Jihočeská univerzita v Českých Budějovicích
Pedagogická fakulta
Katedra informatiky

Bakalářská práce

Výuka programování v prostředí Scratch a Snap!

Autor: Jan Studený
Vedoucí práce: doc. PaedDr. Jiří Vaníček, Ph. D.

České Budějovice 2015

Jihočeská univerzita
v Českých Budějovicích
University of South Bohemia
in České Budějovice

University of South Bohemia
Faculty of Education
Department of Informatics

Bachelor's thesis

Programming in Scratch and Snap! environments

Author: Jan Studený
Supervisor: doc. PaedDr. Jiří Vaníček, Ph. D.

Budweis 2015

JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH
Fakulta pedagogická
Akademický rok: **2012/2013**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Jan STUDENÝ**
Osobní číslo:	**P10369**
Studijní program:	**B7507 Specializace v pedagogice**
Studijní obor:	**Informační technologie ve vzdělávání**
Název tématu:	**Výuka programování v prostředí Scratch a Snap!**
Zadávající katedra:	**Katedra informatiky**

Zásady pro vypracování:

Scratch je jeden z vedoucích světových produktů umožňujících výuku základů programování pro děti. Je zdarma, v češtině a nyní i ve verzi 2.0. Je vyvíjen na Massachusetts Institute of Technology. Nadstavbou původní verze Scratch pro pokročilejší programování je Snap! (v původní verzi pro Scratch 1 s názvem BYOB - Build your own blocks). V teoretické části práce student porovná vývojová prostředí Snap! a Scratch 2, zejména v oblasti možností výuky, komfortu prostředí, snadnosti zápisu a editace algoritmů, práce s daty, práce s grafikou a multimédii, české lokalizace, programovacích konceptů; práce s objekty. Taktéž zpracuje porovnání s původním prostředím Scratch 1 a BYOB. V praktické části pak vytvoří sadu vypracovaných úloh/programů, které budou učit programovat na pokročilejší úrovni a na kterých budou demonstrovány specifické funkce a vlastnosti prostředí Snap!. K některým úlohám lze přiložit i ukázku řešení z prostředí Scratch 2 pro možnost porovnání.

Rozsah grafických prací: **CD ROM**

Rozsah pracovní zprávy: **40**

Forma zpracování bakalářské práce: **tištěná**

Seznam odborné literatury: **viz příloha**

Vedoucí bakalářské práce: **doc. PaedDr. Jiří Vaníček, Ph.D.**
Katedra informatiky

Datum zadání bakalářské práce: **16. dubna 2013**

Termín odevzdání bakalářské práce: **30. dubna 2014**

Mgr. Michal Vančura, Ph.D.
děkan

doc. PaedDr. Jiří Vaníček, Ph.D.
vedoucí katedry

V Českých Budějovicích dne 16. dubna 2013

Příloha zadání bakalářské práce

Seznam odborné literatury:

1. RESNICK, Mitchel. Scratch: Reference Guide [online]. 2009. Dostupné z: http://info.scratch.mit.edu/sites/infoscratch.media.mit.edu/files/file/ScratchRefe

2. HARVEY, Brian a Jens MÖNIG. BYOB: Reference Manual [online]. 2011. Dostupné z: http://byob.berkeley.edu/BYOBManual.pdf

3. HARVEY, Brian a Jens MÖNIG. Snap!: Reference Manual [online]. 2013. Dostupné z: http://snap.berkeley.edu/SnapManual.pdf

4. RESNICK, Michael, Yasmin KAFAI a John MAEDA. A Networked, Media-Rich Programming Environment to Enhance Technological Fluency at After-School Centers in Economically-Disadvantaged Communities. MIT Media Laboratory, UCLA, 2003. Dostupné z: http://web.media.mit.edu/ mres/papers/scratch.pdf

5. RESNICK, Mitchel, Brian SILVERMAN, Yasmin KAFAI, John MALONEY, Andrés MONROY- HERNÁNDEZ, Natalie RUSK, Evelyn EASTMOND, Karen BRENNAN, Amon MILLNER, Eric ROSENBAUM a Jay SILVER. Scratch. Communications of the ACM [online]. 2009, roč. 52, č. 11, s. 60-. ISSN 00010782. DOI: 10.1145/1592761.1592779. Dostupné z: http://web.media.mit.edu/ mres/papers/Scratch-CACM-final.pdf

6. MONROY-HERNÁNDEZ, Andrés a Mitchel RESNICK. Empowering kids to create and share programmable media. Interactions [online]. 2008, roč. 15, č. 2, s. 50-. ISSN 10725520. DOI: 10.1145/1340961.1340974. Dostupné z: http://web.media.mit.edu/ mres/papers/interactions-scratch-08.pdf

7. RUSK, Natalie, Mitchel RESNICK a John MALONEY. LIFELONG KINDERGARTEN GROUP, MIT Media Laboratory. Learning with Scratch: 21st Century Learning Skills [online]. Dostupné z: http://info.scratch.mit.edu/sites/infoscratch.media.mit.edu/files/file/translated-docs/Scratch-21stCenturySkills.pdf

8. The Beauty and Joy of Computing [online]. Dostupné z: http://bjc.berkeley.edu/

9. Programming concepts and skills supported in Scratch [online]. Dostupné z: http://info.scratch.mit.edu/sites/infoscratch.media.mit.edu/files/file/translated-docs/ScratchProgrammingConcepts-v14.pdf

10. Scratch - Imagine, Program, Share: Create stories, games, and animations. Share with others around the world [online]. 2013. Dostupné z: http://beta.scratch.mit.edu/

Prohlášení

Prohlašuji, že jsem svoji bakalářskou práci vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích dne 26.6.2015 .... Jan Studený

Anotace

Scratch 2.0 je výukové, ikonické, skriptovací programovací prostředí zaměřené na děti ve věku od 8 do 16 let; a vyvíjené skupinou Lifelong Kindergarten na Massachusettském technologickém institutu (MIT). Skládáním grafických komponent v podobě různých dílků v něm tvoříme scénář, který jednotliví maskoté odehrávají na scéně a ztvárňují tak náš příběh. K tomuto prostředí však existuje alternativní prostředí Snap*!* (dříve pod názvem BYOB jako modifikace první generace Scratch), vyvíjené Jensem Mönigem ve spolupráci s Brianem Harveym, jež se snaží odstranit nedostatky, překážky a chudší nabídku možností v oblasti programování, kterýmiž právě Scratch trpí.

Práce nejprve představuje ona dvě prostředí v jejich majoritních verzích či generacích (dle pořadí vývoje tedy: Scratch 1, BYOB, Scratch 2; a Snap*!*), dále je všechny podrobně srovnává dle vybraných kritérií, a nakonec popisuje nové programovací koncepty a z toho pramenící možnosti či techniky, které lze v prostředí Snap*!* na rozdíl od Scratch – a v některých případech i jeho předchůdci BYOB – aplikovat. Součástí jsou nejenom různé algoritmické příklady, ale také autorem této práce vyvinuté projekty v prostředích Snap a Scratch.

K pochopení bakalářské práce je podmínkou zkušenost s alespoň jedním z prostředí Scratch 1, Scratch 2, anebo BYOB. Tato práce byla vytvořena sázecím systémem TEX s využitím maker OPmac.

Klíčová slova: Scratch 1 & 2, BYOB, Snap*!*, srovnání, programování, výuka

Abstract

Scratch 2.0 is educational, iconic, scripting programming environment focused on children in range of age from 8 to 16, and being developed by Lifelong Kindergarten group on Massachusetts technology institute (MIT). By composing graphical components in the form of various puzzle pieces we create scripts which individual maskots are acting on the scene and externalize our story. To this environment also exists alternative Snap*!* (previously called BYOB as modification of Scratchs' first generation), developed by Jens Mönig in cooperation with Brian Harvey, which is trying to remove deficiencies, barriers and poorer possibilities in programming by which Scratch is suffering.

Thesis firstly introducing both environments in their major versions or generations (in order of development: Scratch 1, BYOB, Scratch 2; and Snap*!*), secondly comparing all of them based on chosen criterias, and finally describing new programming concepts and from them incoming possibilities and techniques, which can be in Snap*!* unlike in Scratch – and in some cases even in its predecessor BYOB – applied. Included are not only algorithmic examples, but also projects in Snap and Scratch 2 developed by author of this bachelors thesis.

To understand this bachelor thesis is required to have experience with at least one of the environment Scratch 1, Scratch 2, or BYOB. This thesis was created by typesetting system TEX with usage of OPmac macros.

Keywords: Scratch 1 & 2, BYOB, Snap*!*, comparison, programming, education

Poděkování

Děkuji svému vedoucímu práce, panu docentu Jiřímu Vaníčkovi, za jeho důvěru svěřit mi zadání bakalářské práce do rukou, za nespočet absolvovaných konzultací, za nápady k řešení postupně objevujících se překážek; a celkově za pozitivní přístup a výbornou spolupráci.

Dále děkuji všem zúčastněným ve vývoji prostředí Snap*!*, kteří se mnou – ať už na webové službě GitHub či přes e-mail – komunikovali, spolupracovali a též zodpovídali mé dotazy; tj. uživatelům `jmoenig`, `brianharvey`, `cycomachead`, `nathan` a `Gubolin`.

Obsah

1 Úvod do bakalářské práce . . . 12
1.1 Problematika . . . 12
1.2 Cíle práce . . . 12
1.3 Metoda práce . . . 13
1.4 Vysvětlivky k bakalářské práci . . . 14
2 Představení prostředí . . . 16
2.1 Prostředí Scratch 1 . . . 16
2.2 Modifikace Scratch 1 – prostředí BYOB . . . 20
2.3 Současná verze Scratch – Scratch 2.0 . . . 22
2.4 Současná verze BYOB – prostředí Snap 4.0 . . . 24
3 Porovnání prostředí . . . 26
3.1 Rozdíly a možnosti při programování . . . 26
3.1.1 Kategorie bloků a vše s jejich tvorbou spojené . . . 26
3.1.2 Proměnné a konstanty . . . 28
3.1.3 Datové typy . . . 29
3.1.4 Parametry bloků . . . 35
3.1.5 Koncepty objektově orientovaného programování . . . 39
3.1.6 Programování rekurze . . . 47
3.1.7 Kostýmy, grafické efekty, scéna, hudba, kreslení . . . 47
3.1.8 Videokamera, mikrofon, přepis řeči do a předčítání textu . . . 49
3.1.9 Aktuální datum a čas . . . 50
3.1.10 Turbo mód a provedení scénáře bez obnovy obrazovky 51
3.1.11 Porovnání znaků a ověření původu hodnot . . . 51
3.1.12 Události a rozesílání zpráv . . . 53
3.1.13 Spouštění a pozastavení projektu, zastavování scénářů 55
3.1.14 Ladění scénářů . . . 56
3.1.15 Hlášení chyb a zobrazování výjimečných hodnot . . . 58
3.1.16 Zbylé doposud nezmíněné primitivy . . . 59
3.2 Práce s grafikou, zvukem a multimédii . . . 61
3.2.1 Podporované grafické a zvukové formáty . . . 61
3.2.2 Zabudované editory, záznam zvuku, snímky z kamery 62
3.3 Komfort a uživatelská přívětivost prostředí . . . 63
3.3.1 Snadnost skládání a editace scénářů . . . 63
3.3.2 Správa projektu . . . 65
3.3.3 Ztlumení zvuků . . . 67
3.3.4 Nápověda k primitivním blokům . . . 67
3.3.5 Nabídky pro pokročilejší, uživatelský a vývojový mód 67
3.3.6 Velikost bloků, jejich popisků, sledovačů a mód scény 67
3.3.7 Focení scénářů, definic bloků, scény, celého projektu . 68
3.3.8 Krátká zmínka o dalším nastavení prostředí . . . 68
3.4 Převody projektů mezi prostředími . . . 69
3.5 Modifikace, externí a interní rozšíření, knihovny . . . 70

3.5.1 Modifikace prostředí a přispívání k vývoji 70
3.5.2 Interní a externí rozšíření 70
3.6 Propojení prostředí a týmový vývoj projektu 72
3.6.1 Nabízený obsah ze zabudovaných knihoven 73
3.7 Dostupné lokalizace 75
3.8 Závěr porovnání 76
4 Využití nových programovacích konceptů a funkcionalit prostředí Snap v příkladech 77
4.1 Vlastní funkce a podmínky, rozbalovací nabídka parametrů, imitace konstant 78
4.1.1 Vlastní funkce 78
4.1.2 Vlastní podmínky 85
4.1.3 Rozbalovací nabídky parametrů 88
4.1.4 Imitace konstant 94
4.2 Seznamy 96
4.2.1 Seznam jako typ parametru 96
4.2.2 Přidané funkce pro práci se seznamy 97
4.2.3 Vícenásobná varianta parametru 100
4.2.4 Vnořené seznamy 104
4.2.5 Kruhové seznamy 106
4.2.6 Kopírování seznamů 108
4.2.7 Tvorba datových struktur vnořenými seznamy . . . 109
4.3 Zaobalené procedury 112
4.3.1 Zaobalená procedura namísto vlastního bloku . . . 113
4.3.2 Zaobalená procedura použita ke změně chování . . 116
4.3.3 Rekurze u zaobalených procedur a skrývání některých parametrů koncové rekurze 117
4.4 Ostatní dosud neužité typy a varianty parametrů 119
4.4.1 Parametry vyžadující bloky příkazů 119
4.4.2 Parametry vyžadující bloky funkcí či podmínek . . 124
4.4.3 Parametry typu „objekt“ a „pravdivostní hodnota“ 128
4.4.4 Obsah automaticky zaobalující typy parametrů . . 130
4.4.5 Parametr přesahující varianty 133
4.5 Bloky (JavaScript funkce) a [zahaj], přepis vybraných bloků do textové podoby 136
4.5.1 Funkce (JavaScript funkce) 136
4.5.2 Příkaz [zahaj] zahajující scénář v novém procesu . 138
4.5.3 Přepis bloků do textové podoby 140
5 Závěr bakalářské práce 148
A Základní informace k vlastním blokům 149
A.1 Tvorba a editace, přidání nápisů a parametrů 149
A.2 Změna kategorie a typu, mazání bloku 150
B O proměnných scénáře 151
B.1 Vytvoření a přejmenovávání 151
B.2 O přístupnosti a sledovačích 151

B.3 Procvičující ukázky 152
C O referencích na data 153
C.1 Obecný popis (vymezení pojmů) 153
C.2 Uchování a sdílení reference 153
C.3 Porovnání referencí 155
D Důležité informace pro práci se seznamy 156
D.1 Operace nad seznamy 156
D.2 Export a import seznamů 158
E Přehled přidaných bloků k sadě ze Scratch 1 160
F Upravená čeština a její instalace do Snap 161
F.1 Instalace upravené češtiny do on-line verze 161
F.2 Instalace upravené češtiny do off-line verze 161
G Přehled projektů s odkazy do prostředí 162
Reference 163

Kapitola 1
Úvod do bakalářské práce

1.1 Problematika

Programování je bezesporu složitou disciplínou, která vyžaduje nejenom znalost konkrétního programovacího jazyka – jeho syntaxe, knihoven a charakteristických vlastností; ale také odlišný způsob myšlení k řešení specifických úloh. Už i proto se snažíme žáky nejenom základních škol začlenit do tajů programování a podpořit tak jejich rozvoj a přístup k logickému myšlení.

To si dávají za cíl edukační programovací jazyky či prostředí, která jsou navržena především jako výukový nástroj k programování než jako nástroj na řešení problémů reálného světa.[30] Patří do nich i kategorie dětských programovacích jazyků zaměřující se na děti různého věku. Mezi známější patří například jazyk Logo, robot Karel, Baltík, Petr, Alice, Greenfoot, Lego Mindstorms, anebo Scratch 2.

A právě vizuální prostředí Scratch 2, založené na idejích jazyka Logo[8, s. 1], nabírá v posledních letech na své popularitě. Je plně zdarma, v češtině, a v on-line i off-line distribuci. Programováním v něm vytváříme postavám – tzv. maskotům – různé scénáře, kterými pak ztvárňují příběhy na scéně projektu. Zároveň je možné programovat hudbu a želví grafiku, pracovat s bitmapami a vektory, hudbou a multimédii; využívat proměnné, z datových struktur seznam; a mnoho dalšího.

Na své speciální Scratch 2 síti umožňuje programátorům ukládat a sdílet projekty s ostatními uživateli. I přesto, že k programování v tomto prostředí netřeba registrace, je do současnosti registrováno více než 5,75 milionu uživatelů, sdílelo téměř 8,5 milionu originálních či odvozených projektů, a napsáno přes 40 milionů komentářů.[39]

Toto prostředí však nevyužívá z hlediska programovacích konceptů plně svého potenciálu a vede tak začínající programátory k mnohdy zbytečně složitým – a v některých případech i zrůdným – programovým řešením. Z tohoto důvodu se Jens Mönig a Brian Harvey rozhodli vyvinout k němu alternativní prostředí Snap*!*, které se ovládá a vypadá téměř identicky, avšak převyšuje jej svými možnostmi v programování. A právě tuto z programátorského hlediska propracovanější alternativu je zapotřebí představit.

1.2 Cíle práce

Cílem bakalářské práce je představit uživatelům alespoň jednoho z prostředí Scratch 1, Scratch 2, anebo BYOB; ono alternativní prostředí Snap*!* v několika kapitolách.

Práce nejprve seznámí čtenáře s postupným vývojem všech již zmíněných prostředí a také se zásadními rozdíly mezi nimi. Dozví se tedy kdy, kým, v jakém programovacím jazyce a proč tato prostředí vznikla; a v čem jsou či byla unikátní.

Dále nabídne čtenáři představu o stavu, podobnosti, rozdílech, a možnostech všech prostředí skrze poskytnuté srovnání, jež bylo provedeno na základě několika kritérií. Ta byla stanovena těmi oblastmi, se kterými přichází do styku většina uživatelů každého prostředí. Konkrétně jde o srovnání rozdílů: v programování; při práci s grafikou, zvukem a multimédii; komfortu a uživatelské přívětivosti; publikování a převodu projektů; propojování prostředí a týmového vývoje; modifikací, rozšíření a knihoven; a dostupných lokalizací. Tato část je mostem pro snazší přechod uživatelů předchozích prostředí do – pro ně absolutně neznámého – prostředí Snap*!*. Závěrem je celkové shrnutí.

Poslední část vysvětluje autorem vybrané oblasti programování z prostředí Snap*!*, které jsou – na rozdíl od Scratch 2 – v něm aplikovatelné. Jde i o další programovací koncepty, funkcionality, a z toho pramenící techniky; kterých nelze ve Scratch 2 jednoduchou cestou dosáhnout. K demonstraci popisované problematiky slouží autorem vytvořené a v praxi užitné krátké příklady a též vyvinuté projekty převážně v prostředí Snap*!*, výjimečně pak i v prostředí Scratch 2.

1.3 Metoda práce

Při vytváření bakalářské práce se bude postupovat podle následující metody založené na pěti hlavních bodech níže popsaných.

- **Vyhledání zdrojů**

Nejprve je zapotřebí zjistit si potřebné informace o všech popisovaných prostředích. Proto autor práce nastuduje jím vyhledané a vybrané zdroje z různých k tomu určených vyhledávacích portálů a internetových stránek – převážně pak:

  - z portálu ScratchED[1])
  - ze stránek o výzkumných publikacích věnující se prostředí Scratch[2])
  - ze stránek Mitchela Resnicka, lídra Scratch týmu, na nichž uvádí své publikace[3])

- **Zjištění stavu prostředí Snap*!***

V důsledku neustálého vývoje prostředí Snap*!* je zapotřebí o něm získat více informací k sestavení výsledné představy o jeho stavu a možnostech. Po přečtení manuálu tohoto prostředí autor sesbírá další informace z těchto zdrojů:

  - oficiální vlákna na Scratch fóru pro uživatele a vývojáře prostředí Snap*!*[4])
  - repositář prostředí Snap*!* na webové službě GitHub,[5]) který již nyní obsahuje téměř 850 různých hlášení o chybách, nedostatcích, ale i o návrzích a vylepšení
  - soubor history.txt[6]) obsahující záznamy o změnách každé aktualizace; a
  - prohlédne oficiální ukázkové příklady[7]) a dema[8])

- **Porovnání prostředí**

Po prostudování potřebných zdrojů a zjištění aktuálního stavu prostředí Snap*!* se autor vrhne na porovnání všech prostředí mezi sebou. Některá porovnávaná témata budou obsahovat stručné srovnávací tabulky ke zvýraznění rozdílů či podobností.

- **Sestavení pravidel pro vytvářené projekty**

Před vývojem projektů je zapotřebí sestavit základní, na ně aplikovaná, pravidla pro zajištění jednodušší orientace v nich a k zachování podobného způsobu ovládání.

Každý projekt tedy bude podléhat alespoň těmto základním pravidlům:

  - pro čtenáře nedůležité algoritmy, které projekty potřebují pro svou činnost, budou v rámci možností ukryty ve speciální blocích (např. „připrav postavu“)
  - všechny scénáře, popisky a názvy naprogramovaných bloků budou v češtině
  - pokud je to možné, pak se bude projekt spouštět kliknutím na zelený praporek
  - projekty Snap*!* obsahující jen ukázkové bloky můžou mít skrytou scénu
  - každý projekt bude uložen a sdílen na cloud účtu autora této práce

- **Sestavení obsahu a vývoj projektů**

Na základě prostudovaných zdrojů a porovnání prostředí autor sestaví různé oblasti programování, kterým se lze v prostředí Snap*!* věnovat a na které se práce zaměří.

Nakonec dojde k vývoji projektů doplňující ony vybrané oblasti programování a to především v prostředí Snap*!*, výjimečně pak i v prostředí Scratch 2. Stanovená pravidla z předchozího bodu metody práce budou aplikována na každý projekt.

---

[1]) http://scratched.media.mit.edu/resources/
[2]) http://scratch.mit.edu/info/research/
[3]) http://web.media.mit.edu/~mres/papers.html
[4]) Vlákno pro uživatele zde: http://scratch.mit.edu/discuss/topic/4455/ a vlákno pro vývojáře pak zde: http://scratch.mit.edu/discuss/topic/4464/
[5]) http://github.com/jmoenig/Snap--Build-Your-Own-Blocks
[6]) http://snap.berkeley.edu/snapsource/history.txt
[7]) http://snap.berkeley.edu/snapsource/Examples/
[8]) http://snap.berkeley.edu/snapsource/demo/

1.4 Vysvětlivky k bakalářské práci

- **Odkazy na části dokumentu a jejich číslování**

Odkazy na kapitoly, sekce, podsekce, dodatky, obrázky, tabulky a reference jsou vysázeny se zelenou barvou a v elektronické verzi jsou navíc klikacími.

Číslo odkazu však není kompletní. Například 3.1.3 se odkazuje do první sekce třetí kapitoly, ale už nelze odhadnout, jde-li o třetí podsekci, obrázek, či tabulku (viz odlišné 3.1.3, 3.1.3, a 3.1.3). Proto je vždy před odkaz doplněn onen pojem k přesnému určení.

- **Sázení názvů prostředí**

Jelikož práce popisuje několik prostředí a ještě v různých verzích, bylo zavedeno konkrétní značení. Následující výčet čtenáři odlišné názvy vysvětlí:

- Scratch 1 – reprezentuje finální verzi 1.4 první generace Scratch
- Scratch 2 – reprezentuje současnou verzi 2.0 druhé generace Scratch
- Scratch – vztahuje se k oběma generacím, jak k Scratch 1 tak i k Scratch 2
- BYOB – akronym modifikace prostředí Scratch 1; sázeno velkými písmeny
- Snap – prostředí Snap*!* ve verzi 4.0; z typografických důvodů se nesází vykřičník

- **Kategorie, bloky, argumenty, parametry a vrácené hodnoty v prostředích**

Protože je třeba čtenáři výklad co nejvíce přiblížit, zaveďme shodné značení věcí s bloky spojených napříč prostředími. Vše je sázeno strojovým písmem. Začněme kategoriemi:

pohyb	zvuky	události	vnímání	proměnné	ostatní
vzhled	pero	ovládání	operátory	seznamy	data

Tabulka 1.4.1 Výčet kategorií všech prostředí a jejich značení v práci

Bloky prostředí spadají nejen do určité kategorie, ale dělí se ještě dle svého typu. Zde se autor snažil přiblížit grafickému návrhu z prostředí. Mohl sice použít obrázky, ale to by jednak snižovalo kvalitu elektronické verze při přiblížení, dále vytvářelo vetší mezery mezi řádky, a hlavně by nemohl systém TEX zalomit řádek dle slov v názvech bloků.[1])

Jedná-li se o příkaz, pak je jeho název ohraničen hranatými závorkami stejné barvy. Při popisu funkce jsou použity závorky kulaté a u podmínek závorky špičaté. Výjimečně jsou názvy bloků sázeny zkráceně. Obecná ukázka: `[příkaz]`, `(funkce)`, `<podmínka>`.

Blok v prostředích	Sazba bloku v práci	Argumenty	Vrací
jdi na pozici x: 0 y: 0	[jdi na pozici]	{0, 0}	--
po kliknutí na	[po kliknutí na start]	--	--
když	[když]	nepíší se	--
tempo	(tempo)	--	{60}
3 + 2	(sečti)	{3, 2}	{5}
proměnná, parametr	(proměnná), (parametr)	--	{hodnota}
směr z Pes	(z)	{směr, Pes}	{90}
A = a	<je rovno>	{A, a}	{<pravda>}
dotýká se okraje ?	<dotýká se>	{okraje}	{<nepravda>}

Tabulka 1.4.2 Ukázky sazby bloků a všeho s nimi spojeného

Všimněme si, že sázené názvy bloků neobsahují argumenty. To proto, že je pro autora práce podstatné určit pouze o jaký blok se jedná. Proměnné a formální parametry uživatelem vytvořených bloků jsou pak sázeny jako `(název proměnné)` a `(název parametru)`. Formální parametry ze Scratch 2 pak jako `(název parametru)`. Pokud však čtenář nerozumí zmíněným pojmům, dozví se jejich význam později.

[1]) Tudíž by sazba této práce byla pro autora daleko náročnější.

- **Poznámky**

Občas je dobré čtenáře upozornit na dodatečné informace. K tomu jsou v práci použity poznámky. Jedna taková je přítomna v následující části o tabulkách. Jejich priorita je vyšší než u tradičních poznámek pod čarou, které si čtenář může dovolit ignorovat. Jinak řečeno zatímco poznámky pod čarou se vztahují k větě či slovu, tak poznámky pro tuto práci stvořené se vztahují k okolnímu obsahu probírané problematiky.

- **Tabulky**

Každá tabulka je vysázena strojovým písmem. Autor si osvojil z důvodu úspory místa několik výrazů, které v tabulkách preferuje. Zde je jejich vysvětlení:

- `ANO` – znamená určitě ano; bylo potvrzeno; s jistotou; autor práce si ověřil...
- `NE` – ne v současnosti; není zmínky; neplánuje se; mohlo být, ale není
- `--` – významově silnější `NE`; v žádném případě; vylučuje se s předchozím
- `ještě ne` – něco je plánováno, ale není stanoven termín; ano, ale nevíme kdy
- `až v 4.1` – týká se pouze prostředí Snap; potvrzeno, ale objeví se až ve verzi 4.1

• *Poznámka: Čtenář by měl respektovat autorův cit a uvědomit si, že bylo občas složité rozhodnout se mezi* `NE` *a* `--`. *Budou-li informace v tabulkách dělat čtenáři problémy, nechť si zopakuje tuto část. Drobným rozdílům snad už napodruhé lépe porozumí.*

- **Dodatky**

Práce taktéž obsahuje dodatky, na které je čtenář postupně odkazován. Namísto číselného značení jsou označeny písmenem. Týkají se především prostředí Snap a vysvětlují mnohdy podmínečné, avšak pro zkušené čtenáře nadbytečné, informace k pochopení popisované problematiky. Pokud tedy čtenář nezná například pojem *reference*, je v jeho zájmu se konkrétnímu dodatku nevyhýbat.

- **Webová služba GitHub**

Autor práce mnohdy zmiňuje a čerpá z webové služby GitHub, na které si týmy programátorů vytvářejí vlastní repositáře a spravují na nich své kódy společně. Případné nálezy chyb v jejich kódech se popisují v hlášeních (tzv. *issues*). Návrhy a opravy jsou nazývány jako *pull request*. Reakce na uživatele se provádí v kombinaci `@ + přezdívka`. A je-li hlášení/návrh vyřešen či nikoliv určuje popisek `CLOSED` anebo `OPEN`. Tímto je čtenář připraven GitHub pročítat, pakliže si chce ověřit aktuální stav prostředí Snap.

V práci se odkaz na uživatele sází v podobě `@přezdívka` a odkaz na konkrétní hlášení či návrh pak jako `GitHub#Číslo`. Čtenáři tištěné verze musejí url adresu opisovat. Zároveň musí ověřit, jestli v url adrese použít slovo `issues` či `pull`. Šablony:

`https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/issues/ČÍSLO` nebo
`https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/pull/ČÍSLO`

- **Projekty v prostředích Snap a Scratch 2**

Upozornění čtenáře na nadcházející projekt je znázorněno níže. Pořadové číslo projektu je klikacím odkazem a obě prostředí sdílí stejné počítadlo. Pro čtenáře tištěné verze jsou internetové odkazy na on-line projekty umístěny v dodatku G. Ukázka:

NÁZEV PROSTŘEDÍ | Název či popis projektu |	pořadové číslo → #N

- **Upravená čeština prostředí Snap a rozdílné názvy bloků**

Autor si poupravil oficiální češtinu z on-line verze Snap, s níž pak fotil všechny obrázky scénářů. Bude-li čtenář prohlížet publikované projekty z cloud účtu autora v on-line verzi Snap či stáhne-li si off-line verzi tohoto prostředí, uvidí oficiální (odlišný) český překlad. Aby užíval stejný překlad, musí češtinu nainstalovat (viz návod v dodatku F).

Překážkou při odkazování se na názvy bloků v bakalářské práci je rozdílnost oficiálních překladů mezi prostředími. Např. blok v Scratch 1 pod názvem `[oblékni kostým]` se ve Scratch 2 přejmenoval na `[změň kostým]`. Toto ale bakalářská práce neřeší.

Kapitola 2
Představení prostředí

2.1 Prostředí Scratch 1

Scratch 1 je off-line, volně a v češtině dostupné[1]), výukové, vizuální a skriptovací programovací prostředí určené převážně pro děti ve věku od 8 do 16 let.[18, s. 7] Prostředí je naprogramováno v jazyce Squeak, což jej umožňuje spustit na operačních systémech Windows, Linux (Ubuntu a Debian), a na Mac OS.[7, s. 2]

Vyvinuté skupinou Lifelong Kindergarten na Massachusettském technologickém institutu (MIT) si postupně prošlo několikaletým vývojem a to od první oficiální verze 1.0 vydané roku 2007, až po poslední verzi 1.4 z roku 2009.[40] Poté se začalo pracovat na nové generaci tohoto prostředí označované jako Scratch 2,[15, s. 6] ale nepředbíhejme.

Prostředí je založené na idejích jazyka Logo s tím rozdílem, že nahrazuje psaní kódu používáním grafických komponent (bloků) ve spojení s metodou táhni-pusť.[8, s. 1] Jejich skládáním pak tvoříme vlastní *skript*, což do češtiny překládáme jako *scénář*.

Bloky jsou reprezentovány různými tvary, které napovídají začínajícím programátorům kam je lze napojit, jelikož bloky téhož tvaru k sobě přesně pasují. Navíc protože neprogramujeme psaním zdrojového kódu, nýbrž skládáním bloků, pak taktéž odpadá problém se znalostí specifické syntaxe a vznikem různých chyb. Programátoři jistě znají nepříjemnosti spojené se zapomenutou závorkou či středníkem.[11, s. 4]

Svým návrhem se snaží klást co nejmenší nároky na uživatele a nezatěžovat jej procesy odehrávající se na nižší úrovni prostředí. Proto třeba skrývá různá složitě formulovaná chybová hlášení. Totéž platí i pro pokročilou teorii o programování. Jakožto kompilovaný jazyk například nevyžaduje provádět kompilaci ručně přes nějaké tlačítko, ale odstiňuje ji samostatným provedením na pozadí.[18, s. 8]

V prostředí vytváříme různé postavy, které mezi sebou nějak komunikují a sehrávají příběh na scéně. Postavy mohou kromě pohybu po scéně také oblékat různé kostýmy, vydávat zvuky nebo hrát na hudební nástroje, něco povědět či si něco pomyslet, kreslit želví grafiku, aplikovat na sebe grafický efekt, anebo vnímat a rozesílat zprávy dalším postavám. Možná je i komunikace s uživatelem přes počítačovou myš a klávesnici.

Scratch 1 přesahuje průměrná očekávání a nabízí ještě bitmapový editor pro tvorbu a úpravu kostýmů, záznam zvukové stopy mikrofonem, různorodý obsah z dostupných knihoven, propojení s fyzickým světem (stavebnicí LEGO WeDo Construction Kit); a možnost sdílení své tvorby s on-line komunitou dalších programátorů na k tomu určené Scratch síti – speciální webové stránce – přímo z prostředí jediným tlačítkem.

Název je odvozen od „scratchujících“ hip-hop diskžokejů, kteří spojují různé části písní do kreativních celků – stejně jako je tomu i ve Scratch používáním různých programových bloků ke stvoření zajímavých scénářů. Motto tohoto prostředí mimochodem zní „Představ si, naprogramuj, sdílej“, což se podobá práci oněch diskžokejů.[36]

Celkově podporuje tvořivost, sebe-prezentaci a motivuje k programování vytvářením vlastních interaktivních příběhů, her a simulací.[2, s. 1] Jeho velkým přínosem je, že pomáhá se zaměřit na obsah (co programuji), než na zápis (jak to naprogramuji).

2.1.1 Scratch sítě a on-line komunity

Původní návrh vývojářů ohledně prostředí Scratch obsahoval mezi jeho klíčovými rysy (programování skrze skládání bloků, programovatelná manipulace s multimédii, bezešvá integrace s fyzickým světem, a podpora vícejazyčnosti) i pestré možnosti sdílení obsahu a spolupráce s ostatními začínajícími programátory.[16, s. 5]

[1]) Lze jej stáhnout na adrese: http://scratch.mit.edu/scratch_1.4/

Podle Scratch týmu by totiž lidé neměli být pouhými konzumenty interaktivních médií, ale měli by se stát i jejich tvůrci a okusit většinu s jejich tvorbou spojené. To vyžaduje jisté znalosti – matematické, programovací, schopnost návrhu, řešení problému; a podobně.[1, s. 2] Po tom všem se přirozeně každý rád podělí o svoje výtvory.

Na sdílení a prohlížení projektů různých programátorů je oficiální webová stránka prostředí Scratch, kterou lze chápat i jako sociální síť registrovaných uživatelů. Vyvíjela se společně s prostředím Scratch a byla spuštěna v roce 2007.[14, s. 1]

Sociální přínos této Scratch sítě tkví v možnosti odvození projektu nějakého autora (jež poté zjednodušíme, opravíme, rozšíříme, a případně nahrajeme pod svým jménem), týmové spolupráci (když dva a více přátel spolupracují na jednom projektu), přiučování se od řešení druhých (prohlížením si sdílených projektů), či komunikace s ostatními přes názorová fóra nebo přímo pod každým projektem.

- **ScratchEd – síť zaměřená na potřeby vzdělavatelů**

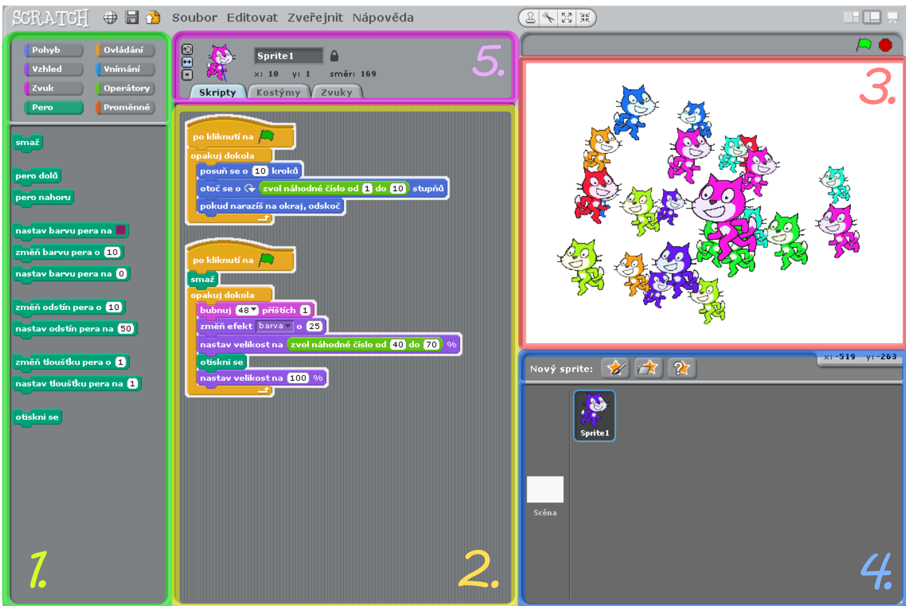
Scratch síť je určena spíše programátorským novicům než vzdělavatelům programování využívající prostředí Scratch. Proto není vhodným médiem ke komunikaci mezi těmito vzdělavateli, jenž by rádi sdíleli své náměty, zdroje, výukové materiály, či diskutovali o svých zkušenostech, anebo poznávali další učitele programování.[13, s. 4]

Naštěstí se Karen Brennan, členka skupiny Lifelong Kindergarten, spojila s agenturou OHO Interactive, a vytvořili novou webovou stránku zaměřenou na potřeby učitelů. Spustili ji v červenci 2009 a pojmenovali jako ScratchEd[1]), což zřejmě znamená „Scratch Educators“ – v překladu „Scratch vzdělavatelé“. Na této stránce (další Scratch síti) tak vznikla nová komunita uživatelů – učitelů –, kteří se vzájemně podporují.[19]

2.1.2 Náhled do prostředí

Pojďme si postupně představit různé prvky prostředí, ale nejprve začněme vzhledem. Obrázek 2.1.1 ukazuje celé prostředí za spuštěné hry. Rozhraní je rozvrženo do několika hlavních oblastí, které usnadňují pohyb a orientaci v něm. Následuje jejich výčet:

1. paleta bloků a jejich kategorie
2. oblast scénářů postavy
3. scéna projektu
4. přehled postav projektu včetně scény
5. další nastavení postavy a přepínání mezi jejími scénáři, kostýmy a zvuky



Obrázek 2.1.1 Ukázka prostředí Scratch 1 s vyznačenými hlavními oblastmi

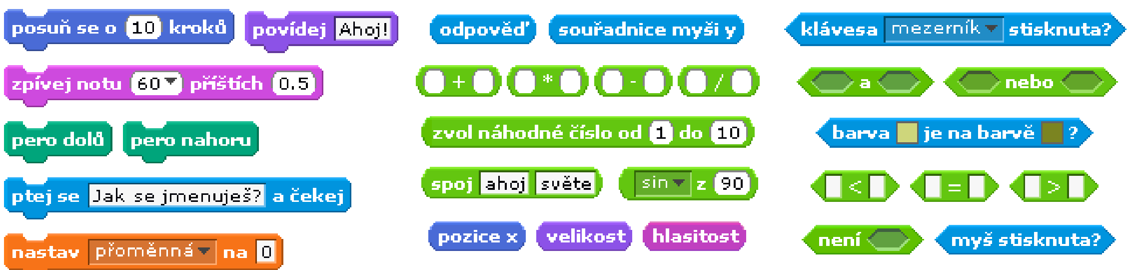
[1]) http://scratched.gse.harvard.edu/

- **Typy bloků ke tvoření scénáře**

Označování začátku scénáře provádíme *bloky událostí*, které mají svůj specifický tvar (viz obr. 2.1.2). Tyto bloky spustí scénář při obdržení žádosti od prostředí – například po kliknutí na zelený praporek symbolizující odstartování projektu.

