Využití grafických programovacích jazyků pro výuku programování

Petr Mather

Bakalářská práce
2019

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2018/2019

ZADÁNÍ BAKALÁŘSKÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Petr Mather**
Osobní číslo:	**A16601**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Informační a řídicí technologie**
Forma studia:	**kombinovaná**
Téma práce:	**Využití grafických programovacích jazyků pro výuku programování**
Téma anglicky:	**Visual Programming Languages for Introductory Programming**

Zásady pro vypracování:

1. **Proveďte průzkum aktuálního stavu grafických programovacích nástrojů, určených pro výuku programování, které jsou šířeny s licencí otevřeného kódu (opensource) nebo jsou dostupné zdarma.**
2. **Vybere jeden z takových nástrojů a na příkladu reálného projektu vyhodnoťte jeho výhody a nevýhody.**
3. **Analyzujte a navrhněte možnosti implementace nových vlastností nebo úprav, které odstraní některou z nevýhod.**
4. **Implementujte rozšíření dle návrhu z předchozího bodu.**
5. **Otestujte novou implementaci na příkladu reálného projektu.**

Rozsah bakalářské práce:

Rozsah příloh:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **ZHANG, Kang. Visual languages and applications. New York: Springer, 2007. ISBN 9780387682570.**
2. **ICHIKAWA, Tadao., Erland. JUNGERT a Robert R. KORFHAGE. Visual languages and applications. New York: Plenum Press, c1990. ISBN 9780306434273.**
3. **BROWN, H. Douglas. Teaching by principles: an interactive approach to language pedagogy. 3rd ed. White Plains, NY: Pearson Education, c2007. ISBN 9780136127116.**
4. **HOLZNER, Steven. JavaScript profesionálně: [kompletní referenční příručka]. Praha: Mobil Media, c2003. iDnes internet knihy. ISBN 8086593401.**
5. **REARICK, Ben. Blockly. Ann Arbor, Michigan: Cherry Lake Publishing, 2017. 21st century skills innovation library. ISBN 9781634727174.**

Vedoucí bakalářské práce: **Ing. Tomáš Dulík, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání bakalářské práce: **21. prosince 2018**

Termín odevzdání bakalářské práce: **15. května 2019**

Ve Zlíně dne 21. prosince 2018

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Ing. Vladimír Vašek, CSc.
ředitel ústavu

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářské práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky. Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně 15.5.2019 .....Petr Mather v.r.....

podpis autora

ABSTRAKT

Předmětem této práce je rozbor současných grafických výukových programovacích jazyků a následná implementace rozšíření jednoho z nich. V první části se zabývá rozborem a procesy výuky programování v současném základním a středním školství. Rozebírá přístupy k výuce programování pro děti ve věku přibližně 10-15 let. Následně obsahuje rozbor nejběžnějších grafických prostředí pro výuku programování. Hlouběji rozebírá prostředí Blockly, pro které je v druhé části popsána implemetace rozšíření. Rozšířením je přidání nových bloků pro struktury, složené datové typy. Zbytek práce se pak zabývá vývojem nových bloků, systému pro jejich obsluhu a ukázkové aplikaci s tímto rozšířením, která je součástí práce.

Klíčová slova: programovací jazyky, grafické programovací jazyky, výukové programovací jazyky, výuka programování, programovací jazyky pro děti

ABSTRACT

Subject of this thesis is analysis of current educational graphical programming languages and implementation of extension for one of them. In the first part it breaks down processes of teaching programming in contemporary primary and high school system. It mentions ways of teaching programming for kids ages 10 to 15. Next it contains description of most common visual programming languages for teaching programming. It explains Blockly in more depth, and in second half describes implementation of extension. That consists of new blocks for structures, composite data types. Rest of the text is about development of new blocks, background systems for management of these new blocks, and demo application to show them, which is part of the thesis.

Keywords: programming languages, graphical programming languages, programming languages for introductory programming, programming languages for children

OBSAH

ÚVOD

V dnešní době je programování a IT obecně oborem, který se výrazně a vysokým tempem zapojuje do všech oblastí lidské činnosti. I malá firma o několika zaměstnancích dnes snadno vyprodukuje alespoň jednu nebo několik IT zakázek, např. webovou prezentaci nebo účetní software. Pro pokrytí takto vysoké poptávky se tak IT stalo oborem, který má stabilní nárust pracovních míst a nedostatek zaměstnanců i přes širokou nabídku pozic, nadprůměrné platové ohodnocení a časté nabídky dovzdělání pro práci v IT.

Pokud se pak podíváme například na systém základního a středoškolského vzdělávání v České Republice, uvidíme velmi pomalé přizpůsobování se tomuto trendu. Výuka informatiky na základních školách má nízkou hodinovou dotaci, z níž je často velká část vyhrazena v podstatě kancelářské práci na počítači (např. obsluha textových a tabulkových editorů).[1] Na středních školách je pak informatika vyučována primárně na technických oborech, u všeobecných oborů jako gymnázií pak často chybí, nebo je dostupná jako volitelný seminář.

V praxi se je pak běžné vidět zaměstnance v IT, kteří nemají žádné formální vzdělání v oboru a jejichž znalosti jsou pouze výsledkem dlouhodobého zájmu ve formě hobby nebo jejich cílevědomým přeškolováním z jiného oboru. V obou těchto případech však jde většinou o samostudium. Vysoké školy přitom nabízejí ucelené informatické vzdělání, nižší stupně škol ale mají problém zatraktivnit a dostatečně nabídnout tuto kariérní volbu.

Jedním z možných řešení tohoto problému je nabídnout ucelené výukové metody, cílené na mladší žáky. Grafické programovací jazyky nabízí možnost prezentovat úplné základy programování jako jednoduchou logickou hru, kterou je schopno absolvovat jakékoliv dítě ve chvíli, kdy je schopno řešit jednoduché logické úlohy. Mají snahu odbourat vysoký počáteční rozsah znalostí potřebných pro programování, čímž si kladou za cíl zvýšit atraktivitu IT u žáků a studentů.

V této práci se budu několika z nich teoreticky věnovat. Rozvedu zde základy pedagogických aspektů práce s nimi, zmíním jejich zařazení z hlediska programování. Popíši jejich možnosti a techniky práce s nimi v tomto kontextu. Následně si jeden z nich vyberu a navrhnu jeho rozšíření, které na základě znalostí nabraných v první části práce podpoří jeho funkci. Rozeberu aspekty a postup tohoto řešení. Nakonec zhodnotím přidanou hodnotu mojí práce zvolenému projektu a rozeberu nově otevřené možnosti pokračování jeho rozvoje.

I. TEORETICKÁ ČÁST

1 Výuka programování

Pokud ponecháme stranou samostudium programování zmíněné výše, můžeme se zaměřit na konkrétní postupy a jevy, které doprovázejí strukturovanou výuku programování. Cílem takové výuky pak může být schopnost studenta vytvářet programy v prakticky uplatnitelném programovacím jazyce. Pro tuto práci se také primárně zaměřím na cílovou skupinu výuky základů programování, tedy primárně žáky druhého stupně základních škol, případně pak středních škol.

1.1 V rámci vzdělávácích plánů

Chceme-li podrobněji rozebrat vzdělávání v Českých školách, je potřeba začít od *Rámcového vzdělávácího programu*[2] (dále RVP). Ten definuje závazné rámce vzdělávání na jednotlivých stupních škol v určených vzdělávácích oblastech. Ve vzdělávací oblasti *Informační a komunikační technologie* ale není vůbec zmíněna algoritmizace či výuka základů programování. Vzdělávácí oblast se jako celek zabývá hlavně obecnou digitální gramotnosti, tedy uživatelským zvládnutím kancelářských nástrojů a internetových služeb. Minimální hodinová dotace podle RVP je pak stanovena na jednu hodinu týdně. Podle výzkumu[3] mezi učiteli informatiky se ukázalo, že až 86% respondentů vyučuje programování, ale často v minimálním měřítku, nebo až v posledním ročníku. Mezi důvody vůbec nevyučovat programování byla hlavně jeho nepřítomnost v RVP a nedostatečná časová dotace nebo technické vybavení.

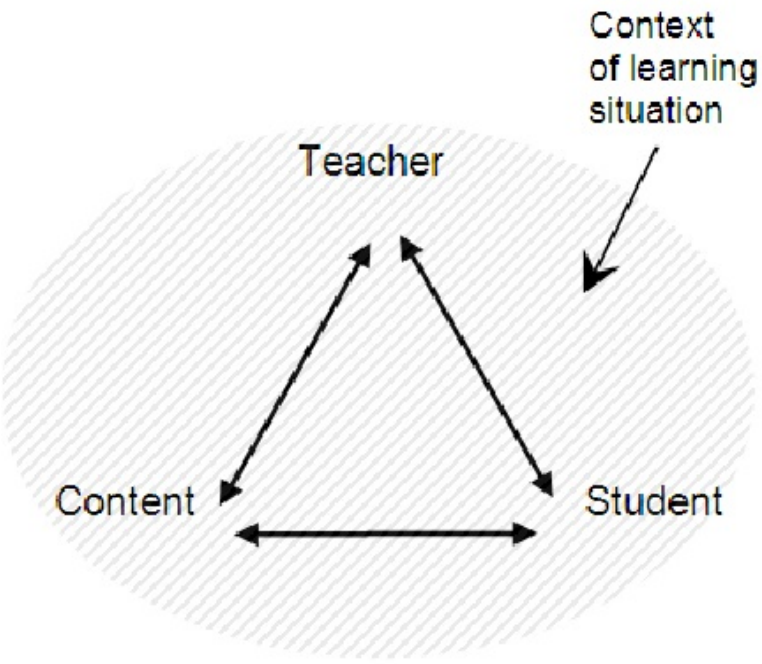
V rámci evropské unie pak na základě publikace *Computing our future*[4] vidíme, že téměř všechny země považují digitální gramotnost za jednu z priorit rozvoje vzdělávání. 16 zemí pak zmiňuje programování ve svých vzdělávacích plánech (včetně České republiky). Ve většině z těchto zemí je programování zahrnuto v rámci plánu pro předmět informatika, v některých pak v matematice nebo fyzice. V statistice ale figurují i země, které nechávají plány na výuku programování na regionální úrovni, nebo na jednotlivých školách.[5]

1.2 Výukové metody

Pro potřeby výuky se jeví vhodným začít pohledem do mechaniky učení. V práci[6] se autor zaměřil na několik teoretických rozdělení postupu učení. Prvním z nich je rozdělení na explicitní a implicitní učení. Explicitním učením rozumíme cílené přijímání informací se záměrem pochopení zákonitostí v probírané látce. Naproti tomu implicitní učení znamená pochopení zákonitostí bez cíleného příjmu informací. Druhým rozdělením pak je deduktivní, kdy jsou předloženy strukturované informace bez příkladu vyžadujícího subjektivní pochopení problematiky. Naproti tomu induktivní

učení znamená doplnění významu informací až po ponechání prostoru ke studiu příkladu. Kombinací těchto dvou veličin vznikají tzv. prostory učení. Z toho pak vyplývá použití jednotlivých kombinací, tedy např. rodný jazyk se učíme implicitně-induktivně. Naopak většina školní výuky probíha v explicitně-deduktivním prostředí.

