VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ROZŠÍŘENÍ NÁSTROJE PRO VIZUÁLNÍ PROGRAMOVÁNÍ V LUA/LÖVE O GENEROVÁNÍ BLOKŮ Z TEXTOVÉHO KÓDU

AN EXTENSION OF A VISUAL PROGRAMMING EDITOR FOR LUA/LÖVE WITH ABILITY TO GENERATE BLOCKS FROM TEXT CODE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. RENÉ REŠETÁR**
AUTHOR

VEDOUCÍ PRÁCE — **prof. RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

154279

Ústav:	Ústav informačních systémů (UIFS)
Student:	**Rešetár René, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Strojové učení
Název:	**Rozšíření nástroje pro vizuální programování v Lua/LÖVE o generování bloků z textového kódu**
Kategorie:	Softwarové inženýrství
Akademický rok:	2023/24

Zadání:

1. Seznamte se s programovacím jazykem Lua a s rámcem LÖVE a s nástrojem pro tvorbu aplikací s tímto rámcem pomocí vizuálních programových bloků. Prozkoumejte způsob, jakým nástroj generuje z bloků textový kód pomocí knihovny Blockly.
2. Seznamte se s podobnými nástroji pro jiné programovací jazyky, které umožňují i zpětné generování vizuálních bloků z textového kódu. Prozkoumejte možnosti detekce vzorů ve struktuře programu.
3. Navrhněte způsob, jakým v programovém kódu v jazyce Lua s rámcem LÖVE rozpoznat dané struktury kódu a generovat jim odpovídající bloky vizuálního programovacího jazyka. Popište části kódu, ze kterých vizuální bloky vygenerovat nelze. Návrh konzultujte s vedoucím.
4. Implementujte rozšíření stávajícího nástroje pro vizuální blokové programování v Lua/LÖVE o generování vizuálních bloků z kódu Lua/LÖVE, případně implementujte nástroj nový s původními funkcemi i výše uvedeným rozšířením. Implementujte také testy, které zkontrolují správnost postupu pomocí dopředného a zpětného generování kódu a odpovídajících bloků s náhodnými změnami.
5. Řešení otestujte, vyhodnoťte a diskutujte výsledky. Výsledný software publikujte jako open-source.

Literatura:

- S. Kenlon. Modular Programming with LÖVE. In: Developing Games on the Raspberry Pi. Apress, Berkeley, CA, 2019. ISBN 978-1-4842-4170-7. Dostupné z: [https://doi.org/10.1007/978-1-4842-4170-7_3]
- E. Pasternak, R. Fenichel a A. N. Marshall. Tips for creating a block language with blockly. In: 2017 IEEE Blocks and Beyond Workshop. IEEE, USA, 2017. Dostupné z: [https://doi.org/10.1109/BLOCKS.2017.8120404]
- P. Medek. Výukový software pro vizuální a textové programování v Lua/LÖVE. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

Při obhajobě semestrální části projektu je požadováno:
Body 1 až 3.

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

Vedoucí práce:	**Rychlý Marek, RNDr., Ph.D.**
Vedoucí ústavu:	Kolář Dušan, doc. Dr. Ing.
Datum zadání:	1.11.2023
Termín pro odevzdání:	17.5.2024
Datum schválení:	30.10.2023

Abstrakt

Táto práca sa zaoberá spracovaním zdrojového kódu v jazyku Lua rozšíreného o rámec LÖVE s využitím Tree-Sitter parseru. Následne spracovaním jeho výstupu a generovaním odpovedajúcej reprezentácie tohto kódu v prostredí Blockly pomocou blokov. V rámci práce vznikol nástroj pre rozšírenie existujúcej aplikácie o spätné generovanie blokov z kódu. Nástroj sa nepodarilo do existujúcej aplikácie integrovať priamo ani pomocou vytvoreného API. To by však malo túto integráciu v budúcnosti umožniť. Okrem toho vznikol aj nástroj, slúžiaci ako vývojové prostredie, ktorý by mal umožniť jednoduchšiu implementáciu ďalšej funkcionality tohto rozšírenia.

Abstract

This thesis deals with the processing of Lua source code extended with the LÖVE framework using the Tree-Sitter parser. Then processing its output and generating the corresponding representation of this code in the Blockly environment using blocks. As part of the work, a tool was developed to extend an existing application to generate blocks backwards from code. The tool could not be integrated directly into the existing application, even with the help of the developed API. However, it should allow this integration in the future. In addition, a tool has been created to serve as a development environment that should allow for easier implementation of the additional functionality of this extension.

Kľúčové slová

vizuálne programovanie, blokové programovanie, programovací jazyk Lua, rámec LÖVE, vizuálne bloky, spätná tvorba blokov, Blockly, detekcia vzorov kódu, generovanie blokov zo zdrojového kódu, interaktívne vzdelávanie, rozšírenie.

Keywords

visual programming, block programming, Lua programming language, LÖVE framework, visual blocks, reverse block generation, Blockly, code pattern detection, block generation from source code, interactive learning, extensions.

Citácia

REŠETÁR, René. *Rozšíření nástroje pro vizuální programování v Lua/LÖVE o generování bloků z textového kódu.* Brno, 2024. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. RNDr. Marek Rychlý, Ph.D.

Rozšíření nástroje pro vizuální programování v Lua/LÖVE o generování bloků z textového kódu

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána doktora Marka Rychlého. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
René Rešetár
16. mája 2024

Poďakovanie

Rád by som poďakoval vedúcemu diplomovej práce, doktorovi Marku Rychlému, za jeho trpezlivosť, cenné vedenie a podporu behom celého procesu.

Obsah

Kapitola 1

Úvod

V dnešnom rýchlo sa meniacom informatickom prostredí je dôležité neustále hľadať inovatívne prístupy k výučbe programovania a vytvárania softvéru. Vizuálne programovanie, ktoré umožňuje tvorbu kódu pomocou grafických prvkov a blokov, sa stáva stále viac atraktívnym nástrojom, najmä pre začiatočníkov a mladších programátorov. Táto diplomová práca sa zameriava na prieskum vizuálnych programovacích nástrojov a ich využitia v kontexte generovania blokovej reprezentácie zdrojového kódu.

Cieľom tejto diplomovej práce je poskytnúť ucelený pohľad na vizuálne programovanie v kontexte Lua a LÖVE, prieskum podobných nástrojov a návrh metód na efektívne generovanie vizuálnych blokov. Následne tieto metódy implementovať a použiť ako rozšírenie pre existujúci nástroj. Týmto spôsobom sa snažíme prispieť k rozvoju vzdelávania v oblasti programovania a zároveň poskytnúť nové pohľady na možnosti vizuálneho programovania v konkrétnom programovacom jazyku a hernom rámci.

Prvá časť práce 2 sa bude venovať úvodu do programovacieho jazyka Lua a jeho väzby s rámcom LÖVE. Taktiež sa zameriame na nástroje pre tvorbu aplikácií s vizuálnymi blokmi, ktoré umožňujú tvorbu kódu prostredníctvom intuitívneho a vizuálneho prístupu. Potom zanalyzujeme proces generovania textového kódu z vizuálnych blokov, jeho súvislosti s knižnicou Blockly a možnosti spätného prevodu z kódu do blokov v tomto prostredí. Ďalej sa pozrieme na podobné existujúce nástroje, ktoré umožňujú nielen programovanie pomocou blokov, ale najmä opačné generovanie vizuálnych blokov z existujúceho zdrojového kódu. V poslednej časti si krátko predstavíme rozširovaný nástroj.

V druhej časti 3 sa venujeme návrhu spôsobu rozpoznávania špecifických štruktúr kódu v jazyku Lua s rámcom LÖVE. Preskúmame možnosti detekcie vzorov v štruktúre programu pomocou nástrojov pre syntaktickú analýzu kódu. Tieto nástroje, si predstavíme a identifikujeme ich spoločné rysy. Nakoniec si odôvodníme výber parsovacieho nástroja pre túto prácu.

Pokračujeme s návrhom nástroja 4, ktorý bude spracovávať výstup vybraného parsovacieho nástroja. Špecifikujeme si jeho funkčné požiadavky a bližšie sa pozrieme na jeho vstup. Podľa tohto vstupu si navrhneme štruktúru objektov, ktoré s ním budú pracovať a generovať odpovedajúcu blokovú reprezantáciu vo vizuálnom programovacom jazyku. Zároveň analyzujeme časti kódu, ktoré môžu byť výzvou pre vizuálnu reprezentáciu a konzultujeme návrh s vedúcim práce. V poslednej kapitole tejto časti navrhneme spôsob pre napojenie nástroja do rozširovanej aplikácie.

Následne si popíšeme implementáciu vyvíjaného nástroja 5. V prvej časti popíšeme prostredie vyvinuté pre implementáciu a testovanie tohto nástroja. V nasledujúcej časti si hlbšie predstavíme spracovávaný abstraktný syntaktický strom (odteraz AST), gramatiku použitú v tejto práci a prepojenie uzlov stromu s navrhnutými objektami. Ďalej si ukážeme implementáciu generovania blokov pomocou knižnice Blockly. Nakoniec si krátko povieme o spätnom generovaní kódu z blokov a rozšíriteľnosti nástroja implementovaného v tejto práci.

V predposlednej časti 6 popíšeme postup integrácie do existujúcej aplikácie a rôzne ťažkosti, na ktoré sme v tomto kroku narazili, a ktoré nám tento krok znemožnili dokončiť. Spomenieme si aj pokus o alternatívne riešenie pomocou API serveru.

Nakoniec si ukážeme ako sme postupovali pri vytváraní testovacích setov, ktoré sme popri vyvíjaní tohto nástroja využívali k jeho testovaniu 7 a popíšeme si ako samotné testovanie prebiehalo.

Kapitola 2

Použité nástroje

V tejto kapitole sú predstavené základné pojmy, nástroje a teória potrebné pre formalizovaný popis nástroja vyvíjaného v tejto práci. Najprv je potrebné predstaviť jazyk Lua, pre ktorý je tento nástroj vyvíjaný a rámec Love2D 2.1. Následne zanalyzujeme open-source knižnicu Blockly 2.2 a pozrieme sa na pár jej alternatív. Potom sa pozrieme na existujúce nástroje 2.3, ktoré ponúkajú aj prevod z textu do blokov a ktorými sme sa v prístupe k tomuto prevodu inšpirovali v tejto práci. V poslednej časti si zľahka predstavíme webovú aplikáciu vyvinutú vrámci diplomovej práce [13], na ktorú by táto práca mala nadväzovať 2.4.

2.1 Lua a LÖVE v praxi

Lua

Lua [17] je výkonný, efektívny, ľahký a jednoducho integrovateľný skriptovací jazyk, ktorý bol vyvinutý za účelom jednoduchej implementácie a integrácie do rôznych aplikácií. Táto kapitola poskytuje stručný prehľad jazyka Lua, zdôrazňujúc jeho kľúčové vlastnosti. Lua podporuje procedurálneho programovania rovnako ako aj funkcionálne programovanie. Tie spája s výkonnými konštrukciami na opis údajov založenými na asociatívnych poliach a rozšíriteľnej sémantike. Jazyk Lua je dynamicky typovaný, beží na základe interpretácie bajtkódu s virtuálnym strojom založeným na registroch a má automatickú správu pamäte s inkrementálnym zberom odpadu. Vďaka tomu je ideálny na konfiguráciu, skriptovanie a rýchle prototypovanie.

Jeho jednoduchá syntax, dynamická typovosť a silná podpora pre výkonné skriptovanie sú ďalšie vlastnosti, ktoré tento jazyk robia vhodným kandidátom pre nováčikov v programovaní. Lua preto zohráva významnú úlohu vo vzdelávacom systéme, najmä v oblasti informatiky a programovania. Lua sa vďaka svojej jasnej syntaxi a minimalistickým vlastnostiam často používa ako výučbový jazyk v úvodných kurzoch programovania. Vďaka tomu, že je tento jazyk ľahko integrovatľný, je tiež cenným nástrojom na výučbu skriptovania v rámci rôznych aplikácií a vývoja hier[1].

[1] Medzi niektoré známejšie hry vyvíjané aj pomocou jazyka Lua patria napr. *Don´t Starve* alebo *Hades*.

Lua sa výrazne využíva v hernom priemysle a vo vložených systémoch, čo svedčí o jeho univerzálnosti a efektívnosti v oblasti rýchleho vývoja. Taktiež sa venuje otázkam výkonnosti a škálovateľnosti v kontexte väčších projektov. Jeho popularita sa čoraz väčšmi rozširuje na hlavné herné platformy. Skripty Lua sa teraz často používajú na implementáciu herných mechaník, umelej inteligencie a ďalších dynamických prvkov. Rozšírené používanie jazyka Lua v hernom priemysle dokazuje jeho všestrannosť a účinnosť pri zlepšovaní procesu vývoja a uľahčovaní vytvárania pútavých a dynamických herných zážitkov.

Framework Love2D: Rozhranie Pre Rýchly Vývoj 2D Hier v Jazyku Lua

Spoločnosti Lua a Love2D (známa aj ako LÖVE) majú v hernom priemysle blízky vzťah. LÖVE je open-source multiplatformový engine pre 2D videohry, ktorý používa programovací jazyk Lua. Jeho rozhranie API využíva knižnice SDL[2] a OpenGL[3], čo umožňuje jednoduchý prístup k obrazovým a zvukovým funkciám počítača. Je úplne bezplatný, multiplatformový a ľahko sa učí a používa. Je to ideálny nástroj pre začatie vývoja hier a precvičovanie programovania. Táto kapitola poskytne prehľad kľúčových aspektov LÖVE, jeho architektúry a výhod pre vývojárov.

Love2D [9] sa vyznačuje minimalizmom a jednoduchosťou, umožňujúc vývojárom sústrediť sa na samotný obsah hier. S jednoduchým API pre okno, grafiku, zvuky a ovládanie vstupu, umožňuje rýchle prototypovanie a vývoj. Programátori majú možnosť preskakovať komplikované konfigurácie a okamžite sa pustiť do tvorby.

Love2D je open-source projekt, ktorý podporuje aktívna a vášnivá komunita vývojárov. Komunita prispieva k rozvoju rámca, vytváraniu nových nástrojov a zdieľaniu skúseností. Táto podpora zabezpečuje aktualizácie a nové funkcie, čím udržuje LÖVE živým a relevantným pre súčasné potreby vývojárov.

Okrem vytvárania hier poskytuje LÖVE priestor pre experimentovanie s interaktívnou grafikou a multimediálnym obsahom. Vzhľadom na svoju jednoduchosť sa často používa na výučbu programovania a tvorbu rôznych interaktívnych vizualizácií.

V záverečnom hodnotení možno konštatovať, že LÖVE je nielen framework pre hry, ale aj nástroj pre kreatívne experimentovanie s interaktívnym obsahom. Jeho jednoduchosť a podpora jazyka Lua ho robia prístupným nielen pre skúsených vývojárov, ale aj pre začiatočníkov, ktorí chcú rýchlo vytvárať svoje prvé 2D vizuálne projekty.

```
function love.load()
end

function love.update(dt)
end

function love.draw()
end
```

Výpis 2.1: Tri hlavné funkcie s ktorými začína každý vývoj v Love2D

[2] https://www.libsdl.org/
[3] https://www.opengl.org/

Pri písaní hier pomocou LÖVE, hrajú najdôležitejšiu úlohu tri fundamentálne funkcie viz. 2.1 a konfiguračný súbor *conf.lua*. Tieto funkcie definujú správanie a štruktúru hry.

- *love.load()* sa volá vždy pri spustení hry a slúži primárne k inicializácií premenných, načítaniu obrázkov či zvukov a nastaveniu dobre definovaného počiatočného stavu hry.
- *love.draw()* je zodpovedná za vykresľovanie grafiky na obrazovku. Volá každý snímok po funkcii *love.update()*. Vývojári v nej definujú kreslenie obrázkov a textu.
- *love.update()* sa volá každý snímok pred funkciou *love.draw()* pre aktualizáciu stavu hry. Táto funkcia zvyčajne obsahuje kód popisujúci logiku hry, akcie závisle od času (napr. pohyb) a kolízie. Je nevyhnutná pre zabezpečenie vývinu hry v postupnom čase a spracovanie všetkých aspektov hry.
- *conf.lua* ovplyvňuje prispôsobenie a konfiguráciu hier. Tento súbor obsahuje funkciu *conf.lua* a je spustený vždy pred *main.lua*. Centralizuje definovanie parametrov týkajúcich sa vlastností okien, herných modulov, zdrojov a metadát. Vieme v ňom definovať veľkosť okna, verziu hry počas vývoja, alebo špecifikovať moduly (napr. pre zvuk alebo fyziku). To zlepšuje flexibilitu a kontrolu nad vývojom hier v Love2D a pomáha vývojárom splniť špecifické požiadavky a preferencie.

2.2 Vizuálne programovanie a framework Blockly

Vizuálne programovanie

Vizuálne programovanie je intuitívny spôsob, ako tvoriť softwarové aplikácie bez písania kódu. Je to grafická reprezentácia toho, ako spolu rôzne časti spolupracujú a tým dovoľujú vývojárom vytvárať komplexné programy rýchlejšie a jednoduchšie. Oproti tradičnému textovo založenému programovaniu, vývojárom dovoľuje vytvárať software pomocou manipulácie grafických elementov. Tieto elementy sú často reprezentované ako vizuálne bloky alebo uzly. Prepájaním a usporiadaním týchto reprezentácií sa definuje logika a tok programu.

Tento prístup je obzvlášť prospešný pre začiatočníkov, keďže namiesto písania kódu do riadkov spájame predom definované reprezentácie konceptov kódu. To redukuje počet syntaktických chýb a následne znižuje počiatočné bariéry a chaos pri vstupe do sveta programovania. Vďaka tomu sa vývojár môže sústrediť na logiku a správnu funkcionalitu výsledného programu. Táto metóda tiež uľahčuje sledovanie toku programu, umožňuje rýchlejšie prototypovanie a podporuje kreatívne myslenie pri tvorbe softvéru. Celkovo to zlepšuje skúsenosť a pochopenie procesu programovania, čím poskytuje prostredie, kde sa noví vývojári môžu komfortne učiť a zároveň skúsení vývojári môžu efektívne experimentovať a inovovať.

Ako je zmienené vyššie v tejto kapitole, existuje viacero druhov vizuálneho programovania, ako je napríklad programovanie toku dát[4], ktoré je reprezentované grafovými štruktúrami a sústredí sa na tok dát systémom, kde uzly predstavujú operácie a hrany tok dát. Ďalším typom je programovanie pomocou vývojového diagramu, kde sa na vyjadrenie toku programu využívajú rôzne tvary a šípky. V tejto práci nás však bude zaujímať vizuálne programovanie založené na blokoch.

[4]Node-RED, LabVIEW

Blokové programovanie

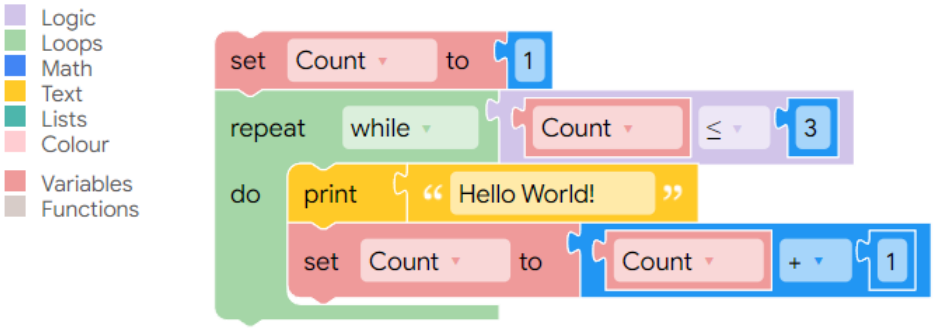
Blockly

V tejto časti diplomovej práce si predstavíme rámec Blockly [12] vyvinutý spoločnosťou Google. Ako jeho názov naznačuje, tento rámec slúži k programovaniu pomocou blokovej reprezentácie kódu.