Obrázek 2.1.2 Bloky událostí označující začátek scénáře

Následující obrázek 2.1.3 zobrazuje některé bloky tří typů – *příkazů*, *funkcí* a *podmínek*. Nalevo jsou – tvarem připomínající dílek – příkazy, které lze připnout pod blok stejného tvaru a složit tak nějaký algoritmus, tj. scénář. Všimněme si, že některé bloky mají taktéž své vstupy, jenž nazýváme *parametry*. Svým tvarem napovídají, jaký typ hodnoty očekávají – například otvor s oválným tvarem očekává číselnou hodnotu.

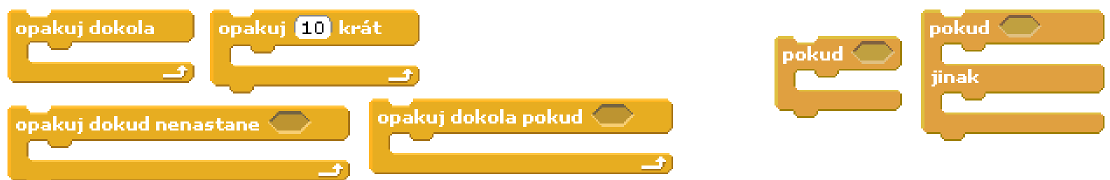


Obrázek 2.1.3 Další tři typy bloků – příkazy, funkce a podmínky

Uprostřed téhož obrázku jsou zobrazeny funkce, které se z hlediska tvaru od příkazů liší zaobalenými rohy a lze je vkládat pouze do vstupů ostatních bloků. Pakliže má nějaká funkce svůj vlastní parametr, může přijmout další – do něj vloženou – funkci. Můžeme tak postupně seskládat několik funkcí do sebe a vytvořit složitější výrazy.

Nakonec na pravé straně obrázku jsou zobrazeny podmínky. Prostředí totiž rozlišuje funkce vracející čísla anebo text od funkcí vracející pravdivostní hodnotu. A právě funkce vracející hodnoty *pravda* či *nepravda* jsou v prostředí reprezentovány jako bloky se špičatými konci. Používají se v řídicích strukturách.

K řízení toku scénáře máme bloky tvarem připomínající písmeno C[1]), jenž přijímají další příkazy. V prostředí jsou dostupné čtyři cykly k opakování části scénáře a dva podmíněné bloky k jeho větvení, které ovlivňuje výsledek podmínky (viz obrázek 2.1.4).



Obrázek 2.1.4 Bloky umožňující řízení scénáře – cyklení a větvení

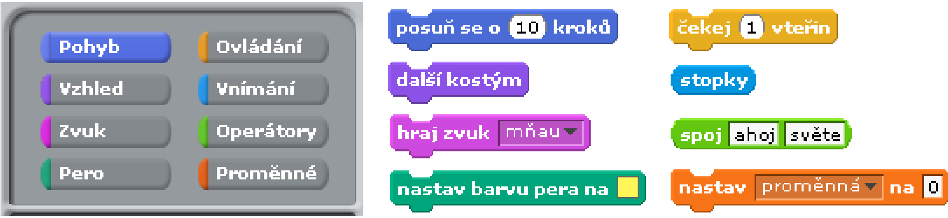
Posledním typem jsou *konečné bloky* ukončující vyhodnocování scénáře, který – pokud se neukončí přirozeným způsobem – lze těmito zastavit. Za ukončující příkaz již nepůjde vložit další příkazy, neboť mu ve spodní části chybí „zobáček“ k jejich napojení. Ostatně cykly `[opakuj dokola]` a `[opakuj dokola pokud]` z obr. 2.1.4 jsou též konečnými.

Obrázek 2.1.5 Bloky ukončující vykonávání jednoho či všech scénářů

[1]) Pokud čtenář vidí na obr. 2.1.4 blok ve tvaru písmen E, mýlí se. Blok má dva parametry ve tvaru C.

- **Kategorie bloků**

Mohli jsme si všimnout z předchozích obrázků, že prostředí třídí bloky do různobarevných kategorií. Ty nejenže zpřehledňují scénář svou vzhledovou odlišností, ale také napovídají programátorům, jakou schopnost má ten či onen blok podle názvu kategorie, do které spadá. Prostředí jich rozlišuje osm.



Obrázek 2.1.6 Panel k přepínání mezi kategoriemi (vlevo) a ukázky bloků

- **Datové typy**

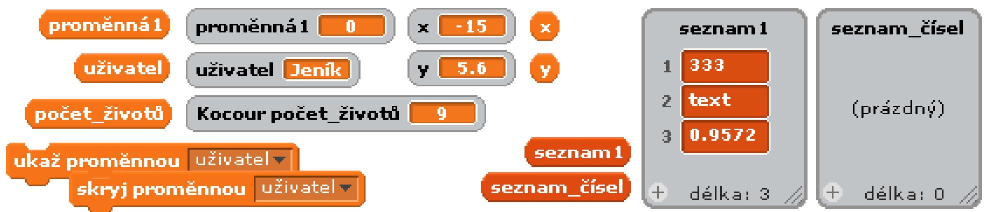
Co se datových typů týče, tak prostředí Scratch 1 disponuje pouze třemi – a to číslem, textem (řetězcem znaků) a pravdivostní hodnotou.[9, s. 9]

- **Proměnné, datové struktury a zobrazení jejich stavu sledovači**

Nedílnou součástí algoritmů jsou též proměnné a datové struktury. Scratch 1 umožňuje vytvořit proměnnou přístupnou buď jedné postavě či všem postavám v projektu a to bez určení datového typu. V proměnných se tak chvíli může vyskytovat číselná hodnota a později například textová. Z datových struktur je možné použít pouze *seznamy*.[1])

Stav proměnných a seznamů (přesněji hodnot a obsahu) je vhodné a někdy i žádané sledovat a proto nám prostředí nabízí tzv. *sledovače* umístěné na scéně, které lze zobrazit či skrýt jak manuálně přes zaškrtávací políčko tak i dvěma příkazy programově. V případě seznamů jsou sledovače zobrazeny vždy a není způsobu, jak je skrýt.

Obrázek 2.1.7 ukazuje několik proměnných a seznamů společně s jejich sledovači ze scény. Při tvoření scénáře se na hodnoty proměnných a obsahu seznamů odkazujeme funkcemi – na obrázku reprezentovány např. `(proměnná1)` a `(seznam1)`, které nám prostředí přidá do příslušné kategorie bloků ihned po jejich vytvoření tlačítky v rozhraní.



Obrázek 2.1.7 Proměnné a seznamy se sledovači ze scény

Na obrázku jsou ještě proměnné `(uživatel)`, `(x)`, `(y)`; a `(počet_životů)` u jejíhož sledovače je ještě uveden popisek „Kocour“. Jde totiž o proměnnou pouze jedné postavy s tentýž názvem. Ostatní již uvedené proměnné jsou k dispozici všem postavám. Ještě je dobré zmínit, že prázdný text (jako hodnota) se v prostředí nijak neoznačuje a sledovač jej nemá jak zobrazit – proto v něm uvidíme pouze prázdné místo.

U seznamů jsou sledovače odlišného vzhledu, neboť mají za cíl zobrazit všechny jeho prvky – tedy obsah. Všimněme si i prázdného seznamu úplně vpravo, který neobsahuje žádný prvek. Závěrem dodejme, že i seznamy lze vytvořit pouze jedné postavě.

[1]) Vzhledem k cílové věkové skupině je logické, že se odborná terminologie nevyskytuje a teorie datových struktur nevyžaduje. Proto neřešme, jestli prostředí využívá *spojový seznam* či *dynamické pole*.

2.1.3 Programovací koncepty

Na závěr zmiňme ještě podporu programovacích konceptů v tomto prostředí.

`Podporované programovací koncepty`[17]

- sestavování sekvence příkazů
- opakování scénáře (iterace)
- podmíněné vyhodnocování
- proměnné
- seznamy (dat. struktura)
- spouštění a obsluhování událostí
- paralelní vykonávání (vlákna)
- odesílání zpráv postavám a jejich synchronizace
- vstup z klávesnice
- posluchače myši
- náhodná čísla
- logické operace
- využívání efektů prostředí
- primitivní ladící nástroj (debugger)

Navzdory relativně široké škále podporovaných konceptů Scratch 1 ke své škodě některé další – nejenom pro výuku základů programování klíčové – koncepty nenabízí.

`Některé nepodporované programovací koncepty`[17]

- tvorba bloků a vracení hodnot
- rekurze
- tvorba tříd a objektů
- dědičnost
- zachytávání výjimečných případů
- práce se soubory (vstup/výstup)

2.2 Modifikace Scratch 1 – prostředí BYOB

Prostředí Scratch 1 je možné vylepšit tvorbou vlastních modifikací, kterých je do současnosti několik desítek k dispozici.[31] Modifikací máme na mysli použití a upravení zdrojových souborů tohoto prostředí. Zpravidla doplňují originální verzi o nové bloky a funkcionality. Nakonec jsou publikovány pod vlastním názvem, který nesmí obsahovat slovo Scratch – s výjimkou dovětku „založeno na prostředí Scratch“.[38]

Za modifikací BYOB stojí vývojář Jens Mönig z MioSoft Corporation a Brian Harvey, lektor z Kalifornské univerzity v Berkeley, který přispěl svými návrhy a dokumentací. Na vývoji se podíleli i další studenti ze stejné univerzity či ostatní uživatelé.[42]

Původně Mönig vyvíjel svou vlastní modifikaci s názvem Chirp, kterou ovšem při setkání s Harveym přestal dále vyvíjet a společně se pustili do vývoje nové a propracovanější modifikace BYOB[1]).[24] Oba totiž neuspokojovaly tehdejší možnosti prostředí Scratch 1, mezi nimiž převažovala absence tvorby vlastních procedur a rekurze.[45]

Prostředí se postupně vyvíjelo a číslovalo od verze 1.0, která byla vázána na verzi 1.3 prostředí Scratch 1 – tj. už někdy v roce 2008. Poslední verze BYOB je 3.1.1 z roku 2011, která je k dispozici volně ke stažení na internetu.[2])[24] BYOB se tedy vyvíjel ještě dva roky poté, co Scratch 1 ukončil svou první generaci verzí 1.4.

• *Poznámka: Jelikož je prostředí BYOB modifikací původního prostředí Scratch 1, zachovává si též stejný vzhled, všechny části a jejich rozložení. Přestože se v něm vyskytují nepatrné změny, stěží bychom je na případném obrázku zaznamenali. Z tohoto důvodu si snímek z prostředí BYOB ukazovat nebudeme.*

Akronym BYOB v podstatě znamená „*Build your own block*“, což lze přeložit do češtiny jako „*Sestav si svůj vlastní blok*“. Tímto názvem se vývojáři snažili vyjádřit hlavní odlišnost (a výjimečnost) své modifikace od původního prostředí Scratch 1.

Slabina prostředí Scratch 1 tkví v dostupných programovacích možnostech neuspokojujících poptávku pokročilejších programátorů, kteří jsou tak nuceni toto prostředí opustit a migrovat do jiných, převážně textově založených, jazyků. Ty mnohdy vyžadují znalost sofistikovanějších integrovaných vývojových prostředí, která jsou převážně

[1]) Ačkoliv je BYOB modifikací, v bakalářské práci jej budeme označovat převážně jako prostředí.
[2]) `http://snap.berkeley.edu/old-byob.html`

v angličtině – stejně jako klíčová slova a knihovny jazyka samotného. Proto důvodem ke vzniku BYOB nebylo pouhé zavedení možnosti tvorby vlastních bloků a rekurze.

Abychom kompletně shrnuli motivaci k jeho tvorbě, ocitujme ve volném překladu vyjádření jednoho z vývojářů:

"*Scratch tým se domníval, že by měla existovat oddělená verze Scratch pro univerzitní studenty informatiky. S tím nesouhlasíme. Následováním principu prvotřídnosti dat můžeme vytvořit mocnější Scratch s několika málo viditelnými změnami na jeho rozhraní.*"[22]

Brian Harvey

Jinými slovy to, čím již Scratch 1 částečně disponovalo, bylo možné rozšířit i bez většího zásahu do vzhledu a ovládání prostředí, aby se BYOB mohlo otevřít novým možnostem a stát silnějším výukovým nástrojem. Plán vývojářů umožnit výuku programování na pokročilejší úrovni ve stejném prostředí proto mohl být naplněn.

Některá rozšíření jsou v důsledku příliš složitá a nelze předpokládat, že by je například žáci druhého stupně základní školy zvládali. S tím ani samotní vývojáři nepočítají. BYOB je variantou pro pokročilejší programátory, kteří „trpěli“ strnulostí a neobratností prostředí Scratch 1 při programování komplexnějších projektů.[3, s. 10]

- **Sdílení projektů**

Modifikace mají zakázané své projekty sdílet na Scratch síť.[38] To je logické, neboť nelze předpokládat, jak ony modifikace fungují (co rozšiřují) a též by při stažení takového projektu k nahlédnutí museli zájemci nainstalovat i modifikaci samotnou.

Uživatelé různých modifikací si musejí projekty vyměňovat jiným způsobem. Proto vznikla například, dnes již neaktivní, webová stránka Mod Share[1]) shromažďující projekty z vícero modifikací, na níž bylo možné sdílet i projekty z prostředí BYOB.

- **Programová rozšíření a další koncepty**

BYOB v celku úspěšně eliminuje hlavní nedostatky Scratch 1. Kromě toho umožňuje vytvářet i vnořené seznamy, kdy seznam je prvkem seznamu jiného, pomocí nichž lze sestavit složitější datovou strukturu.[3, s. 1] Zásadou tohoto prostředí je mimo jiné vytvořit všechny, tj. i nově zavedené, datové typy jako prvotřídní.[2])[4, s. 7] Namísto sáhodlouhých vět shrňme rozšíření modifikace BYOB v následujícím výčtu:

`Hlavní rozšíření modifikace BYOB navíc k Scratch 1.4`[24]	
+ tvorba vlastních bloků všech typů	+ úplné klonování postav (programově)
+ nové a hlavně prvotřídní datové typy	+ hierarchie postav (předci a potomci)
+ 12 typů parametru u vlastního bloku	+ přikazování a dotazování se libovolné postavy jinou postavou nepřímo
+ vnořené seznamy	+ vnořování (skládání) postav
+ lambda výrazy (prvotřídní procedury)	+ pozastavování projektů za jejich běhu
+ tvorba objektů nadefinovanými třídami anebo prototypy	+ vylepšení ladícího nástroje
+ přístup ke všem atributům postavy	+ přístup ke znaková ASCII tabulce

[1]) `http://wiki.scratch.mit.edu/wiki/Mod_Share`

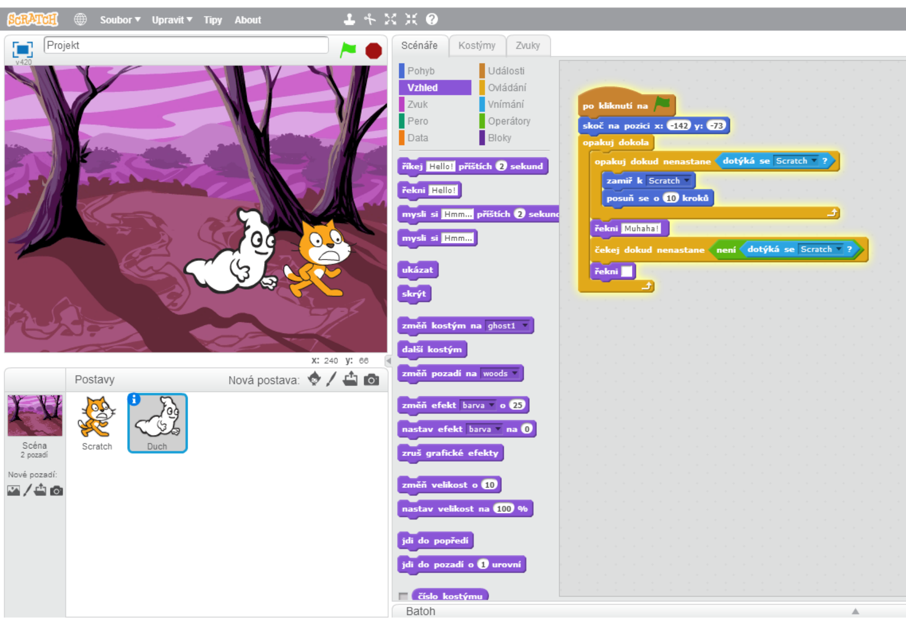
[2]) Někdy též jako *plnohodnotné*, v angličtině pak jako *first-class*. Více až v podsekci 3.1.3.

2.3 Současná verze Scratch – Scratch 2.0

V prvním měsíci roku 2010 Scratch tým ohlásil práci na nové generaci prostředí a obeznámil veřejnost s plánovanými změnami a rozšířeními. Po třech letech vývoje a téměř půlročním veřejném testování uživatelskou komunitou se v květnu téhož roku (2013) nové prostředí Scratch opuštěním statusu beta oficiálně vydalo ve verzi 2.0.[37]

Prostředí je nyní v on-line verzi – tj. programujeme přímo ve webovém prohlížeči po navštívení oficiální stránky Scratch a kliknutí na tlačítko „Tvořit“ –, má změněno uživatelské rozhraní, jež zahrnuje změnu odlišného uspořádání hlavních oblastí; a přešlo na světlé barevné téma (vzhled).

• *Poznámka: Prostředí Scratch 2 toho nabízí oproti Scratch 1 mnohem víc. Některá rozšíření si uvedeme níže v oblasti o současných a budoucích rozšířeních, ale vše podrobně rozebereme až v třetí kapitole bakalářské práce srovnávající prostředí mezi sebou.*



Obrázek 2.3.1 Snímek z on-line editoru prostředí Scratch 2

Motivací k přeprogramování do nového kabátu bylo – krom přidání dalších funkcionalit a rozšíření – spojit dva oddělené světy, jimiž byly Scratch off-line aplikace a on-line komunita.[26] Hlubší vhled získáme volnou citací slov vedoucího Scratch týmu:

“*Vnímal jsem jako frustrující, když jste si prohlíželi nějaký projekt z on-line komunity a chtěli jste zjistit jak funguje – jelikož jste ho museli stáhnout, poté otevřít v aplikaci, podívat se jestli jej chcete upravit, a nakonec nahrát zpátky na webovou stránku. Chtěli jsme udělat pohyb mezi přehráváním projektu a jeho programováním v editoru mnohem jednodušší, což bylo – myslím – naší počáteční motivací a zároveň největší změnou.*”[26]

Mitchel Resnick

Každá webová stránka nějakého projektu na Scratch síti nyní obsahuje i tlačítko „Pohlédni dovnitř“, kterým se přeneseme přímo do editoru prostředí. Uvidíme – a to bez ohledu na to, jestli se projekt již přehrává či nikoliv – jeho scénu a existující postavy s kostýmy, zvuky, a scénáři. Pohodlným přesunem do editoru tak odpadají výše v citaci uvedené potíže. Toto platí pro každý projekt – sdílený samostatně, sdílený ve studiu (skupině projektů), soukromý (prozatím nesdílený), či čerstvě vytvořený.

Vzhledem k razantním změnám muselo být prostředí přeprogramováno do nového programovacího jazyka, kterým je ActionScript. To však znamená, že abychom mohli v prostředí Scratch 2 programovat či přehrát již vytvořený anebo sdílený projekt, musíme mít nainstalovaný software Adobe Flash Player – a to pro různé prohlížeče zvlášť.

Zvolení tohoto softwaru avšak nebylo nejšťastnější. Operační systém iOS vyvinutý společností Apple Inc., na němž běží produkty z jeho dílny a mezi něž patří například tablet iPad, již nějakou dobu Flash Player nepodporují.[20] Také operační systém Android ve svém oficiálním obchodu Google Play přestal nabízet Flash Player ke stažení. Přesto je možné jej sehnat jiným způsobem a doinstalovat manuálně.[33]

Kvůli postupnému úpadku podpory Flash Playeru se Scratch tým rozhodl ony klíčové části prostředí přeprogramovat do jazyka HTML5, který by měl zajistit schopnost spustit Scratch 2 na již zmíněných systémech a zařízeních. Software Flash Player poté nebude potřeba. Není však známo, kdy práci na novém HTML5 přehrávači dokončí.[27]

- **Vylepšení Scratch sítě a sdílení projektů Scratch 1**

Postupem času se vyskytly výtky některých dětí, že jejich díla jsou „kradena" ostatními a vydávána pod vlastními jmény jen s minimálními úpravami. Problémy s autorstvím (převážně vlastními multimédii a nápady) vývojáři vyřešili dodáním tlačítka „*Odvodit*" do editoru, jenž vytvoří kopii prohlíženého projektu na uživatelův účet. Oficiálně se ona možnost nazývá „remixování". Odvozené projekty si zároveň udržují cestu k předešlým odvozeninám – a to postupně až k úplnému originálu. Tento „strom odvozenin" lze na stránce každého projektu přes k tomu určené tlačítko zobrazit v grafické formě, což umožňuje „vystopovat" ty nejúspěšnější odvozeniny. Úspěšnější odvozeniny se vyznačují vetším množstvím svých vlastních odvozenin.

Když se oficiálně vydávala druhá generace Scratch, zásadní změnou prošla i Scratch síť, neboť se prostředí přesunulo do on-line verze. Sdílené projekty vytvořené v Scratch 1 byly zachovány. I v současnosti můžeme sdílet projekt z off-line verze Scratch 1 a poté jej prohlížet přes Scratch síť v on-line editoru Scratch 2. Takový projekt však nebude využívat nové možnosti druhé generace prostředí, neboť jsou Scratch 1 neznámé.

- **Současná a budoucí rozšíření**

Hlavní rozšíření ve Scratch 2.0 navíc k Scratch 1.4[37]	
+ tvorba vlastních bloků – jen příkazy + dočasné klony postav (programově) + cloud proměnné + takzvaný „batoh" k přenosu obsahu mezi postavami a projekty + zvukový editor	+ podpora vektorové grafiky a přidání vektorového editoru + snímání pohybu z videokamery + přístup k aktuálnímu datu a času + přidána nová kategorie `události`

Plánovaná rozšíření ve Scratch 2.0[37]	
+ tvorba vlastní funkce a podmínky + uchování textu v cloud proměnné + podpora cloud seznamů	+ HTML5 přehrávač pro zařízení nepodporující Flash Player

- **Off-line editor**

Pokud někdo nemá stálý přístup k internetu anebo nechce programovat v prostředí běžící v prohlížeči, může si stáhnout off-line editor[1]) a nainstalovat jej na počítač. Předně ale musí stáhnout a nainstalovat Adobe AIR – víceplatformní běhové prostředí pro vývoj desktopových a mobilních aplikací –, což může být nepříjemností při zprovozňování off-line editoru – převážně na školních či univerzitních počítačích.

Off-line editor je zatím stále ve fázi vývoje beta a liší se nepatrně od on-line verze ve smyslu chybějících funkcionalit. Případné obavy o zastaralou verzi na stolním počítači odstraní možnost automatické aktualizace, na kterou uživatele v případě potřeby prostředí samo upozorní. Zde je logicky připojení k internetu vyžadováno.

Přístup k internetu využijeme i při sdílení projektů na Scratch síť, které nám editor stále umožňuje – stejně jako tomu bylo u jeho předchůdce Scratch 1. Formulář je značně zjednodušen a vyžaduje kromě přihlašovacích údajů pouze název projektu.

[1]) http://scratch.mit.edu/scratch2download/

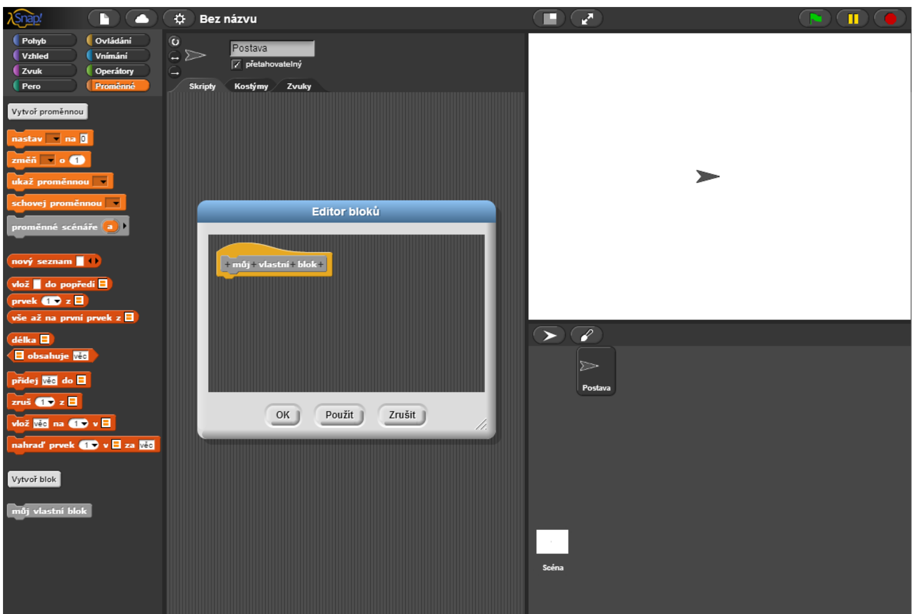
2.4 Současná verze BYOB – prostředí Snap 4.0

Z předchozích popisů víme, že BYOB se vyvíjelo ještě rok po oznámení práce na druhé generaci Scratch. Jelikož se Jens Mönig a Brian Harvey podíleli na vývoji Scratch 1 a znají se tak se členy Scratch týmu, lze předpokládat, že měli přístup k podrobnostem vývoje Scratch 2. Vývojáři BYOB byli postaveni před těžké rozhodnutí, neboť věděli, že originální prostředí Scratch 1 – umožňujíc vznik modifikace BYOB –, již v budoucnu nebude dále podporováno (ve smyslu dalších rozšíření).

Rozhodli se proto přeprogramovat prostředí do jiného jazyka, stejně jako se rozhodl Scratch tým provést s Scratch 1. Nová nadstavba BYOB je též v on-line verzi, zdarma, a dokonce i v češtině. Mohli sice modifikaci BYOB nadále rozšiřovat, ale dalších důvodů k jejímu přeprogramování bylo podle vývojářů hned několik:[23]

1) BYOB bylo údajně pomalé a plné chyb, takže vyžadovalo náročnou údržbu.
2) Někteří učitelé programování si stěžovali, že nemohou instalovat software na školní počítače a tak vznikla myšlenka na prostředí spustitelné ve webovém prohlížeči. Paradoxně si však nepomohli, neboť nyní učitelé argumentují, že mohou navštívit jen školou povolené internetové stránky. Ačkoliv tento problém vývojáři nevyřešili, byl to podle nich jeden z klíčových důvodů ke změně.
3) Protože Scratch tým zavrhl původní verzi a začal vyvíjet novou generaci ve webovém rozhraní společně s několika dalšími změnami, chtěli s nimi udržet krok. Jelikož ale Scratch 2 nepovoluje modifikace a v té době ještě ani nebyly sdíleny zdrojové kódy tohoto prostředí, museli si vývojáři nové prostředí naprogramovat sami.

V květnu 2012, tedy přibližně rok po dokončení poslední verze BYOB, byla spuštěna alfa verze nového prostředí. Již tehdy – časově téměř shodně s Scratch 2 – fungovalo ve webovém prohlížeči. V srpnu téhož roku se mělo prostředí přesunout do vývojové fáze beta. Tato fáze beta byla ukončena v květnu 2015 a prostředí bylo oficiálně vydáno.



Obrázek 2.4.1 Snímek z on-line prostředí Snap

Prostředí se přejmenovalo z BYOB na *Snap!*[1]), protože si někteří učitelé stěžovali na význam předchozího názvu.[2]) Kvůli razantním změnám v současné verzi provedených, se navýšilo číslování majoritní verze prostředí – kompletní název je tak Snap 4.0. Od této chvíle mluvme o prostředí jako o alternativě či rozšíření ke Scratch 2, nikoliv jako o jeho modifikaci – žádné zdrojové kódy ze Scratch 2 totiž použity nebyly.

[1]) Význam slova přibližně znamená „plesk“, „cvak“, anebo „prásknutí o sebe“, což zřejmě reprezentuje spojení dvou bloků secvaknutím (či praštěním). Za ozřejmění velmi děkuji vedoucímu práce.

[2]) Co však natolik pobuřujícího akronym BYOB znamená se autorovi práce zjistit nepodařilo.

Původně vývojáři plánovali ukončit beta status Snap v druhé polovině roku 2013, ale vzhledem k malému počtu vývojářů, neustálému nálezu chyb a téměř nekončícímu procesu odlaďování, mnoha dalších návrhů, a přiznejme si – i nedostatku času; se oddalovalo zdárné dokončení vývoje. Proto se už tehdy vývojáři rozhodli některé změny implementovat až v další verzi 4.1, na které se v současné době pracuje.

Zdrojové kódy BYOB byly kompletně převedeny do jazyka JavaScript. Pro scénu a vše s ní spojené se využívá canvas (plátno) z HTML5. Prostředí Snap disponuje i vlastním serverem nabízející tvorbu uživatelského účtu, na něž můžeme uložit a popřípadě i sdílet naše projekty v prostředí vytvořené. Oproti Scratch 2 má Snap výhodu ve zvolených programovacích jazycích, které nebrání jeho spuštění na libovolném systému a prohlížeči – není tedy problém jej spustit na mobilních zařízeních či tabletech.

Přestože Snap hluboce vychází z obou generací Scratch, tak jeho vývojáři nejsou Scratch týmem vnímáni jako rivalové. Naopak je vítají a podporují.[5, s. 3] Mitchel Resnick dokonce navrhl označit Snap jako „inspirován prostředím Scratch“, což je však vzhledem k množství přebrané práce a nápadů stále velmi skromný dovětek.[34]

- **Současná a budoucí rozšíření**

Jelikož většinu zdrojových kódů museli autoři přeprogramovat do jiného programovacího jazyka, nejsou zatím všechny funkcionality z prostředí BYOB implementovány. Přesto – zřejmě vlastním návrhem a z toho pramenící volností ve vývoji – se v tomto prostředí objevují jiná rozšíření, kterými BYOB nedisponuje.

`Současná rozšíření Snap 4.0 navíc k BYOB:`[24]	
+ další dat. typy a nastavení parametru	+ přístup k Unicode namísto k ASCII
+ dočasné klony – jako ve Scratch 2	+ přepis bloků do textové podoby
+ funkce určené k rekurzi se seznamy	+ možnost vykonat JavaScript kód
+ přístup k webovému obsahu	+ tzv. *call-with-current-continuation*

`Hlavní v budoucnu implementovaná rozšíření do verze 4.1:`	
+ makra z funkcionálního paradigmatu	+ úplné klonování postav z BYOB
+ přístup k atributům postavy z BYOB	+ hierarchie postav z BYOB

- **Sdílení projektů a Snap síť**

Oficiální stránka prostředí Snap neumožňuje procházet sdílené projekty. Na ně je možné se v současné době dostat přímo pouze přes URL adresu zaslanou samotným autorem. V současnosti se vyvíjí podobná webová stránka jako má Scratch,[1]) která sdílené projekty zobrazuje. Nabídne zřejmě shodné či podobné možnosti jako Scratch stránka.

- **Off-line editace**

I ve Snap je možné programovat bez připojení k internetu. Stačí stáhnout archivované zdrojové soubory přes příslušnou nabídku v prostředí. Nejde ale o samostatnou aplikaci, neboť žádné instalace netřeba. Prostředí se spustí též v prohlížeči[2]), je však odříznuto od oficiálního Snap serveru a nelze se připojit na cloud účet k alespoň sdílení projektu.[3])

Uživatelé hostující prostředí na svém počítači musejí počítat ještě s dalšími překážkami. Za prvé, prostředí neupozorňuje na novou aktualizaci a tak musejí vývoj prostředí sledovat samostatně. Aktualizaci provedeme jedině manuálně a to opětovným stáhnutím zdrojových souborů. Za druhé, nemáme přístup do **některých** knihoven (kostýmů, zvuků, aj.). A konečně – a to je problémem spíše technologickým než vývojovým – nelze u bloků pracujících s barvou použít kapátko k jejímu nastavení. Při nastavování barvy pera můžeme použít alternativní blok nastavující barvu podle číselné hodnoty, ale třeba u podmínky ověřující dotek postavy s nějakou barvou máme bohužel smůlu – což může značně zkomplikovat či úplně znemožnit vývoj některých projektů.

[1]) Návrh, jehož autorem je Kyle Hotchkiss, je na `https://github.com/ucb-snap/ucb-snap.github.io`
[2]) A to přes soubor „index.html“ z archivu zdrojových souborů.
[3]) V takovém případě je třeba jej exportovat a poté importovat a sdílet přes on-line verzi prostředí.

Kapitola 3
Porovnání prostředí

Uživatelé prostředí Scratch 1, Scratch 2, či BYOB jsou zvyklí programovat dle naučených dostupných programovacích možností, používat různé zabudované editory, pracovat s interními či externími rozšířeními, knihovnami, a tak podobně. Proto si zřejmě pokládají otázku, v čem se prostředí Snap od ostatních prostředích liší a zdali má smysl jej s jeho současnými možnostmi přijmout jako primární programovací prostředí.

Proto je zapotřebí si všechna prostředí porovnat v těch oblastech, se kterými přichází pravidelně do styku drtivá většina uživatelů. Porovnáním dle stanovených kritérií je možné získat obecnou představu o jejich stavu, podobnosti, a rozdílech.

• Poznámka: *Stanovená kritéria porovnávání netřeba rozepisovat. Čtenář se může vrátit na stranu 9 a přečíst si obsah této, tedy třetí, kapitoly bakalářské práce.*

Je důležité mít na paměti, že prostředí Scratch 2 a Snap se stále vyvíjí a proto nelze zaručit aktuálnost výsledků srovnání s ostatními prostředími. Do porovnání také nebyl zahrnut off-line beta editor Scratch 2, který se nepatrně liší od své on-line verze a to absencí některých funkcionalit. Především jej nelze chápat jako dalšího představitele druhé generace Scratch. Vzhledem k přímé podobnosti by se některé záznamy výsledků musely ve srovnávacích tabulkách zbytečně opakovat.

Předpokladem k porozumění této kapitoly je částečná znalost programování a zkušenost s alespoň jedním ze tří prostředí. Kapitola se odkazuje na různé dodatky. Nadbytečné či přesné popisy některých postupů najde čtenář v manuálech BYOB a Snap.

Obrázky se vyskytnou jen ojediněle a to pouze tehdy, mohou-li čtenáře obohatit či přispět ke srozumitelnosti textu. Cílem není provést grafické, nýbrž funkční srovnání. Pakliže bude mít čtenář zájem si některé části prostředí otestovat, bude si muset takové prostředí sám obstarat, spustit a požadovanou část otestovat.

3.1 Rozdíly a možnosti při programování

• Poznámka: *Je třeba upozornit, že představení podpory programovacích konceptů těchto prostředí tato sekce neobsahuje, neboť to bylo provedeno již v předchozí kapitole.*

3.1.1 Kategorie bloků a vše s jejich tvorbou spojené

• Kategorie bloků

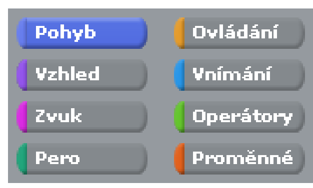
Každý blok v prostředí má přidělenu jednu kategorii. To přináší programátorům snazší orientaci v nabídce dostupných bloků, neboť chtějí-li například posunout postavu, zamíří do kategorie určené pro její pohyb. Svou barevnou odlišností taktéž zpříjemňují čtení scénáře, jelikož dle barvy bloku získáme základní představu o jeho schopnostech, respektive co onen blok přibližně způsobí v dané části scénáře.

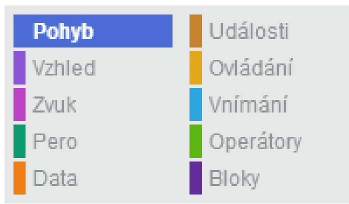
Počet kategorií je mezi prostředími odlišný. Scratch 1 nabízí šest základních, z nichž všechna ostatní vycházejí a rozšiřují je. Zachovávají navíc stejné zbarvení.

Největší změnou prošlo prostředí Scratch 2. Přejmenovalo kategorii `proměnné` na `data`, neboť obsahuje jak proměnné tak i seznamy a předchozí název byl tedy zavádějící. Dále byla bloky přeplněná kategorie `ovládání` upravena vyseparováním některých bloků událostí, které byly přesunuty do nové kategorie `události`. To byl další chytrý tah vývojářů vedoucí ke zvýšení orientace mezi některými bloky v prostředí. Poslední změna spočívala v přidání další kategorie `bloky`, o jejímž účelu si povíme později.

Nyní se vraťme k prostředí BYOB, které jakožto modifikace Scratch 1 nabízí stejný počet tlačítek pro přepínání mezi kategoriemi nad paletou bloků. Ačkoliv se tedy zdá, že má stejný počet kategorií, není tomu tak. Nabízí i kat. `ostatní` ukrytou v `proměnné`.

Prostředí Snap kráčí ve šlépějích svého předchůdce. Není však jasné, zdali počet kategorií navýší. Na jednu stranu vývojáři navýšení striktně odmítají[54][55], na druhou zase připouštějí, že autoři vlastních rozšíření do Snap by si mohli speciální kategorii pro přidané bloky vytvořit.[58] Nezbývá než vyčkat, jak se nakonec rozhodnou.





Obrázek 3.1.1 Kategorie všech prostředí. Zleva Scratch 1 společně s BYOB, další v pořadí Scratch 2; a nakonec kategorie Snap

- **Primitivní a vlastní bloky**

Každý projekt nabízí základní sadu bloků. Prostředí Snap označuje tyto bloky odlišným názvem – a to jako *primitivy*. Primitivy proto, že jsou naprogramované na nižší úrovni (tedy v programovacím jazyce, na kterém je prostředí postaveno) a uživatel nemůže jejich algoritmus číst ani nijak upravovat (ledaže by nahlédl do zdrojového kódu). Proto od nynějška nazývejme základní bloky přítomné ve všech prostředích jako *primitivy*.

Primitivní bloky slouží jako základní stavební jednotky pro tvoru dalších, uživatelem vytvořených bloků, jenž můžeme nazývat *vlastními*, a které Scratch 1 jako jediné prostředí neumožňuje vytvořit. Jak již bylo řečeno v první kapitole, tento fakt omezuje programátory a ve svém důsledku snižuje přehlednost celého projektu.

- **Tvorba vlastních bloků**

Scénář, jenž chceme vykonat na více místech, lze uložit do *vlastního bloku*, který – stejně jako primitivy – můžeme vybavit specifickým názvem a parametry. Definice bloků je tvořena jejich *hlavičkou* a *tělíčkem*. V hlavičce je umístěn název (identifikátor či nápis) společně s názvy a typy parametrů, a v tělíčku je onen scénář k užití na více místech.

Scratch 2 prozatím umožňuje pouze tvorbu vlastních příkazů. Vždy hodnotu vracejícím funkcím a podmínkám se však v programování dostává nezměrnému využití a tak vývojáři plánují tvorbu vlastních funkcí uživatelům časem nabídnout.[1]) Každý vlastní blok se po svém vytvoření automaticky umístí do kategorie `bloky`.