Pokud z roviny učení odstoupíme k pohledu na úlohu ostatních účastníků procesu učení, můžeme využít Kansanenova modelu didaktického trojúhelníku (Obr. 1.1).



Obr. 1.1 Didaktický trojúhelník (Kansanen, 1999)

Učení z pohledu studenta jsme řešili v předchozím odstavci, proto postupme přímo k pozici učitele. Z tohoto pohledu se můžeme zaměřit na orientaci výuky na obsah proti výuce orientované na studenta. Výuka zaměřená na studenta se specializuje na využití induktivní oblasti učení. Ta je naopak výhodná pro výuku vědeckého a analytického přístupu k hledání řešení. To je také podstatnou součástí výuky programování. Naproti tomu u výuky zaměřené na obsah pozorujeme jasné znaky zaměření na explicitně-deduktivní oblast učení, cíl by se dal shrnout jako snahu „dostat x informací do studenta jako do nádoby“. Ačkoliv programování má výraznou teoretickou základnu, kterou se student musí naučit, tato varianta učení se u studentů setkává s problémy s motivací.[7]

Směřování této formy výuky je ale výrazně ovlivněno třetím článkem didaktického trojúhelníku, obsahem. Jens J. Kaasbøll ve své práci[8] zmiňuje výhodnost výuky zaměřené na cyklus analýzy problému a následného vývoje jako posledního kroku. Cílem výběru nástroje pro výuku programování je snaha snížit objem informací nutných pro posun do fáze řešení problému.

1.3 Úloha grafických programovacích jazyků

V tomto bodě nacházíme prostor pro vizuální programovací jazyky. Účelem mnoha z nich je učinit programování dostupnější. Nejvýrazněji je to vidět na práci se syntaxí jazyka. Pouze malá podmnožina možných textových vstupů je validní, nebo dokonce smysluplný program. Grafické programovací jazyky používají syntaxi definovanou geometrickými tvary, takže nefunkční výraz ani nelze sestavit (např. bloky nezapadají do sebe). Dalším benefitem je zpravidla nabídka možných bloků, uživatel tedy nepotřebuje znalost nabízených funkcí nebo dokumentaci. To dává žakovi prostor pro zaměření se na logické řešení problému.[9]

Pokud bych měl na tomto místě zmínit některá omezení grafických jazyků, hlavním z nich je, že jde stále o programování. Algoritmizace není nijak zjednodušena a syntaxe, ač omezena do snáze uchopitelné podoby, musí být dodržena. Pro pokročilé programátory pak nepřináší příliš výhod. V této cílové skupině je textová reprezentace výrazně kompaktnější a také nabízí výrazně širší nabídku funkcí a konstrukcí, které už nemusí jít pokrýt tvary a barvami bloků.[9] Z těchto důvodů je třeba uvažovat při návrhu výukových grafických programovacích jazyků o přechodu studenta na textové programovací jazyky.[10]

2 Programovací jazyky

Programovacím jazykem rozumíme množinu příkazů a dalších syntaktických výrazů pro zápis programu. Ty jsou poté přeloženy na instrukce konkrétního procesoru. Určující pro rozdělení programovacích jazyků je míra abstrakce nad konkrétními instrukcemi procesoru.

Podle míry abstrakce můžeme programovací jazyky rozdělit na nižší a vyšší. Nižší programovací jazyky zpravidla mají nízkou míru abstrakce. Zdrojový kód daného jazyka musí obsluhovat více detailů např. práce s pamětí a zdrojové kódy jsou rozsáhlejší. Běžně tyto jazyky bývají náročnější na znalost programátora. Vyšší programovací jazyky pak představují vývoj směrem k čitelnějšímu zdrojovému kódu. Jejich struktura je častěji organizována do logických bloků. Vypouštějí příkazy obsluhující strojové principy počítače, stávají se tak přenositelnými mezi počítači.[11]

Dalším rozdělením je podle způsobu spuštění na jazyky překládáné a interpretované. Jazyky s překladačem jsou založeny na programu (překladači), který jejich zdrojový kód přeloží do instrukcí instrukční sady konkrétní platformy. Naproti tomu jazyky interpretované se spouští pomocí programu běhového prostředí, které provádí jejich zdrojový kód. Tyto varianty se běžně kombinují u jazyků, které jsou překládány na mezikód, který je pak spouštěn běhovým prostředím. Nejznámějším příkladem tohoto přístupu je jazyk Java.

Posledním rozdělením, které zde zmíním, je podle paradigmatu programování. Nejčastěji využívaným je imperativní. U něj prográmator zadává posloupnost instrukcí, které se mají vykonat v zadaném pořadí. Méně používanou skupinou jsou deklarativní jazyky. Sem patří jazyky logické, u kterých je program zadán množinou logických výrazů a je na programu, aby našel řešení. Podobně u funkcionálního programování je zdrojový kód množina funkcí a jejich řešení hledá program. Tyto méně běžné přístupy se zejména v poslední době objevují v imperativních jazycích pro řešení menších podúloh.

Za zmínku v této kapitole stojí také paradigma objektového programování. To využívá abstrakce rozdělení zdrojového kódu do objektů. Ty jsou navrženy jako ohraničené bloky s konečnou množinou funkcí. Umožňují tak lepší organizaci a strukturování kódu. Využívají také princip zapouzdření, tedy skrytí vnitřního stavu a možnost komunikace pouze pomocí navržených funkcí. Dalšími z výhod je možnost dědění objektů, spojené se snadnou znovupoužitelností kódu, vycházející z jeho modularity.[12]

2.1 Grafické programovací jazyky

Grafické programovací jazyky jsou definované možností vytvářet programy manipulací vizuálních prvků místo jejich textové definice. Jejich abstrakce vychází z myšlenky, že při vysvětlování programu se často využívá grafů, a je tedy možné je využít přímo při programování. Využívají pro tento účel nejčastěji sestavování grafů, nebo seřazování bloků do organizovaných skupin. Běžně používaným prvkem jsou bloky připomínající např. stavebnici LEGO nebo podobnou pro modelování vztahů mezi prvky.

Vzhledem k jejich vzniku z grafů je nasnadě, že drtivá většina používaných grafických programovacích jazyků je založena na imperativním paradigmatu (program lze popsat průchodem orientovaným grafem). Kvůli často výukové povaze těchto jazyků je kladen důraz na jejich snadnou přenositelnost a nenáročnou dostupnost. Z toho důvodu jde zpravidla o vyšší programovací jazyky (i zde ale nalezneme výjimky, např. doplněné proprietálním hardwarem).

Již z jejich popisu je jasné, že jsou založeny na vývojovém prostředí, které obsahuje grafické rozhraní pro sestavování „kódu“. Také vzhledem ke struktuře „kódu“ jde většinou o jazyky interpretované. Některé ale obsahují překladač, zpravidla ne do instrukcí instrukční sady procesoru jako běžné překládané jazyky, ale do jiného programovacího jazyka. Níže uvedené příklady mají bloky zpravidla implementovány v některém z běžně používaných formátů pro zápis dat (např. JSON nebo XML).

2.1.1 Konstrukce v programovacích jazycích

Množinu dostupných syntaktických konstrukcí můžeme rozdělit do celků, které přibližně kopírují běžnou osnovu výuky základů programování. V další kapitole pak budu u jednotlivých procházených příkladů zmiňovat, které z nich jsou u daných prostředí k dispozici.

- práce s proměnnými - vytváření a přiřazování
- aritmetické operace
- logické operace
- příkazy pro řízení toku programu - podmínky, cykly, případně skoky
- pole nebo vícerozměrná pole
- práce s textem
- práce s obrazem nebo zvukem (závislé na běhovém prostředí)
- případné knihovní funkce (řazení, práce s řetězci, atd)

- procedury, funkce
- objekty na základě prototypování
- objekty na základě tříd, dědičnost

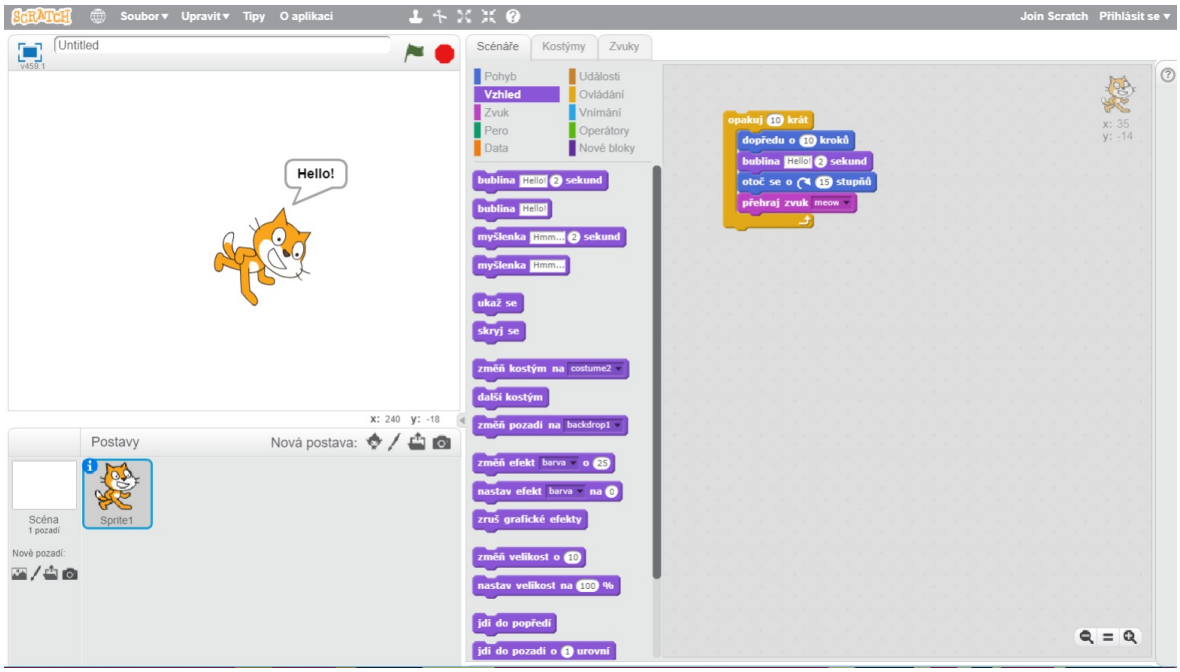
3 Rozbor stávajících řešení

V této kapitole rozeberu některé z existujících grafických programovacích jazyků, konkrétně ty zaměřené na výuku programování. Zmíním některé základní informace ohledně projektu a technologie využité při implementaci. Zaměřím se na jejich návrh výuky programování, dostupné konstrukce jazyka a možnost rozšíření. V případě významných aplikací nebo výukových programů na nich založených zmíním i ty.