Blockly je webová knižnica, ktorá vám umožňuje pridať do aplikácie editor kódu založený na blokoch. Editor používa bloky podobné skladačke na reprezentáciu konceptov kódu, ako sú premenné, logické výrazy, cykly a ďalšie. Blockly ponúka vizuálny prístup k programovaniu prostredníctvom kombinovania blokov reprezentujúcich programovacie koncepty, čo umožňuje tvorbu kódu bez nutnosti písania textového kódu, alebo použitia príkazového riadku. Kapitola sa detailne zameria na architektúru tohto rámcu a preskúma, ako rôzne funkcionality rámcu podporujú proces vizuálneho programovania.

Blockly umožňuje sústrediť sa na používanie blokov bez toho, aby sme sa museli starať o to, ako by sa tieto bloky mali vykresľovať, ťahať alebo spájať. Preto je možné ho použiť na širokú škálu prípadov: práca študentov s textovým programovaním, podpora výpočtového myslenia, automatizácia v robotike, konfigurácia internetových obchodov, atď.

Na obrázku 2.1 môžeme vidieť ako sú tieto bloky vizuálne reprezentované a na obrázku 2.2 vidíme zdrojový kód ktorý tieto bloky reprezentujú. Obrázok 2.3 nám ukazuje JSON definíciu zeleného bloku typu *controls_whileUntil* z predchádzajúcej ukážky.



Obr. 2.1: Ukážka reprezentácie kódu pomocou blokov v *Blockly*. Prevzaté z [5]

```
Count = 1
while Count <= 3 do
  print('Hello World!')
  Count = Count + 1
end
```

Obr. 2.2: Ukážka reprezentácie zdrojového kódu v jazyku Lua, ktorý je zobrazený na 2.1. Prevzaté z [5]

Obr. 2.3: Ukážka JSON definície bloku reprezentujúceho **while** cyklus v prostredí Blockly. Prevzaté z [6]

Okrem vyššie spomínaných predností tejto knižnice, Blockly taktiež poskytuje metódy pre prácu s blokmi priamo zo zdrojového kódu. V tejto časti si ukážeme pár týchto základných metód, ktoré budeme využívať pri implementácií.

Ak by sme napríklad chceli zadefinovať blok pre cyklus *While*, použili by sme definíciu z obrázku 2.3. Táto definícia je už implementovaná Blockly knižnicou a je možné ju použiť okamžite. Pre vytvorenie bloku v kóde, potrebujeme mať inicializovaný hlavný komponent, ktorým je *Blockly.Workspace()*. Povedzme, že máme vytvorenú jeho inštanciu v premennej *workspace*. Spomínaný blok by sme vytvorili následovne:

```
1 let whileBlock = workspace.newBlock('controls_whileUntil');
2 whileBlock.initSvg(); -- musi byt zavolane pred prvou interakciou s blokom
3 whileBlock.render();
```

Výpis 2.2: Vytvorenie a inicializácia bloku v prostredí Blockly

Metóda *newBlock()* vytvára a inicializuje nový blok. *initSvg()* inicializuje SVG (Scalable Vector Graphics) obrázok bloku a je nutné ju volať pred prvou interakciou z blokom. *render()* vykresľuje obsah bloku (texty, ikony a iné časti) do jeho SVG reprezentácie.

Vytvoreným blokom je potrebné nastaviť, alebo vyplniť ich vstupy. Tieto vstupy obsahujú políčka a môžu, ale nemusia obsahovať prepojenia. Bloky majú rôzne typy políčok na vyplnenie podľa toho, aký vstup tam očakávajú. Vypíšeme si najpoužívanejšie z nich: *field_variable*, *field_number*, *field_dropdown*, *field_textinput*,...

Pre nastavenie hodnôt týchto políčok Blockly ponúka metódu *setFieldValue(newValue, name)*. S jej využitím by sme pre **while** blok vytvorený vyššie nastavili hodnotu jeho *field_dropdown* políčka nasledovne:

```
1  whileBlock.setFieldValue("WHILE", "MODE");
```

Výpis 2.3: Pre dropdown políčko je potrebné hodnotu nastaviť na jednu z ponúkaných. V tomto prípade boli na výber WHILE/UNTIL.

Ako vidíme na príklade, prvý parameter tejto metódy je hodnota, ktorú chceme políčku nastaviť. V tomto prípade si musíme vybrať jednu z preddefinovaných možností, keďže sa jedná o políčko *field_dropdown*. Ak by sme menili napríklad *field_number* políčko, mohli by sme mu nastaviť akúkoľvek numerickú hodnotu. K políčku ktoré chceme zmeniť pristupujeme pomocou atribútu *name*, ktorý má v našom prípade hodnotu **MODE**.

V JSON definícií 2.3 *controls_whileUntil* bloku môžeme okrem *field_dropdown* políčka vidieť ešte dve vstupy: *input_value* slúžiaci na pripojenie bloku s podmienkou pre cyklus a *input_statement*, do ktorého sa napoja bloky predstavujúce telo cyklu. Pristupujeme k ním pomocou metódy *getInput(name)*, pomocou atribútu *name*. Uvažujme inicializovaný správny blok pre podmienku, uložený v premennej *conditionBlock*. Tento blok by sme do tohto vstupu napojili takto:

```
1  let connection = whileBlock.getInput("BOOL").connection;
2  connection.connect(conditionBlock.outputConnection);
```

V krátkosti sa tento kus kódu dá popísať nasledovne. *getInput()* vracia vstup reprezentovaný objektom typu *Blockly.Input*. Pomocou *connection* získame spojenie tohto vstupu, ktorým je objekt typu *Blockly.Connection*. Ten ponúka metódu *connect(connection)*, ktorá ako argument berie objekt typu *Blockly.Connection* iného bloku. Ten je získaný pomocou atribútu *outputConnection*. Je dôležité bloky vždy spájať pomocou správne získaných spojení a v správnom poradí. Teda *input* s *output* a *next* s *previous*, viď. 5.6 pre lepšie porozumenie.

Alternatívy ku Blockly

Scratch

Scratch je prostredie pre vizuálny programovanie založené na blokoch, ktorý vyvinula skupina Lifelong Kindergarten Group v MIT Media Lab. Tým, že ponúka prístupné rozhranie typu *drag-and-drop* predstavil revolúciu v kódovaní. Je to rozhranie, v ktorom používatelia spájajú farebné bloky a vytvárajú interaktívne príbehy, hry a animácie. Tento inovatívny prístup odstraňuje syntaktické bariéry a podporuje logické myslenie a riešenie problémov. Živá online komunita programu Scratch uľahčuje spoluprácu a zdieľanie obsahu, čím podporuje tvorivosť a učenie. Hra Scratch, ktorá sa hojne využíva vo vzdelávaní, zábavnou formou predstavuje koncepty kódovania a podporuje informatické myslenie. Vďaka podpore multimediálnych prvkov, spätnej väzbe v reálnom čase a neustálym aktualizáciám vrátane najnovšej verzie Scratch 3.0 umožňuje platforma používateľom objavovať kódovanie, vyjadrovať kreativitu a prispievať k vzdelávaniu v matematike, programovaní a vede. Scratch je dôkazom sily vizuálneho programovania pri sprístupňovaní písania kódu, ktoré je príjemné, inkluzívne a inšpiruje globálnu komunitu tvorivých študentov.

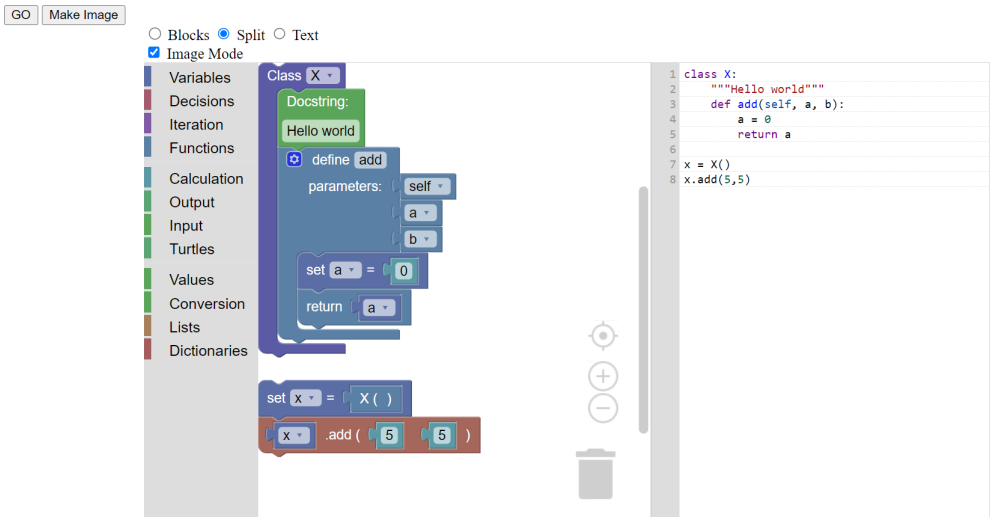
Snap!

Snap! je bezplatný vzdelávací grafický programovací jazyk založený na blokoch, ktorý je navrhnutý tak, aby sprístupnil kódovanie a zaujal používateľov všetkých vekových kategórií. Snap! vyvinula Kalifornská univerzita v Berkeley, má spoločné korene so Scratchom, ale rozširuje jeho koncept a poskytuje pokročilejšie a rozšíriteľnejšie prostredie. V Snap! používatelia používajú funkciu drag-and-drop na zostavovanie blokov, ktoré predstavujú kódovacie štruktúry, čo podporuje intuitívne a kreatívne programovanie. Snap! kladie dôraz na matematické a výpočtové myslenie a prostredníctvom svojho užívateľsky príveti-vého rozhrania oboznamuje užívateľov so základnými konceptami programovania. Podobne ako Scratch, aj Snap! podporuje integráciu multimédií a spätnú väzbu v reálnom čase, čo umožňuje užívateľom okamžite vidieť výsledky ich kódu. Záväzok platformy k otvorenosti a modularite umožňuje pokročilým užívateľom rozširovať jej možnosti. Bez ohľadu na to, či sa používa na vzdelávacie účely alebo na osobné projekty, Snap! predstavuje výkonný nástroj, ktorý umožňuje jednotlivcom skúmať programovanie, vyjadrovať kreativitu a rozvíjať základné počítačové zručnosti.

2.3 Aplikácie s podporou prevodu textu na bloky

Aplikácie, v ktorých je na rozdiel od už spomínaných aplikácií prevod textu na bloky implementovaný a podporovaný v ich užívateľskom rozhraní.

Block-Mirror



Obr. 2.4: Snímka obrazovky zobrazujúca UI BlockMirror aplikácie

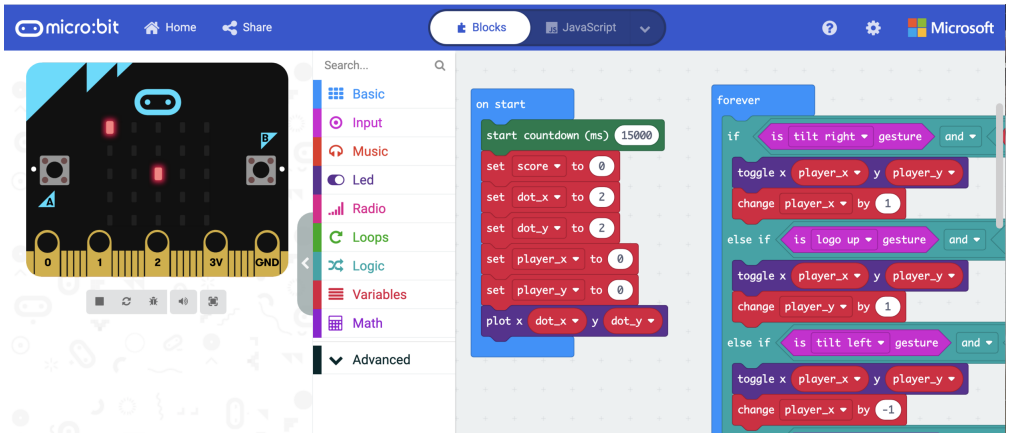
Táto aplikácia ponúka rozhranie pre duálnu reprezentáciu blokov/textov pomocou knižnice Blockly. Jej UI môžeme vidieť na obrázku 2.4 Je napísaná v JavaScripte a dovoľuje užívateľovi preklad kódu v jazyku Python do blokovej reprezentácie a naopak. Tento preklad prebieha automatický s každou zmenou, či už na strane blokov, alebo na strane textového kódu.

Ako textový editor využíva populárny front-end na úpravu textového kódu *CodeMirror*[5]. Pre spracovanie kódu používa *Skulpt*, transpiler z jazyka Python do JavaScriptu (*Pozn. Transpiler je typ kompilátoru, ktorý prekladá zdrojový kód z jedného programovacieho jazyka do druhého*). Aplikácia využíva časť funkcionality tohto nástroja a to konkrétne, ako parser.

Po prejdení zdrojových kódov tejto aplikácie by sme chceli vytvoriť nástroj, v ktorého kóde by sa orientovalo lepšie a bol by prístupnejší novým vývojárom pre pridávanie rozšírení. Aplikácia taktiež dovoľuje prevod kódu, ktorý vo väčšom spektre nemá zmysel a nie je úplne syntaktický správny. Napríklad aritmetický výraz bez priradenia výsledku do premennej, čo nás ponechá z blokom, ktorý má výstup, ale tento výstup nie je nikam pripojený. Podobne rieši ponechanie samotnej premennej na riadku bez priradenia hodnoty.

Aplikáciu si je možné odskúšať v aktuálnom stave vo webovom prehliadači, alebo si môžete stiahnuť jej zdrojové súbory a rozbehnúť si ju lokálne s prípadnými zmenami. K webu aj zdrojovým kódom sa viete dostať z Git-Hub repozitára aplikácie [6].

Makecode micro:bit



Obr. 2.5: Ukážka UI webového nástroja MakeCode micro:bit. Prevzaté z [7]

Micro:bit je malé, programovateľné zariadenie vyvinuté firmou BBC v spolupráci s rôznymi partnerskými organizáciami, ako napríklad Microsoft. Je určené pre vzdelávacie účely a zameriava sa na výučbu programovania a elektroniky. Webová aplikácia od Microsoftu [7], ponúka vlastné vývojové nástroje a platformu pre prácu s *micro:bit*. Je to interaktívne vizuálne programovacie prostredie, ktoré umožňuje vytvárať programy pre spomenuté zariadenie pomocou blokového programovania alebo písania kódu v jazyku TypeScript. Tieto programy sa potom môžu preniesť na *micro:bit* a spúšťať priamo na zariadení. Microsoft taktiež poskytuje rôzne vzdelávacie materiály a tutoriály, aby pomohol učiteľom a študentom využívať *micro:bit* vo výučbe. Na obrázku 2.5 môžeme vidieť ukážku užívateľského rozhrania. Pre vstup pomocou písania kódu by sa užívateľ prepol do *JavaScript* editoru.

Aplikácia MakeCode [1] prevádza kód na bloky s využitím parsovania a tvorby syntaktického stromu (AST). K tomu využíva metadátove komentáre, ktoré definujú mapovanie kódu do Blockly prostredie. Keď užívateľ napíše kód v textovej forme, MakeCode analyzuje

[5] `https://codemirror.net/`
[6] `https://github.com/blockpy-edu/BlockMirror`
[7] `https://makecode.microbit.org/`

tieto komentáre spolu s kódom a vytvára z nich stromovú štruktúru reprezentujúcu syntax kódu. Každý uzol je potom prevedený na ekvivalentný blok. Bloky sú následne usporiadané a umiestnené tak, aby držali hierarchiu kódu a jeho tok. Nakoniec ich vizualizačná vrstva MakeCode zobrazí v blokovom editore.

2.4 Love-blocks-web

Táto kapitola v krátkosti predstavuje diplomovú prácu [13], ktorú by mal nástroj vyvíjaný v tejto práci rozširovať. Jedna sa o webovú aplikáciu pre vizuálne programovanie v Lua/LÖVE. Bola vytvorená so zámerom zjednodušiť výuku programovania pomocou blokového programovania priamo vo vašom prehliadači.

Aplikácia ponúka možnosť tvorby hier v jazyku Lua za pomoci rámcu LÖVE, ktoré sme si už popísali v predchádzajúcej kapitole 2.1. Okrem samostatnej tvorby, ponúka aj možnosť nasledovať už existujúce tutoriály. Taktiež podporuje spúšťanie a vytvorených hier na platforme Android. Software tejto aplikácie je poublikovaný ako open-source a je možné ho nájsť a stiahnuť na stránkach Git-Hubu[8].

Technológie použité v tejto aplikácií:

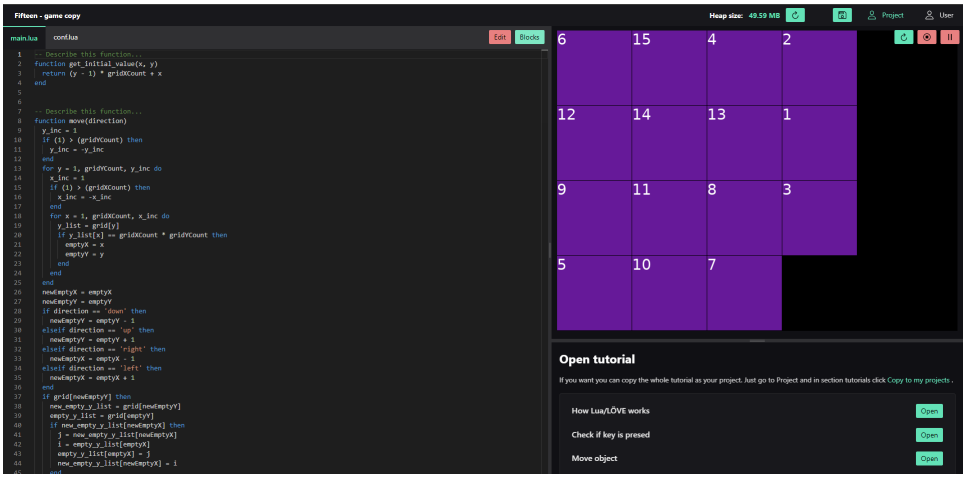
- **Laravel:** PHP framework využívajúci MVC architektúru bol použitý pre backend aplikácie.
- **MySQL:** relačný systém správy databáz, ktorý sa používa na ukladanie údajov aplikácie.
- **VueJS**[9] **+ InertiaJS**[10]**:** JavaScriptové rámce používané na vytváranie dynamických používateľských rozhraní.
- **Monaco Code Editor + Blockly Code Editor:** Editor pre editovanie kódu v jazyku Lua a editor pre tvorbu blokov a prácu s nimi.
- **Tailwind CSS:** CSS, ktorý sa používa na štylizáciu aplikácie.
- **Server jazyka Lua**[11]**:** Poskytuje jazykovú podporu pre programovací jazyk Lua.
- **Firebase Cloud Messaging:** Uľahčuje komunikáciu s Android zariadeniami.
- **Docker:** Kontajnerová platforma používaná na jednoduché nasadenie a škálovanie aplikácie. Konkrétne bol použitý `docker-compose` a aplikácia sa skladá z niekoľkých kontajnerov: `nginx`, `php`, `mysql` a `language-server`. Tieto kontajnery tvoria jadro aplikácie a sú spustené automaticky. Ďalšie použité kontajnery sú: `npm`, `artisan` a `composer`.

