2002, roč. 10, č. 1, s. 55–66.

47. CORDY, James R. TXL-a language for programming language tools and applications. *Electronic notes in theoretical computer science*. 2004, roč. 110, s. 3–31.

48. CHODAREV, Sergej; BAČÍKOVÁ, Michaela. Development of Oberon-0 using YAJCo. In: *2017 IEEE 14th International Scientific Conference on Informatics*. 2017, s. 122–127.

49. FECENKO, Bc. Michal. *Rozšírenie generátora jazykových procesorov YAJCo*. Košice, 2019. Diplomová práca. Technická univerzita v Košiciach. An optional note.

50. CHODAREV, Sergej; BAČÍKOVÁ, Michaela. Syntax-Driven Development of Oberon-0 Using YAJCo. *Journal of Information and Organizational Sciences*. 2019, roč. 43, č. 2, s. 145–162.

Zoznam skratiek

BNF Backus-Naur form.

DSL Domain Specific Language.

EBNF Extended Backus-Naur form.

IDE Integrated Development Environment.

Slovník

Property je neterminálny symbol, ktorý je obsiahnutý v koncepte jazyka..

Syntactic Sugar je výraz označujúci doplnenie syntaxe počítačového jazyka, ktoré uľahčuje vyjadrovanie ale nepridáva žiadnu novú funkcionalitu..

Zoznam príloh