3.1 Scratch

Scratch je projekt Lifelong Kindergarten Group, která působí v rámci MIT Media Lab. První verze byla určena pro desktopy s potřebou instalace. Výrazným rozšířením druhé verze bylo přidání serveru pro sdílení projektů. Je k dispozici zdarma ke stažení a instalaci nebo online bez instalace. Offline verze je k dispozici pro počítače s operačními systémy Windows, Linux nebo Mac OSX. Online verze vyžaduje pro spuštění prohlížeč s Adobe Flash Player.

Výhodou *Scratch*e je výrazná uživatelská podpora. Na stránkách `https://scratch.mit.edu/` autoři zmiňují 15 milionů registrovaných uživatelů. K tomu se váže zaměření komunity na sdílení projektů, kterých autoři zmiňují 22 milionů. Další výhodou je lokalizace do široké nabídky jazyků, včetně češtiny. Scratch bývá často zmiňován jako platforma pro tvorbu a sdílení aplikací, her, ale i animací.



Obr. 3.1 Prostředí Scratch

Po otevření stránky v prohlížeči jsou zobrazeny základní součásti editoru (Obr. 3.1).

Ten můžeme rozdělit na dvě části. Jednu pro umísťování postav do okna výsledné aplikace společně s plochou aplikace, a druhou pro sestavování bloků společně s nabídkou dostupných bloků. Běhové prostředí je výhradně součástí vývojového prostředí (v rámci sdílení projektů je možné skrýt blokovou část). Celé prostředí tak funguje jako uzavřený systém vytváření a zároveň i prezentování obsahu.

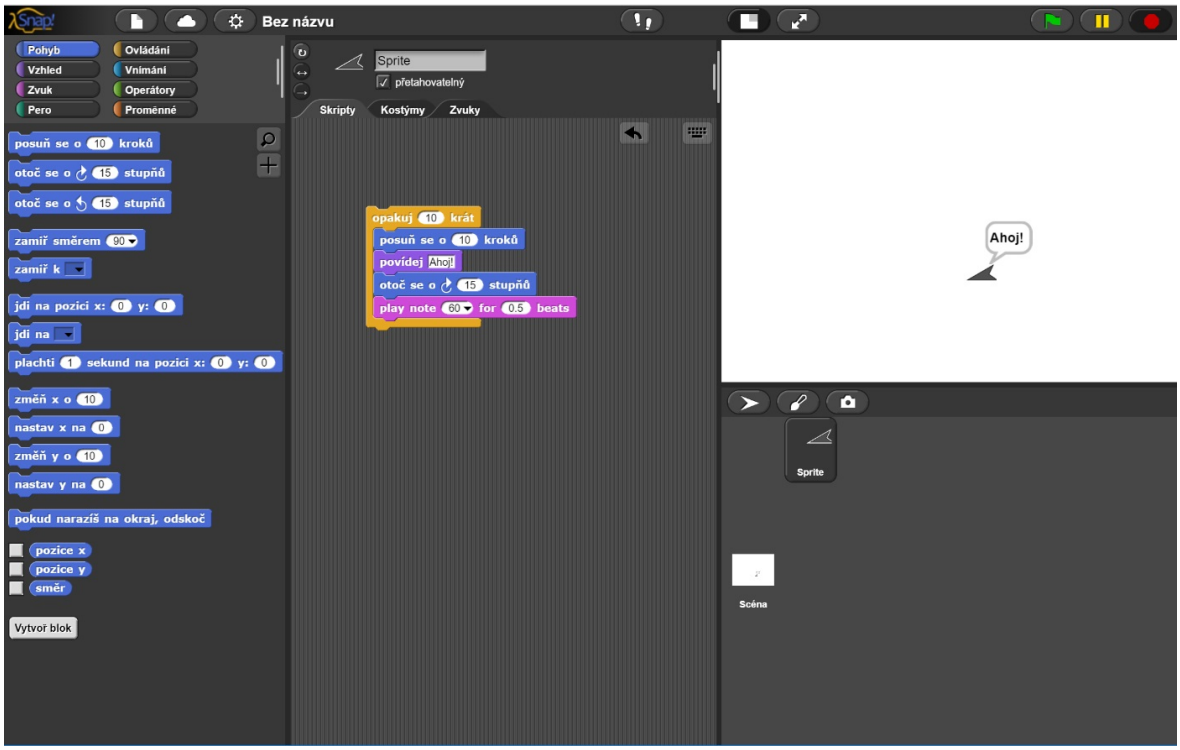
Pokud se podíváme na programování ve *Scratch*i, zjistíme, že základním stavebním kamenem jsou kromě bloků také výše zmíněné postavy. Plochu pro umístění bloků máme samostatnou pro každou postavu v záběru. V nabídce máme příkazy řízení běhu programu, skupinu bloků pro aritmetické a logické operace, vytváření a obsluhu proměnných a několik sekcí zaměřených na pohyb postav v ploše a změny jejich vzhledu. Také je k dispozici přehrávání zvuku. Podstatnou možností pro vytváření např. her je možnost programování na základě událostí a zpráv. Zprávy umožňují kontakt mezi jednotlivými postavami, reakce na události pak ovládání myší a klávesnicí. Také obsahuje možnost tvorby vlastních bloků v rámci vývojového studia, které slouží jako procedury a funkce.

Z hlediska návrhu výuky zvoleného tématu v rámci programování neumožňuje prostředí výrazné zásahy. Využití pro výuku v rámci některých vzdělávácích programů je možné hlavně pomocí doplnění o externí materiály s instrukcemi, případně vytvářením prototypů a ponechání jejich rozšíření na studentech. Naopak přínosem je podpora soutěživosti a sdílení vlastních výtvorů.

3.2 Snap!

Snap! původně vznikl jako rozšíření projektu *Scratch* na univerzitě v Berkeley. Podobně jako Scratch je k dispozici v offline módu a také jako aplikace běžící v prohlížeči. Co jej naopak odlišuje, je jádro systému vytvořené v jazyce javascript. Zdrojové kódy jsou dostupné pod licencí AGPL a jsou k dispozici ke stažení ze služby Github. Prostředí je pak k dispozici na stránkách projektu `https://snap.berkeley.edu/`.

Vývojové prostředí (Obr. 3.2) silně připomíná *Scratch*. Opět jsou přitomny příkazy pro řízení běhu programu, skupinu bloků pro aritmetické a logické operace, vytváření a obsluhu proměnných a několik sekcí zaměřených na pohyb postav v ploše a změny jejich vzhledu, přehrávání zvuku a programování na základě událostí a zpráv. Významným rozšířením z hlediska programování je využití *first-class data types*. To znamená možnost používat funkce, metody a v tomto případě i bloky jako proměnné. Jedná se o implementaci beztřídní varianty objektově orientovaného programování. Také je přítomna možnost rekurze. Dalšími výraznými rozšiřujícími prvky jsou možnost ladění a také možnost generovat zdrojový kód běžných skriptovacích jazyků, jako javascript nebo Python, přímo z aplikace.



Obr. 3.2 Prostředí Snap!

Z výukového pohledu má podobnou podporu jako *Scratch*. Neumožňuje výrazný zásah do stránky pro doprovodné texty a manuály, ale samotný jazyk obsahuje více konstrukcí běžně používaných v „dospělých“ programovacích jazycích, čímž umožnuje delší přesah bez nutnosti přejít na textové programování. Zároveň umožňuje přechodové cvičení díky možnosti generování zdrojového kódu a jeho porovnání s bloky.

3.3 Blockly

Blockly je projekt firmy Google. Jde o open-source vydáván pod licencí Apache 2.0 License. Na rozdíl od ostatních nejde o kompletní vývojové a zároveň prezentační prostředí, ale pouze o knihovnu vyvinutou v javascriptu. Blockly je tak k dispozici pouze online (ale není problém si stáhnout a spustit jeho zdrojové soubory). K jeho spuštění tedy je potřeba pouze prohlížeč s podporou javascriptu. Existuje také verze Blockly pro iOS a Android.

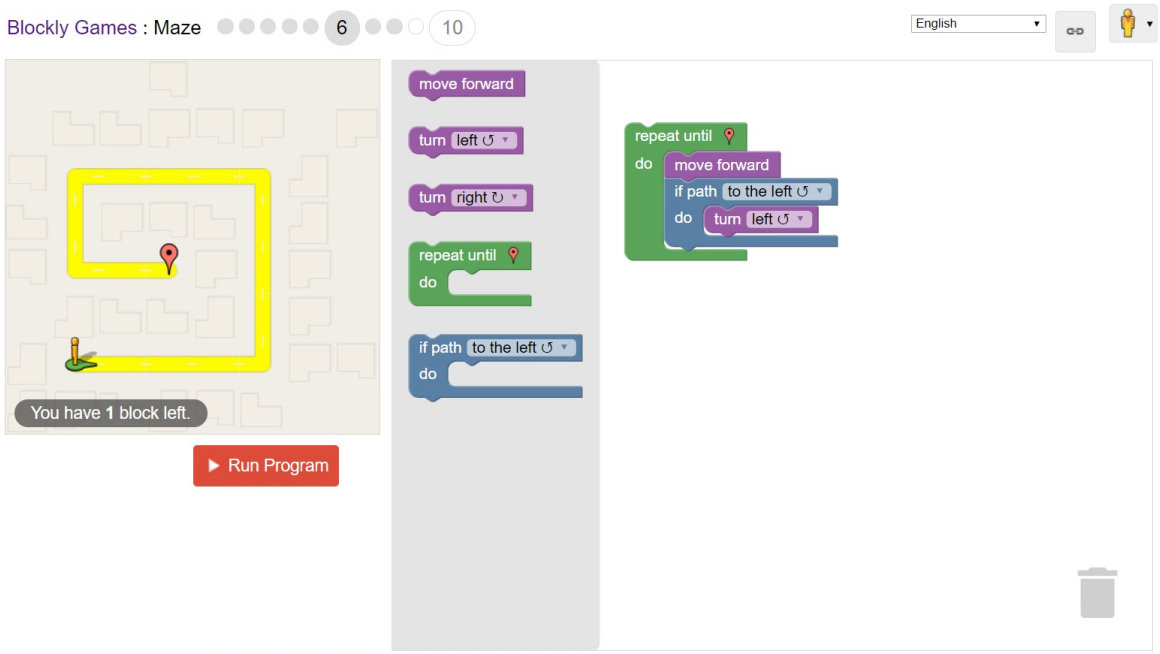
Prostředí blockly vložené do webové stránky pak poskytuje pouze plochu pro sestavování bloků a nabídků bloků. Součástí zdrojových souborů není běhové prostředí vytvořeného „kódu“, ani žádná vazba na grafickou reprezentaci běžícího programu. Naopak součástí je překladač bloků do zdrojového kódu nabídky jazyků, jako javascriptu, Pythonu apod. Další součástí, kterou obsahuje, je nástroj pro vytváření nových bloků přímo v tzv. editoru bloků. Podobně je možné pro jednotlivé stránky měnit obsah „toolboxu“, tedy například přidávat nové bloky s postupem interaktivního kurzu. Bloky,

které prostředí obsahuje, pokrývají pouze minimum očekávaných možností. Skupina příkazových bloků pro běh programu, aritmetiku, proměnné a funkce. Příjemným rozšířením jsou pole.