- *Poznámka: Blok typu podmínky je téměř totožný k bloku typu funkce. V prostředích je podmínka znázorněna se špičatými konci a vrací vždy a pouze pravdivostní hodnotu. Jejich princip je ovšem stejný a proto můžeme podmínku označit za jistý druh funkce.*

Prostředí BYOB a Snap jsou nejpestřejší. Umožňují vytvořit základní typy bloků (tj. *příkazy*, *funkce*, a *podmínky*). Zde již zřejmě nelze očekávat další změny. Prostředí programátora při tvorbě vlastního bloku ještě žádají o výběr kategorie, kterou mu přiřadí. Je proto možné vytvořit blok patřící do kategorie *pohybu*, *vnímání*, *proměnných* a tak dále. Vlastní blok najdeme vždy vespod palety bloků jemu přidělené kategorie.

Vzpomeňme též, že jsou v prostředích i primitivy pracující se seznamy. Jelikož je možné v BYOB a Snap vytvořit blok manipulující se seznamem, dávají nám prostředí příležitost přiřadit vytvářený vlastní blok i do kategorie `seznamy`.

Základní informace o tvorbě, úpravě, změně typu, a mazání vlastních bloků v prostředí Snap jsou uvedeny v dodatku A.

- **Vlastní bloky jedné či všech postav**

Při programování občas vytváříme na konkrétní postavě nezávislé (tedy obecné) bloky, které bychom rádi použili i u postav ostatních. Možnost vytvořit takový blok může být vítanou funkcionalitou, neboť stejný princip již využíváme při tvorbě proměnných.

[1]) Nezávazný návrh byl představen na https://github.com/LLK/scratch-flash/pull/179

Bohužel Scratch 2 umožňuje vytvořit blok pouze v rámci zvolené postavy a pokud jej chceme i u dalších postav, musíme takový blok přenést. Případnou změnu algoritmů či parametrů pak provádíme ve všech postavách ručně a vystavujeme se zanesení chyby.

Na to ovšem myslí BYOB i Snap a nabízejí programátorům vytvořit blok pouze pro *zvolenou* postavu nebo pro *všechny* postavy. Zvolit si mezi těmito možnostmi můžeme pouze při vytváření bloku. Pokud uživatel potřebuje provést změnu později, musí starý blok vymazat a založit nový s opačnou volbou, což je značně frustrující.

Za zmínku ještě stojí, že ačkoliv *scéna* žádné primitivy v kategorii `pohyb` nemá, je možné ji ve Snap oproti BYOB vlastní blok do této kategorie přiřadit. Scéna zobrazuje v této kategorii i bloky všech postav a pokud některý z nich obsahuje primitivy s pohybem spojené (např. `[jdi na]`), pak pokus o jejich vykonání scénou skončí chybou.

• Rozdíl mezi pojmy „příkaz“ a „procedura“

Někteří programátoři si možná pod pojmem *procedura* představí blok typu *příkaz*, jehož cílem je pouze něco vykonat bez vypočítávání a vracení nějaké hodnoty.

V práci ale chápejme proceduru jako konkrétní postup – tj. určení použitých bloků a jejich pořadí, jaké mají a kde použijí své parametry a s jakými argumenty se provedou. Procedurou je scénář jak příkazu, funkce, tak i podmínky. Proto kdykoliv o ní hovoříme, máme na mysli algoritmus, ale ne příkaz – na něj se odkážeme slovem *příkaz*. Pomůckou může být nahrazení slova *procedura* slovy *postup*, nebo *scénář*, či *část scénáře*.

Prostředí	Počet kategorií	Tvorba příkazů	Tvorba funkcí	Tvorba podmínek	Blok jedné či všech postav
Scratch 1	9	NE	NE	NE	--
BYOB	10	ANO	ANO	ANO	lze zvolit
Scratch 2	11	ANO	ještě ne	ještě ne	jen jedné
Snap	10	ANO	ANO	ANO	lze zvolit

Tabulka 3.1.1 Závěrečné shrnutí o kategoriích a tvorbě bloků

• *Poznámka: Do celkového počtu kategorií v tabulce byly započteny i kategorie* `seznamy` *a* `ostatní`*. V podstatě se spočítaly všechny barvy bloků v prostředích se vyskytujících.*

3.1.2 Proměnné a konstanty

Dosud jsou nám známy pouze dva typy proměnných, které vytváříme v obou verzích prostředí Scratch 1. Jedná se o proměnné buď všemi postavami sdílené a nebo proměnné přístupné pouze zvolené postavě. První typ nazývejme *proměnnými projektu* a druhý typ jako *proměnné postavy*. Jelikož lze už od prostředí Scratch 1 při tvorbě proměnné zvolit možnost „pro tuto postavu“, byla *scéna* vnímána jako *postava*, neboť sama tuto možnost nabízela. To změnilo až Scratch 2, které přes *scénu* umožňuje vytvořit pouze proměnnou všech postav. Jde o špatné řešení, neboť i scéna využije vlastní proměnné.

• Životnost a přístupnost k proměnným

V souvislosti s každým typem proměnné existují ještě pojmy *životnost* a *přístupnost* (někdy též *viditelnost*). Pojďme si je vysvětlit, neboť je nutné jim porozumět.

Životnost proměnné představuje dobu, po kterou bude proměnná existovat. Například proměnná projektu bude existovat tak dlouho, dokud ji z projektu neodstraníme či dokud onen projekt nezrušíme. Proměnná postavy zanikne až ve chvíli, kdy odstraníme ji samotnou či smažeme její postavu. Proměnné jsou vždy na někom závislé.

Přístupnost pak určuje, z jakých míst je ještě možné k proměnné přistoupit, tedy jestli je na ni z pohledu scénáře stále „vidět“. Na proměnné projektu se odkazujeme odkudkoliv, z čehož plyne, že se použijí pro uchování hodnot využitelných všemi postavami. Proměnné postavy jsou už v přístupnosti částečně omezené, neboť je možné je použít pouze ve scénářích téže postavy. Slouží k uchování „soukromých“, pro onu postavu důležitých, hodnot. Ostatní postavy k nim nemají přístup (nevidí je).

- **Cloud proměnné**

Prostředí Scratch 2 rozeznává ještě *cloud proměnnou*, což je modifikovaná proměnná projektu, která svou hodnotu po opuštění projektu nezapomíná – tj. ona ani její hodnota nezanikají. Její životnost je svázaná s projektem. Tyto proměnné „sdílí“ registrovaní uživatelé Scratch sítě při prohlížení takovýchto projektů. Využívají se například na uchování největšího skóre či rekordu, kterého kdy nějaký uživatel dosáhl. Dodejme, že jsou uloženy na serveru a prostředí u nich zobrazuje symbol mráčku.

Aktuálně můžeme ve Scratch 2 ukládat pouze čísla. V budoucnu půjde uložit i text a nebo vytvořit *cloud seznam*. Snap bude mít cloud proměnné nejdříve ve verzi 4.1.[53]

- **Proměnné scénáře**

BYOB nabízí další typ proměnných a to *proměnné scénáře*, které může čtenář znát jako *lokální proměnné* z jiných programovacích jazyků. Je možné je označit i za *dočasné* či *místní*. Žádné tlačítko na vytvoření těchto proměnných nehledejme, neboť se vytvářejí ve scénáři k tomu určeným primitivem `[proměnné scénáře]`. Používáme je na různé mezivýpočty nebo na uchovávání těch hodnot, které si postava nemusí pamatovat.

Jejich životnost je závislá na existenci scénáře, tudíž zaniknou až ve chvíli, kdy se scénář obsahující takovéto proměnné ukončí. Vytvoříme-li je uvnitř řídící struktury či vlastního bloku, pak budou existovat pouze dokud jeden z uvedených bloků neopustíme. Proměnné scénáře jsou blokům ve scénáři přístupné až od místa svého vytvoření.

Snap též disponuje proměnnými scénáře, ale s tím rozdílem, že po vytvoření uvnitř například podmínky existují i po jejím opuštění, což se v BYOB neděje. Více o tvorbě tohoto typu proměnných a jejich zvláštností nalezne čtenář v dodatku B.

Nakonec se zastavme u *kolizí* názvů proměnných. Pokusí-li se programátor vytvořit další proměnnou s názvem jiné již existující proměnné, pak k vytvoření nové proměnné nedojde. Paradoxně však toto pravidlo neplatí pro proměnné scénáře, které je možné vytvořit se stejným názvem na více místech jednoho scénáře. V takovém případě bude proměnná znovu vytvořena, změněna na počáteční hodnotou `{0}`, a scénář poběží dál.

- **Konstanty – neměnné hodnoty**

Opakem k proměnným jsou konstanty, tedy jednou přiřazené a dále neměnné (stálé či konečné) hodnoty, které se běžně využívají v moderních programovacích jazycích. Bohužel ani v jednom z prostředí konstanty vytvářet nemůžeme. Avšak bude-li tvorba konstant v budoucnu programátorům umožněna, pak pravděpodobně nepůjdou upravit jejich hodnoty příkazy `[nastav]` a `[změň]`, neboť jsou – podle definice – ihned po přiřazení první hodnoty nadále neměnné.

V prostředích BYOB a Snap je možné vytvořit funkci vracející vždy stejnou hodnotu, jimiž lze konstantu částečně zastoupit. Jde o jakousi imitaci konstant, s níž se seznámíme v podsekci 4.1.4. Konstanty budeme moci imitovat též ve Scratch 2, jakmile nabídne možnost tvorby vlastní funkce.

3.1.3 Datové typy

Každá hodnota je v programovacích jazycích nějakého datového typu, jenž definuje jak s ní pracovat, jak ji ukládat a jakého rozsahu může nabývat. Uživatelé jsou návrhem prostředí od určování datového typu osvobozeni a tak při tvorbě proměnné neřeší, jestli bude striktně toho (číselného) či onoho (textového) typu.

Některé datové typy jsou ještě *prvotřídní*, někdy nazývané též jako *plnohodnotné*. Každý datový typ se stává prvotřídním, pakliže jím reprezentovaná hodnota může být:

- přiřazena a zároveň uchována v proměnné
- součástí datové struktury (jako prvek v seznamu)
- předána jako argument proceduře
- vrácena jako výsledek procedury
- bezejmenně a na čemkoliv nezávisle vytvořena kdekoliv v a za běhu scénáře

Bezejmennost, též *anonymita*, vyjadřuje pouze to, že hodnotě nemusí být určeno nějaké jméno (*identifikátor*), pod kterým bychom ji znovu nalezli. Ukažme si na konkrétním příkladu jak ověříme, zdali je nějaký datový typ plnohodnotným. Zvolme si např. *číslo* a zjistěme, jestli splňuje všechny výše uvedené podmínky:

- Můžeme číslo přiřadit do a uchovat v proměnné? Ano, můžeme.
- Lze uložit číslo do seznamu jakožto jeho prvek? Jistěže. Používáme i seznamy čísel.
- Je možné zapsat číslo jako argument? Ano, je. Třeba nějaké funkci hodnotou `{8}`.
- Dokáží funkce vrátit číslo? Ano, například funkce `(souřadnice x)` vrací číslo.
- A dokážeme číslo bez určování jeho názvu vytvořit kdekoliv ve scénáři? Ano.

Obě verze Scratch mají pouze *číslo* a *text* jako prvotřídní datové typy. Hlavní výsadou BYOB a Snap jest, že mají všechny datové typy plnohodnotné, z čehož plynou různé programátorské výhody.[5, s. 22] Také umějí hodnotu jakéhokoliv datového typu zobrazit ve sledovači proměnné, bublině výsledku funkce, anebo v mluvící bublině postavy.

Pro ověření datového typu námi zadané hodnoty je k dispozici v prostředích BYOB a Snap podmínka `<je typu>`. Například zvolením možnosti `{číslo}` je touto podmínkou možné ověřit, jestli je zadaná hodnota číslem či nikoliv. Jen pro srovnání – takovéto ověřování se ve Scratch provádí vždy doprogramováním přídavného scénáře.

Na ověřování totožného původu (úplné rovnosti) hodnot máme pouze v prostředí Snap blok `<je totožný k>`, o němž se zmíníme blíže až v podsekci 3.1.11. Nyní si ale popišme podrobněji všechny datové typy podporované napříč prostředími.

Datový typ	Scratch 1	BYOB	Scratch 2	Snap
Číslo	ANO	ANO	ANO	ANO
Text	ANO	ANO	ANO	ANO
Pravd. hodnota	NE	ANO	NE	ANO
Seznam	NE	ANO	NE	ANO
Lambda výraz	--	ANO	--	ANO
Postava	--	ANO	--	až v 4.1
Kostým	--	NE	--	ANO
Zvuk	--	NE	--	ANO
Barva	--	NE	--	v návrhu
Celkem	2	6	2	8 + 1

Tabulka 3.1.2 Přehled datových typů – zdali jsou plnohodnotnými či nikoliv

• *Poznámka: V obou verzích prostředí Scratch pravdivostní hodnoty a seznamy existují, jen nejsou plnohodnotnými. Z toho důvodu je u nich v předešlé tabulce uvedeno* NE.

• Pravdivostní hodnota

Obě prostředí Scratch mají *pravdivostní* hodnotu jako datový typ, ale není prvotřídním. Když bychom si například uložili výsledek nějaké podmínky do proměnné, uložilo by se `{pravda}` či `{nepravda}` jakožto *text*.

Představme si, že budeme ve scénáři zjišťovat, zdali je výsledek nějaké podmínky roven slovu `{pravda}` přes `<je rovno>`. Jakmile by uživatel změnil jazyk svého prostředí na například anglický, pak by podmínka nevracela `{pravda}/{nepravda}`, nýbrž hodnoty `{true}/{false}`, což nesouhlasí s očekávanou textovou hodnotou `{pravda}`.

Booleova algebra reprezentuje obě *pravdivostní* hodnoty číselně a to jako `{1}/{0}`, kdy `{1}` stojí pro pravdu a `{0}` pro nepravdu. Zatímco ve Scratch 1 bychom s tímto značením nepochodili, Scratch 2 dokáže porovnat výsledek podmínky jak slovem `{pravda}` tak i číslem `{1}`. Ověřováním čísla namísto textu lze problém se změnou jazyků vyřešit.

Občas ale potřebujeme zaměnit stav *pravdivostní* hodnoty z pravdy na nepravdu či naopak a to s hodnotou v textové či číselné podobě jednoduchým způsobem neuděláme.

BYOB a Snap nás nenutí a ani samy nereprezentují pravdivostní hodnoty textem či číslem. Zavádějí dvě nové podmínky <pravda> a <nepravda>, což jsou konstanty vracející hodnoty sebe samých. Použití těchto konstant činí porovnávání na zvoleném jazyku prostředí nezávislé a prohodit jejich stav je možné použitím unárního operátoru <není>, který z {<pravda>} udělá {<nepravda>} a naopak.

- **Seznam**

Ve Scratch vytváříme seznam manuálně kliknutím na příslušné tlačítko „*Vytvořit seznam*“. Popišme si – alespoň abstraktně –, co se děje uvnitř prostředí při jeho tvorbě.

Každý seznam se uchovává někde v paměti počítače. Abychom k němu mohli opakovaně přistupovat – tedy najít jej opět v paměti –, musíme si na něj uchovat *referenci*[1]). Její získání a uchování zajistí prostředí automaticky. Nazvali jsme-li seznam třeba jako „*seznam čísel*“, pak se v kategorii seznamy objeví funkce (seznam čísel).

Funkce (seznam čísel) vrátí obsah seznamu v textové podobě, jehož prvky jsou odděleny mezerou, namísto vrácení reference samotné. Od tohoto místa Scratch bohužel potenciál seznamů nevyužívá a to proto, že s referencí nemůžeme vůbec manipulovat.

Sice budeme moci vybrat tento seznam v blocích pracujících se seznamy a provést nad ním nějakou operaci, ale stále nebude možné vytvořit obecný blok, který by třeba naplnil **různé** seznamy náhodnými čísly (tj. jednou třeba (seznam1), jindy (seznam2), poté (seznam čísel)...). Je vyloučené naprogramovat scénář tak, aby prováděl stejnou věc nad různými seznamy dle našeho výběru. Řešením je tedy kopírování stejného scénáře do dalších bloků, kterým jen změníme název a určíme odlišný seznam (tj. např. [naplň seznam1], [naplň seznam2], [naplň seznam čísel]...).

Hřebík do rakve seznamům zatloukl i fakt, že jsou reference konstantní (dále neměnné) a tudíž nelze někde za běhu scénáře vytvořit seznam bezejmenný, který bychom vnutili třeba funkci (seznam čísel), aby jej přijala za svůj a zapomněla onen seznam původní. Jsme tak „nuceni“ vnímat seznamy jen jako pouhé kontejnery na hodnoty.

V prostředí Snap jsou seznamy samozřejmě plnohodnotné. Vytvářejí se naprosto odlišným způsobem a **vždy** pracujeme s referencemi na ně. Žádná konstantní reference v podobě (název seznamu) neexistuje stejně jako tlačítko „*Vytvoř seznam*“. Přesto to není přímo seznam samotný, kdo je prvotřídní, nýbrž reference na něj.[2])

Manipulace s referencemi je pro nezasvěcené něčím novým a zřejmě budou v této oblasti ztraceni. Od toho se v práci vyskytuje dodatek C, který se snaží přiblížit způsob získávání, sdílení (předávání), a porovnávání referencí.

Ukažme si ještě jak se seznamy tvoří v prostředí Snap. Na obrázku 3.1.2 jsou vytvořeny celkem tři seznamy funkcí (nový seznam), která každý seznam **bezejmenně** zkonstruuje a vrátí na něj referenci. Co s ní dále uděláme již funkci (nový seznam) nezajímá. Lze ji třeba uložit do proměnné – to je ostatně způsob, jak seznamy v prostředí uchováváme. Všimněme si i černých šipek na jejím konci, kterými měníme počet argumentů, jež se po vytvoření seznamu promění v jeho prvky.

Obrázek 3.1.2 Ukázka odlišné tvorby seznamů v Snap

Bloku [povídej] byla touto funkcí na čerstvě vytvořený seznam reference vrácena, ten ji přijal, přikázal postavě povědět jeho obsah a nakonec referenci „zapoměl“, neboť jsme

[1]) Jinak řečeno *odkaz*. Toto je důležitý a velmi abstraktní pojem, který se v práci hojně vyskytuje.
[2]) To platí i pro následující typy. Řekneme-li, že jsou prvotřídní, myslíme tím jen ony reference na ně.

ji neuchovali v žádné proměnné a ani ji blok [povídej] neposílá někam dál. Přímo ve středu obrázku je dále ukázáno, jak se vytváří seznam prázdný. A konečně úplně dole je seznam s dalším seznamem uvnitř, což rozeznáváme jako *vnořený seznam*.

Prostředí BYOB se nachází někde uprostřed. Umožňuje vytvořit seznam tlačítkem a referenci na něj udržovat konstantní (stejně jako ve Scratch), ale dokáže vytvořit seznam i anonymně někde ve scénáři, jehož referenci lze uchovat v proměnné (ekvivalentní ke Snap). Doplňme, že bloky pracující se seznamy neřeší, jak byly seznamy vytvořeny. Buď za argument zvolíme název seznamu z rozbal. nabídky či předáme referenci.

Z výše uvedeného vzniká mezi prostředími zajímavý rozdíl. Skrývání seznamů nebylo možné provést ve Scratch programově. Pakliže bychom v BYOB vytvořili seznam standardním způsobem přes k tomu určené tlačítko, jeho sledovač bychom též neschovali, neb pro to neexistuje příkaz. Když však vytvoříme seznam druhým způsobem (shodným se Snap), je možné jej skrýt příkazem [skryj proměnnou], neboť si referenci na něj uchováváme v proměnné a tu skrýt umíme. Vývojáři Scratch 2 vyřešili problém s ovládáním sledovačů seznamů novými příkazy [skryj seznam] a [ukaž seznam].

- **Kostým**

Ve všech prostředích přepínáme mezi kostýmy postavy standardním způsobem a to přes příkaz [oblékni kostým]. Postava samozřejmě musí mít nejprve nějaký nakreslený či naimportovaný. To samé, včetně dále uvedených odstavců, platí i pro pozadí scény.

Na různé kostýmy se odkazujeme vždy jejich názvem v textové podobě. Úplný přístup k nim však nemáme. Pakliže chceme, aby několik různých postav používalo jeden a tentýž kostým, musíme jej naimportovat každé zvlášť. A změníme-li název nějakého kostýmu, pak je třeba na změnu názvu zareagovat i ve scénářích těchto postav.

Pro čtenáře bude jistě překvapením, že ve Snap můžeme s kostýmem pracovat jako s objektem. Ke kostýmům nemusíme přistupovat přes jejich název, ale přes referenci na ně získanou funkcí (šatník) vracející seznam dostupných kostýmů postavy.[1]) Referenci na některý z nich lze uložit na později či ihned předat bloku [oblékni kostým], který jej rovnou oblékne namísto aby ho dle názvu vyhledával mezi ostatními kostýmy.

Obrázek 3.1.3 Ukázka práce s prvotřídním kostýmem v Snap

Prostředí samo manipuluje s kostýmy pomocí referencí. Když uchováme přes funkci (šatník) referenci na kostým a ten poté postavě odmažeme, stále k onomu kostýmu budeme mít přístup. Prostředí jej nezapomene, jen ho v kartě „*Kostýmy*“ dále nezobrazí. Pochopme, že držením reference máme v rukou kostým samotný – ne jen název.

Dokážeme tak vytvořit například postavu s několika naimportovanými kostýmy a seznam s referencemi na tyto kostýmy uchovat v proměnné všech postav (dostupné_kostýmy). Jiné postavy budou moci ke kostýmům přistoupit a obléknout je. Správa kostýmů (přejmenování, pořadí atd.) se nepředstavitelně zjednoduší.

- *Poznámka: Způsob jak postavám hromadně rozeslat kostým a přikázat jim ho obléknout si společně s obrázkem představíme v souvislosti s událostmi (viz podsekce 3.1.12).*

Nicméně *kostým*, tak jak byl představen, je stále ve vývoji. Kvůli tomu je funkce (šatník) označena za experimentální a před obyčejnými uživateli skryta. Zobrazí se jen po přepnutí prostředí do vývojového módu, o němž více v podsekci 3.3.5.

[1]) Zde uživatelé BYOB mohou namítat, že ve svém prostředí ke kostýmům postavy přistupují funkcí (atribut) (viz sekce 3.1.5). To je sice pravda, ale ta vrací seznam názvů kostýmů a ne referencí na ně.

- **Zvuk**

Vše výše o kostýmech uvedené platí i pro zvuky. Nejprve je musíme nahrát mikrofonem či naimportovat postavě a až poté je lze přehrát bloky [hraj zvuk] [hraj zvuk až do konce]. Přistupujeme k nim opět přes název a existují jen v dané postavě. Manuálním smazáním v kartě „*Zvuky*" je úplně ztratíme. Když zkusíme přehrát zvuk s názvem již odstraněného zvuku, prostředí ho u postavy nenalezne a žádný zvuk nepřehraje.

Ve Snap jsou zvuky taktéž prvotřídní a tak máme přístup k referencím na ně experimentální funkcí (jukebox). Nejsou zobrazeny graficky, ale textem [object Object].

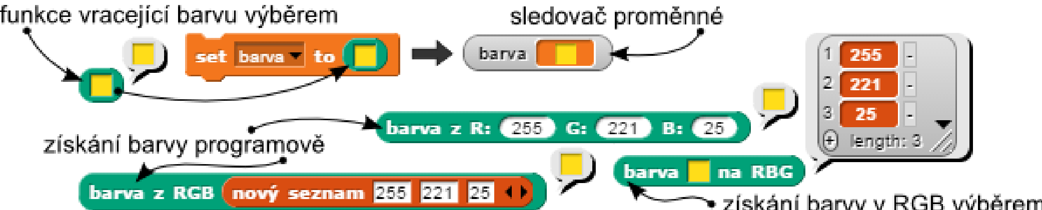
Referenci na zvuk ještě bloky [hraj zvuk] a [hraj zvuk až do konce] zpracovat neumí. Zatím je pro vývojáře přednější *kostým*. Tomu nasvědčuje i fakt, že referenci na zvuk prostředí zapomene při ukládání projektu a po načtení již v proměnné chybí.

- **Barva**

Práce s barvami je v prostředích komplikovaná. Barvu – kromě výběru z barevné palety či kapátkem – a odstín pera nastavujeme programově číselnou hodnotou přes [nastav barvu pera] a [nastav odstín pera]. Též musíme znát jejich číselný rozsah.

Vývojáři Snap plánují vytvořit barvy prvotřídní, což nám umožní je nejenom vytvářet, uchovávat, předávat, ale i míchat v barvy jiné. K vyjádření barvy v informatice používáme různé barevné modely, z nichž Snap bude znát jen modely RGB a HSV.[1])[50]

Prozatímní návrh počítá s přidáním funkcí k získání barvy samotné, k jejímu převodu na již zmíněný model (hodnoty tří složek budou vráceny v seznamu), a k převodu modelu zpět na barvu.[50] Obrázek 3.1.4 zobrazuje nikým nepotvrzenou představu autora této práce o možné budoucí práci s barvami v prostředí Snap.



Obrázek 3.1.4 Nezávazná představa práce s dat. typem *barva* v Snap

- **Postava**

Při komunikaci mezi postavami se setkáváme s totožným problémem. Například funkce (z), umožňující získat základní údaje o jiných postavách včetně scény, se na konkrétní postavu odkazuje jejím názvem v textové podobě. Můžeme tak v druhém parametru této funkce vybrat z rozbalovací nabídky konkrétní postavu anebo dosadit proměnnou, jejíž hodnota je rovna názvu některé z postav. Jenže změníme-li název nějaké postavy, pak musíme změnit i v minulosti provedený výběr v rozbalovací nabídce či přenastavit hodnotu proměnné u funkce (z). I tento problém však eliminujeme zpřístupněním referencí na postavy. Při manipulaci s referencemi na postavy se o jejich název nezajímáme – referencemi totiž přistupujeme přímo k nim. Nemusíme tak znát jejich jména a v případě přejmenování postav nebude náš scénář ovlivněn.

Reference na postavy se v prostředí BYOB získávají funkcí (postava), mezi možnostmi jejíhož parametru lze vybrat buď konkrétní postavu, všechny postavy (vrácené v seznamu); anebo scénu samotnou. Pro získání reference na postavu je sice nutné znát její název, ale po jejím získání se nemusíme obávat závislosti scénářů na jejím jméně. Funkci (postava) bude mít prostředí Snap až s příchodem verze 4.1.

Jelikož jde o relativně komplexní oblast, tak si využití referencí na postavy popíšeme podrobněji až v podsekci 3.1.5 zaměřené na objektově orientované programování.

---

[1]) RGB model definuje barvu smícháním tří barevných složek – červené, zelené a modré – v různém poměru. Model HSV definuje barvu jinými složkami – barevným tónem, sytostí a hodnotou. Další informace musí čtenář v případě zájmu nastudovat sám.

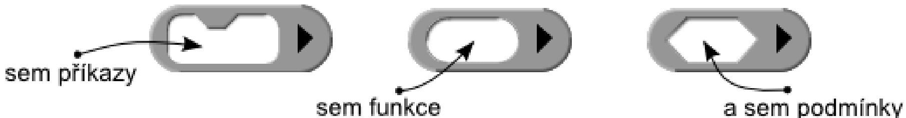
- **Zaobalená procedura (výrazová a příkazová lambda)**

Jedním z nejpodstatnějších rozdílů BYOB a Snap vůči oběma verzím Scratch je možnost bezejmenně vytvořit, uchovat, předat, a v libovolný čas vykonat; nějakou sestavenou proceduru – nepojmenovanou část scénáře tvořenou z příkazů, funkcí, čí podmínek. Můžeme proto tvrdit, že v těchto prostředích lze proceduru uchovat i jako data – a tudíž je, stejně jako všechny ostatní datové typy v prostředích BYOB a Snap, prvotřídní.

Zakladatelem této myšlenky byl Alonzo Church, jenž popsal formální systém v matematické logice zvaný *lambda-kalkul* k vyjádření výpočtu založeném na abstrakci funkcí, který sehrál klíčovou roli při vývoji teorie programovacích jazyků – převážně těch spadajících do funkcionálního paradigmatu chápající výpočet jako vyhodnocení funkcí.[29] *Lambda* je písmeno řecké abecedy se znakem λ. Proto se někdy používá zápis λ-*kalkul.*

Sami vývojáři BYOB a Snap si uvědomují nejenom složitost užívané terminologie, ale hlavně náročnost celé teorie. Dovolili si přesto začlenit i tuto oblast informatiky do svých prostředí, neboť dokázali využít potenciál grafického návrhu Scratch a minimální úpravou rozhraní těchto prostředí docílit jednoduché a elegantní manipulaci s bezejmennými procedurami.

Na obrázku 3.1.5 vidíme tři šedé funkce, které mají v sobě prázdné místo na bloky různého typu, jenž do nich „vsázíme“. Tyto tři funkce slouží k vytvoření bezejmenných procedur a nazýváme je *zaobalujícími funkcemi*, neboť „zaobalují“ dodané bloky – zde je přímá spojitost mezi pojmem *zaobalení* a vzhledem těchto funkcí. Mimochodem – do zaobalující funkce nelze vložit bloky událostí.



Obrázek 3.1.5 Tři zaobalující funkce Snap očekávající bloky

- *Poznámka: Dejme pozor na záměnu pojmů. Zaobalující funkce a zaobalená funkce jsou dvě odlišně věci. Zaobalující funkcí je každá tato: , , ; kdežto zaobalenou funkcí je například* (kostým číslo) *vsazená do , která ji zaobaluje.*

Tyto namísto vykonání bloků do nich vsazených raději vrátí samy sebe – bloky zaobalené jednou z těchto funkcí, které se automaticky prostředím nevykonají. Kdy zaobalenou proceduru vykonáme určujeme pro nás novými primitivy [spusť] a (zavolej), které vyžadují referenci na nějakou takovou zaobalenou proceduru. Příkaz [spusť] použijeme u zaobalených procedur, jejichž výsledkem je nějaká akce namísto vrácené hodnoty. Funkci (zavolej) použijeme u zaobalených procedur vracejících hodnotu.

Vývojáři umožňují přes nabídku vyvolanou pravým tlačítkem myši blok či část scénáře (proceduru) *zaobalit* a prostředí automaticky vybere správnou ze tří zaobalujících funkcí. Pokud se ale rozhodneme provést zaobalení ručně, musíme si pro tyto tři bloky dojít. Ačkoliv mají bloky šedou barvu, nacházejí se překvapivě v kategorii operátory.

Správné názvy zaobalených příkazů, funkcí, a podmínek jsou *příkazová lambda*, *lambda funkce*, a *lambda predikát*. Místo těchto složitých pojmů začali vývojáři ve Snap používat pojem *ringify*, jenž překladatel do češtiny výtečně přeložil na *zaobalit*, z čehož vyústil pojem *zaobalení*, jehož se drží i sám autor této práce.

Ještě si ukažme rozdíl tvorby zaobalených procedur mezi Snap a BYOB v tabulce 3.1.3. BYOB používalo bloky [scénář] a (výraz), které však nebyly tolik intuitivní.

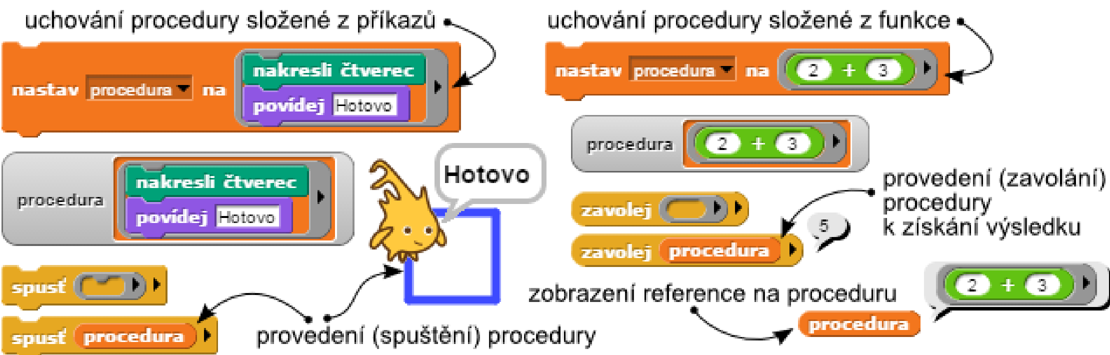
Název	V Snap	V BYOB	Popis
příkazová lambda		scénář	procedura tvořená příkazy
lambda funkce		výraz	procedura tvořená funkcemi
lambda predikát		výraz	procedura tvořená podmínkami

Tabulka 3.1.3 Popis všech lambda druhů a jejich vzhled v Snap a BYOB

Konečně k využitelnosti zaobalování. Potřebujeme-li si nějakou část scénáře vykonat na více místech a nechceme si kvůli tomu vytvářet vlastní blok, můžeme si potřebný scénář zaobalit, uchovat například v proměnné a poté jej vhodným primitivem vykonat. Dále je možné využít zaobalené procedury k záměně chování a to tak, že primitivy `[spusť]` anebo `(zavolej)` budeme vykonávat zaobalenou proceduru uchovanou v proměnné (kupříkladu `(procedura)`), jejíž obsah v určitých chvílích změníme. Jednou tak bude obsahovat referenci na zaobalenou proceduru A, po chvíli na proceduru B, pak na C..., přičemž hlavní scénář s blokem `[spusť]` či `(zavolej)` se odkáže jen na hodnotu (zaobalenou proceduru) proměnné `(procedura)`, která se různě proměňuje.

Jak už bylo nastíněno – namísto tvorby vlastního bloku si můžeme tělíčko onoho bloku (resp. jeho proceduru) zaobalit – bezejmenně a bez zvolené kategorie – a uchovat v proměnné či jako prvek seznamu. Stejně jako mohou vlastní bloky vyžadovat vstupní hodnoty (*parametry*), mohou tak i zaobalené procedury. Blokům `[spusť]` a `(zavolej)` pak ale musíme zvolit potřebné hodnoty přes černé šipky jako u funkce `(nový seznam)`.

Užít zaobalování lze ještě v několika dalších případech, na které zde není prostor. Kompletní popis s příklady zaobalování si ukážeme až v sekci 4.3. Přesto se podívejme alespoň na obrázek 3.1.6, na kterém je znázorněna práce se zaobalenými procedurami. Do proměnné `(procedura)` si ukládáme referenci jednou na zaobalené příkazy v (vlevo) a podruhé (vpravo) referenci na zaobalenou funkci `(sečti)` uvnitř . Ukázán je i způsob, jak obsah zaobalujících funkcí vykonat bloky `[spusť]` a `(zavolej)`.



Obrázek 3.1.6 Práce s zaobalujícími funkcemi a v Snap

3.1.4 Parametry bloků

Vstupní data bloků jsou označována za *parametry*. Umožňují nám proceduru *parametrizovat*, tj. ovlivnit její průběh a tím i konečný výsledek dodáním nějakých hodnot. Včetně toho mají určený typ, jímž napovídají, jakého datového typu by měla dodávaná hodnota být. Přeci jen – chceme-li pracovat např. s číslem, měli bychom to dát najevo.

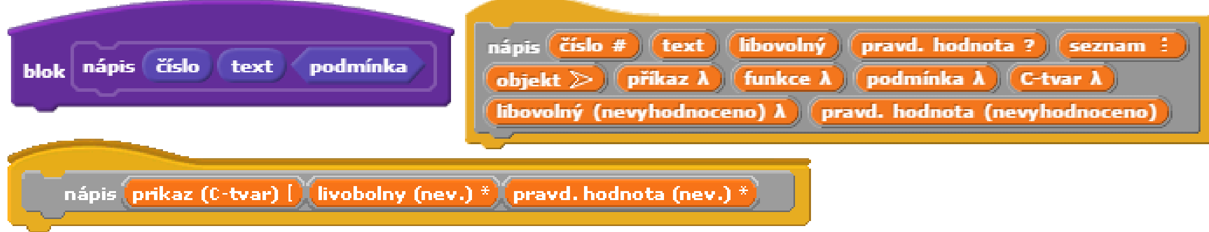
Parametry dělíme ještě na *formální* a *skutečné*. Při editaci bloku s alespoň jedním parametrem pracují jeho *programátoři* s formálními parametry a když jej využívají jako jeho *uživatelé* ke složení scénáře, pracují s parametry skutečnými. Hodnoty, které pak – z pohledu uživatele – skutečným parametrům předávají nazýváme *argumenty*.

• Formální parametry – z pohledu programátora bloku

Programátoři bloků se vždy rozhodují, jestli budou potřebovat nějaké parametry (přesněji kolik a jakého typu). Způsob vytvoření parametru, zvolení jeho názvu, a nastavení typu; se samozřejmě odvíjí od použitého prostředí. Nemá-li blok žádný parametr, pak jej označujeme za *bezparametrický*.

Po vytvoření se parametr objeví v *hlavičce* svého bloku s názvem, tvarem, barvou a případně i se zvláštním symbolem. Abychom si udělali jasno mezi rozdílným značením formálních parametrů napříč prostředími[1]), pohlédněme na následující obrázek 3.1.7.

[1]) Vyjma prostředí Scratch 1, které vlastní bloky – a proto ani parametry – vytvářet neumožňuje.



Obrázek 3.1.7 Rozdílné značení a typy parametrů vlastních bloků

Vlevo nahoře je umístěn blok ze Scratch 2 se třemi parametry, které společně reprezentují množinu typů (včetně jejich názvů) prostředím nabízenou. Mimo jiné parametr s typem očekávající pravdivostní hodnotu a názvem „podmínka" je hexagonálního tvaru. Z jakého důvodu mají formální parametry fialovou barvu si řekneme později.

Napravo je blok z prostředí Snap s dvanácti parametry. Každý taktéž představuje odlišný typ. Všechny formální parametry mají vždy tvar funkce (zakulacený) a některé mají i doplňující symbol k orientačnímu určení typu. Symboly jsou součástí parametru pouze v hlavičce bloku a tak „nepřekáží" ve scénáři. Všechny parametry jsou oranžové.

Poslední uvedený blok pochází z BYOB. Ačkoliv je s nabídkou typů stejně bohatý jako Snap, v ukázce má onen blok pouze tři parametry. Jde o ty typy, jejichž symboly se od prostředí Snap liší. Ostatní typy parametrů v BYOB nemají symbol vůbec žádný.

- **Skutečné parametry – z pohledu uživatele bloku**

Nyní již víme, jak parametry vypadají uvnitř vlastních bloků. Co však museli vývojáři při návrhu těchto vizuálních prostředí vymyslet, bylo jejich zobrazení „zvenku" bloku.

Opusťme představu, že jsme uvnitř něj jakožto jeho programátoři a že ho nyní chceme použít ve scénáři jakožto uživatelé. Parametr nás informuje o žádaném datovém typu svým specifickým tvarem. Má-li tvar například zakulacený, znamená to, že očekává pouze číselnou hodnotu. Uživateli je tak na první pohled jasné, že mu prostředí dovolí vložit pouze číslice či funkci s číselným výsledkem. Netřeba snad připomínat, že mezi typem a tvarem parametru je přímá závislost.