V aplikácií bol použitý docker image `jonasal/nginx-certbot`[12] webového serveru Nginx. Pre databázu je použitý docker image `mysql:8.0`. Image `PHP` je vytvorený pomocou viacúrovňového `Dockerfile`. Ako základ je použitý `node:16`, z ktorého sú kopírované binárne súbory do image `php:8.1-php`. To umožňuje spustiť node vo výslednom

[8] `https://github.com/meda10/Love-blocks-web/tree/master`
[9] `https://vuejs.org/`
[10] `https://inertiajs.com/`
[11] `https://github.com/LuaLS/lua-language-server`
[12] `https://hub.docker.com/r/jonasal/nginx-certbot`

kontajneri, ktorý je potrebný pre spustenie hier vo webovom prehliadači. Posledný image je `language-server`, ktorý podobne ako `PHP` využíva viacúrovňového `Dockerfile` zo základom `node:16`. Binárne súbory sú kopírované do `bitnami/minideb:latest`, čo je minimalistický image založený na Debiane. `Dockerfile` potom obsahuje kopírovanie súborov Lua Language Server a jeho spúšťanie pomocou node.

Pre bližšie detaily o tejto aplikácií, odporúčam prečítať pôvodnú prácu.



Obr. 2.6: Ukážka UI webovej aplikácie Love-blocks-web. Prevzaté z `https://love2d.org/forums/viewtopic.php?t=92894`

Kapitola 3

Syntaktická analýza a jej nástroje

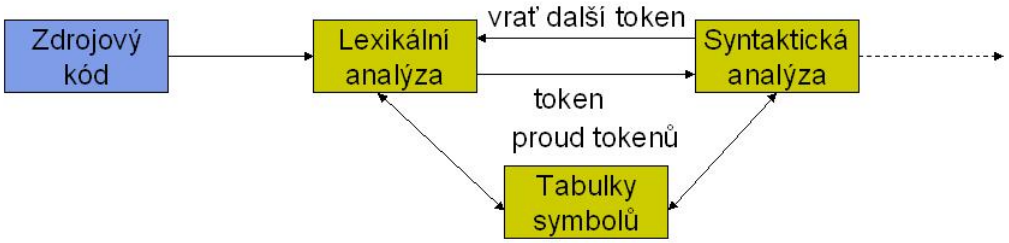
V tejto kapitole sa bližšie pozrieme na to čo je to syntaktická analýza, kde sa používa a ako prebieha 3.1. V ďalšej časti ukážeme existujúce nástroje, ktoré ju vykonávajú 3.2. Na jej konci si detailnejšie predstavíme nástroj Tree-sitter, keďže je pre túto prácu dôležitý.

3.1 Syntaktická analýza

Syntaktická analýza [14], známa aj ako parsovanie, je v počítačovej vede a lingvistike proces analýzy sekvencie tokenov na určenie ich gramatickej štruktúry s ohľadom na danú formálnu gramatiku. Jej hlavným účelom je zabezpečiť, aby kód napísaný v programovacom jazyku dodržiaval pravidlá a gramatiku špecifikovanú syntaxou tohto jazyka. V jej procese sa vstupný text spravidla transformuje na určité dátové štruktúry, väčšinou syntaktický strom. Je to dôležitá časť spracovania programov pri ich preklade (kompilácií) alebo interpretácií.

Syntaktická analýza sa používa aj pri spracovaní ľudského jazyka 3.2 za pomoci parserov, kde je možné napr. rozdeliť slová podľa gramatického tvaru a zistiť správnosť poradia slov. Najprv však musíme mať stanovenú príslušnú formálnu gramatiku. Syntax sa volí na základe zámerov ako lingvistických, tak aj implementačných.

Zjednodušene povedané, je parsovanie analýza vstupu pre zorganizovanie dát podľa určených pravidiel. Tieto pravidlá môžu byť napríklad určené regulárnymi výrazmi, kde Kleeneho hviezdička (*) ako pravidlo indikuje, že nejaký element sa môže vyskytnúť nula alebo nekonečne veľa krát.



Obr. 3.1: Ukážka procesu parsovania so základnými časťami. Prevzaté z Wikipédie.

Ako už bolo spomenuté, syntaktická analýza je často známa ako parsovanie. Presnejšie povedané je to však iba časť celého procesu parsovania zdrojového kódu a jeho prekladu. Tieto časti sú:

- **Lexikálna analýza:** Zdrojový kód je rozdelený na jednotlivé jednotky nazývané tokeny. Tokeny sú najmenšie zmysluplné jednotky v programovacom jazyku, ako sú kľúčové slová, identifikátory, operátory a symboly. Tieto symboly sa ukladajú do tabuľky tokenov.
  Vstup: Zdrojový kód.
  Výstup: Sekvencia tokenov.

- **Syntaktická analýza:** Známa aj ako vlastný rozbor (angl. *parsing proper*). Zahŕňa analýzu syntaktickej štruktúry zdrojového kódu na základe postupnosti tokenov. Hlavným cieľom syntaktickej analýzy je overiť, či usporiadanie tokenov dodržiava pravidlá špecifikované formálnou gramatikou programovacieho jazyka.
  Vstup: Sekvencia tokenov.
  Výstup: Abstraktný syntaktický strom (AST).

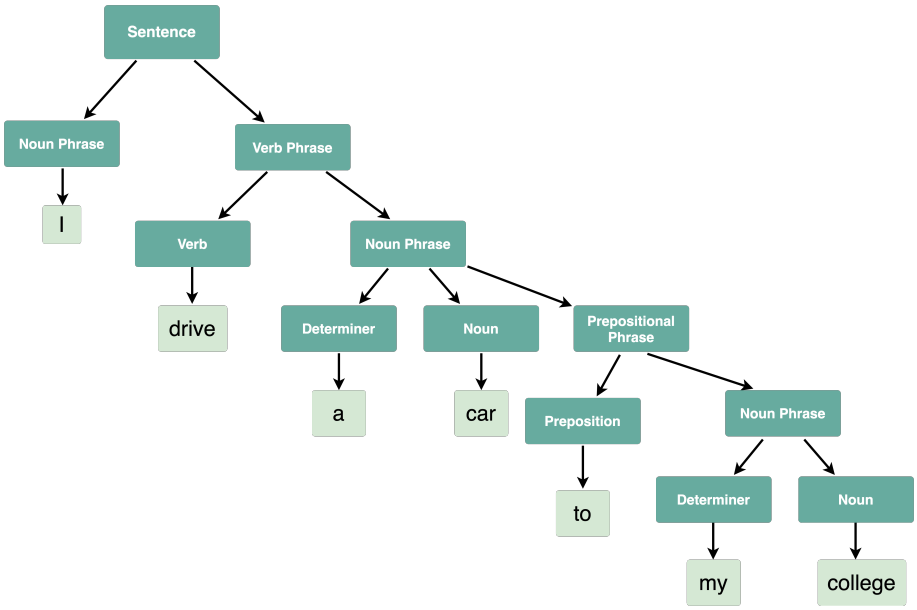
- **Konštrukcia syntaktického stromu:** Počas syntaktickej analýzy sa vytvorí abstraktný syntaktický strom (AST). AST je hierarchická reprezentácia syntaktickej štruktúry kódu, ktorá zachytáva vzťahy medzi rôznymi konštrukciami jazyka.
  Vstup: AST.
  Výstup: Anotovaný AST.

- **Sémantická analýza:** Sémantická analýza sa môže vykonávať ako súčasť parsovania alebo ako samostatná fáza. Presahuje rámec syntaxe a kontroluje význam kódu, pričom zabezpečuje, aby kód dodržiaval sémantické pravidlá a obmedzenia.
  Vstup: Anotovaný AST.
  Výstup: Sprostredkovaný kód.

- **Generovanie prechodného kódu:** V niektorých kompilátoroch alebo interpretoch sa po rozbore môže vygenerovať medziľahlá reprezentácia kódu. Tento medziprodukt slúži ako most medzi zdrojovým kódom vysokej úrovne a cieľovým kódom (strojový kód alebo iná medziproduktová reprezentácia).
  Vstup: Sprostredkovaný kód.
  Výstup: Optimalizovaný kód.

- **Optimalizácia:** Na medziproduktový kód sa môže použiť optimalizácia s cieľom zvýšiť efektívnosť výsledného spustiteľného kódu.
  Vstup: Zdrojový kód.
  Výstup: Sekvencia tokenov.

- **Generácia výsledného kódu:** Posledná fáza zahŕňa generovanie cieľového kódu (strojový kód alebo iný spustiteľný formát) na základe medzikódu alebo priamo z AST.
  Vstup: Optimalizovaný kód.
  Výstup: Cieľový kód.



Obr. 3.2: Ukážka syntaktického stromu vzniknutého pri parsovaní textu ľudského jazyka. Prevzaté z [15]

3.2 Existujúce nástroje

ANTLR (ANother Tool for Language Recognition)

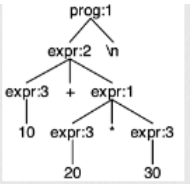
ANTLR[1] [16] je výkonný generátor parserov na čítanie, spracovanie, vykonávanie alebo preklad štruktúrovaných textových alebo binárnych súborov. Široko sa používa na vytváranie jazykov, nástrojov a rámcov. Keďže je určený na generovanie parserov v rôznych programovacích jazykoch vrátane jazykov *Java*, *C#*, *Python* a ďalších, je všestranný a vhodný pre širokú škálu projektov. Používa prístup založený na gramatike LL(*) ku špecifikácii syntaxe jazyka, čo znamená, že používa prediktívny parsovací algoritmus, ktorý zvláda ľavú rekurziu. Jeho kľúčove vlastnosti sú: založený na gramatike, podporuje lexikalizáciu a parsovanie, generuje parsovacie stromy, obsahuje funkcie na vytváranie návštevníkov a poslucháčov pre prechádzanie AST čo dovoľuje vývojárovi vykonávať akcie počas parsovacieho procesu.

ANTLR je často integrovaný do populárnych vývojových prostredí (IDE), ako sú IntelliJ IDEA[2] a Visual Studio Code[3], ktoré používateľom poskytujú funkcie ako zvýrazňovanie syntaxe, automatické dokončovanie a zvýrazňovanie chýb.

[1] `https://www.antlr.org/`
[2] `https://www.jetbrains.com/idea/`
[3] `https://visualstudio.microsoft.com/cs/`

```
grammar Expr;
prog:   (expr NEWLINE)* ;
expr:   expr ('*'|'/') expr
    |   expr ('+'|'-') expr
    |   INT
    |   '(' expr ')'
    ;
NEWLINE : [\r\n]+ ;
INT     : [0-9]+ ;
```

```
$ antlr4-parse Expr.g4 prog -gui
10+20*30
^D
$ antlr4 Expr.g4  # gen code
$ ls ExprParser.java
ExprParser.java
```



Obr. 3.3: Ukážka práce s ANTLR. Popis z ľavej strany: V prvej časti obrázka je zadefinovaná gramatika uložená v súbore. V strede je volanie *antlr-parse* príkazu na výraz $10 + 20 * 30$. Napravo vidíme výstup príkazu [11]

PLY (Python lex-Yacc)

PLY[2] predstavuje čisto pythonovskú implementáciu nástrojov na konštrukciu kompilátorov lex a yacc. Jeho hlavným cieľom je presne dodržať funkčnosť tradičných nástrojov lex/yacc, podporovať parsovanie LALR(1) a zároveň ponúkať rozsiahlu validáciu vstupu, hlásenie chýb a diagnostiku.

Ako je spomenuté v predchádajúcom odstavci, PLY sa skladá z dvoch samostatných modulov: *lex.py* a *yacc.py*. Modul *lex.py* sa používa na rozklad vstupného textu do kolekcie tokenov špecifikovaných kolekciou pravidiel regulárnych výrazov. *yacc.py* slúži na rozpoznávanie syntaxe jazyka, ktorá bola špecifikovaná vo forme bezkontextovej gramatiky.

Tieto dva nástroje majú spolupracovať. Konkrétne *lex.py* poskytuje rozhranie na vytváranie tokenov, zatiaľ čo *yacc.py* pomocou neho získava tokeny a vyvoláva pravidlá gramatiky. Výstupom *yacc.py* je často AST. To však závisí výlučne od používateľa a je možné ho použiť aj na implementáciu jednoduchých jednoprechodových kompilátorov.

Prvé verzie PLY boli navrhnuté ako výučbový nástroj a pôvodne boli vytvorené pre podporu univerzitného kurzu. V dôsledku toho PLY vykazuje dôkladný prístup k špecifikácii tokenov a gramatických pravidiel, ktorého cieľom je identifikovať bežné programátorské chyby začínajúcich používateľov. Táto formálnosť sa ukazuje ako výhodná pre pokročilých používateľov, ktorí konštruujú zložité gramatiky pre skutočné programovacie jazyky. Je dôležité poznamenať, že PLY nemá ďalšie funkcie, ako je automatická konštrukcia abstraktného syntaktického stromu alebo prechádzanie stromom, a neprezentuje sa ako parsovací rámec. Namiesto toho ponúka základnú, ale plne funkčnú implementáciu lex/yacc kompletne napísanú v jazyku Python.

Tree-sitter

Citujem oficiálnu stránku Tree-sitter [3]: *"Tree-sitter je nástroj na generovanie parserov a knižnica na inkrementálne parsovanie. Dokáže vytvoriť konkrétny syntaktický strom pre zdrojový súbor a efektívne aktualizovať syntaktický strom pri úprave zdrojového súboru."*. Tree-sitter je veľmi rýchly, inkrementálny parser vytvorený Git-Hub komunitou, ktorý je pri správnom vygenerovaní schopný inkrementálne analyzovať súbor pri každom stlačení klávesu. Taktiež dokáže vykonávať obnovu chýb, čo znamená, že ak je v súbore chyba, zvyšok AST to neovplyvní a bude vygenerovaný správne.

Inkrementálne parsovanie znamená, že je potrebné znova prejsť iba časti súboru, ktoré boli zmenené. To z neho robí veľmi efektívny nástroj v interaktívnom vývojovom prostredí.

Vývojari si môžu zadefinovať gramatiku 5.2.1 pre programovací jazyk s použitím doménovo špecifického jazyka (angl. *domain specific language*), alebo DSL, ktorý je podobný JavaScript-u. Táto špecifikácia definuje syntaktické pravidlá jazyka. Pre väčšinu jazykov ako je *Python*, *Lua* a mnoho ďalších to nie je nutné, keďže Github komunita pre nich tieto gramatiky už vytvorila. Modulárna povaha knižnice umožňuje pomerne jednoduché pridávanie podpory pre ďalšie jazyky.

Tree-sitter používa rozbor gramatiky výrazov LR(1), alebo PEG (angl. *Parsing Expression Grammar*). Aj tento výber prispieva k jeho efektívnosti a schopnosti pracovať so zložitými gramatikami.

Ako je zvykom pri mnohých nástrojoch tohto typu, výstup jeho parsovacieho procesu je AST. Ten reprezentuje hierarchickú štruktúru kódu a je nápomocný pre analýzu a manipuláciu s kódom.

Tree-sitter je často integrovaný do textových editorov a integrovaných vývojových prostredí (IDE), aby poskytoval funkcie, ako je zvýrazňovanie syntaxe, skladanie kódu a efektívna navigácia v kóde. Bol navrhnutý ako multiplatformný, takže je vhodný na integráciu do rôznych vývojových prostredí bežiacich na rôznych operačných systémoch.

Okrem programovacích jazykov možno Tree-sitter prispôsobiť aj na parsovanie iných formátov štruktúrovaných údajov, čo z neho robí vskutku univerzálny nástroj.

Základné pojmy v prostredí Tree-sitter

- **Súbory parserov:** Srdcom Tree-sitteru sú parsre [8]. Sú to knižnice, po ktorých sa Tree-sitter bude pozerať v runtime adresári parseru. Ak sa nájde viacero existujúcich parserov pre rovnaký jazyk, použije sa prvý. Môžeme taktiež parser určiť a načítať manuálne s použitím celej cesty k nemu.
- **Tree-sitter stromy:** Reprezentujú parsovaný obsah vyrovnávacej pamäte, ktorý môže byť použitý k ďalšej analýze. Je to referencia dát užívateľa na objekt vygenerovaný parserom.
- **Tree-sitter uzly:** Uzol predstavuje jeden špecifický element už spracovaného obsahu vyrovnávacej pamäte. Ten môže byť zachytený pomocou *aQuery* napr. pre vyznačenie.
- **Tree-sitter queries:** Sú spôsobom ako vytiahnuť informácie o parsovanom strome. Stručne povedané, query pozostáva z jedného alebo viacerých vzorov. Vzor je definovaný nad typmi uzlov v syntaktickom strome a zodpovedá konkrétnym prvkom syntaktického stromu, ktoré zodpovedajú vzoru. Píšu sa v jazyku podobnom jazyku Lisp a sú dokumentované na oficiálnej stránke *Tree-sitter*-a[4].

3.2.1 Prečo Tree-siter

Tree-sitter je navrhnutý s ohľadom na modulárnosť. Umožňuje jednoduchú integráciu do rôznych projektov a možno ho rozšíriť o podporu nových jazykov. Je ho taktiež možné použiť z rôznych jazykov ako *Go*, *Haskell*, *Java*, *Python* a mnoho ďalších a to vďaka prepojeniam (tzv. *bindings*), ktoré naviažu parser v jazyku C na požadovaný jazyk.

Poskytuje možnosť integrácie pomocou webového parseru *web-tree-sitter*, ktorý konzumuje jazyk vo forme vygenerovaného binárneho súboru *.wasm*. *web-tree-sitter* je mienený pre aplikácie napísané pre webové prehliadače. Je napísaný v jazyku *JavaScript* a jeho použitie môžeme vidieť na nasledujúcej ukážke 3.1.

[4] `https://tree-sitter.github.io/tree-sitter/using-parsers#query-syntax`

```
const Parser = require('web-tree-sitter');
Parser.init().then(async () => {
    const parser = new Parser();
    const Lang = await Parser.
        Language.load("tree-sitter-lua.wasm");
    parser.setLanguage(Lang);
    const tree = parser.parse(" ... code ...");
});
```

Výpis 3.1: Ukážka inicializácie a použitia parseru v prostredí webového prehliadača.

Taktiež existuje *node tree-sitter*, ktorý je myslený ako rozšírenie (modul) do Node.js projektov a pre parsovanie rôznych jazykov využíva ďalšie node balíčky: *tree-sitter-javascript*, *tree-sitter-lua*,... V tomto prípade nie je potrebný asynchrónny prístup:

```
const Parser = require('tree-sitter');
const Lang = require('tree-sitter-lua');

const parser = new Parser();
parser.setLanguage(Lang);
const tree = parser.parse(" ... code ...");
```

Ako je spomenuté vyššie, je možné použiť už existujúce gramatiky, alebo si napísať a vygenerovať vlastné. K tomuto slúži nástroj *tree-sitter-cli*[5]. V novších verziách je tento nástroj súčasťou *tree-sitter* balíčku. Tento nástroj pre príkazový riadok vyhľadá v adresári v ktorom je spustený súbor *grammar.js* a na jeho základe vygeneruje parser a *bindings* pre použitie z iných jazykov pomocou:

```
$ tree-sitter generate
```

Výstupom tohto procesu bude stromová sada zdrojových súborov, ktoré tvoria parser. Takto vytvorený parser (v procese inicializácie parseru môžeme brať skôr ako jazyk) môžeme použiť napríklad s *node tree-sitter*. Pre *web-tree-sitter* by sme po **generate** príkaze ešte použili:

```
$ tree-sitter build --wasm
```

Bez špecifikácie výstupu tento príkaz vygeneruje súbor s názvom *parser.wasm*, ktorý je možné predložiť *web-tree-sitter* parseru ako parameter pri inicializácii. Okrem týchto argumentov poskytuje aj napríklad **test** pre otestovanie gramatiky, alebo **parse** pre parsovanie obsahu súborov z terminálu. Je potrebné dávať si pozor na to aké verzie týchto nástrojov používame, aby boli vygenerované súbory kompatibilné s parserami. Všetky spomenuté vlastnosti a prednosti robia **Tree-Sitter** vhodnou voľbou pre túto prácu.