Z pohledu dalšího rozšíření nabízí Blockly široké možnosti. Na strankách autorů je dokonce seznam možných rozšíření knihovny pro členy komunity, kteří by se chtěli zapojit do vývoje. Další možností rozšíření je přidat sadu bloků s novou funkcionalitou, např. základy objektového programování. Tuto možnost podporuje také možnost vytvořit libovolnou stránku a upravit toolbox k příslušnému úkolu.

3.3.1 Blockly Games

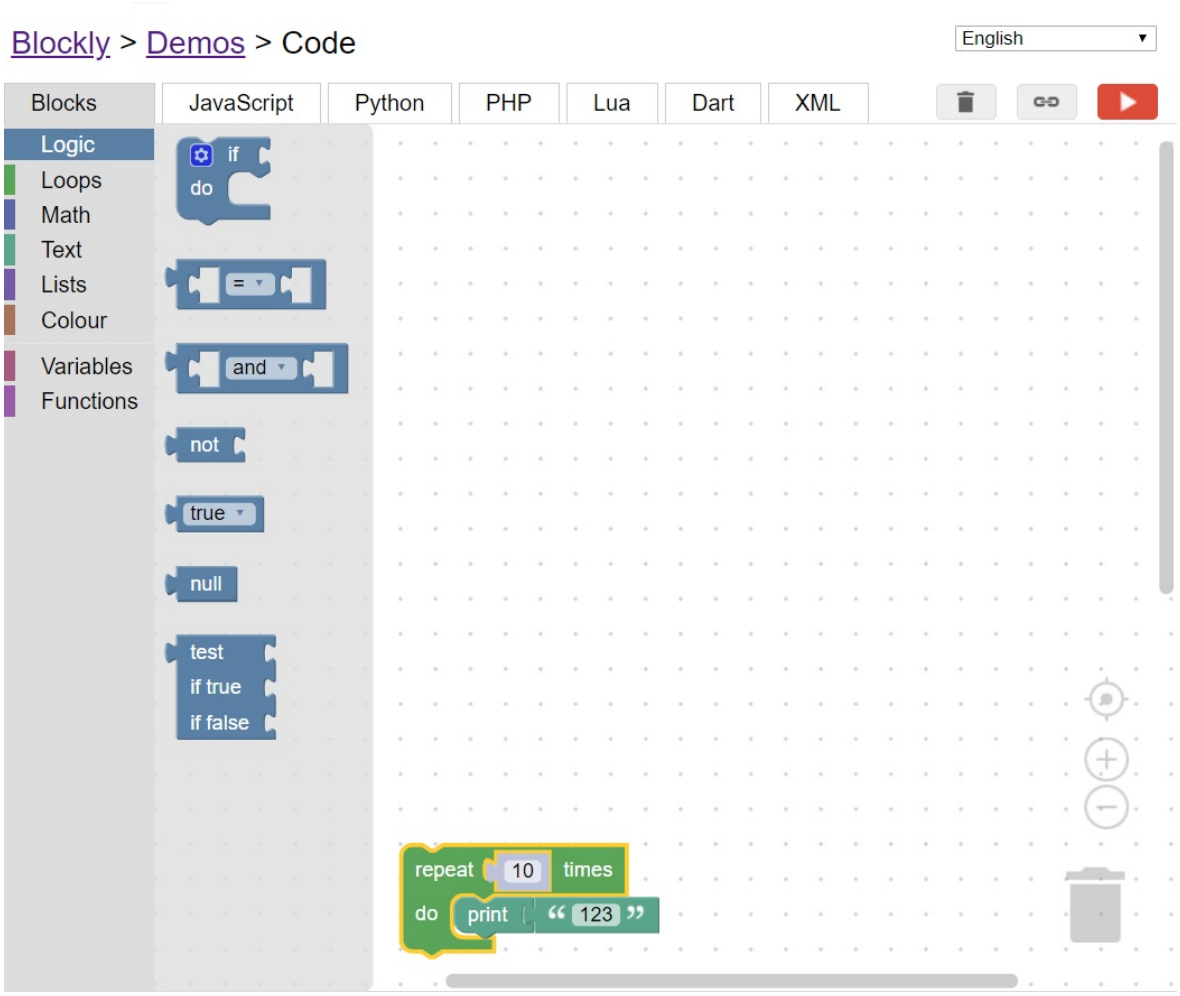
Blockly Games (dostupné na stránkách `https://blockly-games.appspot.com/`) je aplikace prostředí Blockly pro sestavení interaktivního výukového kurzu. Obsahuje několik cvičení, která postupně popisují jednotlivé problémy Blockly. Začíná od obsluhy bloků a pokračuje přes základní řetězení příkazů, podmínky a cykly, až k naprogramování taktiky do jednoduché bojové hry. Cvičení mají minimální instrukce (s postupem kurzu přibývají), ale všechna mají grafickou reprezentaci podobnou prostředí Scratch. Všechny také po úspěšném dokončení uživateli zobrazí podobu příslušného zdrojového kódu v javascriptu, ve snaze usnadnit přechod mezi grafickými a textovými programovacími jazyky.



Obr. 3.3 Prostředí Blockly Games

3.3.2 Blockly Demos

Blockly Demos je stránka s ukázkou vývojového prostředí Blockly (Obr. 3.4). Je možné ji také spustit lokálně ze zdrojových kódu. Je vidět, že obsahuje pouze toolbox s bloky, plochu pro sestavování a v záložkách překladače do výběru programovacích jazyků.



Obr. 3.4 Prostředí Blockly Demos

4 Zvolené prostředí a rozšíření

4.1 Prostředí

Po prostudování několika možných grafických jazyků, zaměřených na výuku programování (viz výše), jsem se rozhodl zvolit pro rozšíření prostředí Blockly.

Hlavním faktorem pro mě bylo samotné pojetí projektu, který je celý veden jako komunitní (je výrazně podporováno přispívání nezávislých vývojářů do projektu). Ostatní projekty zveřejňují svoje zdrojové kódy, ale vývoj vedou primárně v uzavřené skupině vývojářů jako ucelené projekty, včetně běhového prostředí a např. vlastní platformy pro sdílení studentských prací. To komplikuje jakoukoliv kompatibilitu s rozšířenou větví projektu.

Dalším výrazným aspektem pro mě byla implementace daného prostředí. U prostředí Blockly, které je celé implementováno v javascriptu, je patrná důslednost v úpravě zdrojových kódů, škálovatelnost a rozšířitelnost. Zároveň je tato platforma ideální pro úpravy a implementaci odvětvujících se projektů. Z některých těchto projektů, jako výše zmíněného Blockly Games, se dá vycházet při implementaci stránky obsahující příklady nové funkcionality.

K tomu se také váže možnost zasáhnout přímo do prostředí. V prostředích Scratch a Snap! není možnost vytvořit interaktivní lekci přímo ve vývojovém prostředí, je zde snaha spíše uvést zadání práce jako doplňkový materiál. Prostředí Blockly nabízí neomezené možnosti zavedení cílených cvičení apod. přímo do stránky s editorem, bez využití doplňkového materiálu.

4.2 Rozšíření

Zvoleným rozšířením se nakonec stalo přidání bloků pro složené datové typy, ve struktuře podobné jako v jazyce C. Původním návrhem, který mě značně lákal ke zpracování, bylo zaměřit se na objektovou orientaci. Po rozboru jednotlivých části objektového návrhu a značného množství doplňujících mechanik (dědičnosti, přetěžování, zapouzdření apod.) jsem zvolil tuto variantu.

II. PROJEKTOVÁ ČÁST

5 Rozbor Blockly

Nyní rozeberu zdrojové kódy Blockly. Hlavní část tvoří zdrojové soubory ve složce *core*, které obsahují obsluhu hlavních částí Blockly. Jsou zde obecné třídy pro jednotlivá pole na blocích a třída *Workspace*, která obsahuje součásti plochy, jejího uložení a také provolání do metod obsluhujících proměnné, procedury a změny bloků. Také najdeme obecnou třídu bloku, třídy s funkcemi pro jednotlivé typy bloků (procedures, variables) a třídy událostí.

Další důležitou součástí, na kterou se podrobněji podíváme níže, je složka *blocks*. Ta obsahuje definice bloků podle kategorií. Zde bych zmínil, že jednotlivé kategorie bloků se pro přehlednost rozdělují barvou bloku. Pro nové bloky si tak později vytvořím samostatnou kategorii s novou barvou.

5.1 Bloky

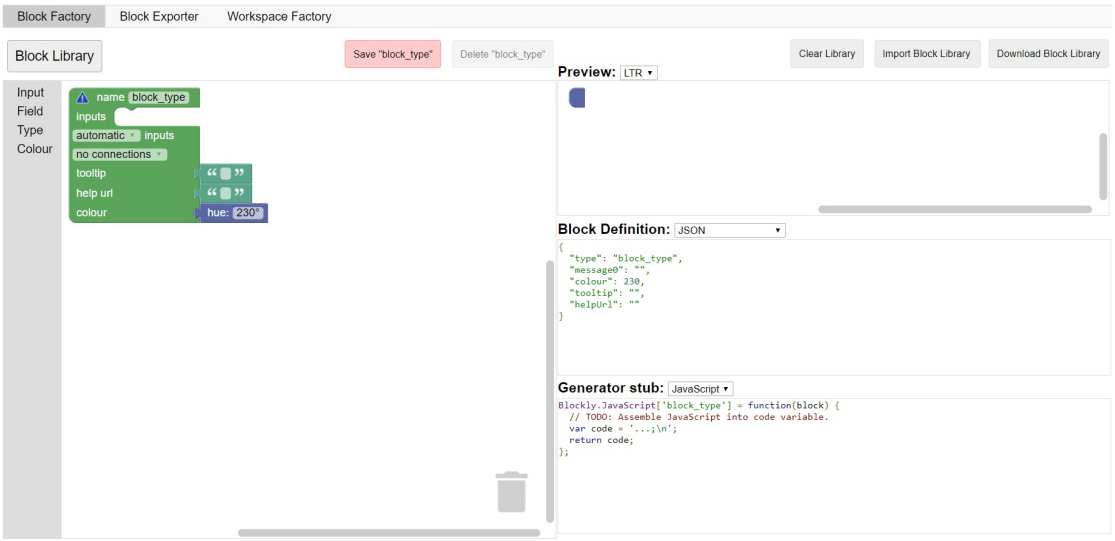
Základní bloky jsou uloženy v jednotlivých kategoriích v souborech ve složce *blocks*. Bloky lze poté do toolboxu vkládat po jednotlivých kategoriích nebo jednotlivých blocích, podle parametru type. Z tříd definujících základní bloky vidíme, že se bloky definují dvěma způsoby. První způsob je pomocí json, kdy Blockly načte definici bloku ze zápisu. Druhý způsob je pak vytvoření bloku a nastavení jeho parametrů přímo v javascript kódu. Způsob pomocí json je preferovaný, ale nezvládne vytvořit některé komplexnější bloky (například s nově vytvořenými poli, nebo mutátory, viz níže). Cílem je tedy vytvořit většinu bloků pomocí definice v json, pouze komplexní bloky definovat přímo v javascript kódu.