Všem čtenářům jsou známy alespoň tyto tři tvary parametrů:

- oválný – očekává číslo, přijme i funkci (obsahuje jej například funkce `(sečti)`)
- hexagonální – očekává pravd. hodnotu, přijme jen podmínku (třeba `[když-jinak]`)
- obdélníkový – nevybírá si, přijme cokoliv kromě příkazů (kupříkladu `[povídej]`)

Mezi uvedenými tvary jsou v prostředích rozdíly. Funkční rozdíl je například u Scratch 2 a Snap, která oproti Scratch 1 a BYOB umožňují zapsat do *číselných parametrů* řádovou čárku jedině tečkou. Rozdíl grafický je pak třeba ve Scratch 2, které oproti ostatním prostředím znázorňuje parametr typu *libovolný* čtvercovým tvarem.

Vrátíme-li se k našim blokům uvedených na obr. 3.1.7, pak by v prostředích mimo jejich editor – tedy zvenku – vypadaly stejně jako na obrázku 3.1.8. Vidíme, že každý typ má svůj specifický tvar (grafickou reprezentaci či znázornění).

Obrázek 3.1.8 Grafická reprezentace parametrů „zvenku" bloků

Jelikož by si čtenář možná rád prohlédnul editor parametru v prostředí Snap (k němuž je téměř totožný editor z BYOB), může se podívat na obrázek 3.1.9. Na něj se budeme odkazovat i v dalších částech textu pro vysvětlení všech dostupných nabídek. Čtenář má také možnost porovnat symboly formálních parametrů z obrázku 3.1.7 podle typů uvedených na obrázku 3.1.9. Nechť se řídí názvy parametrů totožných k názvům typů.

Obrázek 3.1.9 Částečně oříznutý editor parametru v prostředí Snap

Popišme si všechny typy parametrů Snap, jež si blíže představíme až ve čtvrté kapitole:

- `číslo` – očekává číselný argument; prostředí zabraňuje vepsání nečíselných znaků
- `text` – očekává textový argument, lze však doplnit i číselný; zvolí-li autor takovýto typ parametru, dává najevo, že jeho argument nebude užit k výpočtu
- `libovolný` – očekává cokoliv; oproti typu *text* má kratší pole k zadání argumentu
- `pravdivostní hodnota` – očekává blok podmínky vracející pravdivostí hodnotu; lze tedy vložit i konstanty <pravda> a <nepravda>
- `seznam` – očekává referenci na existující seznam
- `objekt` – očekává referenci na postavu či objekt vytvořený zaobalenou procedurou (viz vlastní objekty a třídy v podsekci 3.1.5)
- `příkaz (zaobalený)` – je zaobalující funkcí ; dodané příkazy nevyhodnotí a předá je k vyhodnocení dovnitř bloku; lze pracovat s parametry zaobalující funkce.
- `funkce` – stejné s *příkaz (zaobalený)*; místo příkazů očekává funkci či složené funkce
- `podmínka` – stejné s *funkce*; místo funkcí očekává podmínku či složené podmínky
- `příkaz (C-tvar)` – očekává příkaz či složené příkazy, jenž automaticky vloží do zaobalující funkce bez vědomí uživatele; nelze určovat parametry zaob. proc.
- `libovolný (nevyhod.)` – stejné s *příkaz (C-tvar)*; místo příkazů očekává funkci či složené funkce; automaticky zaobaluje do
- `pravd. hodnota (nevyhod.)` – stejné s *funkce*; místo funkcí očekává podmínku či složené podmínky; automaticky zaobaluje do

Prostředí Snap disponuje ještě jedním speciálním typem parametru, který je při tvorbě vlastních bloků nedostupný – použijí ho jen vývojáři prostředí. Tento typ vypadá stejně jako typ *text*, jen je v něm možné zapisovat text na více řádků, neboť reaguje na stisk klávesy `Enter`. Celý obsah je sázen strojovým fontem určeným k psaní zdrojových kódů. Má ho např. funkce `(JavaScript funkce)`, do které píšeme kód jazyka JavaScript.

• Varianty parametrů

BYOB a Snap ještě umožňují ve spodní části editoru parametru (obrázek 3.1.9), kde se nacházejí tři kulaté přepínače, přidělit parametru jednu z *variant*:

a) jednoduchá varianta – parametr tak jak jej známe, očekávajíc jen jeden argument
b) vícenásobná varianta – vybaví parametr černými šipkami k přidání/odebrání argumentů, které jsou nakonec předány a uchovány v seznamu
c) přesahující varianta – umožní přistoupit k parametru zvenku bloku

Ke zvolení *vícenásobné varianty* musí být parametr jedním z typů: *číslo*, *text*, *libovolný*, nebo *libovolný (nevyhod.)*. Formální parametr *vícenásobné varianty* získává symbol tří teček (...) a u typů *číslo* a *libovolný (nevyhod.)* budou symboly (#, λ) nahrazeny (...).

Přesahující varianta je speciálním typem parametru, který se jeví jako proměnná scénáře vytvořená uvnitř bloku s přesahem ven. V BYOB je hodnota zvenku neměnná, ve Snap naopak. Formální parametr této varianty má symbol šipky (↑).

Závěrem doplňme, že oběma verzím Scratch je známa pouze jednoduchá varianta parametrů. Zbylé dvě varianty popisuje čtvrtá kapitola práce v odlišných sekcích.

- **Výchozí hodnota parametrů**

Některé primitivy předvyplňují hodnoty svých parametrů. V paletě bloků programátor najde třeba funkci `(zvol náhodné číslo)`, jejíž dva parametry ohraničující interval vygenerovaného čísla obsahují hodnoty `{1, 10}`. Nebo například funkce `(spoj)` obsahuje `{hello, world}`. Hovoříme o takzvaných *výchozích hodnotách* parametrů.

Uživatelé prostředí Scratch 2 jistě znají problém, kdy každý číselný parametr vlastního bloku obsahuje předvyplněnou hodnotu `{1}`. Někteří by ji rádi změnili, jiní úplně odstranili. Výchozí hodnotu je však možné přiřadit či odstranit pouze v prostředích BYOB a Snap, jež samozřejmě hlídají, aby výchozí hodnota v číselném parametru nebyla textem. Ve formálním parametru bude výchozí hodnota uvedena za symbol rovnítka (=). Ukázka: `(a = 10)`. Zároveň musí být splněno, že parametr jest:

- v *jednoduché variantě*
- jedním z typů: *číslo*, *text*, *libovolný*, *libovolný (nevyhodnoceno)*

- **Proměnlivé či konstantní parametry**

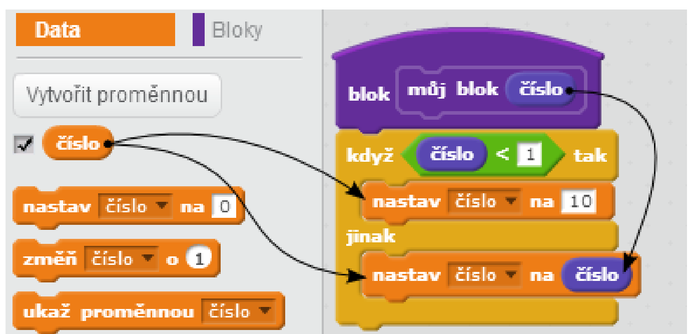
Parametry vlastních bloků jsou ve Scratch 2 zbarveny fialově. Prostředí tím dává programátorům najevo, že si tyto fialové parametry nebudou s oranžově zbarvenými bloky z kategorie `data` rozumět. To má za následek, že jsou parametry ve Scratch 2 konstantní, neboť je nemožné je příkazy `[nastav]` a `[změň]` upravit. Zvolený argument, se kterým se do bloku vstupovalo, zůstane v parametru uchován natrvalo.

BYOB a Snap mají parametry proměnné. Volí tedy opačnou strategii a protože se může jejich hodnota pozměnit, mají logicky oranžové zbarvení.

Na obrázku 3.1.10 jsou dva totožné bloky vytvořené v Scratch 2 a Snap s fialovými parametry `(parametr1)` a `(parametr2)` u Scratch 2 a oranžovými parametry `(parametr1)` a `(parametr2)` u Snap. Projekty, v nichž byly tyto bloky vytvořeny, též obsahují proměnné `(proměnná projektu)` a `(proměnná postavy)`. Jsou to pouze prostředí Snap a BYOB, ve kterých lze v nabídce `[nastav]` vybrat `(parametr1)` a `(parametr2)`.

Obrázek 3.1.10 Proměnné a konstantní parametry v Scratch 2 (vlevo) a Snap (vpravo)

Mnohdy ověřujeme, jestli je hodnota např. číselného parametru ve stanoveném intervalu. Takový, avšak obecný, příklad je zobrazen na obrázku 3.1.10, jenž ověřuje `(číslo)` k intervalu $\langle 1, \infty)$. Překročením intervalu se hodnota parametru nastaví na `{10}`.



Obrázek 3.1.11 Řešení úpravy hodnoty param. v Scratch 2 (vlevo) a Snap

Zatímco ve Snap příklad vyřešíme pohotově, ve Scratch 2 musíme k jeho vyřešení použít navíc proměnnou postavy, přestože s ní samotnou nějaká kontrola čísla vůbec nesouvisí. Navíc s vzrůstajícím počtem parametrů roste i počet pomocných proměnných postavy.

- **Parametr s rozbalující nabídkou možností**

Některé argumenty parametrů jsou předem známé a vyžadované (či doporučené). Proto některé primitivní bloky tyto hodnoty nabízí v rozbalovací nabídce, která je k parametrům „přilepena“. Příkladem budiž příkaz `[nastav]`, který skrze tuto nabídku vyžaduje výběr proměnné k jejímu nastavení. Rozbalující nabídku má i parametr funkce `(z)` s nabízenými možnostmi `{odmocnina/sin/cos/...}`. A zmiňme i číselný parametr bloku `[zamiř směrem]`, který krom `{(90) doprava/(-90) doleva/(0) nahoru/(180) dolů}` povoluje i vložení vlastní hodnoty k určení směru. Funkcionalitu čtenáři jistě znají.

Jenže někteří programátoři by rádi parametrům svých bloků rozbalovací nabídku taktéž přiřadili. To je možné pouze v prostředí Snap. Z popisu zřejmě vyplynulo, že musíme nabídku nějakými možnostmi naplnit. To je možné přes bíle podbarvené dialogové okno vyvolané kliknutím pravého tlačítka myši v editoru parametru jak ukazuje obr. 3.1.9. Parametr zároveň musí být v *jednoduché variantě*. Více až v podsekci 4.1.3.

Prostředí	Počet typů	Proměnné parametry	Varianty parametru	Výchozí hodnota	Rozbal. nabídka	Značení parametru
Scratch 1	--	--	--	--	--	--
BYOB	12	ANO	ANO	ANO	NE	symbol
Scratch 2	3	NE	NE	NE	NE	tvar
Snap	12	ANO	ANO	ANO	ANO	symbol

Tabulka 3.1.4 Rozdíly parametrů u vlastních bloků

3.1.5 Koncepty objektově orientovaného programování

Popišme si některé principy spadající do paradigmatu objektově orientovaného programování (zkráceně OOP), jenž jsou prostředími v různé míře podporovány. V programování může být objektem prakticky cokoliv – věc, živočich, obrázek, barva, apod.; a proto jsou postavy, scéna i projekt v prostředích na nižší úrovni naprogramovány jako *objekty*.

Objektu jsou přidány nějaké vlastnosti (označované jako *atributy*) složené z názvů a hodnot, které si objekt pamatuje. Některé z nich je možné měnit, jiné nikoliv (účelně). Se samotnými hodnotami objektu toho ale moc nesvedeme, a tak je možné jej vybavit chováním (označované jako *metody*), kterým komunikujeme s oním či ostatními objekty. Zajímá nás, co metody provedou objektu, ale nepídíme se po tom, jak toho dosáhnou.

- *Poznámka: V této části řaďme scénu mezi postavy, abychom ji neustále nezmiňovali.*

Bohužel prostředí Snap ve své současné verzi 4.0 oproti předchůdci BYOB nepodporuje všechny (a další plánované) objektově orientované principy, jež mají obrovský potenciál při vývoji projektu. Z toho důvodu se v poslední kapitole této práce věnujeme příkladům OOP minimálně.

- **Úvod do atributů**

Každý *atribut* postavy ovlivňujeme téměř ve všech případech primitivními příkazy. Například příkaz [pero dolů] ovlivní atribut uchovávající pravdivostní hodnotu, zdali má postava pero dole či nahoře. Příkaz [schovej se] ovlivní stejným způsobem atribut uchovávající informaci o viditelnosti postavy. Kupříkladu [změň x] upraví číselnou hodnotu atributu x-ové souřadnice představující pozici postavy na scéně (ta logicky některými atributy nedisponuje, neb se nepohybuje a ani nekreslí). Mezi atributy postavy řadíme i její proměnné s názvy a hodnotami určené příkazy [nastav] a [změň].

Atributy byly dodány i samotnému projektu. Uchovává třeba uživatelovo poslední odpověď na položenou otázku blokem [zeptej se], stopky, tempo hudby; a také všemi postavami sdílené proměnné. Příkazy [vynuluj stopky], [nastav tempo], a dalšími; hodnoty těchto atributů ovlivňujeme. Přístup k nim mají všechny postavy.

U postavy existují i takové atributy, jenž příkazy nenastavíme. Jedná se o jméno, jež volíme jedině ručním zápisem do textového pole v horní části prostředí, dále nastavení způsobu otáčení[1]); a nakonec třeba povolení přetahování postavy myší po scéně. Mezi programově nenastavitelné atributy projektu patří např. zvolený turbo mód.[2])

- **Omezený a kompletní přístup k atributům**

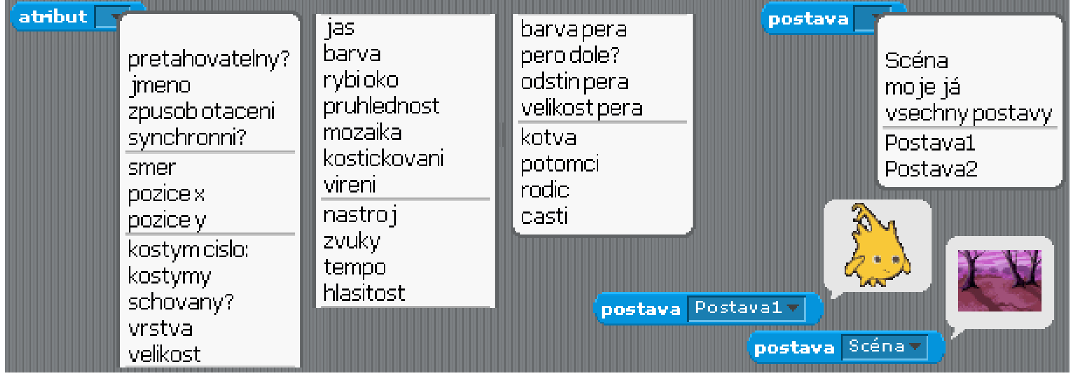
Ačkoliv už víme, že atributy nastavujeme použitím primitivních příkazů, stále nevíme, jak k jejich hodnotám přistupujeme. Na to máme v prostředích různé primitivní funkce. Třeba souřadnici y získáme funkcí (souřadnice y), velikost funkcí (velikost), hlasitost přes (hlasitost), hodnotu proměnné skrze (název proměnné), směr za pomocí (směr); a tak dále. Některými funkcemi jako (odpověď), (stopky), (tempo) a (název proměnné); pak přistupujeme k atributům projektu.

• *Poznámka: Všechny primitivní funkce a podmínky k atributům nepřistupují. Např.* <je větší> *či* (odečti) *atributy vůbec nepotřebují, jen provedou operaci a vrátí výsledek.*

Obě verze Scratch uživatelům umožňují pouze omezený přístup ke všem atributům. Například nelze zjistit celkový počet postavě nahraných kostýmů a zvuků (včetně jejich názvů a pořadí), ani jméno postavy, barvu a odstín pera, zdali je v popředí; a podobně. Jinými slovy přestože postava takové atributy eviduje, prostředí nenabízí způsob, jak k jejich hodnotám přistoupit. Většina programátorů je tak nucena používat pomocné proměnné, do kterých si ukládají informace o tom, že např. zvedli pero postavě.

Kompletní přístup ke všem atributům postav nabízí prostředí BYOB funkcí (atribut), jejichž odhalením pootevírá další možnosti v programování. Dostupné atributy jsou zobrazeny na obr. 3.1.12 a po zvolení jedné z možností funkce vrátí jejich hodnotu.

• *Poznámka: Mezi možnostmi funkce* (atribut) *– kromě toho, že jsou uvedeny bez diakritiky – jsou některé pro uživatele Scratch neznámé. Byly postavám přidány prostředím BYOB. Kvůli úspoře místa obr. 3.1.12 ukazuje i funkci* (postava)*, o níž více později.*



Obrázek 3.1.12 Možnosti funkcí (atribut) a (postava) v BYOB

[1]) Vyjma Scratch 2, který zavedl blok [nastav způsob otáčení] k programové změně způsobu otáčení.
[2]) Což neplatí o prostředí Snap mající k tomu určené primitivy [nastav turbo mód] a <turbo mód>.

Všimněme si, že funkce (atribut) v BYOB nabízí pouze atributy postavy a nikoliv atributy projektu. To se ale vyřeší s příchodem nové verze Snap 4.1, která kromě funkce (atribut) zavede i funkci (atribut projektu) odhalující všechny atributy projektu. Prostředí Snap tak na rozdíl od současnosti nabídne kompletní přístup k atributům.

- **Nastavení skrytých atributů**

Už víme, že k většině atributů máme přístup přes k tomu určené funkce a také příkazy, kterýmiž můžeme hodnoty těchto atributů ovlivnit. Funkce vracející hodnotu atributu mají v paletě zaškrtávací pole k zobrazení sledovače. Ke skrytým atributům – tedy těm, k získání jejichž hodnoty neexistuje přímá primitivní funkce – přistupujeme přes funkci (atribut) (či v budoucnu ve Snap i (atribut projektu)), ale už nemáme k dispozici příkaz, kterým bychom tyto skryté atributy mohli nastavit. V BYOB lze sice nastavit hodnotu skrytého atributu postavy v kombinaci s funkcí (atribut) – a to ovládáním libovolné postavy speciálním způsobem (viz „Dálkové ovládání postavy jinou postavou“ uvedené níže) –, ale takové řešení je příliš komplikované a nepraktické.

Proto se v prostředí Snap plánuje zavést blok [nastav atribut], který bude fungovat podobně asi jako příkaz [nastav] k nastavení hodnoty nějaké proměnné. Bude tedy buď možné vybrat z rozbalovací nabídky konkrétní atribut k nastavení, anebo nějak vývojáři Snap skloubí tento příkaz s funkcemi (atribut) a (atribut projektu), kterými bychom mohli konkrétní atribut k nastavení přes [nastav atribut] určovat.

- **Zobrazení a skrývání sledovačů atributů**

Dalším otazníkem je zobrazování atributů postavy. Sice si můžeme vytvořit proměnnou (např. (hodnota_atributu)), do něhož si uložíme vždy hodnotu vybraného atributu funkcemi (atribut) či (atribut projektu), a jejíž sledovač si zobrazíme na scéně; avšak nejedná se o příliš důmyslné řešení, neboť můžeme vyžadovat zobrazení různého počtu atributů v odlišných chvílích.

Proto se zřejmě v další verzi Snap objeví příkazy [ukaž atribut] a k němu opačný [schovej atribut] (analogicky k [ukaž proměnnou] a [schovej proměnnou]), kterými bychom konkrétní atribut postavy či projektu mohli zobrazit na scéně ve vlastním sledovači. U postavy s názvem „Postava1“ by zobrazení např. atributu *mozaika* zobrazil sledovač s nápisem „Postava1 mozaika“ a skutečnou hodnotou tohoto atributu.

- **O metodách objektu – bloků postavy**

Ačkoliv to pro pochopení následujících částí této podsekce není vyžadované, popíšeme si ještě *metody* objektů. V úvodu o OOP jsme si řekli, že *metoda* umožňuje nějakému objektu se projevit – tedy něco vykonat, vypočítat, nebo nějak se chovat. Aniž bychom si to uvědomili, v předchozích částech popisující atributy jsme několik metod zmínili.

Každý primitivní či vlastní blok, jež postava vlastní, je její metodou umožňujíc jí se nějak projevit či pracovat s jejími atributy. Je tedy lhostejno, mluvíme-li o příkazech, funkcích, či podmínkách, blocích událostí či o konečných blocích. I přesto se nevyhneme určitému dělení na: *přístupové* či *ostatní*, *soukromé* či *sdílené*, a *postavy* či *projektu*.

Přístupovými metodami přistupujeme k hodnotám atributů nebo je jimi nastavujeme. Např. funkcí (směr) přistoupíme k hodnotě směru postavy, kterým je natočena. Funkcí (název_proměnné) získáme hodnotu nějaké proměnné postavy. Příkazem [zamiř směrem] či [nastav] pak hodnoty zmíněných atributů ovlivňujeme – přistupujeme k nim, abychom je nastavili. Mezi *přístupové* metody projektu patří třeba (odpověď) vracející hodnotu atributu (uživatelovo odpovědi) a [zeptej se] k jeho nastavení.

Ostatní metody hodnoty atributů nemění, avšak mohou (a též nemusí) je využívat pro své účely. Atributy získají samy přes *přístupové* metody. Jsou jimi třeba (vynásob), <je menší>, (spoj), <dotýká se>, [čekej], [rozešli všem], [zastav], (délka), <obsahuje>. Funkce (vynásob) k žádnému atributu nepřistupuje – je provede součin. Příkaz [čekej] také ne – jen pozastaví vykonávání scénáře postavy na určenou dobu. Ale <dotýká se> už potřebuje atributy představující pozici a velikost kostýmu u obou porovnávaných postav, aby mohlo dojít k ověření jejich doteku.

Závěrem zbývá dodat, že *soukromé* metody jsou ty, které vlastní jen jedna postava. Může se jedna například o vlastní bloky. Kdežto *sdílené* metody jsou například vlastní bloky dostupné všem postavám. Metody *postavy* jsou bloky pracující s postavou (např. `[zahraj tón]`) a metody *projektu* jsou bloky pracující s projektem (`[nastav tempo]`).

- **Přístup k referencím na postavy**

Samotný projekt si také pamatuje, jaké postavy v něm existují a proto si na ně uchovává reference. Prostředí BYOB tyto reference odtajňuje přidáním funkce `(postava)`, skrze níž máme přístup k scéně či libovolné postavě v projektu (viz obrázek 3.1.12 vpravo). Reference na postavy použijeme při *úplném klonování* postav anebo jejich dálkovém ovládání, kdy jedna postava může ovládat postavou jinou včetně scény (popsáno níže).

Funkce `(postava)` disponuje rozbalovací nabídkou s několika možnostmi. Možnost `{všechny postavy}` vrací seznam s referencemi na všechny postavy projektu – vyjma scény, `{moje já}` vrací referenci na postavu, v níž se `(postava)` provedla, a `{scéna}` vrací referenci na scénu. Nakonec zmiňme, že pokud nevybereme žádnou možnost, bude vrácena „nic“ reprezentující konstanta `{nil}`.[1]) Snap přidá `(postava)` až ve verzi 4.1.

- **Dočasné klony postav**

Scratch 2, a po jeho vzoru i Snap, přichází s tzv. *dočasnými klony* postavy. V souvislosti s nimi byly do obou prostředí dodány nové bloky: příkaz `[klonuj]`, konečný blok `[zruš tento klon]`, a blok události `[když startuji jako klon]`.

Dočasný klon přebírá aktuální stav postavy, jejíž je „otiskem“, což zahrnuje pozici na scéně, grafické efekty, zvolený kostým a tak dále. Pamatují si i její proměnné, jejichž hodnotu si v rámci své existence mohou nezávisle na originálu změnit. Jedná se o kopii přebírající hodnoty atributů původní postavy. Klonování provádíme příkazem `[klonuj]`, jehož rozbalovací nabídka nabízí k výběru konkrétní postavy či možnost `{moje já}`. I scéna má přístup k `[klonuj]` a tak může přikázat klonovat libovolnou postavu. Sama sebe však naklonovat nemůže, tudíž v možnostech `[klonuj]` `{moje já}` chybí.

Vytvořením klonu dojde ke spuštění události `[když startuji jako klon]`, kterou je možné jej ovládat. Klony slyší, stejně jako jejich originál, i na všechny události, takže reagujeme-li kupříkladu na kliknutí myši, zachytí to i klon – klikne-li hráč na něj.

Dočasné klony ve Scratch 2 zmizí ve chvíli, kdy uživatel prostředí klikne na červené tlačítko symbolizující zastavení, či vyhodnocením příkazu `[zruš tento klon]`, anebo dojde-li k restartu projektu kliknutím na zelený praporek symbolizující (re)start. Zde se Snap od Scratch 2 liší. Kliknutím na zelený praporek se dočasné klony neodstraní, neboť ve Snap něco jako restart projektu neexistuje (viz podsekce 3.1.13).

- **Dočasné klony mohou vytvářet klony sebe samých**

Jestliže ve scénáři bloku události `[po kliknutí na start]` figuruje příkaz `[klonuj]` s možností `{moje já}`, pak každý klon tento scénář provede a naklonuje také sám sebe (tzn., že dočasné klony vytvoří své dočasné klony). V projektu se tak mnohonásobně rozmnoží a dojde k radikálnímu zpomalení[2]) prostředí.

Jelikož je `[zruš tento klon]` konečným blokem, nelze na něj napojit další příkazy. A to je překážkou, neboť chceme-li nejprve zrušit klon a až poté zaslat zprávu originálu, musíme prostředí obelstít. Obr. 3.1.13 ukazuje, jak tento problém vyřešit.

Obrázek 3.1.13 Způsoby jak zamezit klonům klonovat sebe sama

O tom, jestli je toto chování u dočasných klonů správné se debatuje na `GitHub#419`.

[1]) Totéž za určitých podmínek platí i pro funkci `(atribut)` u možností vracející referenci.
[2]) V extrémních případech i absolutnímu pozastavení na několik sekund.

- **Úplné klonování postav**

V prostředích je možné udělat přesnou kopii postavy ručně přes rozhraní. Stačí kliknout pravým tlačítkem myši na miniaturu postavy pod scénou a zvolit příslušnou možnost. Nová postava se umístí vedle originálu na scéně a její atributy budou nabývat stejných hodnot, jakých nabývaly atributy originálu v době klonování. Naklonované postavě se změní pouze jméno – to proto, aby nedošlo ke kolizi jmen s již existujícími postavami.

V prostředí BYOB můžeme klonovat postavy ještě jiným způsobem – a to programově funkcí (klonuj)[1]). Od manuálního klonování skrze rozhraní se však tento způsob značně liší. Vytváří nejenom *úplný* klon, ale i určitý vztah mezi původní a naklonovanou postavou. Pojďme si konkrétněji nastínit rozdíly (výhody) tohoto klonování.

Funkce (klonuj) postavu zkopíruje ná **totožné** místo, zobrazí její miniaturu mezi ostatními postavami pod scénou; a vrátí na ni referenci. Dokážeme pomocí ní vytvořit klon postavy kdykoliv za běhu scénáře. Připomeňme opět, že dostane jiné jméno (např. „Postava4“). Oproti dočasném klonu je možné přistoupit k jeho scénářům, proměnným, blokům, kostýmům, a zvukům – tzn. editovat jej, neboť jde o plnohodnotnou postavu. Úplný klon také nezmizí, dokud ho z projektu neodstraníme. Pakliže bychom se pokusili naklonovat scénu, prostředí zahlásí chybu. Scéna sama sebe klonovat nemůže. Úplné klonování funkcí (klonuj) funguje na principu *prototypů*, o němž až na konci podsekce.

Uchování reference na *úplný* klon má svůj význam. S ní, jakožto s *potomkem* klonované postavy, dále komunikujeme zasíláním dotazů a příkazů k ověření či změně jejího stavu, což si později popíšeme v části o dálkové komunikaci a ovládání postavy. S nastíněným pojmem *potomek* se pojí i *předek* (či *rodič*), kterým je pro klonovanou postavu postava původní, čímž vzniká určitá *hierarchie* postav (konkrétněji *dědičnost*), kterou si vzápětí představíme. Ještě si popišme co postavy mezi sebou sdílí.

Úplný klon sdílí se svým předkem proměnné (jen *proměnné postavy*), bloky (pouze *pro jednu postavu*), kostýmy, zvuky, a scénáře (slyší i na stejné události). Sdílené proměnné a bloky mají vybledlejší barvu. Jakmile některou ze zmíněných položek původní postavě upravíme – ať už ručně (třeba vzhled kostýmu) anebo programově (kupříkladu hodnotu proměnné) –, dozví se to i klon a provede identickou úpravu.

Můžeme pozorovat částečnou podobnost ve funkcionalitě s dočasnými klony (přestože úplné klonování vzniklo přes dočasným klonováním), které jakmile změnili hodnotu proměnné, již dále ji nesdíleli se svým předkem. U úplných klonů změnou proměnné tuto vazbu s předkem také narušíme. Jenže zatímco u dočasných klonů vazbu neobnovíme, u klonů úplných můžeme příkazem [smaž] odstranit „vlastní verzi“ upravené proměnnou a „napojit“ se na proměnnou původní patřící originální postavě. Smazaný blok či kostým však klonu zpětně neobnovíme.

Příkaz [smaž] má všestranné užití. Kromě obnovení vazby mezi proměnnými můžeme odstranit další vztahy (např. že postava už dále nebude potomkem postavy původní) přes funkci (atribut). Také můžeme smazat plnohodnotnou postavu, dodáním [smaž] referenci na ni přes funkci (postava), která pak úplně zmizí z projektu.

V prostředí Snap zatím funkce (klonuj) chybí, a tak v něm *úplné* klony nevytvoříme. V další verzi zaměřující se převážně na doplnění OOP principů však tato funkce (klonuj) nahradí příkaz [klonuj] a dočasné klony z Snap nadobro zmizí (viz grafická nápověda ke [klonuj] v prostředí). Příkaz [zruš tento klon] zřejmě nahradí příkaz [smaž], jež přijde též s další verzí Snap, a kterým můžeme odstranit postavu předáním reference na ni. Zdali blok události [když startuji jako klon] zůstane není jisté.

Příkaz [smaž] bude v Snap 4.1 sloužit asi jen k mazání postav. Možná také změní svou kategorii z proměnné na vnímání. Název tohoto bloku je totiž velmi zmatečný, neboť chceme-li například obnovit sdílení hodnoty proměnné se svým předkem v klonu, musíme „smazat“ jeho přepsanou verzi – což nedává moc smysl. Proto se plánuje zavést další příkazy [zděď atribut], [zděď proměnnou], a [zděď blok]; které namísto příkazu [smaž] použijeme. Oproti BYOB tak bude zřejmě možné zdědit smazaný blok.

[1]) Nepleťme si příkaz [klonuj] s funkcí (klonuj). První blok vytvoří klon dočasný a druhý klon úplný.

- **Hierarchie postav**

Jakmile naklonujeme postavu přes (klonuj), prostředí jí i postavě původní změní jeden atribut. Originálu přidá referenci na svůj klon (svého *potomka*) do atributu {potomci}, jež je seznamem obsahující reference na všechny vytvořené *potomky* této postavy funkcí (klonuj). Naklonované postavě naopak změní atribut {rodič} na referenci na původní postavu (*originál*), ze které byla naklonovaná postava vytvořena.

Návrh prostředí stanovuje, že každá postava může mít jen jednoho předka, zatímco v reálném světě savců je potřeba dvou k vytvoření potomka. Potomků však může mít postava kolik chce. Tímto vzniká jakýsi strom dědičnosti (rodokmen).[1])

Jestliže jsme postavu vytvořili klasickým způsobem neprogramově – tj. ručně přes rozhraní prostředí – pak (atribut) s {rodič} vrátí speciální konstantu {nil}[2]), která reprezentuje *prázdnou referenci*. Jakmile nějaký atribut nabývá hodnoty {nil}, znamená to, že očekává referenci na postavu, ale žádnou aktuálně nemá k dispozici.

Hierarchie je především o vztazích. Co si pamatuje a umí předek platí i pro jeho potomky, kteří mohou něco z toho změnit či rozšířit a předat to dále svým potomkům. V rodokmenu se lze pohybovat různými směry. Doprogramováním si vlastních scénářů můžeme zjistit například „prvního z rodu“ nebo jestli je nějaká postava v příbuzenském vztahu s jinou postavou. Správný rozvržením při vývoji projektu jsme schopni simulovat smysluplnou a s realitou srovnatelnou hierarchii postav. Podívejme se na obrázek 3.1.14 s vysvětlujícím popisem níže, který ukazuje popisované atributy na třech postavách.

Obrázek 3.1.14 Zobrazení atributů *rodič* a *potomci* u originálu a klonů

V novém projektu prostředí BYOB byla základní postava přejmenována na „Originál“. Poté se dvakrát zavolala funkce (klonuj), která vytvořila dvě identické postavy (úplné klony) s odlišnými názvy. Pro lepší orientaci byly přejmenovány na „Klon1“ a „Klon2“. Také jim byl přidán barevný efekt, proto je první klon zelený a druhý modrý.

Původní postava „Originál“ nebyla vytvořena funkcí (klonuj) a tak funkce (atribut) s možností {rodič} vrací {nil}, což znamená „nemám žádného předka“. Zavoláme-li funkci (atribut) s možností {potomci}, vrátí se seznam s referencemi na klonované postavy, jež prostředí zobrazí graficky s aktuálně oblečenými kostýmy.

U úplných klonů – postav vytvořených funkcí (klonuj) – nastávají dvě malé změny. Atribut *rodič* byl při klonování poupraven a nabývá reference na postavu „Originál“, čehož si lze všimnout znázorněnou postavou v kostýmu žluté barvy. Funkce (atribut) s možností {potomci} vrací prázdný seznam – to proto, že postavy „Klon1“ a „Klon2“ žádné své úplné klony nevytvořily. Kdybychom v nich vykonali funkci (klonuj), pak by se seznam s jejich *potomky* rozšířil o další referenci na klonovanou postavu – klon se tedy může také naklonovat. Nezapomeňme si všimnout toho rozdílu, že nemá-li postava žádné potomky, pak vrací prázdný seznam namísto konstanty {nil}.

[1]) Jde o klasickou stromovou strukturu s n potomky pro každý uzel.
[2]) Tuto konstantu známe např. z jazyka Pascal. V některých jiných jazycích se místo `nil` používá `null`.

• Nepřímá komunikace – přikazování a dotazování se postavy jinou postavou

Dosud postava komunikovala s ostatními postavami dvěma způsoby a to: 1) rozesíláním zpráv přes příkazy [rozešli všem] a [rozešli všem a čekej], na něž jsme reagovali jejich odchytnutím blokem události [po přijetí zprávy]; 2) funkcí (z), která dokáže dle zvolené možnosti u postavy a scény vrátit hodnoty různých jejich atributů. Obecně se jedná o komunikaci přikazováním či dotazováním se.

Tak či onak – není možné nějaké postavě či scéně přikázat provést libovolný scénář bez jejího vědomí. Ona sama určuje, na jakou zprávu a jakým způsobem zareaguje. Taktéž navzdory existenci funkce (atribut) v BYOB, odhalující většinu skrytých atributů postavy, nedokážeme zjistit hodnoty některých z nich, jelikož si na rozdíl od funkce (z) nemůžeme vybrat, na které postavě se provede – vždy se provádí na té aktivní.

Nicméně vše lze vyřešit *nepřímou komunikací* – jakýmsi dálkovým ovládáním postavy jinou či stejnou postavou, kterému se ovládaná postava nemůže bránit. Toho lze docílí v prostředích BYOB a Snap, které tak narušují právo každé postavy „svobodně" rozhodovat o svém konání a prozrazování soukromých informací.

Nepřímé ovládání se provádí v kombinaci s funkci (z), bloky [spusť] a (zavolej), a zaobalenými procedurami. Ačkoliv jsme si většinu zmíněného pořádně nepředstavili (to až v poslední kapitole), stačí následovat vzor s příklady uvedenými na obrázku 3.1.15 k docílení kýženého výsledku.

Obrázek 3.1.15 Způsob jak v Snap nepřímo komunikovat s postavami

Vykonáním bloků z předchozího obrázku by postava řekla „Ahoj!", dále prozradila svoji pozici x, a nakonec by sdělila, zdali se dotýká okraje scény. Použít funkci (souřadnice x) bylo v podstatě zbytečně, neboť x-ovou souřadnici získáme i výběrem možnosti {souřadnice x} v první rozbalovací nabídce funkce. Namísto (souřadnice x) bychom mohli ale použít například (atribut), kterou se postavy zeptáme na ty atributy, jež jsou skryté a k nimž nemáme nějakou primitivní funkcí přístup. Pokud bychom se nechtěli odkazovat na jméno postavy, je možné do druhé rozbalovací nabídky funkce (z) vložit funkci (postava), vracející referenci na konkrétní postavu.

Postava může takto nepřímo komunikovat i sama se sebou. Dokonce je možné komunikovat i se scénou, jen si musíme dát pozor na užití bloků, kterým scéna nerozumí. Mezi ně se řadí bloky určené pohybu z pohyb a některé další z ostatních kategorií.

Vzhledem k tomu, že není jasné jak se ve Snap vyvine nový koncept práce s atributy postav, úplnými klony, dědičností a tedy i ovládáním postav jinou postavou, nelze zajistit, že tento způsob bude v budoucnu nadále funkční.

• Vnořování postav

Atributy *kotva* a *části* souvisejí s tzv. *vnořováním postav*, kdy nějakou postavu *ukotvíme* k postavě jiné, jíž se stane sou*částí*. Jde o totožný vztah jaký je mezi *rodičem* a *potomky*. *Kotvou* je tedy reference na postavu a *částmi* seznam s referencemi na ukotvené postavy.

Smyslem vnořování postav je vztah mezi jednou a více postavami, které podléhají jakékoliv pohybové akci postavy představující kotvu. Vykonáním [posuň se] na kotvě se posunou i všechny její části, provedením [otoč se vlevo] se také otočí kromě kotvy všechny části, a analogicky stejně při vykonání každého příkazu z kategorie pohyb.

Jakmile se postava ukotví k (vnoří do) jiné postavě, prostředí graficky jejich vztah znázorní na miniatuře vnořené postavy uvedené pod scénou. V levém horním rohu obsahuje zmenšenou postavu, ke které je ukotvena, a v pravém tlačítko k nastavení způsobu otáčení vnořené postavy okolo své kotvy.

Funkcionalitu využijeme například při sestavování těla z částí (rukou, noh, trupu, a hlavy), které se musejí pohybovat pospolu, avšak každá se potřebuje animovat zvlášť. Dalším příkladem může být planetka, okolo které obíhají měsíce. Do planetky se vnoří (ukotví se k ní) měsíce, jenž budou neustále rotující planetku obíhat. Když se bude planetka pohybovat neznámo kam do nekonečně expandujícího vesmíru, její měsíce se posunou spolu s ní. V těchto příkladech by vztah *rodič* a *potomci* nefungoval, proto jej můžeme nahradit vztahem *kotva* a *části*, z něhož těžíme i propojením pohybu postav.

Vnořovat postavy dokáží jenom prostředí BYOB a Snap. Zatímco v Snap lze vnořit postavu do jiné postavy jen manuálně kvůli nedostatku objektově orientovaných bloků, v BYOB lze vnoření provést vnoření i programově využitím dálkového ovládání postavy. Konkrétní návody a ukázky je možné nalézt v manuálech jednotlivých prostředí.

- **Tvorba vlastních objektů přes třídy a z prototypů**

BYOB a Snap umožňují naprogramovat *třídy*, které jsou jakousi továrnou na objekty, skrze něž se pak vytvářejí. Definují objektům atributy, chování (metody), a nastavují je do počátečního stavu. Komunikace s nimi probíhá zasíláním zpráv v podobě dotazů a příkazů ověřujíc či měníc jejich stav, tedy stejně jako je tomu u postav v prostředích.