[5] https://github.com/tree-sitter/tree-sitter/blob/master/cli/README.md

Kapitola 4

Návrh nástroja

Kapitola sa zaoberá návrhom implementácie nástroja vyvíjaného v tejto práci. Prvá podkapitola 4.1 špecifikuje požiadavky, ktoré by mal nástroj spĺňať. V ďalšej sekcii 4.2 si vyvedieme dôležité poznatky a funkčné požiadavky zo špecifikácií definovaných v predošlej sekcii. Následne 4.3 je popísaný návrh nástroja rozdelený do štyroch častí. Tieto časti a ich prepojenie môžeme vidieť na obrázku 4.1. Nakoniec si popíšeme ako presne by mal byť tento proces napojený do existujúcej aplikácie 4.4.

4.1 Špecifikácia požiadavkov

Uvedieme základné špecifikácie pre túto prácu, keďže ďalej sa môže uberať dvoma smermi. Ako je spomenuté v Úvode 1, nástroj by mal byť po jeho implementácii použitý, ako rozšírenie pre aplikáciu vyvinutú v diplomovej práci: *"Výukový software pro vizuální a textové programování v Lua/LÖVE"* [13]. Túto aplikáciu by mal rozširovať o možnosť spätného generovania blokov zo zdrojového kódu. Druhá možnosť je vytvorenie samostatnej aplikácie, ktorá by kládla väčší dôraz na korektnosť prevodu kódu do blokov. V oboch prípadoch je hlavným cieľom vytvoriť nástroj, ktorý spracuje zdrojový kód napísaný v jazyku Lua, správne vygeneruje postupnosť blokov, ktoré tento kód reprezentujú a výslednú reprezentáciu zobrazí užívateľovi.

Špecifikácie pre rozšírenie

- Riešenie musí zapadať do už existujúceho nástroja a správne s ním spolupracovať. Je potrebné hlboké porozumenie nástroja z predošlej práce.
- Podpora aktuálnej verzie Lua s rámcom Love2d použitej v tomto projekte.
- Potreba voľby vhodného implementačného jazyka, ktorý by spolupracoval a mohol by byť použitý vrámci existujúceho riešenia.
- Prevod zdrojového kódu do blokov by mal byť dostačujúci pre základné koncepty a štruktúry známe v prostredí programovania.
- Spätným prevodom pomocou nástroja, ktorý rozširujeme by bolo možné validovať výsledok prevodu nášho rozšírenia.

Špecifikácie pre samostatné riešenie

- Prevod zdrojového kódu by mal byť čo najobsiahlejší a fungovať spoľahlivo aj pre zložitejšie a väčšie štruktúry kódu.
- Potreba hlbšieho porozumenia syntaktickej analýzy a procesu prevodu AST vytvoreného z kódu do blokov.
- Tvorba vlastného užívateľského rozhrania pre zobrazenie vytvorených blokov.

Spoločné

- **Rozšíriteľnosť:** Nástroj by mal byť rozšíriteľný pre ďalšie programové štruktúry a prípadne ďalšie programovacie jazyky.
- **Výkon a škálovateľnosť:** Riešenie by malo byť dostatočne výkonné a škálovateľné pre spracovanie väčších súborov a projektov.
- **Otestovanie:** Dostatočné otestovanie by malo zabezpečiť, že nástroj funguje v prípadoch, ktoré sme anticipovali.
- **Podpora pre vývojárov:** Poskytnutie nástroja a dôležitých informácií vývojárom pre možnosť rozvoju a údržbe riešenia.

4.2 Funkcionalita nástroja

V tejto kapitole si vyvedieme dôležité poznatky zo špecifikácií definovaných v predošlej sekcii 4.1. Bližšie sa pozrieme na jednotlivé časti nástroja, ich zodpovedností a taktiež na ich vstupy a výstupy.

Získanie užívateľského vstupu

Prvá časť by mala spĺňať úlohu textového editoru a umožňovať užívateľovi aktualizovať svoj zdrojový kód, bez použitia blokov mimo Blockly editor. Táto časť je v existujúcom nástroji už implementovaná pomocou textového editoru Monaco, ktorý je súčasťou *Editor.vue* súboru. Nateraz primárne slúži pre posledné úpravy a zmeny kódu, ktoré sa môžu prejaviť iba pri aktualizácií hry, nie v Blockly editore.

Jeho obsah je priamo dostupný zo súboru, v ktorom s ním potrebujeme pracovať (*Show.vue*) a pre naše rozšírenie by nemal byť problém ho získať. Po úprave zdrojového kódu a prepnutí sa do Blockly editoru by sa malo inicializovať naše rozšírenie.

Parsovanie vstupu

Pre túto časť je dôležitá rýchlosť spracovania vstupu a aj z tohto dôvodu sme sa rozhodli pre *Tree-Sitter* parser. Po inicializácii Tree-sitter parseru a načítaní príslušnej gramatiky mu poskytneme zdrojový kód na spracovanie. Po dokončení syntaktickej analýzy vstupu dostaneme AST a ten posunieme na spracovanie nášmu nástroju.

Spracovanie AST a tvorba blokov

Koreň získaného syntaktického stromu z predchádzajúceho kroku sa predá nášmu nástroju. Ten bude zodpovedný za:

- postupné spracovanie jednotlivých uzlov AST,
- identifikáciu a správne priradenie JSON definície bloku pre konkrétny uzol,
- inicializáciu blokov na Blockly rámec,
- vytváranie prepojení (prípadne medzier) medzi jednotlivými blokmi,
- tvorbu vnorených blokov pre cykly, funkcie a podmienky,
- tvorbu parametrov/argumentov správneho typu a ich korektné priradenie
- ich správne vykreslenie do Blockly workspace.

Výstupom tohto kroku by malo byť na novo vygenerované prostredie Blockly (tzv. Blockly Workspace).

Navodenie nového stavu pre Blockly prostredie

V tomto bode je potrebné aktualizovať hodnotu, ktorá drží stav Blockly Workspace. Výsledné bloky, by mali byť správne zobrazené v Blockly editore a malo by s nimi byť možné naďalej pracovať rovnako, ako v stave pred ich vygenerovaním zo zdrojového kódu. Taktiež je naďalej možné z nich spätne vygenerovať správny Lua kód.

Obr. 4.1: Zjednodušený návrh workflow nástroja

Súhrn

Po zvážení týchto funkčných požiadavok a zodpovednosti jednotlivých častí riešenia sme a rozhodli, že bude vhodné pracovať na klientskej strane projektu. Z tohto dôvodu sme sa rozhodli pre implementáciu v jazyku JavaScript. *Vue.js* rámec použitý pre front-end rozširovanej aplikácie je napísaný v JavaScripte a podporuje integráciu JavaScript skriptov. Z tohto dôvodu by integrácia výsledného nástroja nemala byť obtiažna. Okrem toho mi príde rozumné využiť prepojenie cyklického a rekurzívneho spracovania. V hlavnom cykle by sa prechádzali hlavné uzly a pomocou rekurzie by sa riešili vnorené kódové štruktúry (napr. pre cykly, alebo funkcie).

4.3 Návrh rozšírenia

V tejto podkapitole si zadefinujeme hlavnú triedu a jej podtriedy, ktoré budu pracovať s jednotlivými uzlami. Tie budu navrhnuté tak, aby sa im dobre pracovalo s výstupom Tree-Sitter parseru. Taktiež budeme potrebovať triedu, ktorá sa bude podľa obdržaného uzlu starať o inicializáciu správnej podtriedy pre spracovanie tohto uzla. Okrem deľby práce budú atribúty tejto triedy držať informácie potrebné pre správne generovanie blokov. Jej inštancia bude predávaná medzi jednotlivými triedami.

Výstup parseru

Výstupom parseru je objekt predstavujúci strom, ktorý vyzerá následovne:

```
Tree {0: 140816, Language: Language, textCallback: f, rootNode: Node}
```

Z tohto stromového objektu budeme pracovať hlavne s jeho koreňom (*rootNode*) a deťmi tohto koreňa. Na nasledujúcej ukážke môžeme vidieť textovú reprezentáciu koreňa, ktorý má len jeden uzol (jedného potomka). Hneď pod ním môžeme vidieť kód, ktorého spracovaním vznikol tento hlavný koreň AST.

```
(chunk
    (for_numeric_statement
        name: (identifier)
        start: (number)
        end: (number)
        step: (number)
        body: (block
            (variable_assignment
                (variable_list
                    (variable name: (identifier)
                )
            )
            (expression_list
                value: (binary_expression
                    left: (number)
                    right: (number)
                    )
                )
            )
        )
    )
)
```

```
for exp = 1, 10, 1 do
    a = 1 + 2
end
```

Môžeme vidieť, že hlavný koreň je označený ako *chunk*. Jeho jediný uzol je typu *for_numeric_statement*, ktorý bol takto pomenovaný v definícii gramatiky. Podľa tohto výpisu by sme povedali, že má uzol 5 detí. Avšak keď sa pozrieme na výpis jeho detí, zistíme, že ich má 11. Konkrétne v tomto výpise nevidíme uzly pre: *for*, = , čiarku (,), čiarku (,), *do* a *end*. Viditeľné sú iba uzly:

- *(identifier)* pre **exp**
- *(number)* pre **1**
- *(number)* pre **10**
- *(number)* pre **1**
- *(block)* pre **a = 1 + 2**

Na túto skutočnosť si budeme musieť popri implementácii dávať pozor a vždy sa radšej riadiť výpisom všetkých uzlov, namiesto jednoduchej textovej reprezentácie. Ako parsovací nástroj teda budeme primárne používať spomínaný *web-tree-sitter*, s vygenerovaným *.wasm* súborom. Tie by sa mali dať dobre použiť v prostredí webovej aplikácie.

Trieda MyNode a jej podtriedy

Po prvotných pokusoch o implementáciu s čistým procedurálnym prístupom a konzultácii sme sa rozhodli pre prácu s jednotlivými uzlami použiť objektový prístup a vytvoriť jednú hlavnú triedu MyNode. Táto trieda bude implementovať všetky metódy, ktoré sú potrebné pre spracovanie väčšiny typov uzlov.

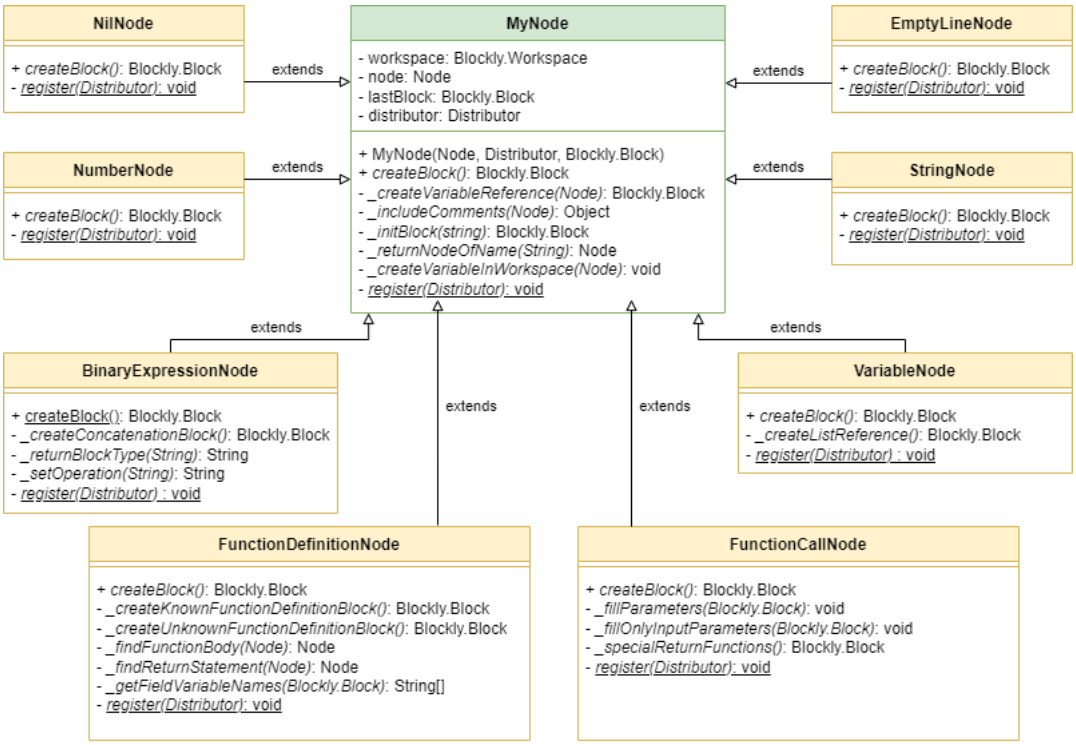
Pre všetky typy uzlov získaného AST, ktoré budeme chcieť spracovávať si zadefinujeme jej podtriedu, ktorá bude implementovať hlavnú metódu pre generovanie príslušného bloku. Okrem toho bude implementovať aj všetky metódy, ktoré sú potrebné k spracovaniu tohto konkrétneho uzlu.

Odôvodnenie využitia objektového prístupu pomocou tried na spracovanie uzlov:

- **Jednoduchšia stavba a zapuzdrenie:** Umožní to oddeliť jednotlivé časti kódu. Každá trieda bude zodpovedná za určitú funkcionalitu, v našom prípade spracovanie konkrétneho typu uzla. Vďaka tomu bude kód prehľadnejší a čitateľnejší.
- **Rozšíriteľnosť:** Objektový prístup uľahčí pridávanie nových funkcií (uzlov na spracovanie a blokov na tvorbu). Zmeny jednej triedy nemusia ovplyvniť ostatné časti kódu, čo by malo do budúcna zlepšiť aj údržbu kódu.
- **Abstrakcia a znovu použiteľnosť:** Ako je už spomenuté vyššie, je možné vytvoriť jednu základnú triedu pre všeobecný uzol stromu a od nej rozšíriť konkrétne triedy pre rôzne typy uzlov pomocou dedičnosti.
- **Jednoduchšie testovanie:** Je jednoduchšie dopátrať sa ku zdroju chyby.
- **Deľba práce:** S týmto prístupom môžeme upustiť od zložitého a zle čitateľného prístupu k deľbe práce pomocou podmienok, ktorý by bol inak nevyhnutný.

Nasledujúci diagram 4.2 popisuje vzťahy medzi jednotlivými triedami. V našom prípade sú tieto vzťahy jednoduché, keďže všetky podtriedy rozširujú hlavnú triedu a pre naše

využitie nie sú potrebné žiadne ďalšie prepojenia medzi podtriedami. Na diagrame nie sú predstavené všetky implementované podtriedy, pretože počet spracovaných uzlov je veľký a na všetky odpovedajúce triedy nebol priestor.



Obr. 4.2: Diagram tried, ktorý popisuje vzťahy medzi hlavnou triedou *MyNode* a jej podtriedami, ktoré ju rozširujú. *Pozn.: Na diagrame nie sú zobrazené všetky implementované podtriedy.*

Hlavná trieda *MyNode* bude jediná s implementovaným konštruktérom. V *JavaScript*-e to znamená, že všetky podtriedy budú volať konštruktér hlavnej triedy a ten sa postará o inicializáciu všetkých potrebných parametrov. Je to pre nás výhodné, keďže podtriedy nepotrebujú žiadne pre seba špecifické parametre.

S týmito parametrami budeme zaobchádzať ako s privátnymi a používať ich iba vrámci metód implementovaných v triedach:

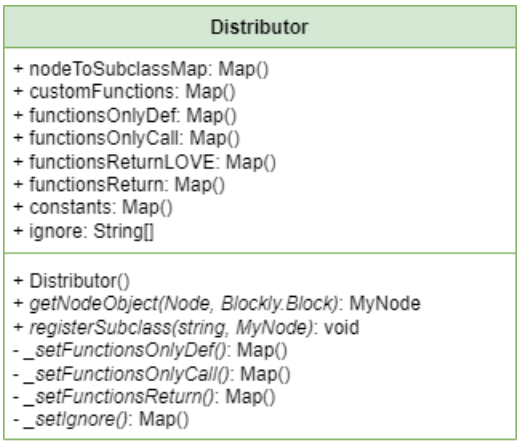
- *workspace:* drží informácie o už vytvorených blokoch (prepojenia, pozícia,...) a je potrebný pre tvorbu nových blokov
- *node:* aktuálne spracúvaný uzol, podľa ktorého sa vytvára konkrétny Blockly blok
- *lastBlock:* tento parameter nie je potrebný pre všetky podtriedy, ale je nevyhnutný pre správny postup tvorby blokov. Okrem iného je potrebný pre tvorbu prepojení medzi blokmi
- *distributor:* inicializovaný objekt triedy *Distributor* 4.3.1, ktorý je posúvaný medzi podtriedami *MyNode*. Sú v ňom zaregistrované všetky uzly na spracovanie a drží informácie nevyhnutné pre korektné vytvorenie niektorých blokov 5.3.4

V čistom JavaScripte nie je natívna podpora pre privátne metódy, ale s metódami, pre ktoré sme zvolili menovaciu konvenciu v štýle *_menoFunkcie()*, budeme pracovať akoby privátne boli. Budú teda používane iba v rámci tried. Príkladom takejto metódy je *_initBlock(name)*, ktorá implementuje vytvorenie a inicializáciu bloku 2.2. Volaním tejto metódy v *createBlock()* začína budovanie každého bloku. Nižšie sú popísané najdôležitejšie metódy implementované v hlavnej triede. Dve metódy si musia podtriedy prepísať, sú to metódy *register()* a *createBlock()*.

- *MyNode()* (alebo *constructor()* v kóde) vytvára novú inštanciu *MyNode*. Potrebuje k tomu tri parametre: *node*, *distributor*, *lastBlock*. Štvrtý parameter *workspace* je inicializovaný pomocou *Blockly.getMainWorkspace()* funkcie. Konštruktér je volaný všetkými podtriedami pri inicializácií ako *super()* metóda.
- *createBlock()* je hlavná metóda pre tvorbu bloku na základe poskytnutého uzlu. Každá podtrieda ponúka vlastnú implementáciu tejto metódy. Budeme ju volať vždy, keď narazíme na uzol, pre ktorý je potrebné vytvoriť blok.
- *_initBlock(name)* je metóda pre vytvorenie bloku na základe poskytnutého typu (parameter *name*) a jeho inicializáciu v Blockly Workspace. Vracia inicializovaný blok.
- *_createVariableInWorkspace(node)* je metóda pre vytvorenie premennej v Blockly Workspace prostredí na základe obdržaného uzlu (zvyčajne typu *variable*, ale môže byť aj pre *identifier*). Bez tejto registrácie nie je možné s premennou konkrétneho pomenovania pracovať. Ekvivalent v programovaní je pokus o priradenie hodnoty nedefinovanej premennej inej premennej.

4.3.1 Trieda Distributor

Podtriedy *MyNode* nám riešia otázku spracovania rôznych typov uzlov. Trieda *Distributor* nám bude zabezpečovať, aby bola pre každý uzol inicializovaná správna podtrieda. Vďaka tomu sa vyhneme spomínanému škaredému a zdĺhavému podmienkovému (*if-else*) zápisu. Pri inicializácii nášho rozšírenia bude inicializovaná jedna inštancia tejto triedy a tá bude používaná v priebehu celého "behu". Definíciu tejto triedy môžeme vidieť na nasledujúcom obrázku 4.3.



Obr. 4.3: Trieda Distributor, zodpovedná za deľbu práce a ukladanie dôležitých informácii.

V krátkosti si popíšeme parametre tejto triedy a ich význam:

- *nodeToSubclassMap:* mapa do ktorej sa na začiatku procesu tvorby nového Blockly Workspace stavu registrujú všetky uzly, pre ktoré je implementovaná trieda na ich spracovanie. V tejto mape uzol reprezentuje kľúč a podtrieda je hodnota priradená tomuto kľúču.
- *customFunctions:* mapa, ktorá drží potrebné informácie pre nové definície funkcií v kóde. Tieto funkcie musia byť spracované iným spôsobom od známych (napr. LOVE2D funkcií). Viac v kapitole o implementácií 5.
- *ignore:* zoznam s menami funkcií, ktorých definícia sa ma preskočiť.
- *functionsOnlyDef:* mapa, ktorá nám umožňuje spracovať definície známych funkcií. Podobnú funkcionalitu majú aj ostatné parametre.