Pro vytváření nových bloků je součástí blockly aplikace *Block Factory*. V té je možné vytvářet nové bloky ve stylu Blockly, sestavováním jednotlivých částí, připojení, polí a podobně. Vytvořené bloky se ukládají v počítači. Zde se setkáváme s další možností uložení bloků, a to jejich uložení ve formátu XML. Tím se vytváří tzv. *Block library*. Definice bloků v XML lze stáhnout pro obnovení *Block library* uživatele v případě smazání historie prohlížeče. Nedílnou součástí Block factory je možnost exportu definic bloků do JSONu, javascriptu a také kostra kódu do generátoru (viz níže - kapitola generátory).


```
Blockly.Blocks['struct'] = {
  init: function() {
    this.appendValueInput("NAME")
        .setCheck("String")
        .appendField("name");
    this.appendStatementInput("VARIABLES")
        .setCheck(null);
    this.setInputsInline(true);
    this.setOutput(true, null);
    this.setColour(60);
 this.setTooltip("Define your structure");
 this.setHelpUrl("Halp");
  }
};
```

```
{
  "type": "struct",
  "message0": "name %1 %2",
  "args0": [
    {
      "type": "input_value",
      "name": "NAME",
      "check": "String"
    },
    {
      "type": "input_statement",
      "name": "VARIABLES"
    }
  ],
  "inputsInline": true,
  "output": null,
  "colour": 60,
  "tooltip": "Define your structure",
  "helpUrl": "Halp"
}
```

Obr. 5.1 Ukázka podoby bloku a jeho definice pomocí javascriptu a JSON - první návrh definice struktury



Obr. 5.2 Block Factory v ukázkách prostředí Blockly

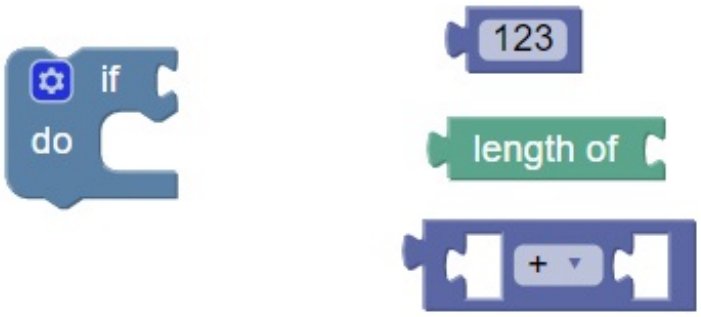
Při návrhu nových bloků musíme rozebrat jednotlivé vlastnosti bloků. Základními prvky jsou typ bloku (jeho identifikátor), nápověda a barva. Dalším hlavním prvkem bloků je jejich připojení.

Máme dva směry připojení bloku. Svislé připojení umožňuje spojit bloky do skupiny za sebou prováděných příkazů, odpovídá odřádkování v běžných programovacích jazycích. Pokud zvolíme žádné, blok půjde umístit pouze samostatně, což je využitelné u definujících bloků (např. definice struktury). Také můžeme zvolit pouze dolní, pouze horní, nebo obojí, což je případ většiny dostupných bloků - je využito u všech bloků, odpovídajícím příkazu v textových programovacích jazycích.

Speciálním případem svislého připojení je vnitřní, tedy že blok obsahuje prostor pro vložení příkazů dovnitř. Typickým využitím toho připojení jsou podmínky, cykly a procedury.

Druhý směr připojení bloku je vodorovné. To se využívá v několika kontextech, primárně jako dosazení nebo přiřazení hodnoty. Nahrazuje tedy řádkové operátory rovnítka, závorek (při volání funkce), případně by šlo využít pro operátor tečky. Pokud bloku přiřadíme výstup (vodorovné připojení vlevo), můžeme definovat i jeho typ. Blockly používá několik zjednodušených typů, jako boolean, number a string.

Připojení vpravo je vstupem bloku. Blok může mít i více vstupů. Vstupy mohou být kromě připojení i hodnotové, v tom případě je místo připojení na bloku umístěno pole. Ta mají více druhů, krom základního textového také např. barvu nebo úhel. Zvláštním typem pole je proměnná. Jedná se o dropdown menu, které nabízí proměnné existující v systému. Používá se u bloků get/set. Tato dropdown pole jsou modifikovatelná. Je ale potřeba navrhnout, jak budou získavat seznam nabízených hodnot. Podívám se tedy, jak fungují pro proměnné a funkce.



Obr. 5.3 Ukázka možných připojení bloku

5.2 Řešení proměnných a procedur

Jak jsem nastínil v minulé kapitole, pro práci se strukturami v návrhu bloků bude potřeba navrhnout správu struktur existujících v ploše na pozadí.

Proměnné jsou vytvářeny externě, pomocí tlačítka *Create Variable*. Po vytvoření alespoň jedné proměnné, bloky pro správu proměnných jsou aktivovány v nabídce bloků. Ty využívají dropdown menu, nabízející všechny proměnné v systému, uživatel si pak vybere, kterou chce použít. Proměnné existující v pracovní ploše nejsou vázány na bloky. Jsou uloženy v objektu *variable map* v podobě mapy podle typu. Každá z nich je reprezentována objektem *variable model*. *variable map* je pak přiřazena na workspace a obsluhována odtamtud. Generátor kódu (viz níže) pak vytváří proměnné z této mapy. Mazat nebo přejmenovávat proměnné je pak možné z nabídky jakéhokoliv bloku pro správu proměnných.

Ač je tento přístup výhodný z hlediska správy proměnných, jejich vytváření nebo modifikace je značně neintuitivní. Jejich správa zcela mimo pracovní plochu nezapadá do konceptu celého systému. Bloky pro obsluhu proměnných využívající dropdown naproti fungují ke svému účelu dobře.

Procedury jsou definovány vytvořením příslušného bloku. Po vytvoření bloku definice procedury jsou vygenerovány bloky pro volání konkrétní procedury. Každá procedura v systému generuje do toolboxu svoje vlastní bloky. Procedury jsou uloženy primárně na pracovní ploše. Generátor (nebo obecně jakékoliv volání *getAllProcedures*) projde všechny bloky na ploše a filtruje je, aby vybral definované procedury. Jako bloky mají metodu *getProcedureDefinition*, která určuje název, návratovou hodnotu a parametry. Není jich tedy sestavena předdefinovaná mapa, ale jejich seznam se volá v případě potřeby.

Tento přístup využívá definice, uložení a správu procedur přímo v pracovní ploše. Slabým místem toho návrhu je vytváření zvláštních bloků pro každou proceduru. Pokud uživatel vytvoří více procedur, obsluha jejich volání se stane značně nepřehlednou. Na druhou stranu je tento přístup, vzhledem k odlišnosti bloků volání jednotlivých procedur kvůli různému počtu parametru, pravděpodobně nejvýhodnějším řešením.

5.2.1 Problémy obou návrhů

Nejvýraznějším problémem je práce s parametry uvnitř procedury. Pokud vytvoříme proceduru s parametry *x,y*, a následně s nimi budeme chtít pracovat, zjistíme, že je problematické zabezpečit práci s těmito „proměnnými“ pouze v rámci metody. Blockly tuto situaci v současné době řeší tak, že skrytě vytvoří proměnné x, y, se kterými můžeme pracovat kdekoliv v ploše. Tím vytváříme značné množství proměnných, které pak budou matoucí pro uživatele, protože je sám explicitně nevytvořil.

S tím souvisí další z problémů proměnných a procedur v Blockly - všechny proměnné jsou globální. Z návrhového hlediska je komplikované donutit uživatele, aby používal proměnnou pouze v rámci metody, ve které je vytvořena. Blok by se musel zamykat proti vyjmutí z metody, případně by musel měnit vybranou proměnnou, což by bylo matoucí. Viditelnost a překrývání proměnných je nicméně výrazným tématem v Blockly komunitě a jeho chování v rámci prostředí je značně nejasné. Nejpraktičtější řešení je tak řešit chování javascript kódu.

6 Implementace rozšíření

V této části práce rozeberu a zdůvodním jednotlivé části mojí implementace v rámci rozšíření Blockly. Zmíním, jak navazuje na dosavadní implementaci a proč jsem zvolil vybrané postupy.

6.1 Návrh bloků

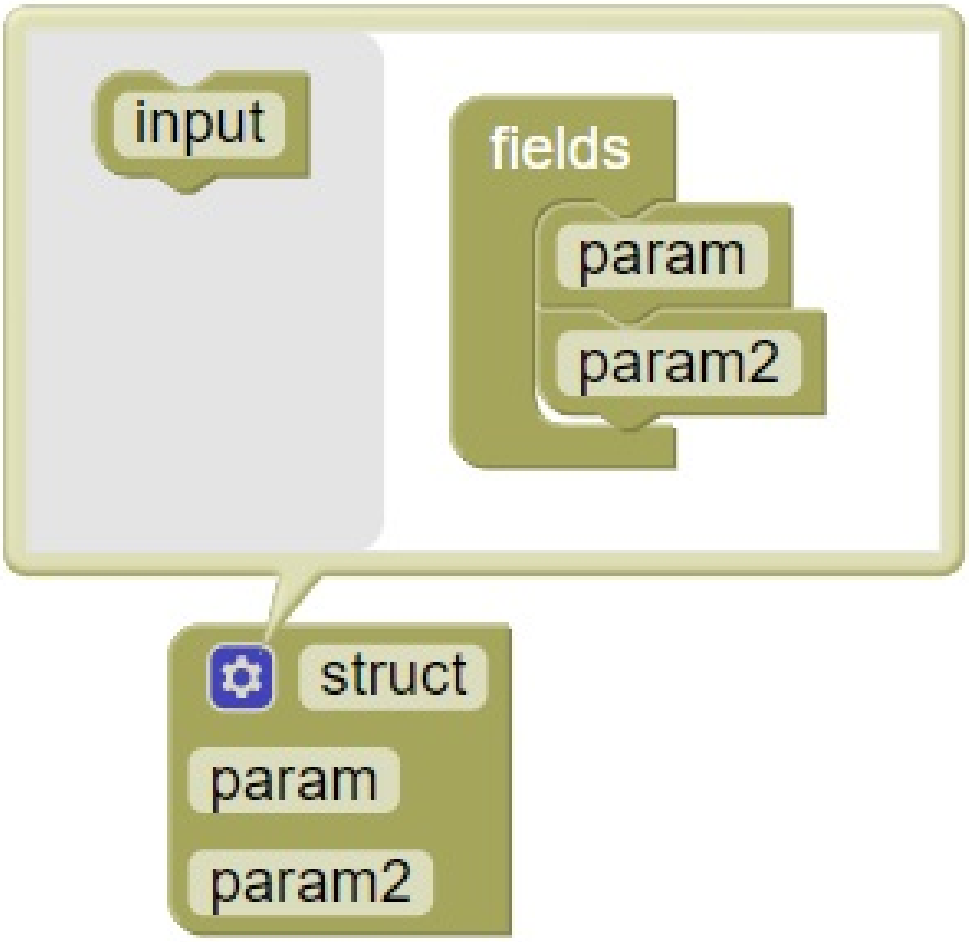
Pro návrh bloků je potřeba se podívat do textových programovacích jazyků, jaké syntaktické konstrukce jsou potřebné pro implementaci struktur, případně objektů. Pokud se podíváme např. do jazyka C, uvidíme že struktura se definuje jako datový typ pomocí konstrukce `typedef` a následně se používá jako jakýkoliv jiný datový typ. V objektových jazycích (vezmeme si jako příklad např. jazyk Java) definujeme třídu s parametry a metodami a následně vytvoříme instanci operátorem `new`. V obou případech pak využíváme operátoru `.` pro přiřazení nebo přečtení hodnoty jednotlivých prvků. Můžeme využít také běžnou konvenci operátorů `getset`, která se stala standardem. V prostředí Blockly je vzhledem ke konstrukci připojení bloků využitá, proto budu postupovat obdobně.