Třída se vytváří vlastní *konstruující* funkcí vracející nový – jí samotnou zkonstruovaný – objekt, který je reprezentován *funkčním uzávěrem* (zaobalenou procedurou). Atributy jsou vytvářeny přes proměnné scénáře a chování je určováno dalšími lambda výrazy, které jsou společně s atributy součástí – funkcí vráceného – funkčního uzávěru.

Zmíněná prostředí také umožňují tvorbu objektů založenou na *prototypech* – tedy kopií nějakého objektu, který poslouží jako předloha. Tento koncept se od tříd liší tím, že objekty přebírají aktuální množinu atributů a metod prototypu včetně jeho stavu s možnostmi dalšího rozšíření, zatímco u tříd je počet atributů a metod objektům pevně definován. Objekty založené na prototypech tak lze za běhu scénáře něčemu přiučit.

K zprovoznění tohoto konceptu je potřeba naprogramovat obecný (prázdný) objekt, který si nic nepamatuje a ani nic neumí, ale dokáže přidat a uchovat nové atributy a metody. K nim taktéž poskytne přístup. Z tohoto objektu se poté odvodí jiný objekt, což znamená, že se onen původní objekt stane prototypem objektu vytvářeného.

Nový objekt můžeme rozšířit, změnit jeho stav a v případě potřeby dále naklonovat. V tu chvíli se on sám stává prototypem nově vznikajícímu klonu. Toto je princip, který se neustále opakuje – 1) vytvoř objekt z prototypu, 2) přiuč jej, 3) naklonuj, 4) opakuj. Dobrým příkladem odvozování z prototypů je funkce `(klonuj)` vytvářející novou postavu jakožto klon původní, který jest jejím prototypem.

Skvělým příkladem prototypů je tvorba úplných klonů funkcí `(klonuj)`. Originální postava je prototypem, která slouží jako předloha svému klonu (další postavě). Úplný klon může přidat další proměnné, vlastní bloky, kostýmy, apod.; případně zrušit vazbu na nějakou sdílenou položku se svým předkem, což se ve výsledku promítne v momentě dalšího klonování, neboť ona klonovaná postava poslouží jako další předloha nově vznikajícímu klonu. Uvědomme si také, že přes atribut *rodič* lze přistupovat k prototypům (*předkům*), ze kterých jednotlivé postavy vycházejí (na kterých jsou založeny).

OO koncepty	Scratch 1	BYOB	Scratch 2	Snap
Přístup k atributům	omezený	úplný	omezený	úplný v 4.1
Reference na postavu	NE	ANO	NE	až v 4.1
Dočasné klony	NE	NE	ANO	ANO
Úplné klony a dědičnost	NE	ANO	NE	až v 4.1
Nepřímé ovládání postavy	NE	ANO	NE	ANO
Tvorba tříd a prototypů	--	ANO	--	ANO
Vnořování postav	NE	ANO	NE	ANO

Tabulka 3.1.5 Podpora objektově orientovaných konceptů

3.1.6 Programování rekurze

Rekurze je v informatice metoda, kdy procedura opakovaně volá sebe sama v určitém svém úseku. Součástí rekurzivní procedury je námi určená ukončující podmínka, která zastaví opakované vnořování se. Rekurzí nahrazujeme složitější řešení některého z algoritmů programovaných iteračním způsobem (využívající cykly), jehož převodem do rekurzivního algoritmu zjednodušíme jeho konečnou implementaci a čitelnost.

Jelikož nebylo možné ve Scratch 1 vytvářet vlastní bloky, nemohli jsme si vyzkoušet naprogramovat rekurzivní scénáře. To však neznamená, že algoritmy primitivních bloků v tomto prostředí nejsou na nižší úrovni jazyka rekurzivními – ba naopak. Překvapivě však existuje ve Scratch 1 způsob jak rekurzi částečně simulovat rozesíláním zpráv a to kombinací bloků `[po přijetí zprávy]` a `[rozešli všem]`.[43]

Protože Scratch 2 podporuje tvorbu vlastních příkazů, lze v něm konečně naprogramovat rekurzivní scénář. U BYOB a Snap lze naprogramovat i rekurzivní funkci, popř. podmínku, které se navíc od rekurzivních příkazů liší v rekurzivním volání. Zmíněná prostředí podporují rekurzi přímou i nepřímou, lineární i stromovou; a dokonce i rekurzi koncovou, která slouží k optimalizaci paměťově náročnějších rekurzivních algoritmů.

3.1.7 Kostýmy, grafické efekty, scéna, hudba, kreslení

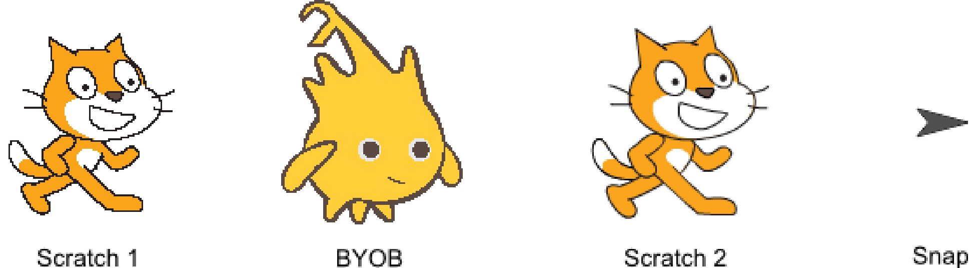
• Základní kostým postavy – maskoti prostředí

Při vytváření postavy postupujeme v prostředích tak, že před jejich uvedením na scénu jim přiřazujeme nějaký kostým. Buď jej namalujeme v příslušném editoru a nebo naimportujeme z počítače či knihovny kostýmů. Bez kostýmu postavu nevytvoříme. Jenže ne vždy je pro nás kostým postavy důležitý, převážně pak u projektů zaměřujících se na kreslení želví grafiky.

Proto prostředí Snap umožňuje vytvořit postavu s automaticky přiděleným kostýmem. Každá takto vytvořená postava získá *základní* kostým ve tvaru šipky a názvem „*Želva*“. Při jejím vytvoření si ještě volí náhodnou barvu pera, což se odrazí ve výplni želvího kostýmu. V kartě „*Kostýmy*“ je možné kliknutím pravého tlačítka na kostým želvy určit, zdali bude kreslit hrotem nebo středem.

Kostým želvy se automaticky oblékne za dvou okolností – a to pokud postavě smažeme její poslední – námi dodaný – kostým, nebo jestliže se pokusíme obléknout kostým neexistující. Při oblečeném kostýmu želvy vrací funkce `(kostým číslo)` hodnotu `{0}`. Je-li zvolen kostým jiný než želví, pak přes `[další kostým]` nikdy nedojde k jeho oblečení. Máme-li oblečen kostým želvy, musíme manuálně či blokem `[oblékni kostým]` obléknout kostým jiný, abychom se přes `[další kostým]` mezi ostatními posouvali.

Na obrázku 3.1.16 si prohlédněme základní kostýmy postav, které jsou prostředím automaticky dodány, napříč prostředími v plné velikosti. Jsou *maskoty* prostředí.



Obrázek 3.1.16 Maskoti prostředí ve stoprocentní velikosti

• Změna pozadí scény postavami

Doposud to byla pouze scéna, která si měnila příkazy své pozadí. Prostředí Scratch 2 novým příkazem `[změň pozadí]` dává možnost ostatním postavám přikázat scéně změnu jejího pozadí. V možnostech tohoto bloku jsou i `{předchozí pozadí}` a `{následující pozadí}`, jimiž můžeme pohodlně přepínat mezi pozadími scény dle jejich pořadí, aniž bychom se museli odkazovat na přesné názvy obrázků.

- **Grafické efekty**

Obě verze Scratch a také BYOB nabízejí sedm různorodých grafických efektů. Prostředí Snap nabízí pouze dva k ostatním prostředím identické efekty a zbylé čtyři jsou odlišné. Ještě však projdou další úpravou, neb jsou nekompletní a údajně chybové.[51]

- **Skrývání obsahu scény**

Snap dokáže zakrýt obsah scény bloky `[schovej se]` a `[ukaž se]`, které přidává scéně do kategorie `vzhled`. Po zakrytí se scéna zahalí tmavě šedým pozadím a všechny postavy, nakreslené čáry pery, otisky postav, a její zvolené pozadí; zmizí. Tuto funkcionalitu můžeme použít u projektů, které obsahují jen nějaké ukázkové bloky, nemají příběh, a nejsou tak vlastně ani hrami. Skrytou scénou uživatelům dáme jasně najevo, že se mají podívat do útrob projektu namísto sledování scény.

- **Změna velikosti scény a přístup k jejím rozměrům i kostýmu postav**

Překážkou pro projekty vyžadující vetší plochu jsou rozměry scény, jež jsou nastaveny na 480 kroků šířky a 360 kroků výšky. Prostředí Snap umí rozměry scény pozměnit. Minimem je rozměr 480 na 180 a maximum překvapivě není stanoveno. Změna rozměrů se provádí ručně přes nastavení a se změnou velikosti programově zřejmě nepočítejme.

Aby bylo možné zjistit střed scény ve vertikálním, horizontálním, či obou směrech, potřebujeme se k jejím rozměrům dostat. Vývojáři zřejmě dodají do `(atribut)` možnosti `{šířka/výska}`, jež vrátí nejenom rozměry scény, ale i rozměry kostýmu postavy.

- **Útěk postavy za okraje scény**

V obou verzích Scratch se postava nemůže dostat za určené okraje scény. Dotek s hranicí scény můžeme ověřovat akorát podmínkou `<dotýká se>` s možností `{okraj}`, případně okamžitě zareagovat příkazem `[když okraj, odraž se]`.

U prostředí Snap se postavy mohou dostat vždy za okraj scény, klidně i tisíce kroků libovolným směrem. Bohužel tomu, alespoň prozatím, není možné zabránit. Můžeme si akorát pomoci smyčkou na pozadí ověřující dotek postavy s okrajem scény společně s reakcí. Při oblečeném kostýmu želvy se detekce s okrajem liší dle zvoleného způsobu kreslení želvy (*středem* či *hrotem*).

Prostředí BYOB je přesně na pomezí Scratch a Snap. V novém projektu je automaticky zvolené chování identické k Scratch, ale přes možnost „*Editovat*“ → „*Povolit postavám utíkat za scénu*“ můžeme útěk za hranice scény postavám povolit.

Útěk postav za okraje scény však není něčím negativním – naopak. Hodí se třeba u her z pohledu ptačí perspektivy, kdy panáček je umístěn stále na středu scény a pouze animuje svůj pohyb střídáním kostýmu, kdežto ostatní objekty se pohybují namísto něj na fádním (jednobarevném) pozadí. Vznikne tak dojem, že panáček chodí na různá místa po rozsáhlé „mapě“, neboť se na scéně postupně objevují (přicházejí a odcházejí) jednotlivé objekty hry. Přesto však dochází pouze k jejich posunu za hranice scény. Útěk za okraje užijeme i tehdy, posouváme-li skupinu postav seřazených v řadě za sebou.

- **Souřadnice myši**

Obě verze Scratch a BYOB zobrazují souřadnice myši x a y na pravé straně pod scénou. Uživatelé se tak mohou jednoduše orientovat. Jediné Snap je na tomto místě nezobrazuje. Namísto toho přidává do palety bloků k `(souřadnice myši x)` a `(souřadnice myši y)` zaškrtávače zobrazující sledovače těchto souřadnic.

Musíme zmínit ještě jednu odlišnost spojenou se souřadnicemi myši – a to pouze u prostředí Scratch 2. Střed scény je ve všech prostředích určen hodnotami `{0, 0}`. Proto najedeme-li myší na střed scény, pak `(souřadnice myši x)` i `(souřadnice myši y)` nabudou také hodnot `{0, 0}`. Jenže kromě prostředí Scratch 2 se souřadnice myši určují dle pozice kurzoru myši v aplikaci prostředí spuštěné na počítači či v prohlížeči. Funkce `(souřadnice myši x)` tak klidně může vrátit hodnoty ve stovkách či tisících (záleží na rozlišení obrazovky). Prostředí Scratch 2 však hodnoty hlídá a – bez ohledu na pozici kurzoru za hranicemi scény – nikdy nepřekročí interval $\langle 480, 360 \rangle$.

• Hudební nástroje a bubny, hlasitost přehrávaných zvuků

Rozdíly mezi oběma verzemi Scratch jsou v nabídce hudebních nástrojů a druhů bubnů. Scratch 1 poskytuje 128 nástrojů a 81 bubnů, kdežto Scratch 2 pouhých 18 nástrojů a 21 bubínků. Nabídka BYOB je totožná ke Scratch 1.

Prostředí Snap nastavit jiné hudební nástroje ani bubnovat neumožňuje. [nastav nástroj] a [bubnuj] se v něm nevyskytují. Hraní not (či tónů) je však možné, akorát je zvolen pouze jeden – blíže neurčený – nástroj. V současnosti se na těchto funkcionalitách pracuje. Vývoj lze sledovat na GitHub#91 a na GitHub#489.

Z předchozího odstavce víme, že Snap v oblasti programování hudby celkem strádá. To bohužel umocňuje i současná absence bloků [nastav hlasitost], [změň hlasitost] a (hlasitost), z čehož můžeme vyvodit, že nelze nastavit hlasitost všech přehrávaných zvuků v tomto prostředí. Vše řeší návrh GitHub#544 od uživatele @gubolin.

• Kreslení želví grafiky

Při kreslení želví grafiky perem nastavujeme též jeho barvu a odstín číselnou hodnotou v prostředích shodnými příkazy [nastav barvu pera] a [nastav odstín pera]. Barvu lze ještě kromě číselné hodnoty nastavit kapátkem. Zde jediné Scratch 2 nezobrazí paletu barev k výběru barvy. Snap v ní zase neumožňuje nastavit stupeň šedi.

Barva pera má ve Scratch 1, Scratch 2, i BYOB rozsah $\langle 0, 200 \rangle$, přičemž dosažením dvoustovky se hodnota vrátí na nulu. Základní postavě nastavená barva je pestře modrá v hodnotě $133.\overline{3}$. Odstín se pohybuje ve stejném intervalu v základní hodnotě 50.

Interval barev ve Snap je $\langle 0, 100 \rangle$ s přednastavenou hodnotou 0. Dosažením stovky se hodnota vrátí na nulu. U odstínu je rozsah též $\langle 0, 100 \rangle$ se základní hodnotou 31.37 a při dosažení maxima se hodnota nemění. Stálé zvyšování odstínu tak nepřekročí stovku.

Též jsme zvyklí, že ihned po provedení příkazu [pero dolů] postava udělá značku (tečku) perem na scéně. Ve Snap tomu tak není. Toto chování je v některých chvílích nežádoucí, zvláště voláme-li příkazy přemisťující postavu před kreslením na odlišné místo (čímž by vznikla čára mezi původní a novou pozicí postavy). Chování změníme přidáním příkazu [posuň se] s argumentem {0} za příkaz [pero dolů].[52]

3.1.8 Videokamera, mikrofon, přepis řeči do a předčítání textu

• Přenášení obrazu z videokamery

Scratch 2 umí přenést na pozadí scény obraz z videokamery. Přidává k tomu nové bloky (video), [přepni video] a [nastav průhlednost videa].

Ve Snap se na této funkcionalitě pracuje. Uživatel @awangenh představil své řešení, které dle jeho slov oproti Scratch 2 trochu vylepšil. Toto řešení však v žádném případě není závazné a nelze odhadnout, jakým způsobem bude snímání obrazu z videokamery implementováno. Další informace o aktuálním stavu návrhu jsou na GitHub#634.

Návrh se drží konceptu ze Scratch 2 vyjma tří rozdílů:

- nelze – alespoň prozatím – nastavit průhlednost videa
- bloky ovládající video jsou pouze ve scéně, neboť jen ona přenáší obraz na pozadí
- video je také kromě vypnutí a zapnutí možné pozastavit

• Mikrofon a snímání hlasitosti

Dosud jsme užívali mikrofon k nahrání zvuků v kartě „*Zvuky*“ a zjištění hlasitosti funkcí (hlasitost), která se pohybuje v intervalu $\langle 0, 100 \rangle$. Scratch 1 a BYOB nabízí ještě blok <hlasitý>, jež vrací {<pravda>}, je-li (hlasitost) přibližně okolo pětadvaceti. Protože je blok <hlasitý> nadbytečným, byl v Scratch 2. Prostředí Snap neumí komunikovat s mikrofonem a tak nelze nahrát vlastní zvuk v kartě „*Zvuky*“, ani zjistit hlasitost mikrofonu – bloky (hlasitost) a <hlasitý> totiž nejsou dostupné.

Zmíněný návrh GitHub#634 uživatele @awangenh však komunikaci s mikrofonem též řeší a přidává bloky [zapni mikrofon] a [vypni mikrofon], kterými lze přenášet zvuk z mikrofonu přímo do prostředí. Je tak možné, že se časem chybějící bloky (hlasitost) a <hlasitý> doplní, stejně jako se možná najde další využití mikrofonu.

• **Přepis řeči do textu a předčítání textu (syntéza řeči)**

V dnešním světě počítačů už umíme rozeznat lidský hlas a přepsat jeho jednotlivá slova do textové podoby. Stejně tak existují programy, které dokáží uměle vytvořit lidskou řeč dle zadaného textu – tzv. *syntezátory řeči*. Uživatel `@bromagosa` představil návrh (též v rámci `GitHub#634`), který právě ony zmíněné schopnosti do Snap doplňuje.

Návrh zavádí dva bloky – `(rozpoznej řeč)` a `[převyprávěj]`. Při zavolání `(rozpoznej řeč)` funkce čeká dokud neřekneme nějaké slovo, které by rozpoznat. Dvě sekundy po posledním proneseném slovu se funkce vyhodnotí a vrátí přepsaný hlasový záznam do textové podoby. Blok sám rozeznává slova dle zvoleného jazyka v prostředí. Příkaz `[převyprávěj]` obsahuje textový parametr, jehož argumentem je text určený k přeříkání. I tento blok by měl přeříkávat text ve zvoleném jazyce prostředí, ale zdá se, že zvolený software zajišťující převod textu na zvuk nepodporuje češtinu. Proto si zatím přeříkáme slova jen v běžných jazycích, jakými jsou angličtina či němčina.

3.1.9 Aktuální datum a čas

• **Funkce** `(aktuální)`

Přistupovat k reálnému času a datu nebylo ve Scratch 1 a BYOB možné. Scratch 2 přichází s novou funkcí `(aktuální)`, která dle zvolené možnosti z `{rok/měsíc/datum/den týdne/hodina/minuta/sekunda}` umožní v uspokojující míře získat podrobnější informace o aktuálním času a datu. Zdůrazněme však, že funkce `(aktuální)` slouží pouze k získávání hodnot a tak nejsme schopni zjistit například počet dnů aktuálního měsíce anebo pracovat přímo s časem nebo datem ve smyslu různých přepočtů z jednotky na jednotku a podobně – to vše lze jedině doprogramováním vlastních algoritmů.

Tyto informace je možné v hojné míře využít. Už dokážeme kupříkladu přizpůsobit pozadí scény dle aktuálního času (den ověříme vraceným výsledkem funkce `(aktuální)` s možností `{hodina}`, jež musí být např. v intervalu $\langle 6, 18 \rangle$). Dáme-li si práci s převedením údajů kalendáře jmen do prostředí, pak zjištěním dne a měsíce možnostmi `{den}` a `{měsíc}` lze získat jméno aktuálního oslavence. A též je možné kromě hodnoty nového rekordu uchovat i datum a čas jeho dosažení, jež uložíme třeba do cloudové proměnné.

Snap drží krok a nabízí totožnou funkci `(aktuální)`, kterou ještě rozšiřuje o možnost `{čas v milisekundách}`. Ta představuje takzvaný „Unixový čas“ – tedy číselnou reprezentaci aktuálního data a času měřených od půlnoci 1. 1. 1970. Ve chvíli, kdy se psal tento odstavec, vrátila funkce `(aktuální)` s možností `{čas v milisekundách}` nicneříkající číslo `{1413677002252}`, které je ekvivalentní k neděli, 19. října 2014, tři minuty a dvaadvacet sekund po půlnoci.[1]) Lze hovořit o uložení bodu v čase.

V souvislosti s časem byla do prostředí Scratch 2 též přidána `(dnů od roku 2000)`. Na zjištění počtu dnů od roku 2000 ve Snap však žádnou takovou funkci nemáme.

• **Využití času v milisekundách**

Čas v milisekundách využijeme třeba při měření času. Přestože můžeme ve všech prostředích použít k měření stopky, vystavujeme se těmto rizikům a nedostatkům:

- stopky jsou sdílené všemi postavami a nelze měřit čas každé postavě zvlášť
- nelze je odstartovat, pozastavit ani úplně zastavit; pouze vynulovat
- hráč může při měření svého výkonu podvádět vynulováním stopek

Krom druhé výtky lze použitím času v milisekundách eliminovat zmíněné nedostatky. Pokud čas v milisekundách uložíme před zahájením měření a po jeho skončení zjistíme rozdíl uloženého času s časem aktuálním, pak dodatečným výpočtem získáme přesný počet uběhnutých sekund. Konkrétní příklad si ukážeme až v podsekci 4.4.1.

[1]) Vzhledem k poloze autora je ovšem nutné k času připočíst další dvě hodiny (GTM +2:00). Rozluštit čas v milisekundách lze například na stránkách `http://www.epochconverter.com/`

3.1.10 Turbo mód a provedení scénáře bez obnovy obrazovky

• Rychlostní mód prostředí

Některé, kupříkladu 3D vykreslující, projekty jsou natolik náročné, že vyhodnocování scénářů normální rychlostí k jejich plynulému běhu nestačí. Autoři takovýchto projektů často žádají různými způsoby uživatele o přepnutí prostředí do *turbo módu*, při němž se několikanásobně zrychlí vykonávání všech scénářů projektu.

Uživatel si ale žádosti o aktivování turbo módu nemusí všimnout a proto jsou autoři trvající na kontrole rychlostního módu nuceni používat ověřené detekční scénáře, jenž jsou uvedené například na Scratch wikipedii.[1])

Jiné projekty vyžadují běh naopak pouze v normálním módu. Se zapnutým turbo módem lze totiž v některých hrách podvádět a to konkrétně při měření času stopkami. Takový hráč je zvýhodněn, jelikož nejsou stopky rychlostním módem ovlivněny. Jenže použije-li autor již zmíněné detekční scénáře k nepřetržité kontrole módu, pak se vzdává možnosti stopky použít, neboť ony stopky již detekční scénáře využívají.

Prostředí Snap nabízí řešení přidáním dvou nových primitiv [nastav turbo mód] a <turbo mód>. Programátor tak nemusí žádat uživatele a rychlostní mód rovnou nastaví příkazem [nastav turbo mód]. Podmínkou <turbo mód> lze zase ověřit, zdali je při spuštění či v průběhu vykonávání projektu požadovaný mód aktivní. Tyto možnosti budou možná v budoucí verzi Snap 4.1 přesunuty do již zmíněného příkazu [nastav atribut] a funkce (atribut projektu).

• *Poznámka: Aby si hráč v průběhu hry rychlostní mód nepřepnul, je dobré mít na pozadí spuštění scénář, který neustále požadovaný mód ověřuje a případně jej přepne zpět.*

• Vykonávání scénáře bez obnovy obrazovky

Více najednou běžících scénářů si předává na určitý čas právo provádět se. To se děje takovou rychlostí, že jejich provádění nakonec působí synchronizovaně. Všechny scénáře jsou si v důležitosti rovny a proto mají stejnou *prioritu*. Některé scénáře se ale vyplatí upřednostnit a nechat je provádět delší dobu na úkor ostatních, takže zvládnou stejné množství práce za méně času. Takové scénáře se mohou týkat třeba vykreslování želví (převážně 3D) grafiky anebo provádění operací nad seznamy s velkými objemy dat.

Vyjma Scratch 2 lze všem prostředím určovat prioritu scénářům, avšak s drobnými rozdíly. Prostředí BYOB a Scratch 1 dokáží upřednostnit jen scénáře vlastních bloků. K tomu editor bloku v BYOB nabízí zaškrtávací pole s popiskem „atomic“ a Scratch 2 zaškrtávací pole s popiskem „Spustit bez obnovy obrazovky“. Prostředí Snap umožňuje upřednostnit libovolnou část scénáře. Používá k tomu nový blok [bez obnovy obrazovky], který očekává příkazy, jež budou upřednostněny. Jelikož se jedná o blok upřednostňující své vnitřní příkazy, může být užit kdekoliv – na začátku, ve středu, u konce scénáře, či uvnitř bloku. V editorů vlastního bloku Snap možnost *atomic* jako má jeho předchůdce BYOB nenabízí a proto chceme-li takový blok vykonat bez obnovy obrazovky, stačí jeho tělíčko (scénář) blokem [bez obnovy obrazovky] obklopit.

• *Poznámka: Uživatele může zaskočit fakt, že je blok* [bez obnovy obrazovky] *umístěn v kategorii* ovládání*, ačkoliv má barvu kategorie* ostatní*. Zřejmě z důvodu odlišení.*

Zajímavostí je také, že se Scratch tým původně inspiroval od prostředí Snap a chtěl zavést blok [bez obnovy obrazovky] do Scratch 2 v trochu odlišném názvu. Nakonec se ale vývojáři rozhodli pro řešení popsané v předchozím odstavci (jako má BYOB).[35]

3.1.11 Porovnání znaků a ověření původu hodnot

• Porovnávání velikosti znaků a jejich abecedního pořadí

Všechna prostředí se nestarají při porovnávání písmen o jejich velikost. Proto výsledkem podmínky <je rovno> s argumenty {A, a} bude {<pravda>}, neboť prostředím záleží pouze na významu porovnávaných písmen. Stejný výsledek bude i pro {AbC, aBc}.

[1]) http://wiki.scratch.mit.edu/wiki/Turbo_Mode_Detection

Porovnávání velikosti písmen nejsme v obou verzích Scratch schopni docílit, na rozdíl od Snap, které disponuje funkcí (unicode), jež vrací číselný kód každého písmene. V informatice mají velká a malá písmena zvlášť, číslice 0–9, interpunkční znaménka, znaky s diakritikou, a další symboly; vlastní číselný kód dle umístění v takzvané *kódové tabulce znaků*. Název již zmíněné funkce tedy vyjadřuje název její kódové tabulky, do které pro číselný kód prostředí nahlíží. Ukažme si, jak s ní porovnáme velikost písmen.

Tajemství rozlišení velikosti písmen tkví v převodu písmen na čísla, která se namísto znaků porovnají. Příkladem budiž porovnání velikosti prvního písmene abecedy. V unicode tabulce má velké písmeno A číselný kód 65 a jeho malá varianta 97. Protože si hodnoty 65 a 97 nejsou rovny, tak si nejsou rovny ani písmena jimi reprezentovaná.

Obrázek 3.1.17 Porovnávání velkých a malých písmen funkcí (unicode)

Opačnou funkcí k (unicode) je funkce (unicode jako znak), která namísto písmene očekává číselný kód z unicode tabulky, do které vzápětí nahlédne a vrátí číselnému kódu odpovídající znak. Že získáme i různé symboly dokazuje argument {8734} vracející ∞.

Ještě se vraťme k porovnávání písmen. Nyní však místo rovnosti řešme jejich abecední pořadí. Výsledek podmínky <je menší> s argumenty {C, Č} vrátí {<pravda>}. To je samozřejmě správně, protože písmenu Č v abecedě předchází písmeno C. Avšak při změně argumentů na {Č, D} se v prostředích mylně dozvíme, že je v české abecedě písmeno D před písmenem Č. Vysvětleme si tento – na první pohled – „nesmysl“.

Nahlédneme-li do unicode tabulky,[1]) zjistíme, že velkému Č je přiřazen číselný kód 268, kdežto velkému D „pouze“ 68. Proto při porovnání těchto čísel dojde k chybnému uspořádání v abecedním pořadí. Česká písmena s diakritikou jsou v tabulce umístěna až za písmeny anglické abecedy a to kvůli historickému vývoji kódování.[2]) Programátor tedy musí s touto překážkou počítat.

• *Poznámka: Použitím bloků* (unicode) *a* (unicode jako znak) *můžeme doprogramovat vlastní podmínku, která vyřeší abecední porovnání písmen české abecedy. Takovýto blok by se hodil i pro zavedení porovnávání českého písmene CH, které se v unicodové tabulce nevyskytuje, jelikož obsahuje pouze samostatné symboly. Podmínka by hleděla vždy o jeden znak dopředu, tedy zdali při nálezu písmena C nenásleduje i písmeno H.*

Prostředí BYOB používá jinou kódovou tabulku zvanou ASCII. Proto místo funkcí (unicode) a (unicode jako znak) nabízí jiné funkce (ASCII) a (ASCII jako znak). Tato kódová tabulka je historicky nejstarší a neobsahuje např. písmena české abecedy.

• Ověření totožného původu hodnot

Podmínkou <je totožný k> ověřujeme, jestli jsou si dvě věci naprosto rovny. Je možné prohlásit, že provádí test úplné rovnosti. Od podmínky <je rovno> se značně liší. Zatímco <je rovno> například u seznamů porovnává počet prvků a v případě shody i jejich hodnoty, podmínka <je totožný k> porovnává pouze reference na ně.

V budoucnu bude ještě provádět úplné porovnání textu a číselných oborů.[46] Takže výsledek <je rovno> s argumenty {AbC, aBc} bude {<pravda>}, zatímco použitím <je totožný k> {<nepravda>}. U argumentů {4, 4.0} podmínka <je totožný k> vrátí {<nepravda>}, zatímco blok <je rovno> vrátí {<pravda>}.

• *Poznámka: Nyní se možná čtenář zamyslel nad využitelností funkce* (unicode)*, jelikož bude v brzké době možné velikost písmen porovnat podmínkou* <je totožný k>*. Netřeba se obávat. Využití najde při posunu písmen v abecedě (například z písmene Z na A), nebo k odstranění diakritického znaménka nad písmenem (záměnou Ň za N); apod.*

[1]) Namísto nahlížení do unicodové tabulky ke zjištění číselných kódů můžeme použít funkci (unicode).
[2]) Tabulku číselných kódů těchto písmen nalezneme např. na http://cs.wikipedia.org/wiki/Unicode

3.1.12 Události a rozesílání zpráv

• Rozdíly v nabízených blocích událostí

Scratch 2 rozšiřuje počet nabízených bloků událostí o blok `[když větší než]` s možnostmi `{hlasitost/stopky/pohyb videa}`, které nás osvobozují od neustálé kontroly uvedených hodnot opakujícím se scénářem. Dalším a posledním přidaným blokem událostí jest `[po změně pozadí]`, kterým lze reagovat na změnu pozadí scény.

Jelikož Scratch 1 a BYOB neumožňují tvořit dočasné klony, nenabízejí tak ani blok události `[když startuji jako klon]` s nimi spojený. Mimo jiné je zajímavé, že tento blok prostředí Scratch 2 ponechává v kategorii `ovládání`. To zřejmě proto, že se klonu událost o jeho vytvoření odešle jenom jedenkrát.

• Rozšíření možností bloku `[po kliknutí na mě]` ve Snap

Blok události `[po kliknutí na mě]` obsluhuje pouze kliknutí počítačovou myši na postavu či scénu. Prostředí Snap však tento blok rozšiřuje o odchytávání dalších událostí, jež lze v grafických uživatelských knihovnách různých programovacích jazyků vyvolat. Autor práce blok přejmenoval na `[když na mě myší]`, neboť mu vývojáři dodali rozbalovací nabídku s možnostmi `{kliknuto/stisknuto/uvolněno/najeto/odejeto}`.

Se zvolenou možností `{kliknuto}` budeme obsluhovat stejnou událost jako doposud blokem `[po kliknutí na mě]`. Možnost `{stisknuto}` reaguje na stisk a držení tlačítka myši. Naopak událost `{uvolněno}` se spustí tehdy, když stisknuté tlačítko myši přímo na oné postavě uvolníme. Možnost `{najeto}` aktivuje `[když na mě myší]` dojde-li ke kontaktu postavy s myší. A `{odejeto}` je opakem k `{najeto}`, tudíž se tato událost vyvolá tehdy, když myš opustí hranice postavy.

U možnosti `{uvolněno}` se však pozastavme. Blok `[když na mě myší]` spustí jen tehdy, bylo-li postavou pohnuto. Nejprve musíme stisknutím levého tlačítka myši uchopit postavu, poté s ní pohnout alespoň o jeden krok, a nakonec pustit – až tehdy dojde k detekci této události. Pakliže postavu pouze chytíme a následně pustíme (bez pohybu myší), prostředí na tuto akci nezareaguje. Závěrem je, že: 1) nastavíme-li postavu jako nepřetahovatelnou (zaškrtávačem v místě kde postavě volíme jméno), pak tuto událost neodchytneme; 2) jelikož je scéna nepřetahovatelná, sama také na tuto událost neuslyší.

• Přijímání jakékoliv zprávy a rozesílání libovolného obsahu

Jediné Snap rozšiřuje koncept rozesílání zpráv. Přesněji řečeno kromě zpráv lze odeslat i určitý obsah, čímž mějme namysli hodnotu libovolného datového typu. Je tak možné rozeslat seznam s hodnotami, nebo zaobalenou proceduru, či např. kostým.

Toto vylepšení ale přináší dva otazníky: 1) jak reagovat na zprávu, která obsahuje jen hodnotu a nemá konkrétní název; 2) jak získat onen odeslaný obsah. Řešení těchto otázek je jednoduché. Bloku `[po přijetí zprávy]` byla přidána obecná možnost `{jakákoliv}` a při odeslání hodnoty přes `[rozešli všem]` a `[rozešli všem a čekej]` se tato událost zachytí. A potom se zavedla funkce `(poslední zpráva)` vracející poslední rozeslanou hodnotu již zmíněnými bloky.

Budeme tak moci rozeslat všem postavám například kostým. Obrázek 3.1.18 ukazuje rozeslání získaného kostýmu ze šatníku, následně na to vrácenou hodnotu funkce `(poslední zpráva)`, a způsob jak událost odchytit a bezejmenný kostým obléknout.

• *Poznámka: Datový typ kostým není prozatím úplně podporován a proto podmínka* `<je typu>` *možnost* `{kostým}` *ještě nenabízí. Obrázek 3.1.18 používá podmínku upravenou.*

Obrázek 3.1.18 Možné budoucí využití rozesílání obsahu namísto zprávy

• *Poznámka: Ideálně ke kontextu by se měl blok* [po přijetí zprávy] *přejmenovat na* [po přijetí] *a možnost* {jakákoliv} *na* {čehokoliv}*, což je výstižnější při rozesílání jak zpráv tak i obsahu. Přejmenování podporuje i fakt, že bloky* [rozešli všem] *a* [rozešli všem a čekej] *neobsahují ve svých názvech slovo „zpráva“.*

• **Zachytávání stisku kláves**

Další událost v prostředích se týká stisku kláves A–Z, a–z, 0–9, šipek, anebo mezerníku; kterou odchytáváme blokem [po stisku klávesy]. Tímto blokem lze však odchytnout stisk pouze jedné klávesy. Popišme si, o jaké možnosti přicházíme.

Nemůžeme reagovat na stisk *libovolné* klávesy, což bychom využili třeba při pobídce uživatele hláškou „Pokračuj stiskem libovolné klávesy“, anebo kdybychom programovali textový editor, jenž by každou stisknutou klávesu otiskl na scéně (podmínkami bychom museli ošetřit stisk šipek, kterými by se posunul kurzor v editoru, a téže stisk mezerníku vkládající mezeru do textu). Také nelze odchytávat stisk jakékoliv *číselné* klávesy, což by nám usnadnilo práci u projektů vyžadující zadání čísel. Může se jednat o kalkulačky, hesla (ke dveřím, trezoru...), či volba jedné z možností dle přiřazeného číselného kódu.

Kolikrát se také snažíme v projektech odchytit kombinace stisknutých kláves. Protože v prostředí lze odchytit jen určité klávesy, pak hovoříme především o kombinacích šipek, čísel, písmen, či mezerníku. Přestože lze v prostředí stisk kombinace kláves ověřovat podmínkou <stisknuta klávesa>, musíme se vždy dotazovat na konkrétní klávesu a nemáme tedy snadný přístup ke všem v onu chvíli stisknutým klávesám.

Vše výše prostředím vytknuté řeší návrh GitHub#387 uživatele @cycomachead, jež implementoval a rozšířil uživatel @gubolin na GitHub#631, do kterých mimochodem zasahoval i autor této práce svými příspěvky. Přidává do bloku [po stisku klávesy] další dvě možnosti {libovolná} a {číselná}. Dále návrh přidává (poslední stisknutá klávesa), kterou přistoupíme k naposledy stisknuté klávese. Tu využijeme právě ve chvíli, kdy zachytíme například stisk číselné klávesy blokem [po stisku klávesy] se zvolenou možností {číselná} a funkcí (poslední stisknutá klávesa) se k číselné hodnotě dostaneme. Konečně přístup k stisknuté kombinaci kláves získáváme poslední přidanou funkcí (stisknuté klávesy), jež vrací seznam s názvy stisknutých kláves. Funkce vrátí prázdný seznam pokud žádné klávesy nejsou stisknuty. Je tedy variantou k podmínce <stisknuta klávesa> a využijeme ji i u příkazu [po stisku klávesy] se zvolenou možností {libovolná}. Popisovaný návrh však ještě nebyl oficiálně schválen.

• **Podpora různých způsobů stisku číselných kláves u** [po stisku klávesy]

Pro čtenáře bude jistě překvapením, že prostředí detekují stisk číselných kláves rozdílným způsobem. Zapsat číslo na klávesnici můžeme několika způsoby. Buď na numerické klávesnici nebo na horním řádku alfanumerické části klávesnice. U druhé možnosti ještě závisí na nastavení jazyku klávesnice v operačním systému, neboť u české můžeme číslo zapsat jen stiskem kombinací kláves SHIFT (např. SHIFT + š => 3), kdežto u anglické klávesnice stačí zmáčknout jen onu klávesu (pak ale SHIFT + š zapíše #). Tabulka 3.1.6 ukazuje, na jaké způsoby stisku číselných kláves prostředí reagují u české klávesnice.

Prostředí	Číslice samostatně	SHIFT + číslice	Číslice numerické kláves.
Scratch 1	NE	ANO	ANO
BYOB	NE	ANO	ANO
Scratch 2	ANO	NE	ANO
Snap	ANO	ANO	NE

Tabulka 3.1.6 Odlišná podpora způsobu stisku číselných kláves

Dodejme, že stisk číselných kláves v prostředích Scratch 1 a BYOB závisí na zvoleném jazyku klávesnice. Pokud máme klávesnici anglickou, hodnoty by byly u těchto prostředí v tabulce 3.1.6 a sloupců „Číslice samostatně“ a „SHIFT + číslice“ opačné.

3.1.13 Spouštění a pozastavení projektu, zastavování scénářů

• Start projektu

V některých prostředích se při spuštění projektu provedou různé změny a restartují se některé hodnoty.[1]) V prostředích Scratch 1 a BYOB se konkrétně provede:

- vymazání uživatelovo odpovědi (funkce (odpověď) tedy vrátí {})
- aktivní dotazování se odpovědi blokem [zeptej se] bude přerušeno
- restartuje se nastavená hlasitost
- přeruší se neukončené zvukové stopy či noty hrané různými nástroji
- postava přestane povídat nebo myslet si
- restartují se grafické efekty postavy
- zastaví se všechny scénáře a začne se znovu od bloku [po kliknutí na start]

Prostředí Scratch 2 ještě vynuluje stopkami naměřenou hodnotu, což nás osvobozuje od neustálého používání příkazu [vynuluj stopky], a smaže dočasné klony postav.