Rovnako pre metódy:

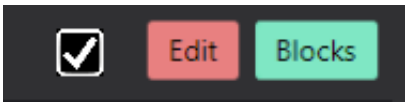
- *Distributor:* (alebo *constructor() v kóde* vytvára novú inštanciu triedy *Distributor* s prázdnym mapovaním *nodeToSubclass* a *customFunctions*. Zbytok parametrov inicializuje s pomocou svojich metód.
- *registerSubclass(type, MyNode):* pomocou tejto metódy sa všetky implementované podtriedy *MyNode* registrujú do *nodeToSubclassMap*, aby ich bolo možné používať.
- *getNodeObject(node, lastBlock):* implementuje hlavnú funkcionalitu tejto triedy. Na základe poskytnutého typu uzlu (*node.type*), ktorý je v textovom formáte, vracia metóda inicializovaný objekt podtriedy zodpovednej za aktuálne spracovaný uzol. Okrem parametrov *node* a *lastBlock* poskytuje ako parameter pre nový objekt aj seba samého. Blockly Workspace nie je potrebné pridávať ako argument, keďže konštruktér hlavnej triedy si ho inicializuje sám pomocou funkcie *Blockly.getMainWorkspace()* poskytnutej Blockly knižnicou.
- Metódy s predponou *_set-*, slúžia pre inicializáciu príslušných parametrov (máp) triedy.

4.4 Napojenie nástroja do Love-blocks

Ako sme si určili v požiadavkách, rozšírenie by malo byť napojené a bežať na klientskej strane projektu a malo by čo najmenej zasahovať do už existujúcich procesov. Funkcionalitu teda chceme iba pridávať a nie meniť. V existujúcom projekte nás najviac budú zaujímať tri súbory:

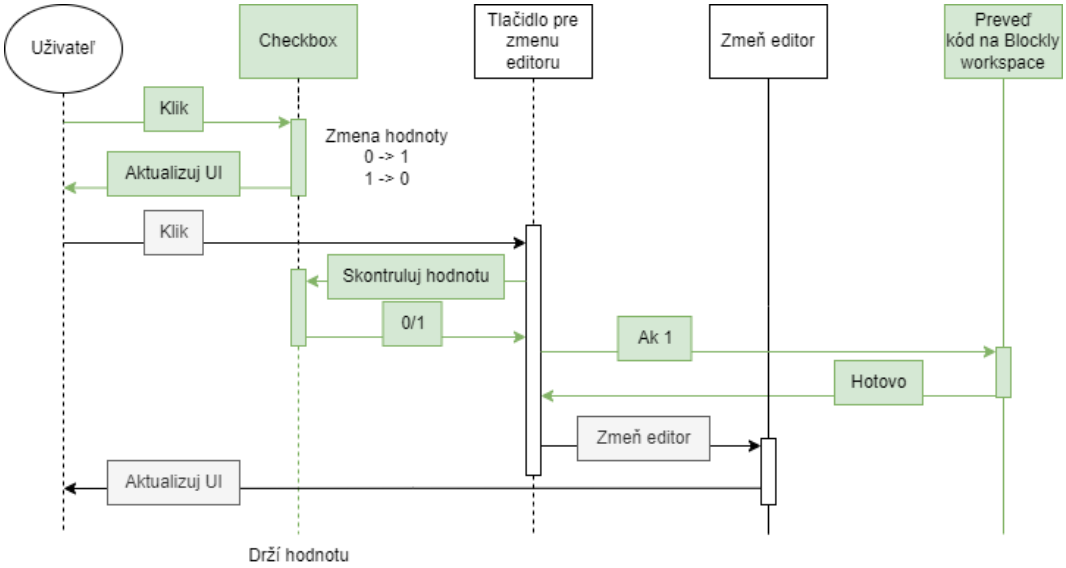
- *Editor.vue:* je v ňom integrovaný *Monaco* editor z ktorého budeme musieť vytiahnuť aktuálnu hodnotu. Tá predstavuje náš vstup, teda zdrojový kód.
- *Blockly.vue:* je v ňom integrovaný Blockly Workspace a po spracovaní zdrojového kódu ho bude potrebné aktualizovať na nový stav.
- *Show.vue:* slúži ako základ pre ďalšie dve spomínané súbory (pohľady). Obe hodnoty, ku ktorým potrebujeme mať prístup by sa podľa mojich zistení mali dať získať v tomto súbore. Tento súbor taktiež implementuje funkcionalitu tlačidla na prepínanie sa medzi editormi.

Funkcionalitu vytvoreného nástroja by sme chceli napojiť zo súboru *Show.vue*. A to konkrétne do implementácie tlačidla na prepínanie medzi editormi. Konkrétne hneď pred začiatok procesu zmeny editorov. Avšak, čo ak by sa užívateľ po zmene zdrojového kódu rozhodol, že zmeny nechce aplikovať? Bude mu potrebné poskytnúť spôsob, ktorým by mohol explicitne povedať, či zmeny aplikovať chce alebo nechce. K tomu by mal byť dostačujúci jednoduchý checkbox, keďže nám stačí držať informáciu logickej hodnoty **áno/nie** teda **true/false**.



Obr. 4.4: Ukážka pridania checkboxu do uživateľského rozhrania rozšírovanej aplikácie

S pridaným checkboxom by sa do procesu prepínania medzi editormi pridala logika načrtnutá na krátkom diagrame 4.5. Táto logika by bola potrebná iba pri prechode z textového editoru *Monaco* do Blockly editoru.



Obr. 4.5: Návrh pridania logiky pre napojenie rozšírenia. Tento jednoduchý diagram nie je založený na žiadnom existujúcom UML (Undified Modeling Language) ani podobnom štandarde, jedná sa o čisto vlastný návrh.

4.5 Zhrnutie

V tejto kapitole sme si popísali základné špecifikácie požiadavkov pre náš nástroj a z nich vyvodili potrebnú funkcionalitu nástroja. Taktiež sme si určili vstupy a výstupy jednotlivých časti nástroja a bližšie sa pozreli na to, ako je reprezentovaný AST. Po ich uvážení sme sa rozhodli pre využitie objektového prístupu pri spracovaní výstupu *Tree-Sitter* parseru. Pre jednotlivé uzly, ktoré bude potrebné spracovať sme si navrhli triedu *MyNode*, jej podtriedy a základné metódy ktoré budú poskytovať a s ktorými budeme pracovať. Zadefinovali sme si triedu *Distributor*, ktorá sa nám postará o mapovanie uzlov k príslušným triedam, predanie potrebných parametrov a ich inicializáciu. Nakoniec sme si načrtli ako by mohol byť náš nástroj zapojený do *Love-blocks-web* aplikácie a logiku, ktorú bude pre túto integráciu potrebné pridať.

Kapitola 5

Implementácia nástroja

V tejto kapitole sa pozrieme na použité vývojové a testovacie prostredie 5.1, vytvorené pre zjednodušenie implementačného procesu. Ďalej 5.2 na konkrétnu implementáciu spracovania AST vytvoreného *Tree-sitter* parserom. V podkapitole 5.3 sa pozrieme na funkcie a triedy implementujúce hlavnú funkcionalitu nástroja, mapovanie uzlov na jednotlivé *MyNode* podtriedy a triedu *Distributor*. V ďalšej časti 5.4 popíšeme ako sa zo spracovaného vstupu skladajú jednotlivé bloky a ako sa medzi nimi vytvárajú závislosti. Potom si ukážeme nevyhnutné riešenie cyklického generovania kódu/blokov 5.5. Nakoniec sa pozrieme na to, ako je možné tento nástroj rozšíriť, či už o existujúce uzly generované *Tree-sitter* parserom, alebo aj o nové pri prípadnej úprave gramatiky 5.6.

5.1 Vývojové prostredie

Čo sa týka IDE vývojového prostredia, v tejto práci bol využívaný **IntelliJ** od JetBrains. V tejto kapitole chcem ale hovoriť o jednoduchom servery, ktorý bol v tejto práci vytvorený. Keďže aplikácia *Love-blocks-web* vyvinutá v predchádzajúcej práci je celkom robustná a má mnoho závislosti a mnoho funkcionalít, ktoré pre nás nie sú dôležité, rozhodli sme sa pre vytvorenie vlastnej lokálnej webovej aplikácie. Jej hlavný účel je zjednodušiť proces vývoja a to nasledujúcimi spôsobmi:

- Zlepšuje orientáciu vo vlastnom kóde. Nateraz budeme pracovať iba so súbormi, ktoré potrebujeme a iba na funkcionalite, ktorú chceme implementovať
- S menším rozsahom kódu a jednoduchšou štruktúrou bude funkcionalita nástroja ľahšie testovateľná.
- Rýchlejšie nasadenie zmien. Nemusíme čakať, pokiaľ sa načítajú všetky závislosti a následne sa postaví a spustí celá aplikácia.
- Jednoduchšia kontrola prevedených zmien a následne lepší proces debugovania a hľadania algoritmických chýb.

Táto jednoduchá aplikácia bola založená na tutoriály poskytnutom od Blockly a dostupnom na ich Git-Hub stránkach [1]. Konkrétne išlo o projekt *examples/getting-started-codelab*. Ide o jednoduchý HTTP server založený na jednom hlavnom HTML súbore *index.html*. V tomto súbore sú pridané všetky závislosti, ktoré budeme používať a aj skripty, ktoré

[1] `https://github.com/google/blockly-samples/tree/master`

budeme implementovať. Vieme si ho spustiť na svojom počítači z adresára obsahujúceho *index.html* súbor cez konzolu pomocou príkazu na ukážke 5.1.

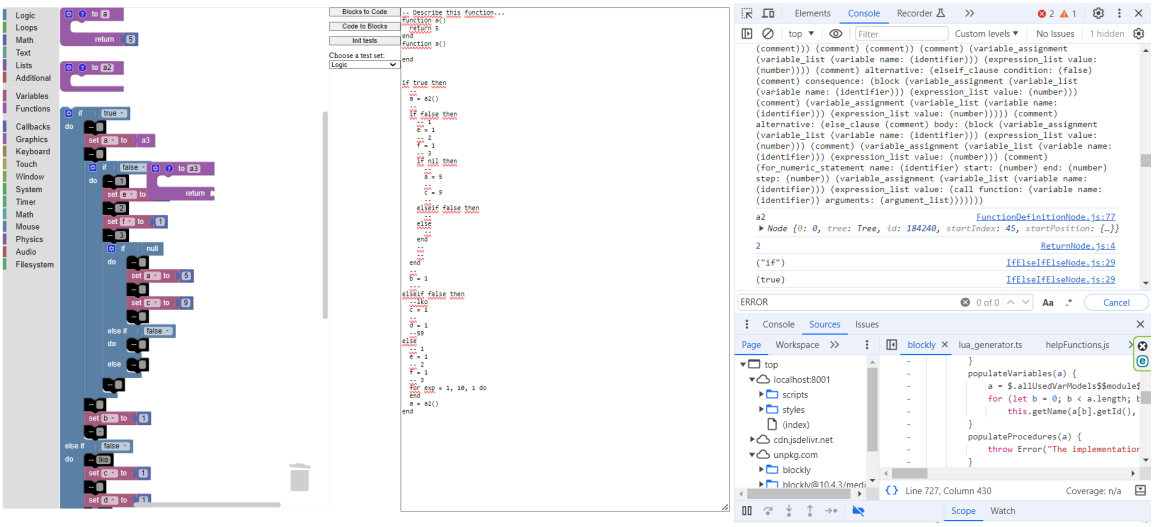
```
$ python -m http.server 8001
```

K takto spustenému serveru by sme pristupovali z webového prehliadača pomocou url *http://localhost:8001*. Toto prostredie nám dovoľuje upravovať bežiaci server a aplikovať prevedené zmeny pomocou znovu načítania stránky (CTRL + F5), čo výrazne zrýchľuje vývoj.

Stiahnutý projekt sme si následne upravili pre naše potreby:

- Vymazali sme nepotrebné bloky a pridali sme všetky definície blokov dostupné v *Love-blocks-web*.
- Pridali sme HTML element *textarea*, aby sme simulovali textový editor a editovanie kódu užívateľom.
- Aktualizovali sme Blockly Toolbox a zväčšili Blockly Workspace.
- Nakoniec sme pridali tieto prvky na ovládanie funkcionality:
  - Tlačidlo **Blocks to Code** nám prevedie bloky nachádzajúce sa v Blockly Workspace na kód a ten vypíše na konzolu vo vývojom nástroji prehliadača.
  - Tlačidlo **Code to Blocks** spustí náš nástroj a prevedie text prítomný v editore na bloky, ktoré vykreslí na Blockly Workspace.
  - Tlačidlo **Init tests** spustí skupinu testov vybranú v *dropdown* políčku.
  - *Dropdown* políčko slúži pre výber skupiny testov na spustenie.

S týmito úpravami nám tento server simuloval všetky dôležité aspekty pre vývoj a dovolil nám pracovať efektívnejšie. Jeho jediná nevýhoda je, že výsledný kód napísaný pre toto prostredie bude potrebné ešte trochu upraviť aby sme ho mohli použiť ako rozšírenie. Výsledné prostredie môžeme vidieť na obrázku 5.1.



Obr. 5.1: Snímka obrazovky jednoduchého HTTP serveru využitého k simulácii pre nás dôležitých častí existujúcej aplikácie pre efektívnejší vývoj nástroja

5.2 Spracovanie Tree-Sitter výstupu

V tejto časti sa lepšie pozrieme na spracovanie výstupu *Tree-sitter* parseru. V prvej podkapitole 5.2.1 si bližšie predstavíme gramatiku, ktorú v tomto nástroji používame. Potom sa pozrieme na to, čo nástroj dokáže spracovať 5.2.2 a čo je nad jeho aktuálne schopnosti 5.2.3. V poslednej podkapitole sa pozrieme na obmedzenia nástroja 5.2.4 mimo gramatiky.

5.2.1 Použitá gramatika

Ako je spomenuté v kapitole 3.2, verzia parseru *web-tree-sitter* používa ako podklad pre spracovanie vstupu vygenerovaný binárny súbor *.wasm*. Tento súbor sa generuje na základe zadefinovanej gramatiky. V tejto časti si popíšeme, ako sa takáto gramatika definuje a zmenu, ktorú sme zaviedli do už existujúcej gramatiky pre jazyk **Lua**.

Súbor *grammar.js* je postavený ako modul, ktorý exportuje objekt gramatiky. Tento objekt má dva hlavné parametre *name* (v našom prípade z hodnotou "lua") a *rules*. *rules* obsahuje definíciu gramatiky, ktorá pozostáva z nasledujúcich krokov:

- **Definícia tokenov:** Sú to pravidlá, podľa ktorých sa kód rozdelí na jednotlivé tokeny. Token môže byť jednoduchý literál, ako je pravdivostná hodnota: *true: () => "true"*, alebo zložitejší token, ktorý sa definuje pomocou regulárneho výrazu.

- **Definícia syntaktickej štruktúry:** Táto časť zahŕňa definície pre rôzne typy výrazov, príkazov, deklarácií atď. Definuje, aké časti kódu môžu byť syntaktický správne a v akom poradí sa môžu vyskytovať.

```
1         variable_assignment: ($) =>
2       seq($.variable_list, "=", alias($._value_list, $.expression_list)),
```

  Táto časť súboru gramatiky nám definuje token *variable_assignment* ako postupnosť *a = 1*, alebo *a, b,... = 1, 2,...*

- **Štart symbol:** Reprezentuje začiatok analýzy kódu. Obvykle koreňový symbol v hierarchii syntaktickej štruktúry. Tento symbol sme už spomínali v kapitole o *Tree-Sitter* parseri 3.2 a v našom prípade, je definovaný následovne:

```
1         chunk: ($) => seq(optional($.shebang), optional($._block))
```

- **Zložité výrazy:** Pre výrazy ako volanie funkcii, podmienky, definície funkcií, atď. sa definujú pravidlá o tom, ako sú tieto výrazy zložené z jednotlivých tokenov, alebo iných výrazov. Pre už spomínaný *for_numeric_statement* vyzerá táto definícia následovne:

```
1         for_numeric_statement: ($) =>
2       seq("for",
3         field("name", $.identifier),
4         "=",
5         field("start", $.expression),
6         ",",
7         field("end", $.expression),
8         optional(seq(",", field("step", $.expression))),
9         "do",
10        optional(field("body", $.block)),
11        "end",)
```

- **Priorita a asociativita operátorov:** Je dôležité definovať, aké operátory majú vyššiu prioritu, a aké sú asociatívne. Bez tejto informácie sa výrazy vyhodnotia nesprávne. V našom prípade má najvyššiu prioritu operátor **CALL** predstavujúci volanie funkcie a najnižšiu prioritu máju logické operátory **AND** a **OR**.
- **Komentáre a medzery:** Ich zachytávanie umožňuje parseru tieto časti kódu ignorovať.
- **Externé moduly:** V použitej gramatike nie sú využité.

Zmena, ktorú sme do tejto gramatiky zaviedli je veľmi jednoduchá, ale aby bola vykonaná správne, bolo potrebné pochopiť tvorbu gramatiky v jej celistvosti. Táto zmena sa týka spomínaných výrazov, ktoré sa majú ignorovať. Pre správnu tvorbu oddelených blokov, bolo potrebné do gramatiky pridať token prázdneho riadku a ten pridať do zoznamu výrokov, aby token nebol ignorovaný.

```
1     empty_line: ($) => "\n\n",
2     ...
3     statement: ($) =>
4       choice(
5         $.empty_line,
6         ...
```

5.2.2 Podporovaná syntax

V tejto časti si vypíšeme kódové štruktúry, ktoré by nášmu nástroju pri spracovaní nemali robiť problém. Cieľom bolo, aby nástroj dokázal spracovať hocijaký zdrojový kód, ktorý je možné vygenerovať pomocou blokov poskytnutých v aktuálnom Blockly Toolboxe rozširovanej aplikácie. Tie už užívateľa obmedzujú v opačnom smere generovania a nemalo by zmysel spracovávať syntax kódu, ktorú nie je možné vytvoriť pomocou existujúcich blokov. Popisujeme tu teda, čo je schopný spracovať celý nástroj, nie parser (ten generuje aj uzly, ktorými sa nebudeme v tejto práci zaoberať).

Najprv si prejdeme základné výrazy. K podporovanému výrazu si vždy uvedemie príklad a v zátvorke pridáme aj uzol, ktorý ho predstavuje v zadefinovanej gramatike.