Je tedy jasné, že bude potřeba blok pro definici struktury, který bude osamocen, tedy bez dolního a horního připojení. Minimálni obsah tohoto bloku musí být název nového typu a výčet prvků, které obsahuje. Při návrhu jsem zvažoval, zda umožnit nastavení výchozích hodnot jednotlivým prvkům při definici typu. Rozhodl jsem se upustit od této možnosti, protože narušuje blok, který je určen čistě pro definici (nového typu), přiřazením. To nepůsobí přehledně a mohlo by komplikovat pochopení problematiky.

Z hlediska samotného vytvoření bloku jsem se rozhodl využít tzv. mutátor. Ten umožňuje na základě podmínek měnit tvar a skladbu bloku. Je typicky využit v bloku vytváření polí, kde umožnuje definovat počet prvků v poli nebo v bloku pro definici procedury pro úpravu vstupních parametrů. Oproti těmto návrhům jsem musel pracovat s jednotlivými prvky struktury jako s objekty obsahujícími id a název, abych dosáhl dynamického přejmenování. Díky tomu je možné přejmenovávat prvky přímo na bloku nebo v dialogu který definuje prvky, jejich počet a pořadí, změna se zachová.

Dále potřebujeme blok vytvoření instance. Ten vytvořím jako jednoduchý blok s levým připojením a výběrovým menu z existujících struktur. Do výběrového menu potřebujeme seznam všech dostupných struktur v pracovní ploše, což rozeberu v rámci uložení struktur v kapitole níže.

Dalšími bloky, o kterých nyní víme, že potřebujeme, jsou bloky přiřazení a přečtení hodnoty. Těm jsem se kvůli komplexnějšímu rozhodování při návrhu rozhodl věnovat celou sekci níže.



Obr. 6.1 Blok definice struktury s otevřeným dialogem mutátoru

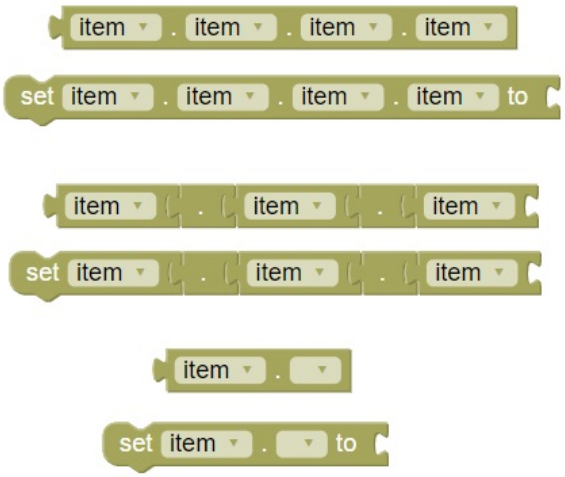
6.1.1 Problém zanořování

Podoba operátorů přiřazeni a čtení z prvku struktury otevřela velkou návrhovou otázku, jak zpracovat tyto bloky. V textových programovacích jazycích (zde myslím jako referenci např. javascript a Java) se přístup k položce objektu provádí přes operátor tečky. Pokud ale budeme mít možnost přiřadit libovolnou hodnotu do prvku instance struktury, můžeme tam samozřejmé přiřadit i další instanci stejné nebo jiné struktury. Tím se dostáváme do vícenásobného zanoření operátorem tečka. Otázkou tedy je, zda vytvořit blok obecného operátoru tečka, bloky get/set s mutátorem, který umožňuje vícenásobné zanořování, které je možné zakončit v libovolné úrovni, nebo výrazně omezit možnosti zanořování na úkor přehlednosti.

První z těchto variant je obecný operátor tečky. Ten umožňuje uživateli využít přesně tolik zanoření, kolik plánuje. Má ale celou řadu problémů. Například pro tři zanoření (`get a.b.c`) vyžaduje využití 5 bloků v řadě. Je tedy zjevně nepraktický pro běžné používání. Další komplikací je potřeba různých bloků pro začátek, střed a konec řetězce podle toho, zda bude zanořování dále pokračovat a blok tedy má mít pravé připojení. Vidíme, že z těchto důvodů není tento přístup příliš vhodný.

Druhou variantou je vytvoření bloku, který by při vybrání prvku struktury, který by sám mohl být instancí struktury, nabídl další zanoření. To by pak šlo zrušit jednoduchým zaklikávacím polem. I tato varianta s sebou ale problémy. Predikce obsahu proměnné, konkrétně zda je instancí určité struktury, je komplexní problém, který rozeberu v další sekci. Navíc i tento blok by byl značně komplikovaný na obsluhu, uživatel by už sice nemusel přidávat značné množství dalších bloků, ale vznikl by dlouhý blok s řadou výběrových menu a k nim přiřazených checkboxů. Navíc při zrušení kteréhokoliv boxu by se blok zkrátil o příslušné zanoření a všechny pod ním. Pokud by to nastalo omylem uživatele, ten by musel znovu vyplnit všechny hodnoty vedoucí až k cílové, což by ve vzdělávacím procesu snadno vedlo ke zmatku a frustraci.

Třetí variantou je omezit obsluhu zanoření pouze na jednu úroveň. Výsledné bloky by pak byly prosté `get(a.b)` a `set(a.b)`. To omezuje uživateli přímý přístup do prvků podstruktur, ale toto omezení je vykoupeno jednoduchostí práce s nezanořenými strukturami. Přístup do hlubších zanoření navíc není zcela vyloučen, pouze je omezen na nepříliš praktické využití pomocné proměnné (např. `var x = a.b; return x.c` pro strukturu `a.b.c`).

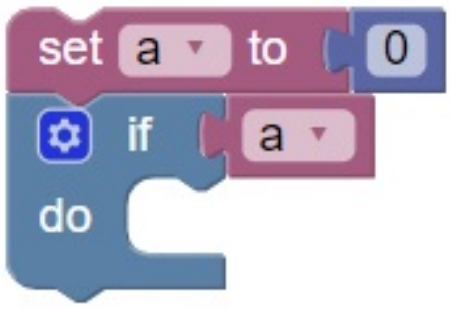


Obr. 6.2 Ukázky variant zanoření A, B a C

Po rozvaze a diskuzi s vedoucím jsem se rozhodl pro třetí variantu. Hlavním argumentem bylo pedagogické zaměření projektu a také s ním související očekávaný rozsah aplikací.

6.1.2 Problém typovosti

Dalším výrazným problémem je typovost bloků proměnných. Ačkoliv Blockly používá zjednodušené typování, např. typ boolean u bloků podmínek, nebo typ number u aritmetických bloků, neověřuje typy u bloků proměnných. Ty umožňuje vložit i do neplatných připojení (viz obrázek).

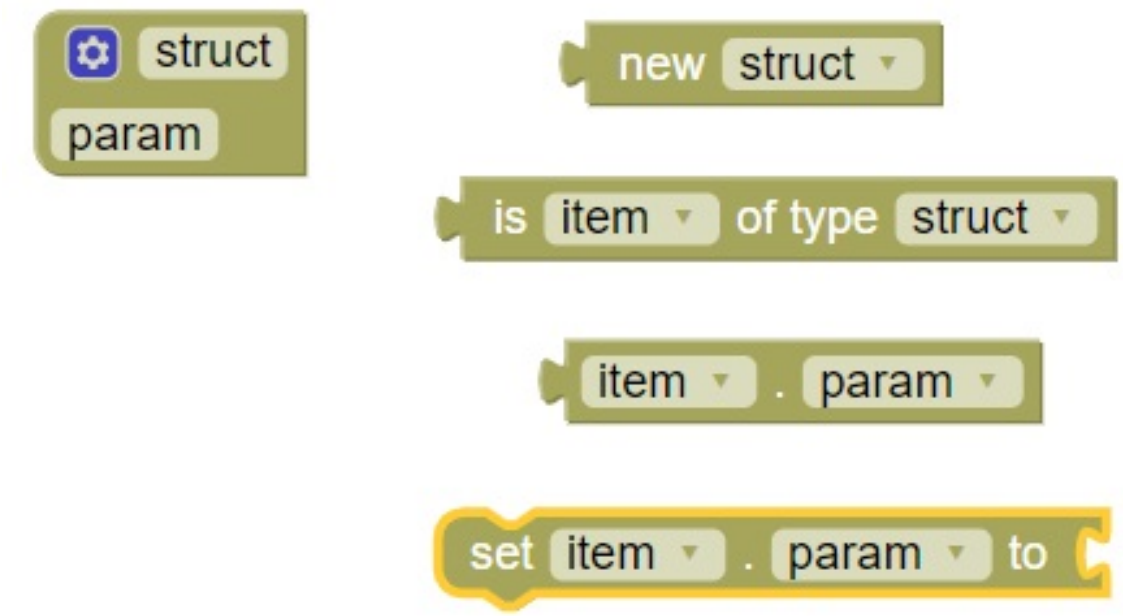


Obr. 6.3 Neplatné připojení bloku do očekávaného logického připojení

Vygenerovaný kód např. v javascriptu sice umožňuje tyto konstrukce, ale nemusí pak proběhnout správně, přičemž je problematické uživatele správně inforomovat o povaze chyby. Tato chyba je pak ještě méně názorná v případě využití procedur a položek procedury jako proměnných. Tehdy není možné zaručit ani zda proměnná vybraná v bloku obsahuje instanci plánované struktury, ani zda její parametr je typu, který chceme použít (jako např. výše zmíněné čislo nebo boolean). Proto jsem se rozhodl přidat do skupiny vybraných bloků také blok `instanceof`, který vrací boolean hodnotu, zda zvolená proměnná je v současném stavu typu struktury, se kterým chceme dál pracovat. To také při rozumném pedagogickém podání umožňuje budovat správné programátorské návyky pro odstranění problému, který je u netypovaných jazyků často výrazně rozšířen.