Snap je v tomto ohledu naprosto odlišným. Žádnou z dosud vyjmenovaných akcí neprovádí. To znamená, že pokud chce programátor stejné chování jako v předchozích prostředích, musí se o to postarat sám využitím k tomu příslušných bloků. Tlačítko se zeleným praporkem tedy nepředstavuje úplné spuštění projektu, nýbrž pouhou událost spouštějící scénář [po kliknutí na start].

• *Poznámka: Řešením může být například vlastní blok* [připrav projekt]*, v němž budou všechny důležité příkazy (restartující stopky, mazající klony, odstraňující grafické efekty; a podobně), a který pak umístíme na začátek bloku* [po kliknutí na start]*.*

Je vhodné poznamenat, že nelze žádným příkazem přerušit aktivní dotazování se uživatele na odpověď. Jediným řešením je zeptat se znovu. Zároveň se neruší dočasné klony. To může při zběsilém klikání na tlačítko start zpomalit celé prostředí, pokud se ve scénáři [po kliknutí na start] vyskytuje příkaz [klonuj]. Uložená poslední rozeslaná zpráva bloky [rozešli všem] a [rozešli všem a čekej], ke které přistupujeme funkcí (poslední zpráva), lze vymazat pouze rozesláním nějaké nové zprávy. Potřebujeme-li i tento obsah po spuštění projektu odstranit, musíme rozeslat nějakou nejlépe jinými postavami nepřijímanou zprávu, například ve tvaru {ignoruj mě}.

• Pozastavení projektu

Při programování je velmi užitečné, když můžeme pozastavit vykonávání scénářů spuštěného projektu a prohlédnout si stav scény a postav. Je proto velkým překvapením, že obě verze prostředí Scratch projekt pozastavit neumí. Tato funkcionalita se proto týká BYOB a Snap z nichž pouze Snap umí pozastavit scénáře i programově.

Nad scénou mezi tlačítky pro spuštění a zastavení projektu se vyskytuje i tlačítko pro pozastavení projektu, které uživatel použije k manuálnímu pozastavení. Aby bylo možné projekt pozastavit, musí být logicky alespoň jeden scénář aktivní. Po pozastavení projektu se přestanou vykonávat všechny běžící scénáře. Pokračovat lze jedině tehdy, až uživatel přikáže projektu pokračovat opětovným kliknutím na stejné tlačítko.

Druhý způsob pozastavování, tedy programově, vyžaduje použití [pozastav vše], který je třeba umístit někam do scénáře. Projekt se sám pozastaví a čeká na uživatele, dokud jej opět manuálně nerozběhne. Toto lze provést pouze ve Snap.

Přirozeně předpokládáme, že při zachycení nějaké události (třeba po stisknutí mezerníku) se pozastavený projekt sám rozeběhne. Není tomu tak. Na události a dokonce ani na stopky projektu se pozastavení nevztahuje. To ale můžeme využít. Provádíme-li hráče nějakými místnostmi, nebo ukazujeme-li mu cíle hry, či chceme-li mu jen vysvětlit způsob ovládání, pak můžeme mezi jednotlivými částmi projekt příkazem pozastavit a čekat na jeho manuální rozeběhnutí. Hráč si tak bude moci číst informace od postavy jak dlouho bude potřebovat.

---

[1]) Děkuji uživateli @nathan, který mne na tuto skutečnost upozornil.

- **Způsoby zastavování projektu a scénářů**

Zastavování celého projektu a scénářů je též odlišné napříč prostředími, nicméně spíše než ve způsobu provedení zastavení se liší svými možnostmi a jejich funkcionalitou.

Ve Scratch 1 a BYOB můžeme kromě kliknutí na červené tlačítko zastavit celý projekt i příkazem [zastav vše]. Pakliže chceme zastavit pouze vybraný scénář nezávisle na ostatních, můžeme do něj umístit konečný blok [zastav scénář]. V prostředí BYOB se tento příkaz používá též uvnitř vlastního bloku. V takovém případě dojde k zastavení nejen scénáře vlastního bloku, ale i scénáře jemu nadřazeného, ze kterého se onen vlastní blok spustil či volal. K zastavení jen scénáře bloku slouží [zastav blok].

Nyní se dostáváme k prostředí Scratch 2, které namísto dvou v předchozích odstavcích popsaných příkazů k zastavení scénářů používá pouze jeden. Nazývá se obecně [zastav] a nabízí možnosti {všechno/tento scénář/jiné scénáře postavy}.

Možnost {všechno} jsme si již představili. Dále možnost {tento scénář} je ekvivalentní k [zastav scénář] a to pouze ke Scratch 1, což znamená, že použijeme-li ve Scratch 2 tuto možnost uvnitř vlastního bloku, pak se zastaví pouze on a jemu nadřazený scénář bude dále ve vykonávání pokračovat.

Konečně možnost {jiné scénáře postavy} zastaví všechny běžící scénáře své postavy. Nicméně je třeba poznamenat, že zvolením možnosti {jiné scénáře postavy} se blok [zastav] změní z konečného na obyčejný a bude tedy možné na něj napojit další příkazy. To znamená, že při zastavení všech ostatních scénářů postavy bude onen scénář, ve kterém žádost o zastavení proběhla, dále pokračovat.

V prostředí Snap je zastavování scénářů ještě sofistikovanější nabídkou celkem dvou k tomu určených příkazů, které se sice jmenují stejně, ale první jest příkazem konečným a druhý obyčejným. Různí se též nabízenými možnostmi, viz obr. 3.1.19.

Obrázek 3.1.19 Stejně pojmenované příkazy k zastavování scénářů ve Snap

U prvního příkazu (na obrázku vlevo) si popíšeme možnosti {tento scénář} a {tento blok}. Chování příkazu [zastav scénář] u prostředí BYOB, který zastavil i sobě nadřazený scénář při umístění ve vlastním bloku, lze docílit zvolením {zastav scénář}. Naopak pro stejné chování ze Scratch 2, ve kterém dojde k zastavení pouze scénáře vlastního bloku, musíme vybrat {zastav blok}.

Nakonec popišme druhý příkaz, který není konečným. Pokud bychom rádi zastavili ostatní scénáře postavy, musíme zvolit {ostatní scénáře postavy}, tedy stejně jako je tomu u prostředí Scratch 2. Snap ještě navíc nabízí možnost {vše až na tento scénář}, která zastaví (až na onen jediný) scénáře v celém projektu. Jde tedy o rozšíření předchozí možnosti ovlivňující i ostatní postavy.

Obě možnosti zastavující ostatní scénáře použijeme k jejich synchronizaci hromadným zastavením scénářů a následným řízeným spouštěním. Dle potřeby můžeme zvolit, jestli budeme synchronizovat scénáře jen v rámci postavy či celého projektu.

3.1.14 Ladění scénářů

Čas od času se v našem projektu vyskytne chyba. Například se špatně vykreslí nějaký obrazec, postava se chová jinak než jsme očekávali, a nebo dojde ke špatnému výpočtu. Ačkoliv se scénář provedl, jeho výsledek byl nesprávný. V takových případech musíme odhalit chybu, tu poté opravit a scénář znovu přezkoušet.

Ladění a testování scénáře lze provádět několika způsoby. Například kontrolou pořadí sestavených příkazů a volaných funkcí, přezkoušením výpočtů, ověřováním hodnot proměnných, či vypisováním stavových hlášek bloky [povídej] nebo [pomysli si].

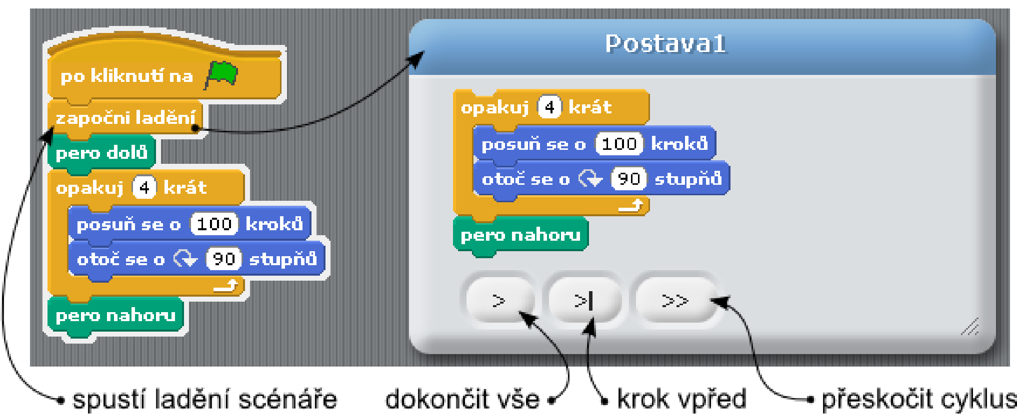
Prostředí Scratch 1 však dává programátorům do rukou další nástroj, který zvýrazňuje navštívené bloky provádějícího se scénáře krok za krokem. Tento nástroj, nazývejme jej ladícím, se zapíná a nastavuje v záložce *Editovat*. Po jeho zapnutí se prostředí přepne do ladícího režimu s určitou rychlostí a při provádění libovolného scénáře postupně zbarvuje jednotlivé bloky do žluta. Programátor tak může sledovat průběh scénářem (například počet opakování cyklu, průchod podmínkami, spouštění scénářů při zasílání zpráv apod.) a snáze odhalit chybu.

Bohužel tento ladící nástroj není příliš sofistikovaný. Nelze totiž určit začátek krokování (vždy se začíná od začátku scénáře), ani jej ovládat (vše běží automaticky a bez možnosti pozastavení) a v nastavení můžeme zvolit pouze ze dvou rychlostí[1]) krokování při průchodu scénářem, mezi nimiž chybí možnost „*velmi pomalu*".

Vývojáři prostředí BYOB si naštěstí dali záležet a tento nástroj vylepšili. Vše vytknuté z předchozího odstavce jejich ladící nástroj eliminuje. Ukázkou budiž následující příklad, který se týká scénáře vykreslující čtverec želví grafikou.

Nejdříve musíme umístit příkaz `[započni ladění]` někam do scénáře.[2]) Ve chvíli, kdy dojde k vyhodnocení tohoto bloku, započne od onoho místa ladění. Jakkoliv dlouhý scénář před tímto příkazem se tedy provede normální či turbo rychlostí.

Při započatí ladění se projekt automaticky pozastaví a v prostředí se objeví dialogové okno. V něm jsou zobrazeny navazující příkazy či funkce[3]), které mají být v rámci onoho scénáře vyhodnoceny. Třemi tlačítky vespodu tohoto okna pak můžeme krokování ovládat – posunout se o krok vpřed, přeskočit cyklus a pokračovat dále, či ladění scénáře neprodleně ukončit (viz obrázek níže).



Obrázek 3.1.20 Ladění scénáře kreslící čtverec želví grafikou v BYOB

Přeskakováním cyklu máme na mysli přeskakování jeho ladění. Cyklus se samozřejmě provede, ale nebude již zobrazen v dialogovém okně. Přeskakovat cykly se hodí v těch případech, kdy jsme si jisti, že neobsahují chybu a že by se prováděli například padesát krát. Klikat na tlačítko posunující nás o krok vpřed by totiž bylo utrpením.

Nezbývá než dodat, že příkaz `[započni ladění]` se může ve stejném scénáři vyskytovat vícekrát. Mezi prvním a druhým (či n-tým) laděním se scénář provede klasickou rychlostí a začne se znovu ladit jakmile se opět narazí na blok `[započni ladění]`.

Co se týče prostředí Scratch 2, pak je s podivem, že takovýto ladicí nástroj na rozdíl od svého předchůdce vůbec nenabízí. Snap tímto nástrojem prozatím nedisponuje, ale je na seznamu vylepšení a v budoucnu bude doprogramován. Vzhled a ovládání si zřejmě převezme z BYOB. Vývoj tohoto nástroje lze sledovat například na `GitHub#453`.

[1]) V nastavení krokování (tedy *Editovat* → *Nastavit krokování...*) jsou na výběr čtyři rychlosti vykonávání scénáře. Pouze dvě z nich se ovšem týkají krokování. Zvolíme-li rychlost *Normální* či *Turbo*, režim ladění se vypne, pakliže byl předtím zapnut.

[2]) Kdyby si někdo chtěl vyzkoušet ladící nástroj v prostředí BYOB, musí blok `[započni ladění]` hledat pod názvem `[debug]`. Prostředí BYOB totiž není kompletně přeloženo (více viz sekci 3.7).

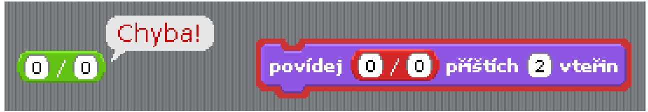
[3]) V BYOB existuje i funkce `(debug)`, kterou můžeme použít namísto `[debug]`, k ladění výsledků pouze funkcí a pořadí jejich vyhodnocování.

3.1.15 Hlášení chyb a zobrazování výjimečných hodnot

V předchozí sekci jsme si popsali ladící nástroj, který využijeme, když se scénář provedl, ale jeho výsledek byl odlišný od našeho očekávání. Nicméně při programování můžeme do scénáře zanést takovou chybu, kterou některá prostředí rozeznají a patřičně nás na ni upozorní – tedy zahlásí chybu, že je scénář neproveditelný. Grafické znázornění chyb, jejich popis a celkově způsob hlášení je však mezi prostředími značně odlišný.

- **Neplatné výpočty a nekonečno**

Klasickým příkladem vedoucí ke vzniku neplatného výpočtu je dělení nulou, tedy vyhodnocení funkce `(vyděl)` s argumenty `{0, 0}`. Tento neplatný výpočet mimo scénář je v prostředích Scratch 1 a BYOB znázorněn obecnou hláškou `{Chyba!}`. Pokud ale budeme dělit nulou někde ve scénáři, tj. blok `(vyděl)` bude jeho součástí, pak jej již zmíněná prostředí celý zastaví a zbarví dočervena (včetně chybového bloku).



Obrázek 3.1.21 Hlášení a zobrazení chyby vzniklé dělením nulou ve Scratch 1

Ačkoliv by se nám více líbilo hlášení „Nemůžeš dělit nulou!“, Scratch 1 svým návrhem předchází vzniku většině takových chyb a tudíž netřeba uvádět konkrétnější zprávu při jejich zachycení. Přesto lze hodnotit kladně, že problém neignoruje a donutí programátora chybu opravit. Stejně tak se chová i BYOB.

Scratch 2 se v této oblasti od svého předchůdce diametrálně liší, neboť nehlásí chyby vůbec (chyby pro něj neexistují). V případě dělení nulou se nezobrazí ani výsledek `{Chyba!}`, ani nezastaví a nezbarví scénář, ale vrátí se speciální hodnota `{NaN}`[1]).

Zatímco ve Scratch 1 byl scénář neproveditelný, ve Scratch 2 vrácení hodnoty NaN scénář nepřeruší. Pokud je nějakému bloku hodnota NaN předána jako argument, nahradí se u číselných vstupů hodnotou `{0}` a při práci s textem hodnotou `{NaN}`.

- *Poznámka: Programátor se tedy nemusí dozvědět, že chyba vznikla při dělení nulou. V tom případě by bylo vhodné použít ladící nástroj, ale tím Scratch 2 nedisponuje, takže musí odhalit chybu sám a to např. způsoby popsánými na začátku podsekce 3.1.14.*

Zobrazování hodnoty NaN však nechápejme jako velkou nevýhodu či nějaké omezení, ba naopak. Zvažme příklad, kdy bychom chtěli vydělit dvě čísla zadaná uživatelem, na která bychom se tázali blokem `[zeptej se]`.

Pokud uživatel zadá jako druhé číslo (jmenovatele) nulu, tak se scénář v Scratch 1 a BYOB přeruší a projekt bude muset být spuštěn znovu, zato ve Scratch 2 scénář poběží nerušeně dál. Jestli uživatel zadal neplatné hodnoty můžeme ověřit podmínkou `<je rovno>` a poté pokud je výsledek dělení roven `{NaN}`, můžeme uživatele upozornit třeba zprávou přes blok `[povídej]` a vyžádat si od něj hodnotu novou.

Završme předešlé příklady popisem prostředí Snap, které je někde na pomezí. Sice zobrazuje chybová hlášení (a to dokonce velmi konkrétní), ale při dělení nulou vrátí stejný výsledek jako Scratch 2, tedy `{NaN}`. Mezi prostředími jsou však dva rozdíly.

Za hodnotu NaN se nedosazuje žádná jiná hodnota, takže třeba součet NaN s jiným číslem vyústí opět v NaN (Scratch 2 by za NaN dosadilo 0 a výsledek by byl číselný).

A přestože Snap hodnotu NaN zobrazí, neumí ji (a ani nebude umět) porovnat podmínkou `<je rovno>`.[57] To znamená, že u předchozího příkladu s dělením dvou uživatelem zadaných čísel není možné ověřit, jestli se výpočet provedl správně.

- *Poznámka: Tento problém je ovšem řešitelný. Namísto kontroly výsledku dělení je třeba scénář přeprogramovat tak, aby nejprve kontroloval uživatelem zadané hodnoty a až poté přešel k operací dělení. Toto řešení musíme aplikovat i u Scratch 1 a BYOB.*

---

[1]) z anglického „not a number“ – tedy „není číslem“

Na závěr si ukažme ještě jeden příklad, v němž budeme též dělit nulou, ale čitatel bude různý od nuly (třeba `{5, 0}` či `{-1, 0}`). Scratch 1 a BYOB zahlásí chybu, neboť dělení nulou je podle nich nemožné. Naproti tomu prostředí Scratch 2 a Snap vrátí `{+Infinity}` nebo `{-Infinity}` (tedy kladné či záporné nekonečno), jenž můžeme porovnávat a to mimochodem v obou prostředích identicky.

- **Volání neexistující proměnné či seznamu**

Na vytvořené proměnné a seznamy se vždy odkazujeme někde ve scénáři. Přesto pokud nějakou proměnnou či seznam smažeme, odkazy na ně ve scénářích zůstanou. O jejich odstranění či výměnu se musíme postarat. Je s podivem, že všechna prostředí reagují odlišně při čtení hodnoty neexistující proměnné či přístupu k neexistujícímu seznamu.

Scratch 1 a BYOB neexistující proměnnou nahradí nulou a žádnou chybu nehlásí. Je ignorováno, že proměnná již není součástí postavy či projektu. Stejně se neprovede žádná operace nad seznamem, pokud takový seznam neexistuje. Programátor si tak vůbec nemusí všimnout, že ve scénáři zůstává něco, co již dávno nechtěl používat.

Scratch 2 zvolil již na první pohled lepší řešení. Neexistující proměnné a seznamy jsou automaticky dotvořeny ve chvíli, kdy na ně scénář narazí. Pokud se tedy proměnná či seznam opakovaně objevují v kategorii `data`, pak stále „trčí“ někde ve scénáři. Dotvořená proměnná se nastaví na nulu a seznam se vytvoří prázdný.

Žádost o neexistující proměnnou a seznam způsobí ve Snap chybové hlášení a dojde k zastavení a zbarvení scénáře dočervena. Snap za uživatele nic automaticky nevytváří. Programátor se tak setká v případě chybějící proměnné s hlášením:

```
Error
a variable of name 'x'
does not exist in this context
```

což můžeme přeložit jako: „Chyba! Proměnná s názvem 'x' není k dispozici v této části scénáře“. Chybová hlášení sice zobrazují podrobné informace o vzniklé chybě, ale zatím nejsou lokalizovatelná (viz `GitHub#462`), takže jim někteří uživatelé nemusejí rozumět.

- **Přespříliš složitá hlášení Snap**

Některá hlášení jsou ve Snap matoucí, jelikož obsahují i ty informace, kterým začátečník neporozumí. Například blok `(http://)` vracející zdrojový kód internetové stránky vypíše nesrozumitelnou chybu, pokud nedostane žádnou cílovou adresu:

```
NetworkError
Failed to execute 'send' on 'XMLHttpRequest': Failed to load 'http:'.
```

z čehož vyplývá, že pakliže není programátor zkušený, hlášením neporozumí.[1])

Dokonce se mohou zobrazit i hlášení o problémech vzniklých na nižší úrovni prostředí, které by měli vývojáři opravit. Taková hlášení jsou začátečníkem od těch běžných téměř nerozeznatelná a jelikož jim nerozumí, bude se zřejmě ptát, zdali jde o chybu jím způsobenou či nikoliv. Zvažme například chybu v předešlé verzi prostředí, která je již opravena. Spuštění `[hraj zvuk až do konce]` bez vybraného zvuku vyvolalo chybu:

```
TypeError
Cannot read property 'ended' of undefined
```

Taková chyba by se neměla vůbec zobrazit, resp. existovat. Vývojáři Snap si tak koledují o dotazy na službě GitHub od těch, kteří si nevědí rady s pro ně nesmyslnými hláškami.

3.1.16 Zbylé doposud nezmíněné primitivy

Do ostatních podsekcí tohoto porovnání se nedostaly některé bloky, které různá prostředí nabízejí navíc k základní sadě bloků Scratch 1. Proto je vhodné se s nimi alespoň částečně seznámit. Jejich celkový přehled je mimo jiné umístěn v dodatku E. Zmíníme pouze ty bloky, jejichž názvy anebo vysvětlení se doposud v práci neobjevilo.

[1]) A pokud je programátorem zkušeným, pak asi nebude programovat ve Snap.

- **Dosud nezmíněné primitivy BYOB**

Kopírování seznamů se provádí ve Scratch vlastním scénářem, a ještě ke všemu můžeme kopírovat prvky jen mezi dvěma existujícími seznamy. V prostředí BYOB funkce `(kopíruj)` převezme referenci na seznam, mezitím si vytvoří seznam nový, překopíruje prvky převzatého seznamu do nového a na ten vrátí referenci. Ve Snap kopírovací funkce chybí, takže si ji musíme doprogramovat, což si ukážeme až v podsekci 4.2.6.

Scratch zobrazuje seznam jakožto text s názvy jeho prvků. K dosažení toho samého v BYOB máme k dispozici referenci na seznam přijímající funkci `(jako text)`, která jeho prvky převede a vrátí v textové podobě. Snap tuto funkci nenabízí, neboť je možné použít funkci `(spoj)` vylepšenou vícenásobným parametrem (viz podsekce 4.2.3).

Závěrem zmiňme příkaz `[zahaj]`, který spustí jemu dodané zaobalené příkazy jako kdyby byly ve vedlejším scénáři.[1]) Je tak možné v jednom scénáři vytvořit více scénářů na sobě nezávislých. I kdyby příkaz `[zahaj]` obsahoval cyklus `[opakuj dokola]`, pak bude jeho původní scénář ve svém vykonávání pokračovat dál, protože se cyklus spustil v odděleném procesu. Tento příkaz je i v Snap, jemuž se věnujeme v podsekci 4.5.2.

- **Dosud nezmíněné primitivy Scratch 2**

Pro oslovení hráčů či ukládání jejich jmen existuje v prostředí Scratch 2 funkce `(jméno uživatele)`. Funkce vrátí `{}` pakliže uživatel není přihlášen. Snap funkci asi přidá.

- **Dosud nezmíněné primitivy Snap**

Pro rozdělení textu je možné použít funkci `(rozděl)`. Na základě zvoleného oddělovače se text rozdělí a výsledky vrátí v novém seznamu. Nabízené oddělovače jsou `{znak/mezera/řádek/tabulátor/cr}`[2]). Je možné zvolit i vlastní oddělovač – třeba pomlčku.

Do prostředí byly přidány ještě dvě funkce pro práci se seznamy. První – `(vlož do popředí)` – je ekvivalentní k funkci „cons“ z prog. jazyka Lisp a druhá – `(vše až na první prvek)` – zase k funkci „CDR“ z téhož jazyka.[24] O nich až v podsekci 4.2.2.

Neopomeňme ani podmínku `<dotýká se>` přidávající do nabídky možností `{stopa pera}`, která ověří, zdali se postava dotýká nakreslené čáry libovolnou postavou bez ohledu na její barvu. Tuto možnost ale asi nevyužijí postavy, které samy kreslí, neboť jakmile nakreslí sebemenší tečku a zůstanou nad ní stát, podmínka `<dotýká se>` detekující kontakt se stopou pera vrátí `{<pravda>}`.

Při programování někdy zjišťujeme délku textu či seznamu. K tomu nám prostředí dávají k dispozici funkce `(délka)` a `(délka)`. Někdy však nevíme – převážně u vlastních bloků a jejich parametrů –, zdali budeme v onu chvíli pracovat s textem či s referencí na seznam. Proto byla funkce `(délka)` v prostředí Snap vybavena schopností přijmout i referenci na seznam a vrátit jeho délku, stejně jako by to udělala funkce `(délka)`.

Rozšířit prostředí či případně komunikovat přímo s jeho zdrojovými kódy můžeme funkcí `(JavaScript funkce)`, která dokáže vykonat doplněný kód na úrovni jazyka JavaScript v kombinaci s `[spusť]` či `(zavolej)`. Je možné naprogramovat třeba časově náročnější algoritmy a tím těžit z efektivity tohoto řešení na nižší úrovni, či si doprogramovat chybějící bloky vyskytující se v prostředí Scratch 2. Více v podsekci 4.5.1.

- *Poznámka: Autor této práce byl zpočátku vůči `(JavaScript funkce)` skeptický a na GitHub argumentoval proti jejímu zavedení. Nelíbilo se mu totiž, že programátoři neovládající jazyk JavaScript neporozumí projektům zmíněnou funkci používající. Nakonec ale pochopil, že jde o výborný rozšiřující nástroj, používá-li se alespoň trochu moudře.*

Dále zmiňme schopnost Snap přepsat scénář či jednotlivé bloky do textové podoby, přiřadíme-li užitým blokům textovou reprezentaci. Lze tak převést scénář do zdrojové kódu nějakého programovacího jazyka, jemuž jsme doprogramovali podporu k převodu (např. JavaScript, Smalltalk, Python aj.).[32] K přepisu určené bloky `[převeď blok]`, `[převeď String]`, `[převeď část]`, a `(kód z)`; je však potřeba zobrazit přes „*Nastavení*“ → „*Podpora přepisu bloků do textu*“. Více o těchto až v podsekci 4.5.3.

---

[1]) Toto se v informatice označuje jako spuštění příkazů v dalším *vlákně*.

[2]) Zkratka „cr“ znamená „carriage return“, což je netisknutelný znak reprezentující stisk klávesy `Enter`.

Tuto podsekci zakončíme posledními přidanými bloky [spusť/sp] a (zavolej/sp), kterými jsme schopni zachytit *pokračování* od jejich umístění ve scénář. Bloky nám toto *pokračování* (následující část scénáře – bloky, argumenty, užité parametry…) umožní uchovat v zaobalené proceduře, kterou můžeme vykonat bloky [spusť] či (zavolej) v libovolné chvíli. Tento princip známý například z jazyka Scheme pod názvem *call-with-current-continuation* si však pro jeho složitost v této práci ukazovat nebudeme.

3.2 Práce s grafikou, zvukem a multimédii

3.2.1 Podporované grafické a zvukové formáty

• Podpora bitmap, vektorů a grafických formátů

Nejprve musíme zmínit hlavní rozdíl v podpoře grafiky mezi současnými a předešlými verzemi prostředí. Scratch 1 a BYOB podporují pouze bitmapovou grafiku, na základě čehož můžeme odhadnout, jakými grafickými editory disponují a přibližně které grafické formáty obrázků dokáží naimportovat. Zato prostředí Scratch 2 ani Snap nejsou přímo vyhraněná a podporují jak bitmapovou tak vektorovou grafiku.

Podpora vektorové grafiky je v prostředí Scratch 2 částečná. Ačkoliv obsahuje svůj vlastní zjednodušený editor, zdá se, že vektorové obrázky z externích editorů zobrazí ve správné podobě jen v případě, že byly v editorech vytvořeny jednoduchými nástroji a to takovými, kterým rozumí právě editor Scratch 2.

• *Poznámka: Zkušenosti autora vypovídají, že vektorové obrázky vytvořené různými sofistikovanými nástroji programu Inkscape se v prostředí Scratch 2 zobrazili s odlišným či narušeným vzhledem, častokrát s jiným fontem, či dokonce vůbec.*

Ve Snap je podpora vektorové grafiky závislá na prohlížeči. Některé verze prohlížeče Internet Explorer projekt obsahující vektorovou grafiku vůbec nenačtou. V prohlížeči Opera se projekt zase načte, ale postavu s oblečeným vektorovým kostýmem nelze přetahovat metodou táhni-pusť. U prohlížečů Mozilla Firefox a Google Chrome funguje podpora vektorů nejlépe. Autoři projektů nechť zváží, jestli je v prostředí Snap vhodné použít takovýto typ reprezentace obrázkových informací. Snap nás při importu vektorového obrázku dokonce varuje, že podpora vektorů není stoprocentní.

• *Poznámka: Při využití kapacitně větších vektorových obrázků může v prohlížeči Google Chrome vzniknout chyba při jejich ukládání na lokální uložiště prohlížeče. To je omezeno svou maximální velikostí, které data vektorových obrázků zaplní. Otevírání některých projektů může skončit stejně neúspěšně. Jde o chybný návrh prostředí, jelikož ukládání a otevírání projektů z cloud účtu funguje bez problému. Aktuálně se hledá vhodné řešení.*

Prostředí	JPEG/PNG /GIF	Animovaný GIF	Průhlednost u PNG	BMP	SVG
Scratch 1	ANO	rozložen	NE	ANO	NE
BYOB	ANO	rozložen	NE	ANO	NE
Scratch 2	ANO	rozložen	ANO	NE	omezená podpora
Snap	ANO	první snímek	ANO	ANO	dle prohlížeče

Tabulka 3.2.1 Podpora grafických formátů a jejich vlastností

Rozepišme závěrem kolonku o animovaném GIF obrázku z tabulky. Všechna prostředí až na Snap jej rozloží dle snímků na jednotlivé kostýmy. Jejich střídáním blokem [další kostým] může programátor „složit“ animaci zpět. Za to Snap naimportuje animovaný GIF jen jako jeden kostým, jehož obrázek je roven prvnímu snímku animace.

• Podpora zvukových formátů

Zohledníme-li pouze formáty WAV a MP3, pak jsou podporovány všemi prostředími.

3.2.2 Zabudované editory, záznam zvuku, snímky z kamery

• **Bitmapové grafické editory**

Rozdílnost bitmapových editorů je napříč prostředími minimální. Nabízejí vesměs základní nástroje pro manipulaci s obrázky (*zvětšování* a *zmenšování*, *otáčení*, *překlápění*, *výběr oblasti* aj.), dále kreslící nástroje (*štětec*, *gumu*, útvary *obdélník* a *kruh*, *kapátko*, *výplň* a další) a umožňují nastavit *střed kostýmu* ke stanovení osy při otáčení postavy.

Jediné prostředí Snap v editoru nenabízí nástroj pro vkládání textu. To je ovšem záměr, neboť se plánuje vytvořit text jako samostatný editovatelný objekt, který bude ovladatelný interaktivní cestou či příkazy – tedy stejně jako je tomu u postavy.[49]

Zajímavostí také je, že Scratch 1 a BYOB nabízejí širokou škálu systémových fontů (zřejmě protože jsou nainstalovány v počítači), na rozdíl od Scratch 2, které nabízí vlastních fontů jenom šest. Pouze dva z nich rozeznávají všechna písmena české abecedy.

• **Vektorové grafické editory**

Editor Scratch 2 funguje v jednom ze dvou režimů (bitmapový a vektorový), mezi kterými uživatel přepíná dle své potřeby. Přepne-li uživatel editor do vektorového režimu, objeví se jednoduché nástroje určené pro práci s vektory, kterými lze kostým vytvořit.

Prostředí Snap vektorový editor vůbec nevlastní. Z toho plyne, že se vektorová grafika do projektu dostane pouze importem obrázků vytvořených externími programy. Když se o editaci vektorového kostýmu ve Snap pokusíme, otevře se editor umožňující nastavit pouze střed kostýmu. Manipulace s body a křivkami v něm však není možná.

• **Zvukové editory**

Scratch 2 jako jediné prostředí nabízí zvukový editor. Jakýkoliv uživatelem nahraný či importovaný zvuk v podporovaných formátech lze různě editovat. Editor zobrazuje zvukovou stopu, jejíž část je možné označit a následně poupravit – tj. *umazat*, *upravit*, či na ni aplikovat vybraný *efekt* apod. Nabízené efekty umožňují označenou část *otočit* pozpátku, *ztlumit* či *zesílit*, nebo například *umlčet* atd.

• **Záznam zvuku z mikrofonu**

Má-li uživatel připojen mikrofon, může nahrát vlastní zvukovou stopu ve všech prostředích kromě prostředí Snap. To se však časem změní, neboť se vývojáři Snap vyjádřili pozitivně k myšlence v budoucnu implementovat podporu pro nahrávání zvuku.[48]

Scratch 2 jako jediný nabízí možnost pozastavit nahrávání zvuku a to tak, že uživatel nahrávání vypne (což simuluje pozastavení) a poté znovu zapne. Dojde sice k novému nahrávání, ale bude se pokračovat ve stejné zvukové stopě, která se navíc objeví ve zvukovém editoru Scratch 2 a tak můžeme její část editovat způsobem již popsaným v předchozí oblasti o zvukových editorech.

• **Pořizování snímků z videokamery**

Zde je stav stejný jako u předchozí části o zvuku. Jen prostředí Snap je stále pozadu. I zde vývojáři plánuji doprogramovat funkcionalitu k pořízení snímku z kamery.[48]

Prostředí	Bitmapový editor	Vektorový editor	Zvukový editor	Nahrávání zvuku	Pořizování snímků
Scratch 1	ANO	NE	NE	ANO	ANO
BYOB	ANO	NE	NE	ANO	ANO
Scratch 2	ANO	ANO	ANO	ANO	ANO
Snap	ANO	NE	NE	ještě ne	ještě ne

Tabulka 3.2.2 Editory prostředí a nástroje pro nahrávání zvuku a obrazu

3.3 Komfort a uživatelská přívětivost prostředí

3.3.1 Snadnost skládání a editace scénářů

• Záměna bloku za jednu z alternativních variant

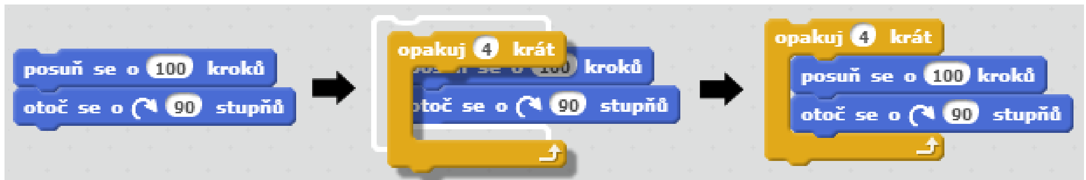
Rozhodneme-li se zaměnit nějaký blok ve scénáři, musíme jej pracně (rozumějme ručně) pozměnit. Abychom nemuseli rozpojovat scénář na dvě části a bloky složitě obměňovat, můžeme využít nástroj pro záměnu bloků za bloky jím podobné a to za předpokladu, že prostředí eviduje varianty k takovým blokům. Prostředí se navíc postará o přepis hodnot argumentů z původního do zaměněného bloku.

Funkcionalita je ve všech prostředích přítomna, jen Snap disponuje daleko větším počtem zaměnitelných variant. Lze např. zaměnit bloky událostí či cyklus v podmínku.

V BYOB a Snap můžeme navíc zaměnit vlastní blok za jiný vlastní blok téhož typu (tedy např. vlastní příkaz za jiný vlastní příkaz), jen BYOB na rozdíl od Snap ještě selektuje bloky podle kategorie (třeba vlastní funkci za vlastní funkci stejné kategorie).

• Obehnání příkazů bloky řídících struktur

Příkazy vkládáme do podmínek a cyklů směrem dovnitř – tj. nejprve si z palety bloků vytáhneme blok řídící struktury a poté do něj vkládáme příkazy. Existuje však varianta opačná. Máme-li připravenou část scénáře, kterou chceme cyklicky či za určitých podmínek vykonat, můžeme takový blok řídící struktury přiložit na ony bloky a prostředí nám spolu s grafickou signalizací nabídne jejich automatické vložení dovnitř. Celý proces je tedy opačný a lze jej chápat jako „obehnání příkazů“ cyklem či podmínkou.



Obrázek 3.3.1 Vložení příkazů do bloku cyklu opačným způsobem

Všechna prostředí kromě Snap mají tuto funkcionalitu implementovanou. Programátoři tohoto prostředí se však časem této užitečné funkcionality též dočkají (viz GitHub#351).

• Kopírování jediného bloku

Ve všech prostředích máme možnost kopírovat soustavu bloků od námi zvoleného bloku až po konec scénáře (resp. konec jeho kopírované úrovně). Problém ale nastává ve chvíli, kdy potřebujeme samostatně zkopírovat první či jiný – mezi vícero bloky a na jiném než posledním místě umístěný – blok. Doposud jsme museli zkopírovanou část scénáře umístit stranou, bloky pracně rozpojit, zbytky smazat, a nakonec přesunout kopii bloku na své nové místo.

BYOB a Snap umí zkopírovat i jediný vybraný blok bez na něj navazujících bloků. Zatímco je v nabídce BYOB možnost „– tento blok“, Snap namísto toho zobrazí zmenšený grafický náhled bloku, abychom si byli jisti, že kopírujeme ten správný. Při kopírování cyklů nebo podmínek BYOB zkopíruje pouze blok samotný bez vložených příkazů. Snap naopak zkopíruje cykly a podmínky i s obsaženými příkazy.

• Vyhledávání bloků v projektu napříč kategoriemi

Jediné prostředí Snap nabízí možnost vyhledat si blok v projektu dle jeho názvu bez ohledu na kategorii. Stisknutím klávesové zkratky `CTRL + F` lze proměnit paletu bloků libovolné kategorie v paletu s výsledky hledání. Nalezené bloky se průběžně aktualizují s každou změnou textu ve vyhledávacím poli. Také nemusíme psát celé názvy bloků. Stačí napsat pár začátečních písmen a je vyhledáno. Skrytá primitiva se nevyhledávají.

- **Dokumentační komentáře bloků**

Každý programátor by měl dodat svým vlastním blokům obecný popis o jejich funkci. To se zpravidla provádí připnutím obyčejného komentáře k jejich hlavičce. Tyto komentáře se nazývají *dokumentační*. Uživatelé tak nemusejí podrobně studovat scénář.

Protože jsou ve Snap definice bloků uvedeny uvnitř (nejsou vidět na ploše scénářů), pak toto prostředí zobrazuje dokumentační komentáře ještě dvěma způsoby:

- ponecháním kurzoru myši nad vlastním blokem v paletě bloků se zobrazí dokumentační komentář v bublině (v současné verzi bublina po prvním zjevení problikává)
- možností „nápověda“ vyvolanou pravým tlačítkem myši na vlastním bloku se zobrazí dialogové okno obsahující jeho ikonku (náhled) a dokumentační komentář.

- **Kontrastní zbarvení typů proměnných a střídavé barvy**

Proměnné scénáře a formální parametry bloku mohou mít stejný název jako proměnné projektu a postavy. Proto aby se mezi nimi programátor lépe orientoval, nabízí prostředí BYOB možnost barevného odlišení ve svém nastavení. Proměnné scénáře a parametry jsou se zapnutou možností zbarveny do tmavší oranžové.