- Konštanty: **42** (*number*), **"hello"**/**'hello'** (*string*), **true** (*true*), **false** (*false*), **nil** (*nil*)
- Premenná: výraz, ktorý nie je medzi registrovanými konštantami napr. **a** (variable)
- Priradenie hodnoty do premennej: **a = 1** (*variable_assignment*)
- Priradenie hodnoty do lokálnej premennej: **local a = 1** (*local_variable_assignment*). Pozn. tento výraz nie je podporovaný aktuálnymi blokmi, ale uvážili sme, že bude vhodné túto základnú funkcionalitu pridať a blok zadefinovať
- Všetky aritmetické operácie: $1 + 2 - 3 * 4/5^6$ (*binary_expression*)
- Unárne operácie: logická negácia **not**, matematická negácia $-(1)$, dĺžka textového reťazca **#"string"** alebo poľa **#array** (*unary_expression*)
- Konkatenácia reťazcov: **a = Ähoj".. "Svet"**
- Logické porovnanie: **x == y**, **x ≃ y**, **x < y**, **x > y**, **x <= y**, **x >= y** sú všetky reprezentované uzlom (*binary_expression*)

- Logické operácie: **(x and y)** a **(x or y)** (*binary_expression*)
  (Pozn.: k logickým operáciám patrí aj negácia **not**, ale tá je u nás braná ako unárny operátor)
- Vytvorenie poľa: prázdneho **x = {}**, aj s počiatočnými hodnotami **c = {1,2,3,...}** (*table*)
- Indexácia do polí: **array[1]** (*variable_assignment*)
- Výrazy v zátvorkách: **(1 + 2)** (*parenthesized_expression*)
- Kľúčové výrazy **break** (*break_statement*) a **return** (*return_statement*)

Teraz sa pozrieme na zložitejšie kódové štruktúry a podobne ako pri výrazoch si uvedieme príklad a patričný uzol s ním spojený:

- Blokové štruktúry **if-then-else**: V tomto prípade je celý blok reprezentovaný uzlom (*if_statement*). Časti **elseif** a **else** sú jeho pod-uzlami a sú typu (*elseif_clause*) a (*else_clause*)

```
if podmienka then
    ...
elseif dalsia_podmienka then
    ...
else
    ...
end
```

- Numerický **cyklus for**: je spracovaný ako (*for_numeric_statement*)

```
for i = 1, 10, 1 do
    ...
end
```

- Generický **for cyklus**: predstavuje ho uzol (*for_generic_statement*)

```
for i, j in ipairs(pole) do
    ...
end
```

- **While** cyklus: reprezentovaný uzlom (*while_statement*)

```
while podmienka do
    ...
end
```

- Cyklus **repeat until**: uzol v AST (*repeat_statement*)

```
repeat
    ...
until podmienka
```

- Definícia funkcie: podporované funkcie s návratovou hodnotu aj bez nej. Ekvivalentným uzlom pre obe je (*function_definition_statement*).

```
function foo(parameter)
    return parameter
end
```

- Volanie funkcie: či už volanie kde očakávame hodnotu, alebo bez návratovej hodnoty, vždy je reprezentované uzlom (*call*)

```
a = foo(parameter)
```

- Ternárny operátor: Lua tento operátor implicitne nepodporuje, ale dá sa zapísať takto.

```
a = (x > y) and x or y
```

- Funkcie pre podporu LÖVE2D rámcu. Napr.

```
love.graphics.setNewFont( 0 )
```

- Komentáre: Ich spracovanie na bloky je podporované iba vo vnorených štruktúrach (funkcie, podmienky,...). Predstavuje ich uzol (*comment*).

```
-- Toto je komentar
```

Okrem týchto spomenutých výrazov je podporovaný vnorený kód (telo funkcie, cyklu, podmienky,...). Pre tie sa používa uzol typu (*block*) a jeho potomkovia sú všetky výrazy prítomne v danom bloku kódu. Je tiež možné používať cyklus v cykle alebo if-else podmienku v tele ďalšej if-else podmienky. Taktiež je možné tieto vnorené výrazy spolu kombinovať, pokiaľ je to syntaktický správne. Volanie funkcie ako argument pre inú funkciu je tiež povolené, pokiaľ je táto funkcia definovaná a je návratového typu. Čo sa týka počtu parametrov funkcie pri jej definícií, ten nie je nijako obmedzený.

V prílohách práce môžeme nájsť ukážku kódu jedného z testovacích reťazcov A, ktorý bol vygenerovaný pomocou ChatGPT [2]. Pre jeho vygenerovanie sme AI popísali podporovanú a nepodporovanú syntax, aby sme odskúšali čo najviac podporovaných funkcií, výrazov a štruktúr.

5.2.3 Nepodporovaná syntax

V tejto kapitole si popíšeme syntax jazyka Lua, ktorá je síce vo všeobecnosti používaná, ale v našom nástroji nie je jej podpora implementovaná. Pre niektoré prípady by to vyžadovalo lepšiu gramatiku, pre iné by bolo dostatočné zadefinovať bloky, ktoré by vedeli túto syntax reprezentovať. Oba tieto prípady by bolo potrebné dodatočne implementovať do nášho nástroja. O možnostiach rozšírenia viac v kapitole 5.6.

Nástroj teda nepodporuje nasledujúce výrazy a kódove štruktúry:

- Maticový prístup: je to obmedzené existujúcimi blokmi. Maticu je možné vytvoriť, ale pristúpiť k jej hodnote musí byť cez dva priradenia do premennej.

```
b = matica[1][2] -- Nie
a = matica[1], b = a[2] -- Ano
```

[2] https://chatgpt.com/

- Slovník

```
slovnik = {kluc = hodnota}
```

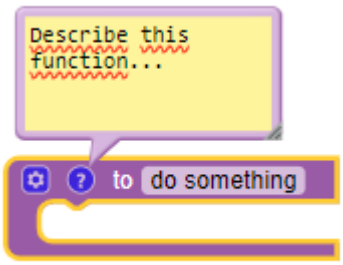
- Textový reťazec ako kľúč/index: je obmedzené existujúcimi blokmi. Ekvivalentný zápis bodkovej notácie, tiež nie je podporovaný.

```
array["kluc"]
a.kluc
```

- Priradenie do viacerých prmenných v jednom výraze

```
a, b, c = 1, 2, 3
```

- *Callback* funkcie[3]: je možné definovať, ale nie je možné im priradiť inú funkciu ako ich definíciu.
- Iterátory
- Lokálne funkcie
- Moduly
- Niektoré natívne funkcie ako: **tostring, type, next, tonumber,...**
- Blokové komentáre a blokové textové reťazce
- Komentáre v hlavnom bloku kódu budú ignorované. Generátory blokov pre definíciu nových funkcií totiž vždy generujú komentár implicitne priradený k tejto definícií funkcie 5.2. Ak by sme ich spracovali viedlo by to k duplikovaniu tohto komentáru pri každom behu.



Obr. 5.2: Tento komentár by bol duplikovaný v každom behu nášho nástroja.

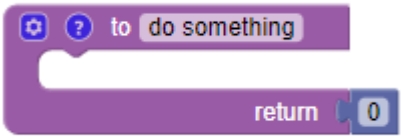
Spracovanie komentárov v tele **if** podmienky je nedokonalé. Ak je komentár prvý alebo posledný riadok v tele, je pri parsovaní spracovaný ako potomok *if_statement* uzla a nie ako potomok *block* uzla, predstavujúceho telo tohto výrazu. Implementáciu sa nám do určitej miery podarilo opraviť, ale sú prípady kedy môže zlyhať a blok pre daný komentár sa nevytvorí (napr. viac komentárov v tele za sebou).

[3] *Callback* funkcia je v prostredí LOVE2D volaná po splnení určitej podmienky, napríklad: užívateľ stlačil klávesu.

5.2.4 Poznámky k funkcionalite

V tejto kapitole si popíšeme menšie obmedzenia nástroja a jeho schopnosti spracovať uzly podporované gramatikou. Tieto obmedzenia sú prítomné buď z dôvodu zložitosti ich implementácie, alebo sme ich objavili až v neskoršom štádiu testovania a z krátkosti času ich označujeme ako *Fact of life*.

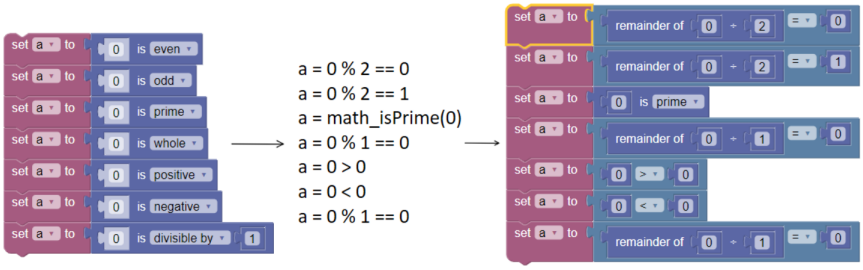
Pre spracovanie funkcie ako návratovej pri uzle *function_definition_statement* je potrebné, aby bol *return_statement* uzol prítomný v prvej generácii potomkov. Takže na konci definície funkcie musí byť explicitný riadok s **return** výrazom, inak nebude funkcia definovaná ako návratová a namiesto bloku *procedures_defreturn* sa vytvorí blok pre *procedures_defnoreturn*. To povedie ku chybám pri generovaní, keď výstup volania funkcie budeme chcieť priradiť napríklad do premennej. Taktiež výrazy vo funkcii po poslednom **return** výraze, už nie je možné správne spracovať a vytvorené bloky nebudú správne napojené do prostredia Blockly. To je z dôvodu, že **return** výraz je súčasťou bloku predstavujúceho definíciu funkcie 5.3 a vytvorené bloky nie je kam pripojiť.



Obr. 5.3: Posledný **return** výraz je súčasť bloku definíce funkcie.

Premenné a funkcie používané v kóde musia byť inicializované a definované predtým ako k ním budeme pristupovať a používať ich v kóde. Ak tomu tak nebude, nástroj nebude schopný vytvoriť patričné bloky reprezentujúce tieto výrazy.

Tento bod nie je ani tak obmedzenie ako vlastnosť nástroja. Jedná sa o to, že proces generovania kódu z blokov a proces generovania blokov z kódu nepredstavujú vzťah 1 k 1. Vygenerovanie blokov do kódu a následne vygenerovanie blokov zo získaného kódu nevedie k rovnakému stavu Blockly prostredia. Príklad môžeme vidieť na obrázku 5.4.



Obr. 5.4: Prevod nevedie vždy k rovnakému stavu Blockly prostredia. Výsledný kód je však ekvivalentný.

Je potrebné si dať pozor na menovaciu konvenciu. Ak je v kóde dva krát definovaná funkcia s rovnakým menom. Blockly zmení názov v druhej definícií na nový (napr. pridaním čísla za meno).

Generický **for** cyklus, ktorý v jazyku Lua využíva funkciu *ipairs()*, je spracovaný tak, že túto funkciu ignoruje a pozerá sa iba na jej parameter, ktorým by mal byť zoznam.

Jednou z najväčších výziev bolo vytvorenie blokov pre volanie funkcií. Hlavne pre funkcie, ktoré sú reprezentované blokmi z kategórií **Math**, **Text** a **Lists**. K spracovaniu niektorých funkcií z týchto kategórií (hlavne **Math**) bolo potrebné pristupovať špeciálne. Je teda pravdepodobné, že nie sú ošetrené všetky možné stavy a bloky sa nie vždy vygenerujú správne. Príkladom sú funkcie *math.sin/cos/tan*, ktoré musia byť volánе takto *math.sin(math.rad(vstup))*. Alebo blok *math_constrain* 5.5, ktorý generuje tento kód *math.min(math.max(50, 1), 100)*. Problematika spracovania funkcií je krátko popísaná v implementačnej časti 5.4 o zložitejších blokoch. V tejto časti sme sa snažili vypísať všetky obmedzenia nástroja o ktorých vieme, ale je možné že nám niečo uniklo.

Obr. 5.5: Prevod nevedie vždy k rovnakému stavu Blockly prostredia. Výsledný kód je však ekvivalentný.

5.3 Hlavné funkcie

5.3.1 createBlocks()

Je to hlavná funkcia, ktorú budeme volať v rozširovanej aplikácii a ktorú bude nástroj ponúkať (exportovať). Ako parameter by jej mal byť predložený vstupný kód a ak bude treba, tak aj Blockly prostredie. Jej pracovný postup môžeme popísať takto: V tejto časti sa inicializuje parser pre jazyk Lua pomocou nástroja *Tree-sitter*, vytvára sa distribútor, ktorý priraďuje uzly k ich príslušným podtriedam a načíta sa zoznam tried reprezentujúcich rôzne typy uzlov. Je načítaný vstupný zdrojový kód a ten sa potom pokúsi analyzovať pomocou parsera. Ak analyzovaný strom obsahuje nejaké uzly, snaží sa vytvoriť Blockly bloky z týchto uzlov a nastaviť ich do pracovného prostredia Blockly. To robí pomocou funkcie *createBlocklyCode()* 5.3.2, ktorej predá koreň AST a inicializovaný objekt triedy *Distributor*. Ak sa počas tohto procesu vyskytne chyba, vráti pracovné prostredie Blockly do jeho predchádzajúceho stavu.

5.3.2 createBlocklyCode()

Funkcia vytvára bloky podľa zozbieraných údajov z prvej generácie uzlov AST stromu. Prebieha v nej hlavná iterácia cez všetky uzly a každý uzol sa spracuje na príslušný blok podľa definícií v distribútorovi. Vytvorenie inštancie príslušnej triedy a jej použitie pre vytvorenie bloku v tejto funkcii vyzerá následovne:

```
1  const nodeObject = distributor.getNodeObject(node, lastBlock);
2  currentBlock = nodeObject.createBlock();
```

Bloky sú následne spojené dohromady podľa ich logického usporiadania a vzťahov medzi nimi. Funkcia zabezpečuje aj prípadný pohyb blokov pre lepšiu vizuálnu prezentáciu. Jedná sa o vytvorenie medzery medzi blokmi v týchto troch prípadoch: narazíme na *empty_line* uzol, vytvárame definíciu funkcie, alebo predchádzajúci uzol vytváral definíciu funkcie. Ak pri spracovaní nedôjde k chybe vráti na konci nový Blockly Workspace.

Pripomeňme si ukážku z kapitoly o Blockly 2.2, kde sme si popisovali ako vytvoriť *controls_whileUntil* blok, vyplniť jeho políčka a napojiť blok s podmienkou do jeho vstupu. Neukázali sme si však, ako napojiť blok do tela tohto cyklu. Keďže telo cyklu môže pozostávať z viacerých blokov a dokonca aj blokov s ďalším telom na vyplnenie, budeme k tomu využívať rekurzívne volanie tejto funkcie. V tomto prípade okrem koreňového uzla a distributóra, však funkcia obdrží ďalšie dva parametre. Tie sú *bodySetter* držiaci meno *input_statement* vstupu bloku a *outerBlock* držiaci blok, do ktorého budeme bloky napájať. Pri volaní tejto funkcie z *createBlocks* majú tieto parametre hodnoty: *undefined* a *null* aby funkcia vedela, že sa nenachádza v tele žiadneho výrazu. Vytvorenie blokov tela by teda pre spomínaný **while** blok vyzeralo takto:

```
1  createBlocklyCode(uzolTelaWhile, this.distributor, "DO", whileBlock);
```

Do vstupného napojenia tohto bloku bude napojený iba prvý blok. Nasledujúci blok sa pripojí na predošlí a na neho zasa nasledujúci atď. Namiesto vstupného a výstupného spojenia musíme pre takéto prepojenie blokov využiť *nextConnection* predchádzajúceho bloku a *previousConnection* aktuálne vytváraného bloku. Týmto spôsobom sú prepojené skoro všetky bloky s výnimkou vyššie spomínaných prípadov.

```
1  currentBlock.previousConnection.connect(lastBlock.nextConnection);
```

Connection	Image
Output Výstupné spojenie	
Input Vstupné spojenie	
Previous Predchádzajúce spojenie	
Next Nasledujúce spojenie	

Obr. 5.6: Ilustračný obrázok pre jednoduchšie pochopenie o akých vstupoch a výstupoch v tejto časti hovoríme.

5.3.3 MyNode podtriedy

Nasledujúca sekcia práce, nám spojí popisované uzly s implementovanými podtriedami *MyNode* triedy. Každá trieda obdrží objekt uzlu určitého typu obsahujúci všetky potrebné informácie pre jeho spracovanie. Týchto tried je dosť veľa, takže si tu bližšie ukážeme iba niektoré zaujímavejšie z nich a ukážeme si ako spracúvajú uzly, ktoré sú im priradené.

Triedy, ktoré boli čo sa týka implementácie spracovania uzla a vytvorenia bloku jednoduchšie:

- **BooleanNode:** pre uzly (*true*) a (*false*)
- **BreakNode:** pre uzol (*break_statement*)
- **EmptyLineNode:** pre uzol (*empty_line*)
- **IdentifierNode:** pre uzol (*identifier*)
- **NilNode:** pre uzol (*nil*)
- **NumberNode:** pre uzol (*number*)
- **ParenthesizedExpressionNode:** pre uzol (*parenthesized_expression*)
- **StringNode:** pre uzol (*string*)
- **CommentNode:** pre uzol (*comment*)
- **ShebangNode:** pre uzol (*shebang*)

Pre tieto triedy s výnimkou *ParenthesizedExpressionNode* platí, že sú zodpovedné za uzly, ktoré môžeme brať ako listy stromu. To znamená, že majú maximálne jedného alebo žiadneho potomka. Vytvárame z nich elementárne bloky, alebo nevytvárame žiaden blok a trieda je potrebná iba pre predanie informácie.

Napríklad trieda *EmptyLineNode* iba vracia blok vytvorený v predchádzajúcej iterácií pre predchádzajúci uzol. Aj keď to vyzerá byť zbytočné a mohli by sme prázdne riadky jednoducho ignorovať už v parsovacom procese, túto informáciu potrebujeme ak chceme vytvoriť presnejšiu reprezentáciu kódu s medzerami medzi blokmi.

Spomínaná trieda *ParenthesizedExpression* je výnimkou, pretože uzol, ktorý má na starosti má troch potomkov: (ľavú zátvorku), (*napr. binary_expression*), (pravú zátvorku). Rovnako sama nevytvára žiaden blok, namiesto toho iba posunie svojho stredného potomka pre ďalšie spracovanie. Vďaka tomu nám dovoľuje vytvoriť blokovú reprezentáciu pre aritmetické výrazy so zachovaním správnej priority jednotlivých výrazov.

Nasledujúce triedy si pre správne spracovanie uzlov vyžadovali zložitejšiu implementáciu:

- **BinaryExpressionNode:** pre uzol (*binary_expression*)
- **ForGenericNode:** pre uzol (*for_generic_statement*)
- **ForNumericNode:** pre uzol (*for_numeric_statement*)
- **FunctionCallNode:** pre uzol (*call*)
- **FunctionDefinitionNode:** pre uzol (*function_definition_statement*)

- **IfElseIfElseNode:** pre uzol (*if_statement*)
- **ReturnNode:** pre uzol (*return_statement*)
- **TableNode:** pre uzol (*table*)
- **UnaryExpressionNode:** pre uzol (*unary_expression*)
- **VariableAssignmentNode:** pre uzol (*variable_assignment*)
- **VariableNode:** pre uzol (*variable*)
- **WhileNode:** pre uzol (*while_statement*)

Tento zoznam pozostáva z tried, ktoré sa starajú o uzly s dopredu neznámym počtom potomkov. Tým máme na mysli generačných potomkov. Teda nevieme s určitosťou povedať či je potomok uzlom listovým, alebo je to uzol s ďalšími potomkami. Ako môžeme vidieť na príklade nižšie 5.1, uzol pre binárny výraz má vždy rovnaký počet potomkov, ale jeho potomkom môže byť ďalší binárny výraz atď.

```
(binary_expression
    left: (binary_expression left: (...) right: (...))
    right: (...)
)
```

Výpis 5.1: Ukážka vnorených uzlov. Pre jednoduchosť sme vynechali tretieho potomka tohto uzlu predstavujúceho operátor.

5.3.4 Distributor trieda

Objekt tejto triedy slúži na distribúciu uzlov do príslušných podtried na základe ich typu. Definuje a inicializuje sa na začiatku spracovania výstupu parseru. Pre registráciu triedy a typu uzlu, na ktorý bude naviazaná je potrebné v *MyNNode* podtriede implementovať statickú metódu *register()* 5.2 a tú zavolať s inicializovaným objektom *Distributor*, ktorý budeme používať.

```
static register(distributor) {
    distributor.registerSubclass("typ_uzlu", this);
}
```

Výpis 5.2: Ukážka implementácie metódy pre registráciu triedy

Takto registrovaná trieda sa uloží do *nodeToSubclassMap* mapy a objekt ju bude môcť invokovať, keď obdrží uzol, ktorý si trieda registrovala. K tomu potrebujeme zavolať metódu *getNodeObject(node, lastBlock)*. Z parametru *node* si metóda vytiahne jeho *type* atribút a podľa neho vráti z registračnej mapy korešpondujúci objekt. Ak pre poskytnutý typ neexistuje žiadna trieda, vyhodí chybu.