6.1.3 Výsledné bloky

Z výše uvedeného jsem nakonec vyvodil následující skupinu výsledných bloků.



Obr. 6.4 Výsledný soubor nových bloků

Skupinu bloků v kategorii v *toolboxu* jsem navrhl tak, aby prvním a jediným zobrazeným blokem byl zpočátku blok definice struktury. V okamžiku, kdy existuje alespoň jedna nadefinovaná struktura v pracovní ploše, zobrazí se další bloky pro práci se strukturami.

6.2 Uložení/vyhledávání struktur

Z hlediska návrhu správy struktur jsem se rozhodl zkombinovat oba přístupy zmíněné v kapitole výše(5.2). Struktury se vytváří umístěním bloku definice struktury do pracovní plochy. Na tomto bloku je pak možné provádět úpravy na struktuře. Lze ji přejmenovat nebo smazat, případně lze přidat, přejmenovat nebo smazat její parametry. Každá struktura a každý parametr struktury jsou uloženy pod unikátním identifikátorem (generované uuid), a tedy se správně spárují při přejmenování.

Uloženy jsou ve struktuře `struct_map`, která je parametrem pracovní plochy, podobně jako to je v existujícím kódu u proměnných. Problém je obsluha struktur změnou bloků v ploše. Ten jsem vyřešil aktualizací této mapy struktur při každé změně bloku definice struktury. Definice struktur jsou tak uloženy jak v ploše, tak v mapě na pracovní ploše. Při aktualizaci se naopak změny načítají obdobně jako u procedur pomocí metody na bloku `getStructDef`, která vrátí trojici jméno, id a seznam prvků ve tvaru jméno, id. Ty se vloží do mapy v podobě objektu `struct_model`, případně naleznou podle id v této mapě a aktualizují.

Důvodem, proč využívám oba přístupy, je kombinace jejich výhod. Cílem je možnost snadno měnit definice struktur přímo v ploše a současně zachovat možnost použít u obsluhujících bloků výběrová menu pro existující struktury a jejich prvky. Ty tvoří očekávanou většinu operací nad strukturami; proto se jeví výhodné udržovat mapu struktur na pozadí; z ní je pak snadné nabízet struktury a prvky k dispozici do těchto menu.

Výběrových menu pro blok jsem vytvořil dva druhy. První z nich nabízí dostupné struktury pro bloky `new` a `instanceof`. Druhý pak nabízí prvky z dostupných struktur pro `get` a `set`. Pro oba volám mapu struktur z pracovní plochy a z té vyberu list příslušných prvků pomocí metod `getAllStructs` a `getAllStructsFields`.

7 Generátory kódu

Nedílnou součástí prostředí Blockly jsou také generátory, které převádějí bloky vysázené v pracovní ploše do zdrojového kódu různých programovacích jazyků. Vzhledem k rozdílům mezi jazyky, které jsou součástí prostředí Blockly, jsem musel přistoupit k různým variantám, jak přidané bloky, zmíněné výše, překládat. Generátory mají několik součástí. První částí je generování kostry kódu, kdy se definují proměnné, případně se vytváří struktura zdrojového souboru u jazyků s pevnějšími pravidly pro syntaxi (Dart). Druhou částí je samotný překlad jednotlivých skupin bloků v pracovní ploše. Každý blok má definován svůj zdrojový kód, doplněný o hodnoty vstupů a s úpravami pro komplexnější bloky. Generátor také využívá výčet priorit operátorů u jednotlivých jazyků, aby vyhodnotil správně syntaxi pro např. operátory přiřazení, logické, matematické apod. Pro tyto případy mají bloky kromě kódu také definovanou prioritu operátorů u jednotlivých připojení na bloku.

7.1 PHP a Python

Prvním použitým přístupem je využití třídy u třídních objektových jazyků. V tomto případě nenastávají žádné komplikace. Pro definování struktury mohu využít vytvoření třídy, operátor `new` je pak nativní. Ani jeden z nich navíc není silně typovaný, PHP dokonce umožňuje dynamicky upravovat parametry objektu, a přesto má `typeof` jako vestavěnou funkci, konkrétně `isa`. Python obsahuje dokonce dvě varianty, funkce `type` a `isinstance`. Z nich jsem vybral využití funkce `type` a porovnání. Narozdíl od jazyka PHP jsem v jazyce Python při definici typu definoval i prázdné parametry.

7.2 JavaScript

Domovský jazyk prostředí Blockly je samozřejmě také součástí nabídky. Jeho použití je nejširší, protože je využíván i pro demo aplikace, které jsou součástí knihovny. Podobně jako v jazyce PHP jsem mohl využít objekty, které nemají pevnou strukturu, tedy jsou nemusel definovat jejich parametry. Komplikace, která se vyskytuje, je absence principu tříd. JavaScript jako prototypovaný jazyk nevyužívá koncept tříd, což znesnadňuje operace `new` a `typeof`. Rozhodl jsem se tyto problémy vyřešit vytvořením vzorového objektu, který bych operátorem `new` následně kopíroval. v Javascriptu ale není ani nativní podpora hluboké kopie objektu. Byl jsem tedy nucen použít méně praktické řešení, a to uložení objektu ve formátu JSON a následně vytvořením nového načtením řetězce tohoto formátu. Operátor `typeof` pak představoval další problém, neboť JavaScript nepočítá s konceptem vytváření více objektů náležejících pod nějakou entitu. Rozhodl jsem se tedy nakonec parametry vkládat při definici struktury, a operátor `typeof` pak porovnává pole parametrů.

7.3 Dart

Jazyk Dart má z programovacích jazyků obsažených v Blockly syntaxi nejvíce podobnou jazyku C. HLavní komplikací Dartu je poměrně striktní struktura kódu programu, daná metodou `main` a vyžadující definice mimo tuto metodu. Proto jsem u tohoto překladače zcela vynechal kód vytvořený z bloku definice struktury. Ty místo toho definuji v sekci kostry kódu přimo z mapy struktur na pozadí. Další komplikací je silná typovost v objektech. Tu jsem musel obejít pro nedostatečnou typovost Blockly využitím typu řetězce u všech objektů. Jinak je třídně typový, obsahuje nativní operátory `new` a `typeof`.

7.4 Lua

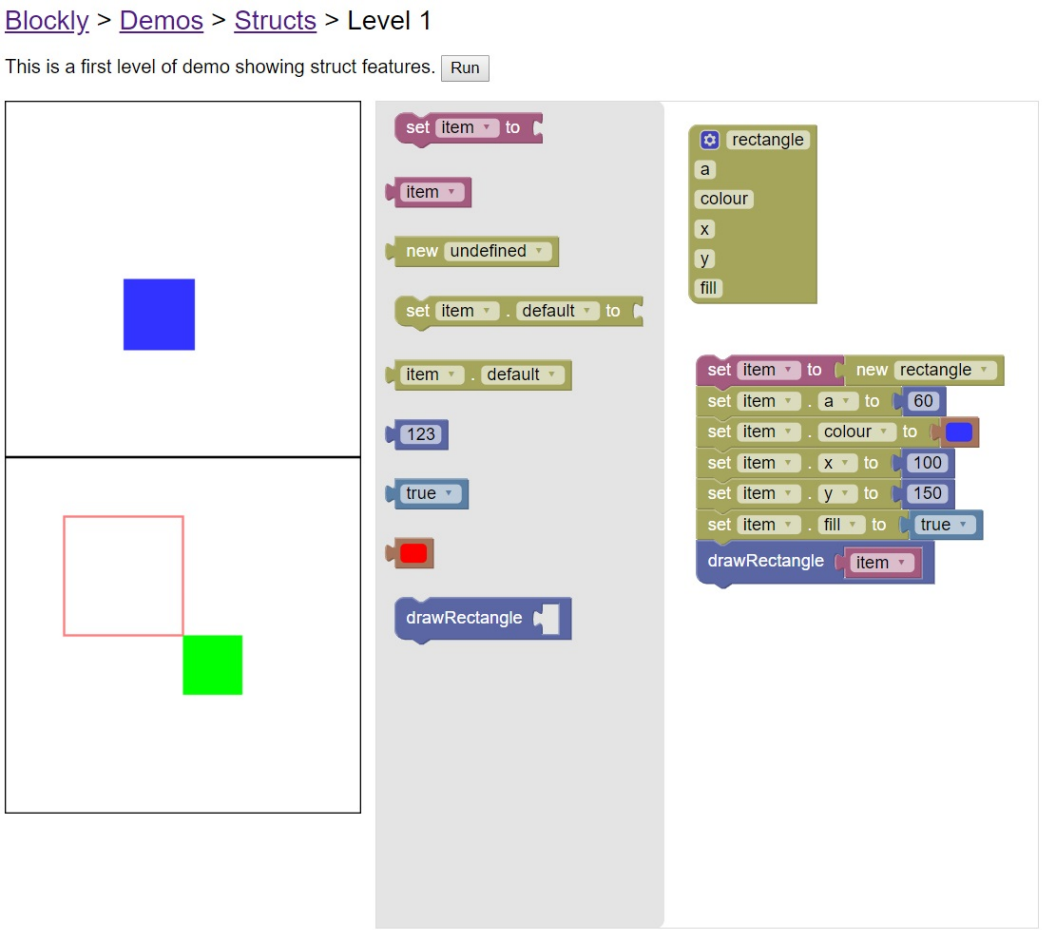
Jazyk Lua je nejvíce vzdálen tradiční predstavě o objektovém (případně třídné objektovém) programovacím jazyku. Obsahuje některé prvky objektového přístupu, konkrétně možnost definice metod a třídy, ale nevyužívá dědičnost apod. v pravém slova smyslu. Parametry třídy pak fungují jako asociativní pole, které je základním datovým typem. Přístup k parametrům objektu je vzhledem k této povaze objektů velmi jednoduchý. Problém je s vytvářením nových instancí objektu a zvláště s jejich porovnáváním pro operátor `typeof`. Pro tyto jsem byl nucen doprogramovat jednoduché metody, které obsahuje každá struktura v kódu pro blok definice, a které se potom využívají. Jde o jednoduchý bezparametrový konstruktor, který vytváří novou instanci a naplní všechny parametry nulovými hodnotami, a také o metodu `type`. Tu jsem navrhl jako vracející statický řetězec sdružující název typu a list názvů všech jeho parametrů. Vzhledem k podstatě vytváření isntancí objektů kopírováním třídy pak tyto metody má i třída; operátor `typeof` je jen porovnáním řetězců.

8 Ukázková aplikace

Jako příklad využití nové funkcionality v prostředí Blockly jsem navrhl dvě různá cvičení využívající nové bloky. V obou jsou předdefinované struktury, které se při spuštění kódu vykreslují do plochy pomocí HTML canvas.