V Snap lze aktivovat „střídavé barvy“. Vnořené bloky stejné kategorie budou jednou světlejší a podruhé přirozené barvy, čímž se značně zpřehledňuje čitelnost scénáře.

- **Zobrazení mezery v textových argumentech a speciální typ parametru**

Při zadávání argumentů v podobě textové hodnoty může být pro některé obtížné odhalit, jestli se v určitých částech textu vyskytuje mezera. Není ani tak problém odhalit mezery mezi slovy jako mezery na začátku a na konci textu. Také je těžké všimnout si více mezer umístěných přímo za sebou. Proto prostředí Snap reprezentuje mezeru vertikálně vystředěnou tečkou, což eliminuje výše zmíněné problémy. Existuje však speciální typ parametru přítomný jen u primitivů, který umožňuje klávesou `Enter` zapsat text na více řádků. Slouží k zápisu programové kódu, proto mezeru graficky neznázorňuje.

- **Znázornění chybějících bloků použitých v importovaných scénářích**

Při importu postav, scénářů z batohu, či definic vlastních bloků se může stát, že využívají nějaký jiný vlastní blok, který jsme do prostředí zapomněli předem naimportovat. Chybějící blok je vždy nahrazen speciálním blokem, který prostředí znázorňují odlišně.

Ve Scratch 2 se chybějící blok nahradí blokem `[undefined]` (v překladu *nedefinovaný*), který – ačkoliv má výraznou rudě červenou barvu – nespadá do žádné kategorie a za normálních okolností k němu nemáme přístup. Sám nic nevykonává, jen vyplňuje chybějící vlastní blok v importovaném scénáři. V prostředí Snap se příkaz nahradí příkazem `[obsolete]` (*zastaralý*), který je téže barvy i významu. Protože ve Snap mohou chybět i vlastní funkce a podmínky, prostředí takové vždy nahradí funkcí `(obsolete)`. Jelikož prostředí netuší, jestli chybí funkce či podmínka, nahrazuje tak raději chybějící blok vždy funkcí, která vrátí prázdný text `{}`.

- **Automatické uspořádání scénářů**

Všechna prostředí umožňují každé postavě automaticky uspořádat scénáře na ploše scénářů do několika sloupců. Prostředí Snap ale uspořádá scénáře jen do jednoho sloupce pod sebe. To může být záměr, neboť vzhledem k rozměrům různých zařízení lze očekávat nedostatek místa pro scénáře, a tak je lepší seřadit je do jednoho sloupce.

- **Maximální šířka bloků aneb zalamování bloků ve scénáři**

Při mnohonásobném složení funkcí či podmínek dosahují scénáře extrémních šířek a jejich obsah „utíká“ za hranice okna se scénáři. Programátoři se proto musí horizontálním posuvníkem přemisťovat z jednoho místa prostředí na druhé.

Pouze prostředí BYOB a Snap v takových případech bloky raději zalomí a udržují tak maximální šířku scénáře. Jenomže v některých případech vlivem automatického zarovnávání může dojít k zalomení v takovém místě, ve kterém je to spíše nežádoucí. Jelikož tato funkcionalita nelze vypnout a ani nelze upravit maximální šířku scénáře, nevyhneme se horší čitelnosti některých scénářů trpících automatickým zalomením.

3.3.2 Správa projektu

- **Přejmenování proměnných**

Mnohdy potřebujeme přejmenovat proměnné všech typů, ať už kvůli chybně zadanému názvu při jejich vytváření, anebo v důsledku změn ve vývoji projektu. Je to však pouze prostředí Scratch 2, které umožňuje přejmenovat proměnné postavy a proměnné všech postav. Prostředími BYOB a Snap podporované proměnné scénáře lze bez problému přejmenovat kliknutím na jejich název v `[proměnné scénáře]`. Parametrů vlastního bloku standardně přejmenováváme v editoru bloku, kde se definuje jeho hlavička.

- **Pořadí vlastních bloků v paletě**

Ve Scratch 2 jsou nově vytvořené vlastní bloky řazeny pod ostatní již existující. Jejich pořadí se však promíchává při přetahování definic každého z nich. Dávat si proto záležet na pořadí při vytváření bloků nemá smysl. Prostředí BYOB řadí bloky dle jejich názvu. I u něho tedy nejsme schopni docílit seřazení dle vlastní představy.

Snap vlastní bloky seřadí podle pořadí přidání. Pokud tedy někteří programátoři lpí na přehlednosti a znají bloky, které se v jejich projektu budou vyskytovat, mohou je vytvořit ve chtěném pořadí. Mnohdy se ale rozhodneme s bloky manipulovat (některé smazat, dodat nové, apod.) a v případě neplánovaných zásahů do pořadí či počtu vlastních bloků jsme nuceni všechny doposud vytvořené bloky smazat a znovu je vytvořit.

Žádné prostředí zatím neumí bloky uspořádat dle přání autora projektu, ale v prostředí Snap je tato možnost na seznamu vylepšení. Více na `GitHub#271`.

- **Skrývání primitivních bloků**

Někteří (zejména učitelé) mohou namítat, že by se například cyklus `[opakuj dokola]` neměl v prostředích vůbec vyskytovat. Přeci jen nic se neopakuje donekonečna a vždy existuje podmínka, za které scénář skončí přirozeným způsobem. Není-li tomu tak, pak zřejmě programujeme špatně a měli bychom náš scénář poupravit. Proto je lepší takovýmto „excesům“ předejít a některé primitivy před uživateli (tedy žáky) skrýt.

Snap dokáže skrývat buď jedno primitivum či všechny primitivy libovolné kategorie přes nabídku vyvolanou kliknutím pravého tlačítka na primitivní blok či paletu bloků. Odhalovat již schované primitivy lze pouze hromadně, proto schováme-li omylem nějaký blok, musíme všechny schované odkrýt a celý proces – tentokrát bez chyby – zopakovat.

Tuto funkcionalitu mohou využít i autoři výukových projektů, ve kterých mají připravené vlastní bloky k vyřešení zadání a úkolů v projektu. Hromadné skrytí primitivů se v takovém případě může hodit, protože se kategorie bloků velmi zpřehlední.

- **Mazání vlastních bloků**

K odstranění vlastního bloku ve Scratch 2 nesmí být takový ani jednou použit v jakémkoliv scénáři. Je proto frustrující hledat – byť jediný – výskyt vlastního bloku, pakliže projekt obsahuje velké množství scénářů rozesetých po více místech.

BYOB a Snap mažou bloky naštěstí ihned. Je ale třeba upozornit, že BYOB blok okamžitě smaže, aniž by se zeptal uživatele, jestli si je tímto krokem jist. Velmi často programátor přijde o blok omylem, neboť přímo nad možností mazající blok je umístěna možnost k jeho editaci. A tak při překliknutí se nelze již smazaný blok obnovit. To je nejenom frustrující, ale převážně demotivující.

- **Zobrazování částečného množství prvků ve sledovači seznamů**

Některé seznamy v projektech dosahují extrémní délky v řádech stovek či tisíců prvků. Proto je sledovač seznamů vybaven posuvníkem, jehož velikost je úměrná počtu prvků seznamu. Se vzrůstajícím počtem prvků se posun mezi prvky ve sledovači provádí velmi obtížně, neboť je posuvník příliš malý a pohyb je velmi „citlivý“.

Sledovač seznamu v prostředí Snap zobrazuje automaticky prvních 100 prvků. Pakliže seznam tuto délku překročí, musíme přes černou šipku směřující dolů umístěnou na sledovači samotném provést „výřez“ v náhledu prvků po stovkách. Lze tak

zobrazit 1–100 prvek, 101–200, 201–300; a tak dále. Zobrazování po stovkách zajistí posuvník větší velikosti s menší citlivostí při posunu. Proto nelze zobrazit všechny prvky najednou, sledovač stále obsahuje informaci o celkovém počtu prvků v seznamu.

- **Funkce navracející změny v prostředí**

Scratch 1, BYOB, a Scratch 2 dokážou vrátit poslední smazaný blok či scénář, komentář, kostým, zvuk, anebo postavu. Scratch 2 obnovuje ještě definici vlastního bloku a to jak jeho hlavičku tak i tělíčko. Vrátit se však můžeme pouze o jediný krok. Smažeme-li kupříkladu kostým a poté zvuk, můžeme obnovit pouze zvuk, nikoliv kostým. Programátorům Scratch 1 a BYOB se prvek vrací „do ruky", tudíž jej musí vrátit na své původní místo. Scratch 2 obnovené prvky okamžitě vrací, avšak nikoliv „do ruky", nýbrž na předem stanovené místo. I v tomto případě musíme prvky manuálně vrátit tam, odkud jsme je odmazávali. Touto možností prostředí Snap nedisponuje.

BYOB a Snap mají ještě další funkcionalitu pro návrat posledního bloku či skupiny bloků, se kterými bylo manipulováno – tj. přesouváno po ploše scénářů a napojováno na konec či odpojováno z jiného scénáře. V obou prostředích se navracené prvky vrací „do ruky". Zatímco první popisovaná funkcionalita sloužila k obnovení **smazaných** prvků, tato funkcionalita se vztahuje pouze k pohybu bloků na ploše scénářů. Programátoři z ostatních prostředí budou zřejmě tyto dvě funkce ve Snap zaměňovat.

Propracovanější návrat změn, alespoň jak je nám známo z jiných aplikací (třeba z internetových prohlížečů), není ani v jednom z prostředích k dispozici. Uvítali bychom nejenom návrat o několik kroků zpět, ale i možnost posunu vpřed, což známe pod názvy „undo" a „redo". Tímto bude v budoucnu vybaveno prostředí Snap (viz `GitHub#78`).

- **Přenášení obsahu mezi projekty stejného prostředí**

Export je jednou z možností přesunu obsahu mezi projekty. Je asi zřejmé, že exportovaná data je možné do prostředí kdykoliv zpět naimportovat. Síla exportu je především v opětovném použití obecných dat – např. zvuků (zvuk kliknutí), kostýmů (vzhled tlačítka), nebo třeba bloků (reakce na tlačítko), které mají využití ve více projektů.

V Scratch 1, BYOB a Scratch 2 můžeme exportovat kostýmy, zvuky a postavy. Soubory se ukládají na počítač přes dialogové okno. Snap neumí exportovat pouze zvuky a kostýmy s postavami se exportují do nové záložky prohlížeče, jejíž obsah musíme ještě uložit na počítač zvlášť. Snap také umí exportovat vlastní bloky.

Scratch 2 ještě přichází s tzv. „batohem" nacházející se vespod části plochy scénářů. Lze jej otevřít a skrýt dle potřeby. Můžeme do něj „vhodit" kostýmy, zvuky, komentáře, postavy samotné, definice bloků, či různé bloky a scénáře; které v něm zůstanou nastálo. Odstraňují se stejně jako jakýkoliv prvek v prostředí – přetažením na paletu bloků.

Batoh je vázán na cloud účet, takže bez ohledu na používaný počítač či otevřený projekt je možné obsah batohu využívat, popř. rozšiřovat. Je tedy logické, že jej bez přihlášení použít nemůžeme.

- **Přenášení obsahu z počítače či jiných zdrojů do prostředí**

Přenášet obsah do prostředí můžeme ještě jedním způsobem – metodou „táhni-pusť". Jelikož jsou prostředí Scratch 1 a BYOB samostatnými aplikacemi nainstalovanými na počítačích, lze do nich obrázky, zvukové stopy, a postavy; přetáhnout. To samé můžeme udělat i u prostředí Snap, jež umí zpracovat i `.xml` soubor obsahující definice bloků.

Bohužel on-line verze prostředí Scratch 2 spuštěná ve webovém prohlížeči při přetažení obsahu metodou „táhni-pusť" opustí internetovou stránku editovaného projektu. Prohlížeč se totiž pokusí obsah otevřít v záložce sám. Ztratíme tím pádem všechno, co jsme v projektu od jeho poslední uložené verze změnili.

Prostředí Snap ještě podporuje tzv. „Cross-Origin Resource Sharing" (CORS).[10] Jestliže nalezneme na nějaké internetové stránce obrázek či zvuk, který chceme použít v projektu, pak je za normálních okolností musíme nejprve uložit na počítač a poté ručně naimportovat do projektu v prostředí. Podporuje-li však taková internetová stránka CORS (volí si její majitel), stačí požadovaný obsah přetáhnout (například z dalšího

okna webového prohlížeče) do prostředí Snap a to se o převod automaticky postará. Například přetažený obrázek okamžitě přidělí aktivní postavě za její kostým.

3.3.3 Ztlumení zvuků

Při prohlížení projektů, zejména na Scratch síti, se setkáváme s různými zvuky dokreslující atmosféru hry nebo příběhu. Některé jsou však příliš hlasité, nepříjemné, anebo se velmi často opakují. Uživatel by měl mít právo si zvolit, jestli chce přehrávání zvuků v prostředí ztlumit. Ztlumením máme na mysli okamžité snížení hlasitosti na nulu, tedy že se zvuky stále přehrávají, ale uživatel je neslyší, dokud ztlumení zvuků neodstraní.

Scratch 1 a BYOB ztlumit zvuky neumí. Tuto funkcionalitu nahradíme příkazem `[zastav všechny zvuky]`, který je třeba umístit do nekonečné smyčky a neustále opakovat. Zasáhneme tak ale do projektu, což je nevhodné. Kliknutím na zelený praporek v kombinaci s klávesou `Ctrl` je ztlumení zvuků možné ve Scratch 2 dosáhnout, nicméně chybí mu indikátor přepnutého stavu (není jasné, zdali jsou zvuky ztlumené).

Návrhem autora této práce se ve Snap začalo na této možnosti pracovat a v současnosti je téměř implementována. Indikace stavu bude spočívat ve změně ikonky tlačítka ke ztlumení zvuků, které bude umístěno nad scénou, což je intuitivnější způsob tlumení zvuků na rozdíl od prostředí Scratch 2. Více na `GitHub#424` a `GitHub#544`.

3.3.4 Nápověda k primitivním blokům

Aby se začínající programátoři lépe orientovali mezi základní sadou bloku, tj. primitivů, nabízí jim prostředí nápovědu. V Scratch 1, BYOB, a Snap jsou nápovědy obrázkové a tím pádem nepřeložitelné. Scratch 2 má většinu popisků u nápověd překladatelných, ale občas také obsahují obrázky s anglickými popisky, jenž není možné přeložit.

3.3.5 Nabídky pro pokročilejší, uživatelský a vývojový mód

Pro začínající programátory anebo nepříliš zasvěcené osoby je vhodné některé nabídky skrýt. Ti pokročilejší si je mohou zobrazit. V kombinaci se stisknutou klávesou `Shift` lze napříč prostředími zobrazit různé nabídky s rozšířenými možnostmi. Prostředí Snap mění barvu jejich textu na červenou.

Prostředí je nám známo v *uživatelském* módu, ve kterém je po spuštění automaticky. Scratch, BYOB, a Snap dokáží fungovat ještě v módu *vývojovém*, který otevírá další možnosti vývojářům a to ať už k připravovaným – ale ještě neoficiálním – rozšířením, anebo k testování různých funkcionalit. Prostředí Snap například bloky přítomné jen ve *vývojovém* módu označuje za *experimentální*.

Do vývojového módu se v každém prostředí dostáváme jiným způsobem. V rámci této práce nás zajímá jen způsob pro prostředí Snap. Stačí kliknout se stisknutou klávesou `Shift` na logo Snap → „*Přepnout do vývojového módu*“ – jedná se mimochodem o skrytou nabídku, která nemá být obyčejným uživatelům přístupná.

3.3.6 Velikost bloků, jejich popisků, sledovačů a mód scény

- **Změna velikosti bloků a fontů u jejich popisků**

Nejenom učitelé využívající projektor ve třídě, uživatelé tabletů a jiných zařízení, ale i osoby se zrakovými potížemi; uvítají možnost změny velikosti bloků a jejich popisků.

Scratch 1 a BYOB bohužel pracovat s velikostmi bloků a jejich popisků neumí. Ve Scratch 2 můžeme bloky přiblížit společně se zvetšením jejich popisků anebo můžeme změnit jenom popisky a bloky ponechat menší velikosti – to provedeme v kombinaci s klávesou `Shift` kliknutím na ikonku zeměkoule → „*set font size*“.

Prostředí Snap umožňuje měnit velikost bloků, ale už neumí změnit velikost jejich fontu. Přesto co se týče přiblížení je toto prostředí nejdále. Umožňuje totiž nastavit libovolnou hodnotu, přičemž je možné si vybrat z přednastavených velikostí. Dobré je pamatovat na to, že s větším přiblížením se projekt rapidně zpomalí.

- **Minimální režim zobrazení (formát) scény**

Dosud známe jen tři režimy zobrazení scény v prostředích: 1) *plná scéna*, 2) *malá scéna*, 3) *prezentační režim*. Prostředí Snap zavádí další režim: 4) *minimální scéna*, jenž zmenší scénu do ještě menších rozměrů než má *malá scéna*. Musíme však kliknout na tlačítko měnící scénu z *velké* na *malou* v kombinaci se stisknutou klávesou `Shift`.

- **Velikost sledovačů v závislosti na zvoleném režimu scény**

Scéna je vždy spuštěna v určitém režimu zobrazení a sledovače proměnných a seznamů se jí velikostně přizpůsobí. Přepneme-li se například do prezentačního režimu, sledovače se zvětší tak, aby byly čitelné i pro osoby pohlížející z větší vzdálenosti. Se zmenšením scény se naopak zmenší i sledovače, jelikož by jinak překáželi postavám odehrávající příběh. Sledovač bude mít na scéně vždy stejné umístění a velikost ve vztahu k velikosti postav, které svojí velikost též přizpůsobují zvolenému režimu scény.

Na rozdíl od ostatních pouze prostředí Snap velikost sledovačů nepodmiňuje režimem scény, tudíž mají neustále stejnou velikost. Výhodou je, že při zmenšeném formátu scény je obsah sledovače stále čitelný a nenutí nás neustále měnit formát scény či přepínat se do prezentačního režimu. Nevýhody však převažují – alespoň podle autora této práce. Nelze se spolehnout na stejné umístění sledovače a svoji velikostí neustále překáží postavám při zmenšeném formátu scény.

3.3.7 Focení scénářů, definic bloků, scény, celého projektu

Při vytváření výukových materiálů, prezentací, anebo evaluací různých úkolů je třeba dokumentovat některé scénáře či jiné části projektu. Proto nám prostředí nabízejí možnost tyto části vyfotit (exportovat) a uložit do samostatného souboru na počítač.

Všechna prostředí umí exportovat kostýmy, zvuky, postavy samotné, i obsah scény. Zde je třeba rozlišit export jednotlivých položek a export hromadný. Jediné Snap dokáže exportovat média projektu (tedy kostýmy a zvuky) hromadně do souboru s příponou `.xml` přes nabídku pro pokročilejší uživatele s názvem „*Exportovat pouze média projektu*“. V ostatních prostředích lze exportovat zmíněný obsah jedině jednotlivě.

Prostředí narážejí na rozdíly v podpoře exportu scénářů a definic bloků. Všechna však dokáží exportovat obrázek scény. Scratch 1 a BYOB umějí vyexportovat všechny scénáře, ale už ne scénář námi vybraný. BYOB ještě umí exportovat definice vlastních bloků. Scratch 2 export scénářů vůbec nenabízí. Prostředí Snap disponuje vším zmíněným i vytknutým. Navíc dokáže exportovat hromadně celý projekt – scénu, všechny definice bloků, a užité scénáře postav.

Prostředí	Focení všech scénářů na ploše	Focení samostat. bloků či scénářů	Focení definic	Hromadné focení celého projektu
Scratch 1	ANO	NE	--	NE
BYOB	ANO	NE	NE	NE
Scratch 2	NE	NE	NE	NE
Snap	ANO	ANO	ANO	ANO

Tabulka 3.3.1 Přehled focení různých částí v prostředích

3.3.8 Krátká zmínka o dalším nastavení prostředí

Je logické, že se budou prostředí lišit i v nabízeném nastavení. V možnostech práce však není takové srovnání udělat. Zájemci musí prostředí prozkoumat či pročíst manuály.

Autor práce přesto nemůže vynechat dvě poznámky. Aby BYOB a Snap fungovala stejně jako obě verze Scratch, je důležité v nastavení zapnout možnost „*Vláknově bezpečné scénáře*“. V opačném případě by docházelo k „přetěžování“ událostí a scénáře by se tak znovu a znovu resetovali ještě než by se kompletně provedli. Pro hladší animace je potom lepší zaškrtnout možnost „*Zapnout plynulou animaci*“ jen v nastavení Snap.

3.4 Převody projektů mezi prostředími

Rozhodne-li se programátor vyzkoušet odlišnou verzi prostředí Scratch, jeho modifikaci, či alternativu v podobě jiného softwaru založeného na tomto prostředí, pravděpodobně se mu dostane nových možností v programování – jinak by do takového programovacího prostředí nepřecházel. Proto programátoři, jenž mají dokončené projekty v předešlých prostředích, takovéto projekty přirozeně převádějí do těch aktuálně užívaných za účelem aplikace nových změn, rozšířeních a funkcionalit v nich se vyskytujících.

Právě proto je zařazena i tato část o převodech projektů mezi různými prostředími a jejich verzemi do bakalářské práce. Dobrou zprávou je, že projekty většinou netřeba složitě převádět. Jiných než níže uvedených kombinací převodů projektů nelze docílit.

- **Převod projektu z Scratch 1 do BYOB**

Vraťme se na chvíli do doby, kdy ještě neexistovala prostředí Scratch 2 a Snap. Když se onehdy programátoři rozhodli si vyzkoušet anebo natrvalo přejít do prostředí BYOB, zřejmě se kromě vývoje úplně nových projektů zajímali i o převod těch starých z prostředí Scratch 1 – a to za účelem zjednodušení a rozšíření.

Jakožto modifikace nemá s otevíráním projektů z Scratch 1 žádný problém, nicméně po uložení projektu získá zdrojový soubor příponu `.ypr` a tak se zpětným převodem do Scratch 1 nelze počítat. To je logické, neboť se předpokládá, že projekt bude vybaven těmi funkcemi, kterým Scratch 1 nerozumí a tudíž by neuměl projekt správně otevřít.

- **Převod projektu z Scratch 1 do Scratch 2 a naopak**

S příchodem druhé generace Scratch vyvstala stejná otázka ohledně převodu starších projektů ze Scratch 1. Jelikož si Scratch tým zakládá na zpětné kompatibilitě s první majoritní verzí prostředí, Scratch 2 umí projekty z Scratch 1 taktéž otevřít. Zdrojový soubor převedeného a znovu uloženého projektu změní svou příponu z `.sb` na `.sb2`.

Jestliže potřebuje autor z jakéhokoliv důvodu převést projekt z prostředí Scratch 2 do svého předchůdce Scratch 1, pak je možné použít k tomu určený on-line převodník Retro Converter[1]). Stejně jako u BYOB ani zde nelze zpětně převést projekty využívající funkcionality z Scratch 2 prostředí Scratch 1 neznámé. Jde hlavně o dočasné klony, vlastní příkazy, cloudouvé proměnné a vektorovou grafiku.[44]

- **Převod projektů ze všech prostředí do prostředí Snap**

Přiznejme si, že Snap je v programovacích možnostech nejpestřejší. Proto nelze předpokládat převod v něm vytvořených projektů do ostatních – z hlediska programování zaostalejších – prostředí. Existují však možnosti jak do něj převést projekt z libovolného prostředí. To však za předpokladu, že se budeme řídit níže popsanými postupy. Ještě zmiňme, že prostředí Snap ukládá zdrojové soubory projektů s příponou `.xml`.

Začněme s převodem z BYOB do Snap, který je analogický k převodu mezi generacemi Scratch. Stačí projekt s příponou `.ypr` otevřít v Snap a o převod je automaticky postaráno. Při převodu však není konvertován text uvnitř kostýmu, který mu byl přidán uživatelem v grafickém editoru jednoho z původních prostředí.

Převést projekt z prostředí Scratch 1 do Snap můžeme dvěma způsoby:

- z Scratch 1 → BYOB → Snap; obejde se bez převodníků, ale vyžaduje BYOB
- z Scratch 1 → Scratch 2 a přes Snapin8r → Snap; vyžaduje převodník (viz níže)

Na převod projektů z prostředí Scratch 2 do prostředí Snap je nutné použít další on-line převodník Snapin8r[2]). Konvertovat nelze projekty využívající cloudové proměnné, text v kostýmu přidaný grafickým editorem; a primitivy, které prostředí Snap nenabízí.[41]

---

[1]) Je k dispozici na `http://kurt.herokuapp.com/20to14`

[2]) Tento je k dispozici na `http://hardmath123.github.io/Snapin8r/`

3.5 Modifikace, externí a interní rozšíření, knihovny

3.5.1 Modifikace prostředí a přispívání k vývoji

V této části práce bychom už měli vědět, co znamená „modifikace prostředí“. Pakliže stále někteří tápou, připomeňme znaky, kterými se modifikace vyznačují.

Zpravidla upravují a doplňují zdrojové soubory modifikovaného prostředí o další programový kód stejného programovacího jazyka. Jde především o rozšíření možností, nové funkcionality, koncepty, přidané bloky a jiné. Experimentující uživatelé, kterým nestačí dostupné možnosti původního prostředí, tyto modifikace vyhledávají a zkoušejí.

Do současnosti má Scratch 1 několik desítek modifikací. Jednou z nich je i BYOB. Scratch tým vydal pravidla k jejich tvorbě a zveřejnil zdrojové soubory prostředí na GitHub repositáři[1]) skupiny Lifelong Kindergarten Group.

Čtenář pravděpodobně nepředpokládá, že by modifikace mohla býti dále modifikovatelná. I to je samozřejmě možné. Stalo se tak dokonce i u prostředí BYOB, jehož modifikace Enchanting[2]) se zaměřuje na ovládání robotů NXT Lego Mindstorms.[21]

Scratch 2 se na rozdíl od ostatních vydal jinou cestou. Oficiálně nepodporuje tvorbu vlastních modifikací. Zdrojové soubory sice byly zveřejněny na GitHub repositáři[3]), ale to spíše za účelem podílení se na vývoji a opravách hlášených chyb dobrovolnými programátory, kteří si zdrojové kódy stáhnou, upraví, a zpětně nahrají ke schválení. Scratch 2 preferuje cestu interních rozšíření, které si popíšeme až za chvíli.

Snap je možné modifikovat stažením zdrojových souborů v prostředí výběrem možnosti *zdrojové soubory* po kliknutí na logo Snap v levém horním rohu. Zajímavou modifikací tohoto prostředí je BLOck Programming[4]), jenž využívá grafické prvky prostředí k programování v jiných jazycích. Původní primitivy jsou skryté a nahrazené vlastní sadou bloků, která je přizpůsobena potřebám konkretního programovacího jazyka, přičemž lze dodat další sady (podporu) pro ostatní programovací jazyky. V současné době umožňuje programovat například v jazyce C či Logo.[25]

• *Poznámka: Pokud má čtenář v plánu zkusit si vytvořit nějakou modifikaci, autor této práce doporučuje přečíst licenční podmínky oněch prostředí. Totéž platí i pro rozšíření.*

3.5.2 Interní a externí rozšíření

Úpravy grafického uživatelského rozhraní, doplnění nabídek prostředí, či nové kategorie bloků jsou rozsáhlými rozšířeními vyžadující zásah do zdrojových kódů prostředí, jenž povětšinou skončí vznikem další modifikace. Přesto však existují taková rozšíření, která mohou existovat i v tom původním – nemodifikovaném prostředí. Bylo by hloupé, kdyby si vývojáři neponechali nástroj, kterým by tato méně významná rozšíření nainstalovala uživatelům do prostředí ve chvíli, kdy si o ně různými způsoby zažádají.

Hovoříme nyní o interních a externích rozšířeních, tj. o rozšířeních, která rozšiřují prostředí o další funkcionality, jen není k jejich zavedení potřeba prostředí složitě upravovat. Obsah, který přidávají, přesahuje základní rámec prostředí. Též mají minimální dopad na jeho uživatelské rozhraní. Využívají se především k „propojením s fyzickým světem“[5]), kdy se potřebujeme spojit s některými zařízeními – převážně robotickými stavebnicemi. Jejich nainstalování do projektu je vždy dobrovolným krokem a instalují se pro každý projekt zvlášť.

• Interní rozšíření

Interní rozšíření jsou zabudovaná přímo do prostředí samotnými vývojáři. Programátor projektu je může okamžitě využít a nemusí je složitě shánět a ani doinstalovávat žádný

[1]) https://github.com/LLK/Scratch_1.4
[2]) http://enchanting.robotclub.ab.ca/tiki-index.php
[3]) https://github.com/LLK/scratch-flash
[4]) http://www.blocklanguages.org/
[5]) Vnejším jak ve smyslu „mimo počítač“ tak i ve smyslu „mimo prostředí, avšak stále v počítači“.

pomocný software anebo ovladač. Dodávají zpravidla nové bloky rozšiřující možnosti prostředí. Jde-li o rozšíření umožňující komunikaci s hardwarovým zařízením, pak jej stačí většinou propojit s počítačem přes USB kabel a prostředí se s zařízením automaticky propojí. Interní rozšíření jsou povětšinou skryta a tak svou přítomností neobtěžují uživatele, kteří je nevyužívají. Přes různé nabídky lišící se napříč prostředími je možné tato rozšíření v prostředích zobrazit.

Scratch a BYOB přes nabídku „Editovat“ → „Zobrazit motor. bloky“ přidají bloky do kategorie pohyb umožňující ovládat zařízení LEGO WeDo Construction Kit, což je stavebnice z Lega, se kterou můžeme složit jednoduchého robota, propojit ho s počítačem a ovládat skrze přidané bloky. Tatáž prostředí podporují i PicoBoard – malou základní desku se senzory, na jejichž hodnoty se v prostředích dokážeme napojit a skrze ně ovlivňovat odehrávání příběhu na scéně.

Na zjišťování hodnot LEGO WeDo a PicoBoard slouží v prostředích další rozšiřující bloky (hodnota senzoru) a <senzor>, které jsou v kategorii vnímání vždy přítomné. Nejsou automaticky skryté a ani je nelze odstranit z projektu.

Scratch 2 interní rozšíření nabízí pod kategorií bloky po kliknutí na tlačítko „Přidej rozšíření“ zobrazující knihovnu dostupných rozšíření, ze kterých lze jedno po druhém instalovat do editovaného projektu. Aktuálně existuje vylepšená podpora u LEGO WeDo oproti svému předchůdci a stejně tak podpora pro PicoBoard.

Scratch tým chce do budoucna umožnit uživatelům tvorbu vlastních interních rozšíření, jimiž by doplnili knihovnu o sady specifických bloků, ke kterým by měla přístup Scratch komunita v každém projektu.[1]) Rádi by také umožnili přidat rozšíření spojené s webovými službami, aby mohlo prostředí Scratch 2 například převést text na řeč – což v současné době neumí. Takové rozšíření by nějakým blokem text z parametru odeslalo službě převádějící text na řeč a zpětnou integrací by hlas v prostředí přehrálo.

Mezi prostředí nabízející interní rozšíření se řadí i Snap. Jeho funkce (JavaScript funkce) umí vykonat programový kód jazyka JavaScript. Skrze ni je možné nejenom rozšířit možnosti prostředí, ale dokonce některé jeho části upravit, neboť se lze dostat ke zdrojovým souborům prostředí. Funkce (JavaScript funkce) netřeba v projektu odhalovat – je vždy přítomna.

- **Externí rozšíření**

Externí rozšíření nejsou přímo zabudovaná do prostředí jejich vývojáři a je proto nutné je sehnat a ručně naimportovat. Některá rozšíření mohou požadovat instalaci podpůrného software a ovladačů. V externích rozšířeních dominuje především prostředí Snap.

Microsoft Kinect je zařízení k herním konzolím Xbox 360 a Xbox One umožňující ovládat ona zařízení hlasem, gesty, a také hrát hry, u nichž je možné použít lidské tělo jakožto ovladač. Stephen Howell vyvinul externí rozšíření propojující toto zařízení s prostředím Scratch 1 a pojmenoval ho Kinect2Scratch.[2]) Abychom však mohli propojit toto zařízení s prostředím, musíme nainstalovat vyvinuté ovladače, jenž fungují pouze na operačních systémech Windows 7 a 8. Kinect2Scratch propojíme i s BYOB.

Snap na svých oficiálních stránkách nabízí mnoho externích rozšíření propojující jej s: ovladačem WiiMote k herní konzoli Nintendo Wii, robotickou koulí Orbotix Sphero, robotickou stavebnicí Lego Mindstorms NXT, dále roboty Finch, Hummingbird, a Parallax S2. Následuje rozšíření k zařízení LEAP Motion detekující pohyby rukou a prsty, syntezátor řeči (převod textu na řeč), a mini počítači Arduino. Většina z rozšíření vyžadují nainstalovanou distribuci programovacího jazyka Python, ve kterém se spouští a provádí klíčové scripty. Doplňující bloky ovládající propojené zařízení se importují do prostředí souborem XML, který obsahuje jejich definice v podobě vlastních bloků. Návod k propojení rozšíření s prostředím je povětšinou uveden na jejich webové stránce.

---

[1]) Zdá se však, že ani tato možnost není v současné době (Květen 2015) k dispozici. Vzhledem k zvyšující se popularitě tohoto prostředí je s podivem, že stále nabízí pouze dvě hardwarová rozšíření.

[2]) Na žádost autora rozšíření je přiložen odkaz na jeho webovou stránku: http://scratch.saorog.com

3.6 Propojení prostředí a týmový vývoj projektu

- **Propojování prostředí**

Scratch 1 a BYOB dokáží vzájemně propojit vícero instancí těchto prostředí.[1]) Dojde-li k jejich propojení, a přesný postup si popíšeme níže, budou všechny propojené instance sdílet některé své části – a je úplně jedno, jestli propojujeme prostředí s novými, ještě neuloženými, projekty anebo máme-li nějaké již jednou uložené znovu otevřené. Propojit instance různých prostředí můžeme na jednom počítači, ale i na dvou a více různých počítačích – mají-li k sobě, ať už přes místní síť či jiné virtuální sítě, přístup.

Propojování vývojáři nazývají slovem „Mesh“. V horní nabídce „Zveřejnit“ se vyskytují „Host Mesh“ a „Join Mesh“, které k propojení prostředí používáme. Původně jsou tyto možnosti v Scratch 1 skryty a zájemci musejí upravit část zdrojového kódu přes skryté uživatelské rozhraní určené pokročilejším uživatelům.[2]) Poté se v něm tyto nabídky zobrazí v kombinaci se stisknutou klávesou `Shift`. BYOB nabídky automaticky odkrývá a nenutí tak uživatele tento složitý proces nastavování provádět.

Jeden z účastníku propojovaných prostředí se musí ujmout role hostujícího, na nějž se ostatní aktéři se spuštěnými prostředími připojí ze svých počítačů. Hostující počítač jest hlavním uzlem v relaci spojení, jímž se stane po kliknutí na tlačítko „Host Mesh“, které zároveň zobrazí jeho IP adresu, jenž ostatní zadají do textového pole zobrazeného po kliknutí na „Join Mesh“, aby došlo k nalezení hostujícího počítače se kterým se chtějí spojit. Propojujeme-li instance prostředí jen na jednom počítači, nacházíme se v místní síti (LAN) a propojení není překážkou. Propojujeme-li však počítače v různých sítích, je vhodné použít například software LogMeIn Hamachi, který vytváří privátní virtuální síť v síti internet simulující onu lokální síť potřebnou ke spojení instancí.

Propojená prostředí sdílí rozesílání zpráv přes příkazy `[rozešli všem]` anebo `[rozešli všem a čekej]`, na něž mohou postavy reagovat blokem události `[po přijetí zprávy]`. Když jedna z propojených instancí rozešle zprávu svým postavám, zbylé instance tutéž zprávu jejich postavám také rozešlou. Přes `(hodnota senzoru)` můžeme ještě přistoupit k proměnným všech postav, jež existují v nějaké z propojených instancí. Konečně uvažme několik příkladů, ve kterých bychom mohli propojení využít.

Někteří již v minulosti vyvíjeli hru, jež byla určena pro dva a více hráčů. Jednalo se buď o hru tahovou, kdy se hráči střídali po provedení tahu (jako například v šachách), anebo o hru odehrávající se v reálném čase, ve které se různé postavy ovládali různými klávesami (první hráč většinou ovládá postavu šipkami, druhý třeba písmeny WSAD) a jejíž dění tak ovlivňovali všichni aktéři najednou. Přesto však všichni její hráči museli být doposud přítomni u stejného počítače, skrze jehož vstupní periferie projekt ovládali.

Připravíme-li takovou hru na propojení s dalšími prostředími, můžeme ji teoreticky hrát na vícero počítačích. Uvažme hru odehrávající se v reálném čase. V tomto případě bychom asi měli vytvořit dvě verze hry – první projekt by sloužil hostujícímu počítači a druhý ostatním hráčům napojujících se z jiných počítačů. Projekt hostujícího počítače by sloužil jakožto centrální a jakékoliv změny ve hře by přijímal od ostatních instancí. Zároveň by je sám úkoloval rozesíláním dalších zpráv, aby sjednotil zobrazovaný obsah hry na scéně ve všech prostředích. Každý hráč by zastal jednu z postav a po stisknutí jejích ovládacích kláves by se odeslala zpráva o detekci jejího pohybu. Projekt otevřený na hostujícím počítači by tuto informaci zaznamenal a odeslal všem ostatním instancím žádost o aktualizaci pozice postavy. Pokud by bylo cílem hry kupříkladu sebrat nějaký předmět, pak by centrální projekt na hostujícím počítači vyhodnotil, která z postav se dotkla předmětu jako první a též by o tom informoval zprávou napojené instance.

Zmiňme další příklad, který bychom mohli využít třeba ve výuce při hodině informatiky. Učitel by zaujal pozici hostujícího počítače a ostatní žáci by se na něj napojili.

[1]) Instanci prostředí představuje každé spuštěné prostředí v počítači. Instancemi pak máme na mysli dvě a více spuštěných prostředí. Ačkoliv Scratch 1 vyžaduje instalaci, lze jej společně s BYOB spustit libovolně krát i na jednom a tomtéž počítači a tak instance v množném čísle mohou představovat například čtyřikrát spuštěné prostředí Scratch 1 s čtyřmi různými či naprosto identickými otevřenými projekty.

[2]) http://wiki.scratch.mit.edu/wiki/Mesh

Domluví se, že po odeslání zprávy `{vykresli_dům}` žákův projekt vykreslí želví grafikou obyčejný panelový dům. Požadovaný počet pater a oken budou určovat proměnné `(počet_pater)` a `(počet_oken)`, které definuje učitel ve svém projektu, k jejichž hodnotám žáci přistoupí funkcí `(hodnota senzoru)`. Po ukončení práce na projektu učitel odešle zmíněnou zprávu z prostředí svého počítače a následně zkontroluje vykreslený obrazec na monitorech jednotlivých žáků ve třídě. Hodnoty proměnných by sám měnil.

Scratch 2 propojování prostředí nepodporuje a to se asi ani v budoucnu nezmění. Výhody plynoucí z propojování částečně nahrazují cloudové proměnné (a v budoucnu i cloudové seznamy), kterými můžeme získat informace o aktuálním stavu projektu ostatních uživatelů. Cloudové proměnné však sdílejí všichni návštěvníci projektu a nelze si vybrat skupinu přátel, s nimiž bychom si onu hru pro více hráčů zahráli. Snap touto funkcionalitou také prozatím nedisponuje, avšak vývojář Brian Harvey informoval, že by časem uvítal i ji implementovanou v prostředí.