Okrem distribúcie uzlov tento objekt slúži aj na zbieranie informácií. Tie sú potom poskytované ďalším triedam, ktoré ich potrebujú. Príkladom sú informácie týkajúce sa novo definovaných funkcií, ktoré sú potrebné pri ich volaní. Konkrétne sa jedná o uzly parametrov, mutácii bloku (ak nejaká bola vykonaná) a o tom, či je funkcia návratového typu. Ukladajú sa do mapy *customFunctions* 5.3.

```
{
"meno_funkcie": {
    "params": params,
    "mutation": mutationXml,
    "return": isReturnNode
    }
}
```

Výpis 5.3: Ako sú ukladané dôležité informácie o nových funkciách v mape customFunctions

Taktiež obsahuje informácie a mapy potrebné k správnemu generovaniu blokov funkcií s vlastnými definíciami blokov. Kvôli týmto informáciám je dôležité, aby triedy používali v jednom behu generovania ten istý objekt a preto si ho medzi sebou posúvajú ako parameter.

5.4 Generovanie Blockly blokov

Snahou bolo nevytvárať nové bloky, ale generovať tie, ktoré už boli definované v rozširovanej aplikácií. Tieto bloky sú rozdelené do kategórií: **Logic**, **Loops**, **Math**, **Text**, **Lists**, **Variables**, **Functions** a funkcie rámcu LÖVE. Bloky poslednej kategórie sú ďalej rozdelené, ale pre naše potreby, sme si ich rozlíšili iba ako definičné a volacie. Volacie sa potom ešte delia na funkcie s návratovou hodnotu a bez návratovej hodnoty. Podľa tohto rozdelenia boli potom aj pripravené testovacie sety.

Jednoduché bloky

Tieto bloky sú vytvárané triedami z prvého zoznamu tried, ktorý sme videli v predchádzajúcej kapitole. Sú to bloky predstavujúce výrazy ako: číslo, pravdivostná hodnota, textový reťazec, atď. Pre ich implementáciu je potrebné akurát blok inicializovať, nastaviť mu správnu hodnotu v jeho políčku, alebo ho napojiť ako vstup do iného bloku. O toto prepojenie sa však v našej implementacií vždy stará blok do ktorého tento blok napájame. Pre príklad, implementácia *createBlock()* metódy pre triedu *NumberNode* je nasledovná:

```
1 createBlock() {
2     let numberBlock = this._initBlock('math_number');
3     numberBlock.setFieldValue(this.node.text, 'NUM');
4     return numberBlock;
5 }
```

Zložité bloky

O tvorbu týchto blokov sa starajú triedy z druhého zoznamu vypísaného v predchádzajúcej kapitole. Jedná sa o bloky, ktoré predstavujú kódové štruktúry ako: cykly, podmienky, aritmetické výrazy, definície funkcií, atď. Väčšina týchto blokov má viacero vstupov, políčok, vyžaduje mutáciu 5.4, alebo pre vyplnenie vstupu vyžaduje rekurzívne volanie funkcie *createBlocklyCode()*.

Na implementáciu boli najzložitejšie triedy *FunctionDefinitionNode* a *FunctionCallNode*, ktoré generuje bloky pre definície a volania funkcií. Tieto triedy sú na sebe závisle v tom zmysle, že vygenerovaný blok pre volanie funkcie musí zodpovedať bloku vygenerovanému pre definíciu tejto funkcie. Doteraz sme si pri jednom uzle vystačili s menším

množstvom blokov, ktoré mohli byť pri ich spracovaní vytvorené. Problém pri spracovaní uzlov *function_definition* a *call* bol, že máme veľké množstvo blokov pre ich reprezentáciu. To sú konkrétne: Blockly definície pre nové funkcie, bloky definované pre prácu s LÖVE rámcom, funkcionálne bloky definované pre kategórie **Math**, **Text** a **Lists**. Bloky z týchto kategórií sú definovane rôzne, majú rozličné vstupy pre parametre a k ich tvorbe bolo potrebné pristupovať osobitne[4]. Každý blok má taktiež iné meno, ktoré sa používa pre jeho inicializáciu.

Riešili sme to pridaním parametrov do triedy *Distributor*. Každý parameter predstavuje skupinu funkcií, ktorých bloky sa generujú podobným spôsobom a drží v sebe mapu s menami funkcií prepojenými s typmi blokov, ktoré generujú. Niektoré parametre v sebe držia ešte dodatočné informácie ako názov vstupu, alebo hodnotu pre nastavenie parametru podľa názvu funkcie. Tieto parametre sú: *functionsOnlyDef*, *functionsOnlyCall*, *functionsReturn* a *functionsReturnLOVE*. Prvý z nich sa použije pri definíciách a zvyšok pri volaniach funkcií.

Na nasledujúcej ukážke kódu z implementácie *FunctionDefinitionNode* triedy môžeme vidieť použitie mapy *functionsOnlyDef* v *createBlock()* metóde. Vďaka nej zistíme či vytvárame blok pre známu, alebo neznámu funkciu. Zvyšné spomenuté mapy sú využité podobným spôsobom v implementácii *FunctionCallNode*, ale podmienka je trochu zložitejšia a ukážka by sa nám do textu práce vkladala ťažko.

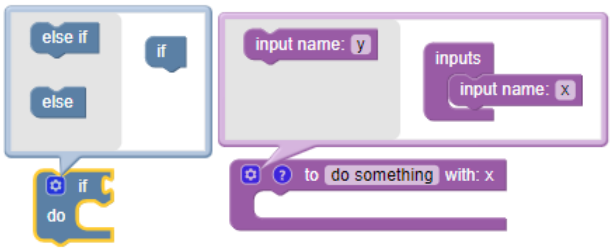
```
1 if (this.distributor.ignore.includes(this.key)){
2     return this.lastBlock;
3 }
4 else if (Object.keys(this.distributor.functionsOnlyDef).includes(this.key.trim())) {
5     tempBlock = this._createKnownFunctionDefinitionBlock();
6 } else {
7     tempBlock = this._createUnknownFunctionDefinitionBlock();
8 }
```

Prvá podmienka v tomto kóde nám tiež ukazuje, ako sú preskakované niektoré definície funkcii o ktorých ešte budeme hovoriť v kapitole 5.5.

Bloky vyžadujúce mutácie

Niektoré bloky v prostredí Blockly ponúkajú možnosť mutácie pomocou mutátoru. Mutátor je špeciálny druh rozšírenia, ktoré do bloku pridáva vlastnú serializáciu a niekedy aj používateľské rozhranie. Medzi bloky s touto možnosťou patria: *controls_if*, *lists_create_with*, *lists_create_empty*, *procedures_defreturn*, *procedures_defnoreturn* a ďalšie. Mutácie nám dovoľujú používať rovnakú definíciu bloku aj v prípadoch, keď potrebujeme iný počet vstupov, alebo parametrov. Na obrázku 5.7 môžeme vidieť príklad toho ako sa mutátor používa v užívateľskom rozhraní Blockly. Pre *controls_if* blok nám dovoľuje pridať *else_if* a *else* doložky. Do bloku *procedures_defnoreturn*, môžeme pridať nové parametre funkcie. Podobne sa pristupuje aj pri mutovaní iných blokov, ale nie je tam potrebné si ukladať žiadne informácie.

[4]Tieto rozdiely nie je možné zachytiť v parsovacom procese, keďže sa jedn´o špecifikúm blokov. Preto sme museli tieto problémy riešiť až tu.



Obr. 5.7: Bloky *controls_if* a *procedures_defnoreturn* s rozkliknutým rozšírením mutácie v užívateľskom rozhraní Blockly.

Teraz si na príklade bloku *procedures_defreturn* ukážeme, ako by sme do neho pridali mutáciu z kódu. Povedzme, že chceme spracovať zdrojový kód s definíciou funkcie, ktorá vyzerá následovne:

```
function add(a,b)
  return a + b
end
```

V zdrojovom kóde sa mutácia definuje pomocou XML zápisu. Vidíme, že funkcia má dva parametre a teda potrebujeme pridať mutáciu s dvoma XML prvkami <arg>. Musíme im nastaviť správny atribut *name*, ktorý prezentuje ich meno. Nakoniec tieto prvky uzatvoríme do <mutation> XML prvku. Tým sme postavili mutáciu, treba ju však ešte previesť z textu na DOM (Document Object Model) objekt a aplikovať na inicializovaný blok. K tomu využijeme metódy *Blockly.utils.xml.textToDom()* a *domToMutation()*. Pre bloky *procedures_defreturn* a *procedures_defnoreturn* je tento proces v našom kóde rovnaký a vyzerá takto:

```
1  if(this.node.child(3).type === 'parameter_list') {
2      let args = ""
3      for (const param of params) {
4          args = args + '<arg name="' + param.text + '"></arg>';
5          this._createVariableInWorkspace(param);
6      }
7      const mutationXml = Blockly.utils.xml.textToDom(`
8          <mutation>` + args + `</mutation>
9      `);
10     // Apply the mutation to the block
11     functionBlock.domToMutation(mutationXml);
12     // Save mutation to apply when you call this function
13     this.distributor.customFunctions[this.key] = {"params": params, "mutation":
           mutationXml, "return": isReturnNode};
14 }
15 else{
16     // Save information, that there is no mutation, to apply when you call this function
17     this.distributor.customFunctions[this.key] = {"params": null, "mutation": null, "
           return": isReturnNode};
18 }
```

Najprv zistíme, či má funkcia parametre. Potom vytvárame <arg> pre každý parameter s príslušným menom. Parametre vložíme do <mutation> objektu a prevedieme na DOM objekt. Ten aplikujeme na vytváraný blok. Nakoniec si uložíme informácie o mutácii funkcie s týmto menom do inštancie triedy *Distributor*, aby sa nám ľahšie vytváral blok pre volanie tejto funkcie.

5.5 Spätné generovanie kódu

Prevod z blokov do kódu prebieha v Blockly pomocou tzv. generátorov. Ak chceme použiť nové nami definované bloky, je potrebné tieto generátory implementovať. V tejto práci používame zväčša bloky z Blockly knižnice, alebo bloky, ktoré boli implementované v predchádzajúcej práci (reprezentujúce funkcie pre prácu s LOVE2D rámcom). Obe tieto skupiny majú generátory pre svoje bloky už definované, ale aj napriek tomu bolo potrebné pre nich urobiť pár zmien.

Generátory z rozširovanej aplikácie bolo potrebné upraviť na nový Blockly zápis. A teda syntax na prvých troch riadkoch ukážky 5.5 bola nahradená syntaxou z posledných troch riadkov:

```
1  import Blockly from 'blockly' // stare
2  Blockly.JavaScript.love_draw = function(block) { // stare
3  Blockly.JavaScript.valueToCode // stare
4
5  import lua from 'blockly/lua' // nove
6  lua.luaGenerator.forBlock["love_draw"] = function(block) { // nove
7  lua.luaGenerator.valueToCode... // nove
```

Bolo potrebné implementovať aj novú definíciu pre dva generátory už implementované v Blockly knižnici. Prvým bol generátor *procedures_ifreturn*, ktorý je v aplikácií používaný, ako jednoduchý návratový blok a jeho podmienková časť nie je potrebná. Druhým bol generátor bloku *controls_for*, ktorý pred úpravou viedol ku cyklickému generovaniu nových podmienok. Ak boli ako parametre v tomto cykle použité premenné, tak sa pri vygenerovaní kódu z blokov, pred tento cyklus vygenerovala podmienka kontrolujúca tieto premenné. Tá by bola pri spätnom prevode z kódu do blokov vygenerovaná ako blok. Pri ďalšom prevode z blokov do kódu by sme už vygenerovali dve takéto podmienky. Preto bolo potrebné generátor implementovať bez tejto kontroly.

Pri generovaní kódu z vytvorených blokov pre volanie niektorých funkcií sa vyskytol podobný problém s cyklickým generovaním. Tieto bloky okrem svojho volania generovali aj kód definície tejto funkcie. V tomto prípade, sme nemohli túto definíciu z procesu generovania odstrániť, pretože funkcia bez svojej definície je v kóde nepoužiteľná. Preto sme do triedy *Distributor* pridali pole *ignore* so všetkými názvami funkcií, pre ktoré sa má generovanie blokov ich definície preskočiť 5.4. Príkladom takejto funkcie je *math_sum()*.

Okrem toho sme ešte implementovali dve jednoduché generátory pre nami pridané bloky: *shebang* a *comment*.

Ďalšou raritou medzi blokmi funkcií boli niektoré bloky z kategórie *Graphics*. Tie sa generovali vo vnútri poľa a to by ich robilo veľmi problematickými pre spracovanie. Takáto funkcia je napríklad: *a = love.graphics.getBackgroundColor()* a jej blok sa generoval takto *a = {love.graphics.getBackgroundColor()}*. Preto sme ich generátory prepísali a blok tejto funkcie je odteraz v Blockly editore potrebné správne priradiť ako vstup do bloku reprezentujúceho pole.

5.6 Rozšíriteľnosť

Nástroj je možné rozšíriť nasledujúcimi spôsobmi:

1. Pridaním triedy pre uzol už podporovaný v gramatike.
2. Pridaním nového pravidla pre uzol do gramatiky a následným vytvorením triedy pre jeho spracovanie.
3. Úpravou gramatiky, pridaním potrebných tried a prípadnou zmenou implementácie tried ovplyvnených zavedením tejto zmeny.

Pridanie novej triedy pozostáva z: jej vytvorenia ako podtriedy rozširujúcej *MyNode* triedu, implementácie povinných metód *createBlock()* a *register()*, pridaním volania registračnej metódy do funkcie *registerAllSubclasses()*. Pre pridanie nového uzlu do pravidiel gramatiky by sme potrebovali správne upraviť *grammar.js* súbor a pomocou *tree-sitter-cli* z neho vygenerovať nový *.wasm* súbor. Tu by bolo potrebné dať si pozor, či sa pridaním nového uzlu neovplyvnili už funkčné časti nástroja a prípadne ich upraviť. Ak medzi ponúkanými blokmi nebude existovať nijaký, ktorý by dokázal spracovaný kód reprezentovať, bude potrebné si nový blok definovať.

Ako príklad, môžem uviesť pridanie podpory pre *repeat_statement* s triedou *RepeatNode*, ktorá nebola explicitne potrebná pre túto prácu. Vystačili by sme si s podporou uzla *while_statement* spracovaného *WhileNode* triedou. Generátor bloku vytváraného vo *WhileNode* triede generuje kód **while** cyklu, pre obe možnosti, REPEAT aj WHILE. Pridaním triedy *RepeatStatement* sme umožnili spracovať **repeat** cyklus. Ale po jeho vygenerovaní na blok a následnom spätnom vygenerovaní kódu z tohto bloku, dostaneme **while** cyklus. Funkcionálne budú rovnaké, zmení sa len zápis.

Čo sa týka vylepšení nástroja, jednou z možností by bolo zbaviť sa funkcie *createBlocklyCode()* a implementovať jej funkcionalitu do hlavnej triedy *MyNode*. Tá by si potom pomocou *Distributor* triedy registrovala koreňový uzol (*chunk*) a (*block*). To by nás priviedlo bližšie k čistému objektovému riešenie, bolo by však nutné doriešiť správne napájanie blokov a podobné záležitosti. Následne by sme tiež mohli pomocné funkcie prepísať ako metódy jednotlivých tried.

Kapitola 6

Integrácia aplikácie do Love-blocks-web

V tejto kapitole si popíšeme, ako konkrétne malo byť rozšírenie integrované do Love-blocks-web aplikácie. Najprv sa pozrieme na rozbehnutie aplikácie 6.1. Ďalej si v rovnakej podkapitole popíšeme, na aké problémy sme narazili, a ako sme ich skúsili riešiť 6.1.3. Nakoniec odôvodníme, prečo sa integrácia nepodarila a pozrieme sa na alternatívne riešenie pomocou API serveru.

6.1 Postup

Aplikáciu sme si pomocou *git clone* príkazu stiahli k nám do počítača. Stiahli a pripravili sme si `Docker Desktop`. Potom sme postupovali podľa inštrukcií v README súbore, ktorý podrobnejšie popisuje postup nasadenia aplikácie. Vo všeobecnosti tento proces zahŕňa nastavenie premenných prostredia, vytvorenie kontajnerov Docker, konfiguráciu MySQL, inicializáciu aplikácie Laravel a inštaláciu závislostí. V tomto procese sme sa potýkali s mnohými problémami a trval omnoho dlhšie ako sme predpokladali. Týchto prekážok bolo viacero, ale popíšeme si pár, ktoré zabrali najviac času.

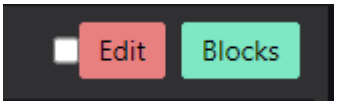
Prvým z väčších problémov bolo, že po nasadení sa server správal akoby nebol inicializovaný a vo webovom prehliadači sme sa k nemu nevedeli dostať pomocou *localhost*-a. Keď sme ho chceli spustiť, dostali sme chybu, že na tomto porte už niekto počúva, čo znamená, že aplikácia bežala a niekde nastala chyba. V tomto prípade sme konkrétnu chybu neidentifikovali a problém sa vyriešil až po niekoľkých pokusoch o nasadenie aplikácie na čisté prostredie po jej vymazaní.

Ďalším problémom bolo, že po nainštalovaní všetkých balíkov pomocou *docker-compose run –rm composer install*, zlyhával *post-autoload-dump event* a vracal chybu 255. Po nepodarených pokusoch s aktualizáciou verzie *composer*-u alebo inštalovaných balíčkov, nakoniec pomohol postup: *composer install -> composer update -> composer install*. Po poslednom príkaze už spomínaný *post-autoload-dump event* prešiel bez chyby.

Posledným z väčších problémov boli npm balíčky. Po spustení *docker-compose run –rm npm install*, hlásili mnohé balíčky problémy. Tie sme sa taktiež snažili vyriešiť aktualizáciou verzií, ale to pri množstve závislosti, ktoré tento projekt má bolo ručne nemožné. Nakoniec pomohol `npm audit fix`, po ktorom síce stále dostaneme pár upozornení, ale aplikácia naštartuje. Táto chyba bola z časti na našej strane, keďže sme spomínaný npm príkaz nevyskúšali hneď, ale namiesto neho sme skúsili `npm audit fix -force`. Ten urobil aj

nežiadúce zmeny a po jeho zlyhaní opraviť tento problém sme začali riešenie hľadať na inom mieste.

Keď už máme aplikáciu rozbehnutú, začneme aktualizáciou kódu používajúceho Blockly knižnicu. Niektorá funkcionalita sa premiestnila z 'blockly' do 'blockly/lua' balíčka a pozmenila sa jej notácia. Potom, ako sme si popísali v návrhu 4.3, pokračujeme pridaním spomínaného checkboxu do užívateľského rozhrania a logiky s ním spojenej do akcie tlačidla *Blocks*. Následne si overíme, či sa dokážeme z nami upravovaného súboru *Show.vue* dostať k potrebným hodnotám. Aktuálne Blockly workspace aj zdrojový kód boli prístupné a tak sme vyskúšali napojiť proces parsovania zdrojového kódu pomocou *web-tree-sitter* 6.1.1. Tu sme však narazili na ďalšie problémy.



Obr. 6.1: Screenshot pridaného checkboxu z lokálne rozbehnutej aplikácie.

6.1.1 Web-tree-sitter zlyhanie

Pre pridanie tejto funkcionality existujú dve možnosti. Prvou je stiahnuť *web-tree-sitter* pomocou npm ako závislosť a používať parser odtiaľ. Druhou možnosťou je stiahnuť súbory *tree-sitter.js* a *tree-sitter.wasm* a používať ich ako súčasť projektu [4]. Oba tieto prístupy pre nás nakoniec zlyhali.