8.1 Příklad 1 - kreslení čtverců

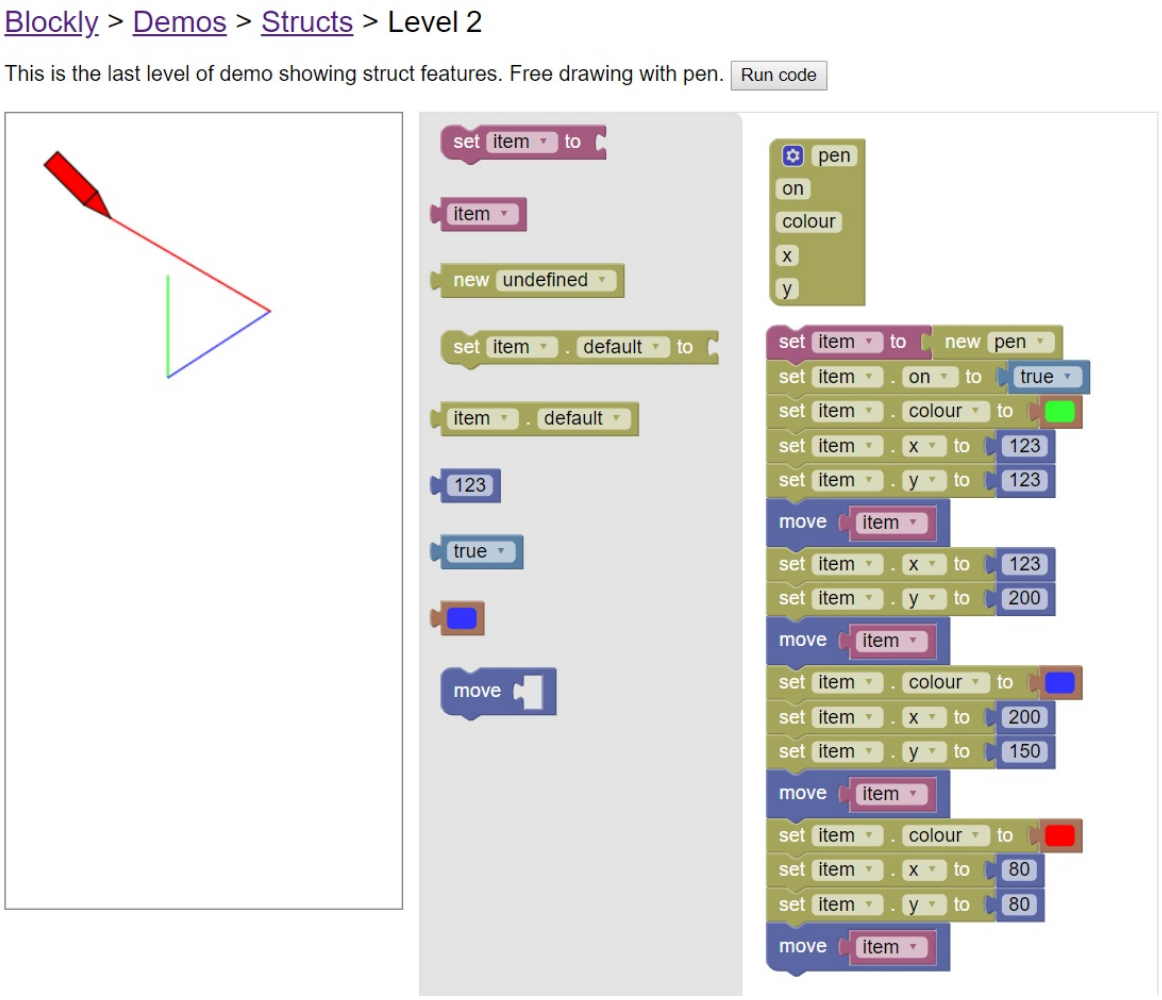
V první aplikaci je definovaná struktura čtverce. Ta má parametry určující souřadnice středu, délku strany a, barvu a zda bude vyplněn. Uživatel si může čtverce libovolně vytvářet, upravovat a vykreslovat. Pro tento účel je přidán specifický blok `drawRectangle`, který vykreslí čtverec do něj zaslaný jako parametr. V případě vložení jiné hodnoty jako parametru tohoto bloku program zobrazí chybové hlášení. Součástí stránky je také vzorový obraz s několika čtverci, vykreslený pomocí stejného kódu; toho lze využít jako předlohy pro cvičení. Spuštění kódu se pak provádí po překladu a kontrole vygenerovaného kódu jednoduchým `eval`. Přiklad je na obrázku níže.



Obr. 8.1 Ukázka prvního cvičení

8.2 Příklad 2 - kreslící pero

Tato aplikace nabízí strukturu pera. Tomu lze nastavit souřadnice, barvu a zapnout/vypnout kreslení. Blokem přidaným v tomto cvičení je `move`, který umožnuje pohyb perem na nové souřadnice. Podobně jako v předchozím cvičení, při zaslání jiného parametru než pera, zobrazí chybovou hlášku. Vyhodnocení zdrojového kódu probíhá složitěji než v předchozím cvičení. Po přeložení a kontrole se kód obdobně spustí funkcí `eval`. V tomto bodě se ale pouze naplní pole objekty typu pero. Poté se ve smyčce vykresluje pero postupně v jednotlivých kreslicích polohách s příslušnou částí vykresleného obrazu. K tomu jsou využity funkce `window.requetsAnimationFrame` se zpožděním pro vykreslování krok po kroku. Příklad je opět vidět níže.



Obr. 8.2 Ukázka druhého cvičení

ZÁVĚR

V práci jsem rozebral jednotlivé varianty grafických programovacích jazyků, zmínil jejich technologie a využití. Nastínil jsem možnosti jejich rozšíření včetně výhod a nevýhod. Z uvedených jsem si vybral knihovnu Blockly. Pro tu jsem navrhl rozšíření o bloky pro vytváření struktur, podobně jako v jazyce C. Vytvořil jsem nové bloky pro operace potřebné k obsluze struktur a zpracoval režii na pozadí prostředí Blockly pro jejich obsluhu, včetně vzájemných kontrol při úpravách, přidávání a odebírání. Následně jsem pro tyto bloky upravil generátory obsažené v knihovně Blockly o schopnost překládat nově vytvořené bloky do výběru programovacích jazyků obsažených v knihovně. Posledním krokem bylo vytvoření ukázkové aplikace pro prezentaci výhod a využití nových bloků.

Největší komplikací a překvapením při vypracovávání práce byla rozhodně náročnost úprav jádra knihovny. Při vypracovávání této části jsem byl nucen několikrát přepracovat svůj návrh pro vyřešení mnoha problémů, které jsou způsobeny vzájemně logicky závislými bloky. Na pozadí probíhají aktualizace výběrových menu, seznamů existujících struktur, kontrola úprav jednotlivých struktur apod. při každé změně libovolného bloku. Oproti tomu vytváření bloků samotných, generování kódu a návrh ukázkových aplikací již patřilo, z velké části díky nástrojům a příkladům poskytnutým v knihovně, mezi méně náročné části práce.

Pokud bych zde měl zmínit možnosti pokračování a dalšího rozšíření práce, zjevným dalším krokem by bylo rozšířit knihovnu Blockly o kompletní funkcionalitu objektů. Nejprve pravděpodobně možnost vkládání metod, dále implementace např. přetěžování nebo dědičnosti. Při návrhu rozšíření v počátečnich fázích této práce byla tato možnost zvažována, ale bylo zjištěno, že by tyto úpravy řádově převyšovaly předpokládaný rozsah bakalářské práce. V průběhu vypracovávání jsem ale s těmito možnostmi počítal, a proto jsou výrazné části kódu navrženy pro eventualitu dalšího rozpracování tímto směrem.

Na tomto místě bych také rád uvedl, že ačkoliv zapojení tohoto rozšíření zpátky do centrálního repozitáře projektu Blockly není nedílnou podmínkou zadání, v době psaní této části práce se rovněť zabývám splněním podmínek pro ucházení se o zapojení do projektu, což považuji za logický výstup této práce. Jedná se ale o rozsáhlý proces a neočekávám, že bude dokončen v době obhajoby této práce.

SEZNAM POUŽITÉ LITERATURY

[1] *Výuka informatiky na školách se mění, zaměří se na programování* [online]. [cit. 2017-03-27] Dostupný z WWW: `https://archiv.ihned.cz/c1-65984410-vyuka-informatiky-na-skolach-se-meni-zameri-se-na-programovani`

[2] *Rámcový vzdělávací program pro základní vzdělávání.* [online]. Praha: MŠMT, 2013. 142 s. [cit. 20147-26-03]. Dostupné z WWW:`http://www.msmt.cz/file/43792_1_1/`

[3] KREJSA Jan, *Výuka základů programování v prostředí Scratch*, České Budějovice 2014. Diplomová práce, Jihočeská univerzita v Českých Budějovicích, Pedagogická fakulta, Katedra informatiky

[4] *Computing our future* [online]. [cit. 2018-03-27]. Dostupný z WWW: `http://fcl.eun.org/documents/10180/14689/Computing+our+future_final.pdf/746e36b1-e1a6-4bf1-8105-ea27c0d2bbe0`

[5] Ugur Tevfik Kaplancali, Zafer Demirkol. Teaching Coding to Children: A Methodology for Kids 5+. International Journal of Elementary Education. Vol. 6, No. 4, 2017, pp. 32-37. doi: 10.11648/j.ijeedu.20170604.11

[6] WESTLAND Frank, *Teaching Strategies for Programming Languages: Deductive vs. Inductive Learning*, Tilburg,TheNetherlands October, 2013. Master thesis, Tilburg University, School of Humanities, Department of Communication and Information Sciences

[7] *Introductory Programming and the Didactic Triangle* Berglund, Anders and Lister, Raymond [online]. [cit. 2018-03-27] Dostupný z WWW: `http://crpit.com/confpapers/CRPITV103Berglund.pdf`

[8] KAASBØLL Jens J., 1998, *Exploring didactic models for programming*, Department of Informatics, University of Oslo

[9] *The maturity of visual programming* Dehouck, Rémi [online]. [cit. 2018-03-27] Dostupný z WWW: `http://www.craft.ai/blog/the-maturity-of-visual-programming/`

[10] FRASER, Neil. *Ten things we've learned from Blockly.* In: 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond) [online]. IEEE, 2015, 2015, s. 49-50 [cit. 2018-03-29]. DOI: 10.1109/BLOCKS.2015.7369000. ISBN 978-1-4673-8367-7. Dostupné z: `http://ieeexplore.ieee.org/document/7369000/`

[11] *Programming language* [online]. [cit. 2018-03-27]. Dostupný z WWW: `https://techterms.com/definition/programming_language`

[12] *object-oriented programming (OOP)* Rouse, Margaret [online]. [cit. 2018-03-27] Dostupný z WWW: `http://searchmicroservices.techtarget.com/definition/object-oriented-programming-OOP`