V projekte je pre prácu so závislosťami využívaný *webpack* [10], čo je nástroj, ktorý spracuje aplikáciu, interne zostaví graf závislostí z jedného alebo viacerých vstupných bodov a potom spája všetky moduly, ktoré projekt potrebuje, do jedného alebo viacerých zväzkov. Tieto zväzky sú statické prostriedky, z ktorých sa servíruje obsah. Bolo potrebné upraviť konfiguráciu tohto nástroja v súbore *webpack.config.js* tak, aby sme vyriešili dostupnosť *.wasm* súboru pre parser pri inicializácií na klientskej strane aplikácie. Potom, ako sa nám to podarilo s využitím *CopyWebpackPlugin* plugin-u, parser už dokázal súbor *.wasm* nájsť a začal ho načítať. Tu sme však narazili na problém s nedefinovanými C funkciami vo *.wasm* súbore. Cez tento problém sme sa nedostali ani po použití najjednoduchšej gramatiky pre vygenerovanie *.wasm* súboru. A nepomohlo ani pridanie C-čkových súborov (*parser.c* a *scanner.c*) vygenerovaných z rovnakej gramatiky. Zlyhal taktiež aj pokus s implementovaným vlastným npm balíčkom.

6.1.2 Node tree-sitter zlyhanie

Po neúspechu s *web-tree-sitter* verziou parseru sme skúsili použiť node moduly *tree-sitter* a *tree-sitter-lua*. Tieto balíčky obsahovali všetky potrebné súbory a mali by sa dať použiť v našom prípade. Pri nich sme však narazili na iný problém. Vygenerované C-čkové parsere potrebujú pre funkcionality z iných jazykov tzv. bindings (v našom prípade pre JavaScript). Tie pre funkcionalitu potrebujú modul *node-gyp-build*, ktorý je natívny node modul. Natívny v tomto kontexte znamená, že je navrhnutý pre použitie v Node.js prostredí a spolieha sa na funkcie a API typické pre toto prostredie. To znamená, že nie je priamo kompatibilný s prostredím webových prehliadačov, a je obtiažne ho použiť z klientskej strany aplikácie, pretože Node.js a webový prehliadač majú odlišne prostredia behu. Z týchto dôvodov sa nám tieto moduly nepodarilo použiť ani po vyskúšaní rôznych spôsobov ako tieto problémy obísť (napr. *brewserifiy*).

6.1.3 Ťažkosti práce s existujúcim riešením

V tejto časti by sme chceli podotknúť, že nasadenie tejto aplikácie a následná práca s ňou zabrali omnoho viac času ako sme predpokladali. Pracovalo sa na PC s 8GB RAM, čo je pre aplikáciu bežiacu na dockeri, spustené IDE a webový prehliadač málo. Aj po prechode na lepší stroj s dvojitou RAM kapacitou sme neprestali pracovať v spomalenom režime a zlepšila sa iba odozva IDE prostredia. Nasadenie a skontrolovanie akejkoľvek zmeny zabralo v priemere 15-20 minút. Podľa online diskusií, ktoré sme pri hľadaní riešenia tohto problému čítali, by mohlo isť o problém komunikácie Linuxu s Windowsom v docker prostredí. Riešením by mohlo byť, pracovať s aplikáciou v Unixovom operačnom systéme (to sme však nestihli otestovať). Ďalšia nepríjemnosť, bol spôsob, akým sú súbory spracované pred spustením aplikácie pomocou `webpack` nástroja. Všetky zdrojové súbory sú zabalené do 1-2 súborov v upravenom textovom formáte, čo značne sťažuje hľadanie chýb v zdrojovom kóde a použitie vývojového prostredia vo webovom prehliadači.

Kvôli vyššie spomenutým ťažkostiam s kompatibilitou parseru použitého v našom nástroji, sme neboli schopný urobiť posledný krok v tejto práci a hotové rozšírenie sa nepodarilo integrovať do aplikácie, ktorá k tomu už bola pripravená.

6.2 Alternatíva pomocou API

V posledných týždňoch práce sme ako alternatívne riešenie implementovali jednoduchý server, ktorým sme chceli obísť spomínané problémy. Server sa spúšťa pomocou *node app.js* a funguje ako jednoduché API bežiace na *http://localhost:PORT* v prostredí, v ktorom je spustená aplikácia. Na zvolenom porte počúva a prijíma POST správy obsahujúce zdrojový kód pre spracovanie. Prípadne správy môžu obsahovať prostredie Blockly serializované do JSON objektu. Táto funkcionalita je využívaná aj v nástroji pre navodenie počiatočné stavu pre generovanie:

```
1  const state = Blockly.serialization.workspaces.save(workspace);
2  Blockly.serialization.workspaces.load(state, workspace);
```

V našom vývojovom prostredí sme pre prácu s ním implementovali jednoduchého klienta a pridali tlačidlo pre jeho invokáciu, aby sme mohli odskúšať schodnosť tohoto riešenia.

V pláne bolo zdrojový kód zasielať API a spracovať ho na strane serveru. Potom vytvoriť odpovedajúce bloky a odpovedať klientovy s aktualizovaným stavom Blockly prostredia, ktoré by si mohol jednoducho nastaviť. To však nebolo možné, pretože na to aby Blockly prostredie mohlo uchovávať informácie o pozícií a spojení blokov, potrebuje byť priradené k HTML prvku webovej stránky. Ak nie je ku žiadnemu priradené, potom nie je možné bloky správne inicializovať pomocou *block.initSvg()* a pracovať s nimi. Bloky sa teda vygenerovali, ale vrátené prostredie bolo nepoužiteľné.

Taktiež sme skúsili poslať naspäť iba výsledný stromový objekt parseru. Zbavili sme sa tým závislosti aplikácie na *Tree-Sitter* parsery a vytváranie blokov by sme už prevádzali na klienktskej strane, teda v rozširovanej aplikácií s inicializovaným Blockly prostredím. Stromový objekt však bolo pred poslaním potrebné serializovať. *Tree-Sitter* stromové objekty však majú v niektorých parametroch cyklické závislosti, ktoré sú v implementácií nášho nástroja pomerne často využívané.

To spôsobilo problémy, pretože ak chceme zaslať objekt s cyklickými závislosťami, nepomôže nám ani jeho serializácia do JSON-u. Vyskúšali sme rôzne dostupné serializačné balíčky (*msgpack-lite*, *serialijse*, *circular-json*, *JSON.stringify()*,...) a nakoniec aj vlastnú

implementáciu serializácie. Ale prevod bol buď stratový, alebo ponechával cyklické závislosti. Aj keď sa nám podarilo JSON reprezentáciu stromu dostať ku klientovy, ten hlásil chýbajúce parametre (napr. *node.firstChild.firstChild* bol nepoužiteľný). Keď sme sa snažili zaslať JSON s cyklickými závislosťami, *res.json(serializedTree)* pre zasielanie odpovede klientovy nám hlásil chybu. Použiteľná bola serialiácia kde sme sa ná už serializované uzly odkazovali už iba pomocou ich id-čka, čo vyriešilo cyklické zavislosti, ale tento prístup by vyžadoval kompletnú refaktorizáciu práce s uzlami v našom nástroji.

Popri API sme ešte skúsili využiť *Google Cloud Functions*[1], ale narazili sme na rovnaké problémy.

S našim nástrojom sme neplánovali pracovať týmto spôsobom, a táto možnosť bola odskúšaná, až keď sa nám nástroj nepodarilo integrovať priamo. Využitie API však ostáva možnosťou, ktorá by sa mohla dať použiť po správnych úpravách nástroja, na ktoré nám však už neostal čas.

[1] `https://cloud.google.com/functions`

Kapitola 7

Testovanie

Testovanie prebiehalo prevažne vo vývojom prostredí 5.1. Pôvodne sme plánovali vytvoriť si súbory obsahujúce útržky zdrojových kódov, ale skončili by sme so zbytočne veľkým adresárom. Potom sme chceli použiť JSON súbory a v nich si vytvoriť objekty s reťazcami, ale ten nepodporuje viacriadkové reťazce, ktoré by boli vhodné pre otestovanie rozsiahlejších zdrojových kódov. Nakoniec sme skončili pri jednoduchom JavaScript súbore *tests.js* s objektom *codeSnippets*, ktorý obsahuje mapy so všetkými testovacími reťazcami. Tieto testy sme rozdelili do kategórii podľa blokov ktoré generujú. Tieto kategórie sú: *function_calls*, *function_definitions*, *variables*, *math*, *text*, *lists*, *logic*, *loops*, *custom_functions*, *big* a *both_way_check*. Dokopy pokrývajú všetky bloky, ktoré máme definované a dokážeme vygenerovať. V jednotlivých kategóriach najprv vždy testujeme primitívny zdrojový kód pre vygenerovanie všetkých blokov v ich najjednoduchšej podobe. Potom ich inkrementálne spájame dokopy a testujeme ich v zložitejších situáciach. Nakoniec pre testy v *big* časti spájame všetky kategórie dokopy a vytvárame kód, ktorý predstavuje reálne prostredie.

Pre načítanie a spúšťanie jednotlivých testov sme mali v UI definované tlačidlo a políčko, kde sme si zvolili aký súbor testov chceme spustiť. Iterovali sme cez všetky textové reťazce dostupné v danom súbore a použili ich ako vstup pre parser a teda aj pre náš nástroj. Pre jeho verziu ako dodatočného modulu by bolo vhodné tieto testy adaptovať a rozšíriť o testy na pozadí.

V testoch testujeme najmä jednosmerné generovanie blokov, keďže prevod naspäť nedodržuje vzťah 1-ku-1, ale bolo vytvorených aj pár testov pre porovnanie vstupného a vygenerovaného kódu. Tieto testy boli primárne vytvorené pomocou kódu, ktorý sme si vygenerovali z blokov v našom vývojom prostredí a simulujú teda reálne prostredie s obojstranným prevodom. Pre ilustráciu sme niektoré testy v *both_way_check* ponechali ako nesprávne, aby sme poukázali na to, že rozdiely medzi vygenerovaným a vstupným kódom sú kozmetické. Niektoré testy sme si vygenerovali aj pomocou AI, konkrétne ChatGPT, ktorému sme zadali obmedzenia vzťahujúce sa na náš nástroj. Jeden z takto vygenerovaných vstupov môžeme vidieť v prílohe A spolu s časťou Blockly prostredia, ktoré nástroj vygeneroval B.1.

Táto testovacia metóda nepatrí medzi najtradičnejšie, ale pre náš prípad sme ju uznali za vhodnú. Odhalili sme pomocou nej mnoho chýb, ktoré sme mohli opraviť a taktiež nám ukázala nedostatky popísané v kapitole 5.2.4. Nástroj nie je otestovaný dokonalo, ale vytvorili sme niečo málo pod 1000 testovacích prípadov, ktoré nástroj prechádza na zelenú a testovali sme ho aj sami ručne, Preto hot teda považujeme za dostatočne otestovaný.

Kapitola 8

Záver

Cieľom diplomovej práce bolo vytvoriť nástroj pre generovanie blokov zo zdrojového kódu napísaného v jazyku Lua, rozšírenom o rámec LÖVE. Tento nástroj mal byť použitý ako rozšírenie existujúcej webovej aplikácie pre vizuálne programovanie v Lua/LÖVE. Vrámci práce vznikol jednoduchý HTTP server, ktorý primárne slúžil ako vývojové prostredie, ale ktorý by v budúcnosti mal vývojárom dovoliť jednoduchú implementáciu rozšírenia stávajúceho nástroja. Taktiež vznikla verzia tohto nástroja pripravená ako rozšírenie existujúcej aplikácie, ktorú sa však nepodarilo integrovať ani po odskúšaní mnohých metód.

V rámci práce sme sa najprv zoznámili s jazykom Lua a rámcom LÖVE, ktorý ho rozširuje a slúži k tvorbe 2D hier. V práci sme sa oboznámili s existujúcimi nástrojmi pre vizuálne programovanie, ktoré primárne slúžia k vzdelávacím potrebám v oblasti programovania, alebo rozvíjania logického myslenia. Medzi týmito nástrojmi sme našli aj také, ktoré ponúkajú obojstranný prevod medzi blokmi a zdrojovým kódom. Pre prevod z blokov na zdrojový kód využívali skoro výlučne AST (Abstraktný syntaktický strom), pre ktorý sme sa taktiež rozhodli. Bližšie sme sa pozreli na knižnicu Blockly a preskúmali jej podporu pre tvorbu blokov z kódu. Následne sme sa zoznámili s rozširovaným nástrojom.

Ďalej sme sa zamerali na spomínaný AST a preskúmali sme nástroje, ktoré ho produkujú ako svoj výstup. Priblížili sme si aj spôsob, akým ho získavajú a reprezentujú. Z týchto nástrojov sme sa bližšie pozreli na *Tree-Sitter* parser, ktorý sme v našej práci použili pre potreby syntaktickej analýzy zdrojového kódu. Bol vybraný hlavne kvôli jeho rýchlosti, rozšíriteľnosti a možnosti použitia z iných jazykov aj napriek jeho implementácii v jazyku C.

Vrámci práce sme navrhli nástroj, ktorý dokáže z poskytnutého zdrojového kódu v jazyku Lua vygenerovať jeho reprezentáciu pomocou blokov v prostredí Blockly. Pre tento nástroj bola navrhnutá trieda *MyNode*, ktorej podtriedy sú zodpovedné za spracovanie uzlov AST, tvorbu blokov a ich prepájanie. Trieda *Distributor* bola navrhnutá pre správnu inicializáciu týchto tried a možnosť jednoduchej registrácie nových tried pre rozšírenie nástroja.

Pre implementáciu sme použili dostupné Blockly metódy pre tvorbu blokov z kódu, spomínaný *Tree-Sitter* parser pre syntaktickú analýzu vstupného kódu a vytvorenie odpovedajúceho AST. Pre implementáciu API servera sme využili npm *express* balíček. Bola implementovaná hlavná funkcia *CreateBlocks*, ktorá mala byť volaná z rozširovaného nástroja a *createBlocklyCode*, ktorá implementuje hlavný cyklus pre spracovanie uzlov AST a logiku spájania vytvorených blokov.

Tento nástroj sme chceli napojiť do existujúcej webovej aplikácie Love-blocks-web [13], ale nepodarilo sa nám to kvôli problémom pri integrácií *Tree-Sitter* parseru. Ani po mnohých alternatívnych riešeniach, ktoré sme vyskúšali a hojnom množstve času, ktorý sme tomu venovali sa nám funkcionalitu nakoniec nepodarilo prepojiť s rozširovanou aplikáciou.

Hlavný cieľ práce však bol splnený a nástroj pre prevod zdrojového kódu v jazyku Lua do blokovej reprezentácie pomocou Blockly bol správne implementovaný. Nástroj dokáže efektívne spracovať aj zložité kódové štruktúry jazyka Lua rozšíreného o rámec LÖVE a vytvoriť z nich odpovedajúcu reprezentáciu pomocou blokov. Jeho ľahká rozšíriteľnosť o nové bloky a schopnosť spracovávať nové štruktúry robí tento nástroj vhodným štartovacím bodom pre vývojárov, ktorí sa zaujímajú o tvorbu blokov v prostredí Blockly. Nástroj je otestovaný a jeho prepísaná verzia je pripravená pre integráciu do aplikácie. Problémy pri samotnej integrácii boli spôsobené externým nástrojom, ktorý používame a nie funkcionalitou nástroja samotného.

Výsledný nástroj je plne použiteľný v našom vývojom prostredí a poskytuje solidný základ pre tvorbu blokov v prostredí Blockly. Jeho prepísanie ako rozšírenia, ktorého použitie sa podarilo odskúšať iba z časti a aj vývojové prostredie, v ktorom sme pracovali, sme publikovali ako open-source a sú dostupné v našom Github repozitári[1].

[1] `https://github.com/Figliar`

Literatúra

[1] Ball, T., Chatra, A., Halleux, P. de, Hodges, S., Moskal, M. et al. Microsoft MakeCode: embedded programming for education, in blocks and TypeScript. In: *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. New York, NY, USA: Association for Computing Machinery, 2019, s. 7–12. SPLASH-E 2019. DOI: 10.1145/3358711.3361630. ISBN 9781450369893. Dostupné z: https://doi.org/10.1145/3358711.3361630.

[2] Beazley, D. *PLY: Introduction and Overview* [online]. David Beazley, 2001-2020 [cit. 2024/01/21]. Dostupné z: https://ply.readthedocs.io/en/latest/ply.html.

[3] Git Hub. *Tree-sitter introduction* [online]. Tree-sitter, (n.d.) [cit. 2024/01/21]. Dostupné z: https://tree-sitter.github.io/tree-sitter/.

[4] GitHub. *Web-tree-sitter - npm* [online]. npm., (n.d.) [cit. 2024/05/12]. Dostupné z: https://www.npmjs.com/package/web-tree-sitter.

[5] Google. *What is Blockly?* [online]. 2023, 2023-12-19 [cit. 2024/01/21]. Dostupné z: https://developers.google.com/blockly/guides/get-started/what-is-blockly.

[6] Google. *Blockly Git-Hub* [online]. 2024, 2024-01-21 [cit. 2024/01/21]. Dostupné z: https://github.com/google/blockly/tree/develop.

[7] Jam, Y. C. *MakeCode for micro:bit* [online]. Youth Code Jam, (n.d.) [cit. 2024/05/10]. Dostupné z: https://www.youthcodejam.org/blog/makecode-microbit.

[8] Neovim. *Treesitter: help page* [online]. Neovim, 2024. 2024-05-14 [cit. 2024/01/21]. Dostupné z: https://neovim.io/doc/user/treesitter.html.

[9] None. *About Love2D* [online]. MediaWiki default., 2022. 2022-01-19 [cit. 2024/01/21]. Dostupné z: https://love2d.org/wiki/Main_Page.

[10] None. *Webpack Documentation* [online]. Webpack, (n.d.) [cit. 2024/05/10]. Dostupné z: https://webpack.js.org/.

[11] Parr, T. *What is ANTLR?* [online]. ANTLR / Terence Parr, (n.d.) [cit. 2024/01/21]. Dostupné z: https://www.antlr.org/.

[12] Pasternak, E., Fenichel, R. a Marshall, A. Tips for creating a block language with blockly. In:. Október 2017, s. 21–24. DOI: 10.1109/BLOCKS.2017.8120404. ISBN 978-1-5386-2480-7.

[13] Petr, M. *Výukový software pro vizuální a textové programování v Lua/LÖVE* [online]. Brno, CZ, 2022. Diplomová práca. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedúci práce Rychlý Marek, P. Dostupné z: https://www.vut.cz/studenti/zav-prace/detail/145428.

[14] Roberts, I. *Beginning Syntax*. Cambridge University Press, 2023. ISBN 9781009010580.

[15] samishawl. *Syntax Tree – Natural Language Processing* [online]. GeeksForGeeks, 2021 [cit. 2024/05/13]. Dostupné z: https://www.geeksforgeeks.org/syntax-tree-natural-language-processing/.

[16] Terence Parr, D. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013. ISBN 9781934356999.

[17] Tišnovský, P. *Programovací jazyk Lua*. Internet Info, 2014. ISBN 999-00-001-2651-4.

Príloha A

Ukážka vstupu

```
-- Describe this function...
function multiply(a, b)
  return a * b
end
-- Describe this function...
function add2(a, b)
  return a + b
end
-- Describe this function...
function isPrime(number)
  if number <= 1 or 2 * 8 > 5 then
    return false
  elseif number == 2 then
    return true
  else
    for i = 2, (math.sqrt(number)), 1 do
      if number % i == 0 then
        return false
      end
    end
    return true
  end
  return false
end

x = 10
y = 5
z = 0

if x > y then
  -- koment
  for i = 1, x, 1 do
    z = z + i
  end
else
  -- Cyklus while
  while y > 0 do
    -- comment s cislami 5689789
    z = z + y
    --
    y = y - 1
    -- comment znaky /()!"__:?
  end
  -- comment --
end
z = z + multiply(x, y)
string1 = 'Hello'
string2 = 'world!'
concatenated_string = table.concat({string1, ', ', string2})
prime = isPrime(x)
array = {1, 2, 3, 4, 5}
for _, value in ipairs(array) do
  -- Volanie funkcie ako parametra
  z = z + add2(z, value)
end
print('Výsledok je: ' .. z)
print(concatenated_string)
print(table.concat({'Je číslo ', x, ' prvočíslo? ', prime}))

```

Príloha B

Časť výstupu pre vstup z A