- **Týmový vývoj projektu**

Sdílení zdrojových kódů vytvářeného programu s ostatními programátory je v dnešním programátorském světě klíčovou částí vývoje. Myšlenka na týmový vývoj tak neunikla ani vývojářům prostředí. Týmovým vývojem rozumějme možnost centralizovaného přístupu k projektu více osobami, v lepším případě ještě schopnost editovat projekt těmito osobami ve stejném čase najednou – stejně jako to známe třeba u dokumentů Google.

Scratch 1 a BYOB týmový vývoj jednoho projektu nepodporují. U Scratch 2 bylo již v období jeho oznámení rozhodnuto, že týmový vývoj – alespoň zatím – nebude. Jediní vývojáři Snap mají zájem se na implementaci této funkcionality podílet.

Vývojáři chtějí nejprve vyřešit vlastnění jednoho projektu více uživateli, kteří společně utvoří tým o velikosti prozatím dvou lidí. Při návrhu projektu si mohou určit role (programátor postav, programátor scény, grafik, zvukař. . .) a editovat jej v rozdílném čase. Taktéž budou osvobozeni od přeposílání si obsahu (obrázků, zvuků, scénářů atd.).

Poté je nutné vyřešit otázku, jak a jestli vůbec editovat projekt více lidmi najednou. Zde je prostor pro různá řešení. Například by si mohli vyměňovat právo na editaci skrze příslušné tlačítko. Nebo by se – v ideálním případě – každá programátorem editovaná postava uzamkla ostatním členům týmu.

3.6.1 Nabízený obsah ze zabudovaných knihoven

Projekty se většinou neobejdou bez obrázků na pozadí, kostýmů pro postavy, různých zvuků, anebo již naprogramovaných scénářů či vlastních bloků z jiných projektů. Sehnat takové materiály není jednoduché a to nemluvě o autorských právech. Prostředí nám usnadňují hledání nabízeným obsahem, který je uložen v zabudovaných knihovnách.

Uživatelé zřejmě vědí, jak se do knihoven ostatních prostředí dostanou. Část knihoven Snap je však ukryta v nabídce zobrazené tlačítkem v levém horním rohu.

- **Knihovna obrázků – kostýmy a pozadí**

Jelikož Scratch 1 a BYOB podporují pouze bitmapovou grafiku, nabízejí proto ve svých grafických knihovnách tentýž typ kostýmů a pozadí. Scratch 2 krom bitmapových kostýmů nabízí i několik vektorových. Některé bitmapy ze Scratch 1 byly dokonce předělány do novější podoby ve vektorech. Snap, leč podporuje i vektory, nabízí jen bitmapy.

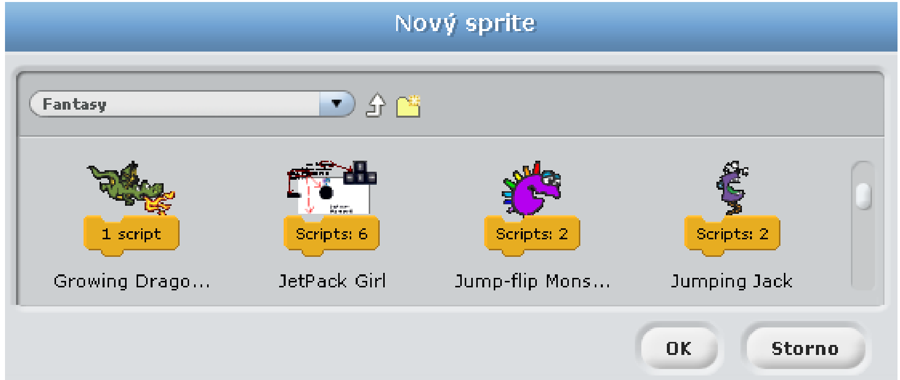
- **Knihovna zvuků**

Tato knihovna je dostupná ve všech prostředích a liší se pouze nabídkou zvuků.

- **Knihovna kostýmů s přidanými scénáři**

Když se programátor rozhodne přidat novou postavu, musí ji buď nakreslit či přiřadit nějaký dostupný kostým. Zvolí-li si druhý způsob, může naimportovat některé kostýmy společně se scénáři „vdechující“ postavě život, čímž ji umožní se projevit. Tato možnost je k dispozici jen u prostředích Scratch 1 a BYOB. Obrázek 3.6.1 prozradí více.[1])

[1]) Slovo „*Script*“ na obrázku 3.6.1 znamená v překladu „*Scénář*“.



Obrázek 3.6.1 Knihovna kostýmů se scénáři při vytváření nové postavy

- **Ukázkové projekty**

Začínajícím programátorům přijdou vhod ukázkové projekty k inspiraci a nabití dalších zkušeností. Scratch 1 vládne širokou škálou projektů.[1]) BYOB tyto ukázkové projekty z prostředí Scratch 1 nahrazuje svými vlastními, avšak v daleko menším počtu.

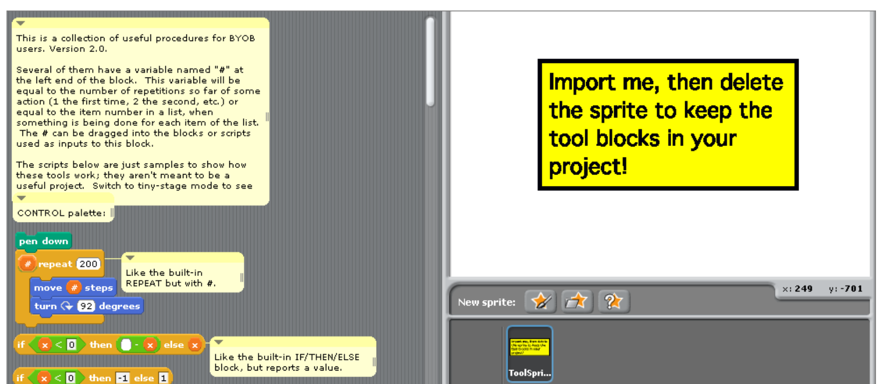
Přímo ve Scratch 2 nejsou ukázkové projekty přítomny. To ale neznamená, že neexistují. Je možné je spustit z příslušné stránky[2]) v on-line verzi anebo stáhnout[3]) přímo do počítače a poté je postupně importovat do off-line verze prostředí.

Prostředí Snap přímo v sobě obsahuje do současnosti na deset *ukázek*. Zobrazíme je k výběru přepnutím na záložku „*Ukázky*" při otevírání projektů v dialogovém okně. Kromě těchto ukázek existují i *dema*, jež je možné stáhnout z oficiálních stránek.[4])

- **Nástrojová sada – demonstrační bloky prostředí**

Pro nové uživatele BYOB a Snap je v těchto prostředích připravena sada bloků, kterou mohou do svých projektů importovat. Vývojáři tyto bloky nazvali „nástroji", ale jednodušší je vnímat je jako *reprezentační bloky* demonstrující schopnosti prostředí. Každý takový blok byl vytvořen uvnitř prostředí (není primitivem) a je proto možné do něj nahlédnout, inspirovat se, popřípadě jej poupravit přes editor bloků. Jsou naprogramovány v angličtině – tj. s anglickými popisky, proměnnými a komentáři.

Prostředí BYOB má tyto bloky uloženy v postavě s názvem „Tool Sprite" (překládejme jako „Postava s nástroji"), kterou je třeba naimportovat do projektu. Nachází se v knihovně s kostýmy. Jelikož obsahuje bloky pro všechny postavy, můžeme ji smazat bez obavy z jejich ztráty. Ztratíme však scénáře v ní uložené, které představují importované bloky různých v příkladech – většinou algoritmicky zaměřených. Nutno dodat, že definice bloků neobsahují dokumentační komentáře.



Obrázek 3.6.2 Reprezen. sada importovaná otevřením postavy „ToolSprite"

[1]) Čtenář se možná setkal s názvem „*Starter projects*".
[2]) http://scratch.mit.edu/starter_projects/
[3]) http://scratch.mit.edu/scratch2download/
[4]) http://snap.berkeley.edu/snapsource/demo/

V prostředí Snap je tato demonstrační sada bloků umístěna mezi ostatními knihovnami a to pod názvem „Oficiální nástroje“. Při importu nástrojů se žádná postava do projektu nepřidá, ani žádné příklady demonstrující jejich užití v praxi – importují se pouze bloky. Dokumentační komentáře jsou mimochodem přiřazeny každému bloku.

Sada nástrojů v prostředí Snap obsahuje bloky z kategorií `ovládání`, `operátory` a `seznamy`. Jakmile budou doplněny další funkcionality (především s příchodem verze 4.1), můžeme očekávat doplnění této sady o chybějící bloky. V prostředí BYOB sada obsahuje ještě bloky z kategorií `pohyb`, `vzhled`, `vnímání` a `proměnné`. Naimportované bloky se však mezi těmito oběma prostředími nepatrně odlišují.

- **Rozšiřující sady bloků**

Snap ještě nabízí další bloky s užitečnými algoritmy. Jde o *rozšiřující sady* doplňující vždy jednu kategorii o další bloky. Využití najdou jak při výuce programování tak při vývoji projektu. Jsou dokonce vlastními bloky, tudíž lze studovat jejich definici, ale jsou naprogramovány v angličtině. Následuje výčet rozšiřujících sad s přeloženými názvy:

- Iterace, kompozice .... rozšiřují kategorii `ovládání`
- Bloky pro práci se seznamy .... `seznamy`
- Proudy („líně“ vyhodnocované seznamy) .... `seznamy`
- Funkce a podmínky s vícenásobnými parametry .... `operátory`
- Bloky pro práci se slovy a větami .... `operátory`
- Více-větvené podmíněné struktury (přepínač) .... `ovládání`

- **Grafické a funkční provedení knihoven**

Zmiňme se ještě o rozdílech v provedení knihoven. Nejlépe je na tom prostředí Scratch 2, které v knihovně kostýmů zobrazuje náhledy a v knihovně zvuků je umožňuje přehrát a zastavit. Scratch a BYOB náhledy kostýmů též zobrazují, ale přehraje-li si uživatel zvuk v knihovně zvuků ještě před jeho importem, nemá možnost jej zastavit. Musí tak čekat, než se mnohdy několika sekundové zvukové stopy přehrají. Nejhorší provedení knihoven má Snap, jež v knihovnách obsahuje pouze názvy bez náhledu kostýmů a bez přehrávání ukázky zvuku, takže se uživatel předem nedoví, co přesně importuje. Přesto dle vývojářů by alespoň knihovna kostýmů měla být podobná té ze Scratch 2.[56]

Prostředí	Kostýmy bitm./vekt.	Postavy se scénáři	Ukázkové projekty	Reprezen. sada bloků	Rozšiřující sady bloků
Scratch 1	bitmapové	ANO	ANO	NE	NE
BYOB	bitmapové	ANO	ANO	ANO	NE
Scratch 2	obojí	NE	ANO	NE	NE
Snap	bitmapové	NE	ANO	ANO	ANO

Tabulka 3.6.1 Shrnutí nabízeného obsahu v knihovnách prostředí

3.7 Dostupné lokalizace

Další ocenitelnou vlastností prostředí je jazyková vybavenost. Kromě BYOB disponují všechna prostředí relativně širokou podporou překladů a proto je můžeme považovat za vícejazyčná. Dostupné překlady nazýváme lokalizacemi. Některé lokalizace prostředí BYOB nejsou součástí distribuce a proto je nutné je stáhnout na oficiálních stránkách.[1])

Následující tabulka zobrazuje počet překladů napříč prostředími. Jelikož čtenářem této bakalářské práce může být i člověk se slovenským mateřským jazykem, zařaďme do přehledu kromě podpory češtiny i jazyk jemu nejbližší. Všimněme si též vzrůstajícího počtu překladů s vývojem obou hlavních prostředí.

[1]) Šest lokalizací je přímo v prostředí a dalších šest je dostupných na internetu. Ty jsou k dispozici na oficiálních stránkách – `http://snap.berkeley.edu/old-byob.html`.

Prostředí	Lokalizací	Čeština	Slovenština
Scratch 1	50	ANO	ANO
BYOB	12	NE	ke stažení
Scratch 2	68	ANO	ANO
Snap	29	ANO	NE

Tabulka 3.7.1 Přehled dostupných lokalizací napříč prostředími

- **Částečný překlad prostředí BYOB**

Přestože z předchozí tabulky víme, že prostředí BYOB vůbec nedisponuje českou lokalizací, lze i přesto dosáhnout alespoň částečného překladu. Jelikož je BYOB modifikací prostředí Scratch 1, používá taktéž identické lokalizační soubory.

Nahrajeme-li libovolnou nabízenou lokalizaci z příslušného adresáře původního prostředí Scratch 1 do stejného adresáře BYOB, pak bude tato lokalizace nabízena i tímto prostředím. Stejné položky, které obě prostředí sdílí, budou přeloženy. Toto ovšem vyžaduje přístup k lokalizačním souborům Scratch 1, které získáme v příslušném adresáři po instalaci tohoto prostředí na počítač. Teoreticky i BYOB disponuje padesáti lokalizacemi, ale jen šest z nich nabídne doplňující překlad nových popisků, nabídek a bloků.

3.8 Závěr porovnání

Po celkovém porovnání prostředí je možné konstatovat, že s jejich postupným vývojem se neustále vylepšují a tudíž je na místě nejenom ocenit práci, ale také pochválit všechny osoby na jejich vývoji se podílející. Nyní však nastal čas zhodnotit možnosti prostředí dle zadaných kritérií závěrečnou srovnávací tabulkou 3.8.1. Ještě zmiňme, že nelze jasně určit vítěze porovnání, jelikož každé prostředí vyniká v různých oblastech.

Autor této práce si zavedl vlastní vyhodnocovací pravidla k určení vítězného prostředí v každém porovnávaném kritériu. Každé kritérium musí mít prostředí s alespoň jednou hvězdičkou značící minimální podporu v kritériu. Je-li nějaké prostředí na vyšší úrovni v podpoře, dostane hvězdičku navíc – atd. pro další prostředí. Teoreticky mohou mít všechna jednu hvězdičku, nabízela-li by stejné možnosti. Jestli má prostředí tři nebo čtyři hvězdičky není podstatné. Důležité je, zdali jich má více než jiné prostředí.

Kritérium	Scratch 1	BYOB	Scratch 2	Snap	Vítěz
Programování	*	***	**	****	Snap
Grafika, zvuk, multimédia	*	*	***	**	Scratch 2
Komfort a přívětivost	*	**	***	**	Scratch 2
Publikování projektů	**	*	***	*	Scratch 2
Převody projektů	*	**	**	***	Snap
Modif., rozšíření, knihovny	***	**	*	****	Snap
Lokalizace	***	*	****	**	Scratch 2

Tabulka 3.8.1 Závěrečná srovnávací tabulka definující vítěze každého kritéria

Pro mladší programátory, ať už žáky druhého stupně základních škol či nadšence, je určeno spíše prostředí Scratch 2, jehož programovací paleta není příliš pestrá, přesto září v oblastech práce s grafikou, zvuky, a multimédii; v komfortu a přívětivosti, publikování projektů na k tomu určené Scratch síti a komunitou uživatelů, a nabízených lokalizací.

Začínajícím – ale už z ostatních prostředích zkušenějším – programátorům, dychtíc po rozšíření svých programátorských obzorů, je bezpochyby určeno prostředí Snap. Dominuje v oblasti programování, možností k převodů projektů z jiných prostředí, a též v otázce rozšíření a knihoven. Jakmile jeho uživatelé dosáhnou mistrovství v nabízených funkcionalitách, konceptech, a principech; mohou se přesunout do jiných jazyků, z nichž se nabízí třeba JavaScript (kvůli totožnému OOP) či funkcionální Scheme.

Kapitola 4

Využití nových programovacích konceptů a funkcionalit prostředí Snap v příkladech

Nyní, poté co jsme si představili prostředí a provedli jejich srovnání, se můžeme vrhnout na různé příklady a projekty demonstrující sílu (programovací funkcionality, koncepty, či principy) prostředí Snap vůči svému současnému konkurentovi Scratch 2. Ne všechny možnosti Snap však budou ukázány. Některé totiž stále nejsou implementovány v současné verzi prostředí a další by přesahovaly rámec této bak. práce. Čtenář přesto získá základní informace o nabízených možnostech Snap, které pak sám může zkombinovat do ještě sofistikovanějších celků k vytvoření mnohem důmyslnějších funkcionalit.

Předpokladem k pochopení této kapitoly je prostudování si dodatků A, B, C a D. Též je vhodné si přečíst předchozí kapitolu porovnávající prostředí, neboť některé již popsané principy nebudou v této kapitole nadbytečně opakovány. Pokud si čtenář nebude jistý, měl by se vrátit do druhé kapitoly a osvěžit si v ní paměť.

Férové je přiznat, že některé algoritmické ukázky jsou známy z různých knižních publikací, používané napříč univerzitami, a některé z nich jsou přítomny i v knihovnách či manuálu prostředí Snap. Jsou však na demonstraci popisované problematiky ideální.

V rámci možností se autor snažil též vymýšlet takové příklady, které by procvičovali všechny kategorie prostředí (`pero`, `zvuky`, `vzhled`, `ostatní`...) v různém pořadí.

Některé projekty byly naprogramovány v obou prostředích Snap i Scratch 2. Proto bude někdy nutné mít obě prostředí k dispozici – popis k tomu bude směřovat. Mnohdy ukazujeme řešení v Scratch 2 k porovnání s lepším, čitelnějším, či jednodušším řešením v Snap. Projekt občas postupně vylepšujeme dalším projektem. Čtenáři tištěné verze musejí navštívit dodatek G, aby si mohli otevřít jednotlivé projekty. Projekty prostředí Snap je nejlepší otevírat v internetovém prohlížeči Google Chrome, neboť ten dosahuje dobrého výkonu, zobrazuje čitelně bloky, a podporuje vektorovou grafiku.

Čtenář by též neměl pozapomenout na upravenou češtinu autora této práce, pakliže chce mít bloky přeložené stejně jako je tomu na uvedených obrázcích. V takovém případě nechť si přečte dodatek F. Zároveň by neměl zapomenou přes nastavení zapnout střídavé barvy (zbarvování scénáře), není-li tomu tak v prostředí automaticky.

Protože se při popisu různých příkladů či projektů odkazujeme na konkrétní vlastní bloky, jež jsou v obou prostředích naprogramovány, nabývají sice stejného názvu, ale odlišné kategorie. Tudíž na blok s názvem „můj blok“ se budeme odkazovat přes `[můj blok]` značící, že mluvíme k prostředí Scratch 2, kdežto přes `[můj blok]` (či jiné barvy) značíme, že mluvíme k projektu z prostředí Snap. Nezapomeňme si všimnout rozdílu mezi `[můj blok]` a `[můj blok]`. První blok je z kategorie `bloky` a druhý z `vzhled`.

Závěrem je třeba upozornit, že projekty z prostředí Snap nemusí v budoucnu fungovat, přestože se povede je bez chyby otevřít. Navzdory tomu, že bylo prostředí oficiálně vydáno a ukončilo vývojovou fázi beta, zahajuje se v současnosti vývoj další verze 4.1, jež může přinést pár zásadních změn ústících v nefunkčnost projektů (viz GitHub).

4.1 Vlastní funkce a podmínky, rozbalovací nabídka parametrů, imitace konstant

4.1.1 Vlastní funkce

Další z typů bloků, jenž uživatelé Scratch 2 nemohou zatím vytvářet, jsou funkce. Jejich podstatou je provést nějaký výpočet a vždy vrátit hodnotu libovolného datového typu. Bloky funkcí jsou znázorněny se zaobalenými rohy a na rozdíl od příkazů se skládají do sebe namísto pod sebe. K předčasnému ukončení vykonávání scénáře vlastního příkazu jsme dosud užívali blok [zastav], kdežto u funkcí použijeme příkaz [vrať] (viz níže).

Obrázek 4.1.1 Ukázka několika primitivních funkcí z prostředí Snap.

Stejně jako u příkazů i funkce mohou mít libovolný počet parametrů jakéhokoliv typu. Na obrázku 4.1.1 vidíme mezi primitivními funkcemi jak funkce bezparametrické, tak i funkci s jedním anebo dvěma parametry. Bezparametrické funkce si vrácenou hodnotu zjišťují samostatně, kdežto parametrické funkce využívají argumenty těchto parametrů k získání výsledku. Výsledek se může určit přímo (zadáním hodnoty do příkazu [vrať]) anebo získáním výsledku voláním jiné funkce. Vracet lze i hodnoty proměnných.

Aby došlo k vrácení hodnoty, musíme to ve scénáři funkce přikázat. Vracení hodnoty se může odehrát v různých částech scénáře (stejně jako lze užít příkaz [zastav] na více místech). Příkaz [vrať] ukončí vykonávání scénáře a vrátí konkrétní hodnotu libovolného datového typu. Příkaz [vrať] tedy:

- ukončí vykonávání scénáře funkce, vrátí se na místo, odkud byla ona funkce volána, a hodnotu, se kterou se vrací, nahradí za volání funkce
- může být užit na více místech ve scénáři
- vrací se s hodnotou libovolného datového typu
- objevuje se v něm rekurzivní volání u rekurzivně založených funkcí

Přejděme raději k ukázkám, v nichž se zaměříme převážně na příkaz [vrať], neboť je důležité pochopit jeho schopnosti. Začněme prvním projektem, který byl naprogramován v obou prostředích s identickým vzhledem. Popišme si zjednodušeně cíl projektu.

Každou sekundu se smažou stopy pera na scéně a znovu se vykreslí želví grafikou analogové hodiny zobrazující aktuální čas. Dále jednou za minutu dojde k aktualizování informací ve sledovači proměnné, jenž zobrazuje aktuální datum a den týdne. Čtenář nechť projekt spustí a prohlédne si jeho scénu. Hlavní scénář vykreslující hodiny jednou za sekundu je zachycen na obrázku 4.1.2.

```
po kliknutí na [zelená vlajka]
opakuj dokola
  smaž
  nakresli ciferník délka čáry: 10 vzdálenost od středu: 125
  nakresli ručičku ú: (aktuální sekunda * 360 / 60) d: 110 tl: 1
  nakresli ručičku ú: (aktuální minuta * 360 / 60) d: 110 tl: 3
  nakresli ručičku ú: ((aktuální hodin * 360 / 12) + (aktuální minuta * 360 / 12 / 60)) d: 75 tl: 5
  hraj zvuk tik
  čekej 1 sekund
```

Obrázek 4.1.2 Scénář vykreslující každou sekundu aktuální čas želví grafikou v prostředí Scratch 2

SCRATCH 2 PROJEKT: Ručičkové hodiny s akt. časem, datem a dnem týdne #1

Příkaz [nakresli ciferník] není pro nás důležitý a je proto určen zájemcům o algoritmus vykreslující modrý ciferník hodin želví grafikou. Na zmíněném obrázku můžeme dále vidět příkaz [nakresli ručičku], jenž využíváme k vykreslení vteřinové, minutové, a hodinové ručičky. Jelikož jsou hodiny jednobarevné, nezáleží na pořadí kreslení ručiček. Vyjma úhlu naklonění ručičky tento blok vyžaduje i délku a tloušťku její čáry, jež vykreslí vždy od středu ciferníku. Podstatné pro nás je nyní zaměřit se na výpočet úhlů jednotlivých ručiček.

Proveďme trochu matematiky k porozumění číslům figurujícím ve výpočtech. Prohlédneme-li si libovolné analogové hodiny, zjistíme, že mají ciferník rozdělen na šedesát dílků. Dvanáct z nich jsou vyznačeny silnější čarou reprezentující celé hodiny. Ostatní reprezentují vteřiny i minuty. Abychom získali úhel jednotlivých ručiček, musíme využít počet těchto dílků při výpočtu. Nezapomeňme, že ačkoliv má den čtyřiadvacet hodin, ciferník analogových ručiček ukazuje pouze hodin dvanáct – což musíme zohlednit.

Vydělíme-li počet hodin, minut, či vteřin zobrazených na ciferníku počtem stupňů kruhu, pak získáme úhel, o který musíme ručičku posunout vždy se změnou hodnoty jí reprezentovanou časovou jednotkou. Takto například první výskyt příkazu [nakresli ručičku] z obrázku 4.1.2 zjišťuje úhel vteřinové ručičky. Aktuální počet sekund vynásobí hodnotou odpovídající úhlu jednoho dílku ciferníku. Pro minutu aplikujeme stejný výpočet, jen u funkce (aktuální) musíme změnit možnost na {minuta}. Analogický postup platí i pro zjištění úhlu hodinové ručičky. Tím její výpočet ale zdaleka neskončí.

Vypočteme-li úhel hodinové ručičky stejně jako u ručiček předešlých, pak tyto analogové hodiny nebudou příliš sdílné. Hodinová ručička by totiž stála vždy a pouze na jednom z dvanácti k tomu určených dílků. Většina analogových hodin ale posouvá hodinovou ručičku v závislosti na minutách, takže čím více minut v jedné hodině uběhne, tím blíže k dalšímu hodinovému dílku se hodinová ručička přibližuje. Proto se u třetího příkazu [nakresli ručičku] používá složitější výpočet. Výpočtem {360 / 12} získáme kruhovou výseč odpovídající jedné hodině na ciferníku, tu dále vydělíme šedesáti k získání úhlu jedné minuty na této výseči, a vynásobíme výsledek s aktuální minutou, kterou přičteme k úhlu aktuální hodiny. Pohyb ručičky tak ovlivní i počet minut.

Zřejmě jsme si všimli, že k pochopení scénáře z obrázku 4.1.2 je zapotřebí znalostí k výpočtu úhlů jednotlivých ručiček. Bez funkce (aktuální) a její zvolené možnosti bychom stěží odhadli, který z příkazů [nakresli ručičku] vykresluje tu či onu ručičku – tedy pakliže nerozumíme výpočtům. Ty sice mohli být v jednodušší podobě, ale ve funkcích by se objevila nicneříkající čísla, která by k pochopení scénáře nepřispěla. Podívejme se, jak si můžeme pomoci tvorbou vlastních funkcí v prostředí Snap. Řešení stejného scénáře je zobrazeno na obrázku 4.1.3.

Obrázek 4.1.3 Scénář vykreslující každou sekundu aktuální čas želví grafikou v prostředí Snap

Vytvořeny byly funkce (úhel vteřinové ručičky), (úhel minutové ručičky), a (úhel hodinové ručičky), které ony složité výpočty skrývají uvnitř své definice. Výsledek výpočtu vracejí příkazem [vrať]. Scénář se zpřehlednil a navíc nevyžaduje podrobnou znalost problematiky. Vraťme se ale ještě na okamžik k výpočtům.

• *Poznámka: Nebuďme zmateni užitím bloku* [bez obnovy obrazovky]*, který se na obrázku řešení z prostředí Scratch 2 nevyskytoval. Ve Scratch 2 jsme totiž vlastním blokům nastavili jejich vykonání bez obnovy obrazovky v editoru bloků, což Snap řeší odlišně.*

U výpočtů úhlů všech ručiček se opakovala část násobící aktuální hodnotu časové jednotky s počtem stupňů kruhu vydělených odpovídajícím počtem dílků na ciferníku. Tuto část je možné vyseparovat do další vlastní funkce (zjisti úhel dle času), jež budou využívat funkce ostatní. Obrázek 4.1.4 konečně ukazuje definice všech dosud zmíněných funkcí. Všimněme si, že jen jedna je na rozdíl od ostatních parametrizovatelná. Parametrická funkce využívá parametry k získání výsledku, kdežto bezparametrické funkce si výsledek obstarají samy. Vlastními funkcemi jsme výpočty rozdělili do oddělených částí, čímž jsme radikálně zpřehlednili řešení celé problematiky.

Obrázek 4.1.4 Vlastní funkce vypočítávající úhly všech ručiček

Projekt vytvořený v prostředí Snap nyní porovnejme s projektem z prostředí Scratch 2. Přesto s těmito projekty ještě nekončíme. Je třeba probrat chybný výpočet dne týdne.

`SNAP PROJEKT: Ručičkové hodiny s akt. časem, datem a dnem týdne` #2

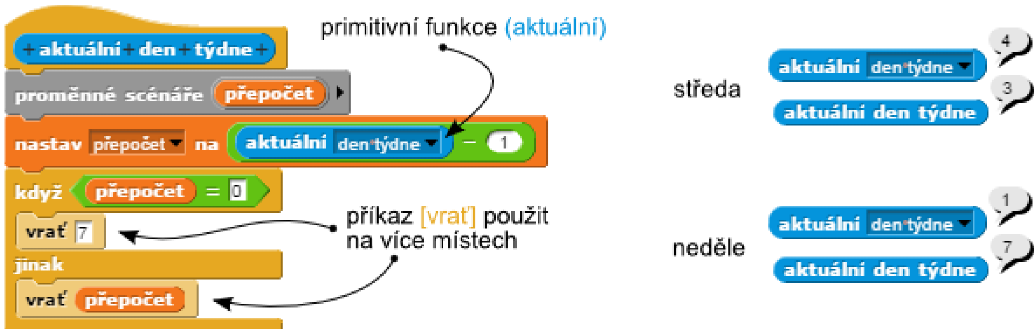
• Příkaz [vrať] užitý na více místech jedné funkce

Do našeho projektu se hodí přidat i informaci o aktuálním dnu týdne. Jenže při užití primitivní funkce (aktuální) s možností {den týdne} zjistíme, že nefunguje „správně“.

Podle mezinárodního standardu ISO 8601 je prvním dnem týdne pondělí a posledním neděle.[28] Přestože se tímto standardem řídí převážná část Evropy, pro obyvatele Spojených států amerických je prvním dnem týdne neděle a posledním sobota. Odůvodnění této odlišnosti přesahuje rámec bakalářské práce, přesto si můžeme říci, že má co do činění s náboženstvími a jejich historií.

Proto vyhodnotíme-li funkci (aktuální) s možností {den týdne} například ve středu, vrátí se hodnota {4} namísto očekávané {3}. O číslo větší výsledek vrací funkce pro každý den. Čtenář si zřejmě řekne, že toto není přílišnou překážkou. Vždyť přeci stačí v místě volání funkce (aktuální) odečíst jedničku funkcí (odečti). Tento způsob funguje u většiny dnů, jenže vyhodnotíme-li funkci (aktuální) v neděli, vrátí se {1} (namísto chtěné {7}) a po odečtení další jedničky bude výsledek {0}, což je samozřejmě nesmysl, neboť nultý den v týdnu přeci neexistuje. Problém je třeba vyřešit vlastním scénářem, jenž ve Snap můžeme umístit do vlastní funkce (aktuální den týdne).

Scénář funkce je uveden na obr. 4.1.5. Proměnnou scénáře (přepočet) nastavíme na vrácenou hodnotu funkce (aktuální), ze které hned odečteme jedničku. Následující podmínkou ověříme, jestli je výsledek po odečtení roven nule či nikoliv. Je-li tomu tak, víme, že jde o neděli a vrátíme příkazem [vrať] nám Evropanům odpovídající hodnotu {7}. V opačném případě vracíme hodnotu proměnné (přepočet), jelikož reprezentuje jiný den v týdnu a jejíž hodnota se pohybuje v platném intervalu $\langle 1, 6 \rangle$.



Obrázek 4.1.5 Definice vlastní funkce `(aktuální den týdne)`

Kromě vyřešení získání správného dne týdne jsme si hlavně ukázali vícenásobné umístění příkazu `[vrať]` do jedné funkce. Naší povinností je zajistit, aby funkce vždy vrátila nějakou hodnotu. Proto se příkaz `[vrať]` musí vyskytnout v obou větvích bloku `[když-jinak]`. Zapamatujme si tedy, že se blok `[vrať]` může ve funkcích vyskytnout vícekrát, stejně jako blok `[zastav]` v příkazech.

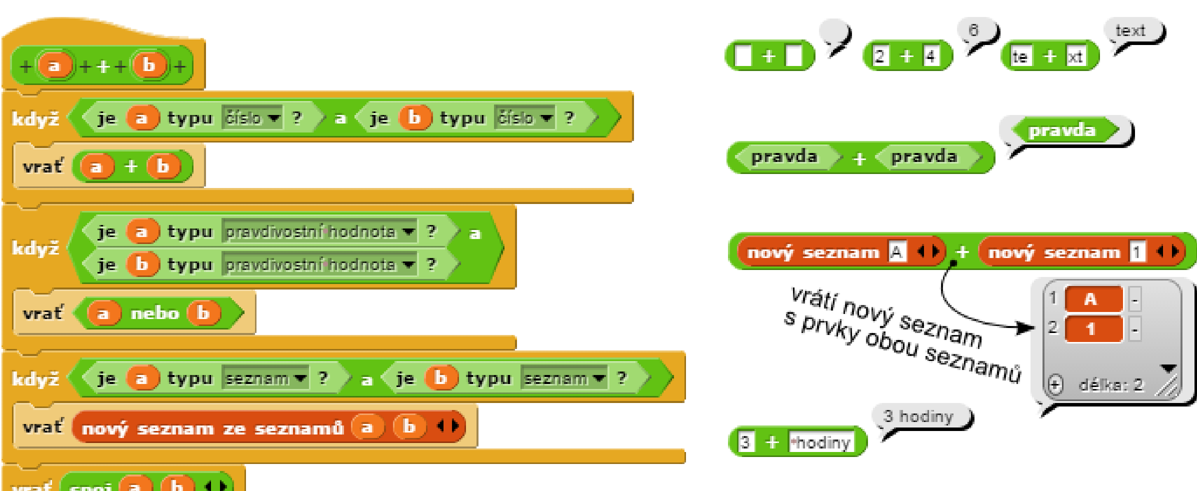
Zmiňme taktéž, že ošetření aktuálního dne týdne provádíme i v projektu Scratch 2, které se ale vyskytuje na scéně a využívá proměnnou všech postav. Tímto s projekty zobrazující aktuální čas a datum končíme.

- **Vracení různých datových typů v rámci stejné funkce**

Na začátku této podsekce o funkcích jsme si řekli, že příkaz `[vrať]` může vracet hodnotu libovolného datového typu. A jelikož lze příkaz `[vrať]` umístit na různá místa ve scénáři, může také nějaká funkce vracet hodnoty různých datových typů.

Čas od času sjednocujeme dvě hodnoty stejného datového typu. Prostředí Scratch 2 nás ale nutí užívat konkrétní funkce – k součtu čísel funkci `(sečti)` a k spojování textů zase `(spoj)`, což je nepříjemné – zvláště zadává-li hodnoty uživatel odpovědí na otázku vyvolanou blokem `[zeptej se]` a my musíme pracně ověřovat jejich datové typy. Práci by nám usnadnila obecná sjednocující funkce s dvěma parametry, která by sama dle datových typů argumentů vybrala odpovídající funkci a vrátila výsledek. Delegovali bychom tak veškerou práci na ní. A právě v prostředí Snap lze takovou naprogramovat.

Její definice je uvedena na obrázku 4.1.6. Název má totožný s primitivní funkcí `(sečti)` – tj. beze slov a pouze se znamínkem +. Její parametry lze označit za operandy a volané funkce za operace. Umí sečíst/spojit čísla, pravd. hodnoty, seznamy, a texty.



Obrázek 4.1.6 Definice vlastní sčítací funkce s ukázkami

Popišme si scénář funkce. Parametry `(a)` a `(b)` jsou typu *libovolný* (obdélníkové tvaru), takže mohou nabývat libovolného datového typu. Několika podmínkami `[když]` ově-

řujeme shodnost datových typů těchto parametrů primitivní podmínkou <je typu>. Abychom vstoupili do větve řídící struktury [když], musí být (a) i (b) stejného typu.

Jsou-li oba argumenty číselné, funkce vrátí příkazem [vrať] součet operandů operací (sečti). Jsou-li pravdivostní hodnotou, vrátí příkaz [vrať] logický součet operandů operací <nebo> (disjunkce).[1]) Jestliže (a) a (b) jsou referencí na seznam, spojí se speciálně vytvořenou vlastní funkcí (nový seznam ze seznamů) jejich prvky do nového seznamu, na něhož vrátí referenci.[2]) Konečně – pokud ani jedna podmínka nebyla splněna – nezbývá než sjednotit argumenty primitivní funkcí (spoj).

Podíváme-li se na pravou stranu obrázku, uvidíme různé ukázky volání této funkce. Jelikož jsou parametry funkce *libovolnými*, jejich implicitní hodnotou je prázdný text. Proto sjednocením textových argumentů {, } funkce (spoj) vrátí stejnou hodnotu {}. V druhém případě jsou oba argumenty {2, 4} číselnými a tak se sečtou funkcí (sečti). Třetí ukázka sjednocuje opět dvě hodnoty textového datového typu {te, xt}, jenž se opět spojí funkcí (spoj) do výsledného {text}. Dále za argumenty předáváme logickou hodnotu konstantami <pravda>, což funkce rozpozná a vrátí výsledek logického součtu {<pravda>}. Poté zkoušíme sjednotit prvky dvou bezejmenně vytvořených seznamů, čímž získáme referenci na seznam nový s prvky {A, 1}. Nakonec sjednocujeme číselný argument {3} s textovým argumentem { hodiny}. Protože druhý argument není číslo, do žádné z větví podmínek [když] nevstoupíme a tak naše funkce spojí tyto argumenty funkcí (spoj) do výsledné hodnoty textového typu {3 hodiny}. Naše sjednocující funkce může dle zadaných argumentů vrátit hodnoty až čtyř rozdílných datových typů.

Pozornější čtenáře zřejmě napadlo, co se stane, když budeme chtít sjednotit naší funkcí například text s referencí na seznam. Jelikož jsou oba argumenty rozdílné, dojde k jejich spojení funkcí (spoj). A protože každý datový typ umí prostředí Snap vyjádřit v textové podobě, prvky seznamu se převedou do textové podoby a nebudou odděleny mezerou. A tak text {Sn} a seznam s prvky {a, p, !} naše funkce spojí do {Snap!}.

• *Poznámka: Naše sjednocující funkce neobsluhuje všechny datové typy. V prostředí lze funkcí* <je typu> *detekovat ještě , , . Na ně je však ještě příliš brzo.*

Všimněme si ještě v definici funkce, že nepotřebujeme použít příkaz [když-jinak], ale vystačíme si jen příkazem [když]. Jakmile se totiž jedna z větví podmínek navštíví, vykonávání bloku se zastaví příkazem [vrať] a na vyhodnocování následujících podmínek v pořadí nedojde. Stejně tak jsme se mohli zachovat i u námi vytvořené předchozí funkce (aktuální den týdne) z obrázku 4.1.5, kde jsme namísto [když-jinak] mohli použít [když] a pod něj umístit příkaz [vrať]. Z programátorského hlediska by byly obě možnosti funkční, jen je otázkou, zda bychom tím zpřehlednili scénář či nikoliv.

• Rekurze ve funkcích

Jakmile přišlo prostředí Scratch 2 s tvorbou vlastních příkazů definovaných uživatelem, otevřela se programátorům možnost programovat rekurzivní algoritmy. Rekurze však není jen záležitostí příkazů. Lze vytvářet i rekurzivní funkce a podmínky. Rekurzivní volání se ale na rozdíl od příkazů provádí trošku odlišným způsobem.

Jelikož nelze funkci napojit na předchozí příkaz a můžeme ji tudíž vložit pouze do skutečných parametrů bloků, je téměř pravidlem, že rekurzivní volání funkce provádíme v příkazu [vrať]. Rekurzivní zarážka pak tento příkaz také obsahuje, stejně jako tomu bylo u rekurzivních příkazů a bloku [zastav], ale vrácená hodnota v rekurzivní zarážce již není rekurzivně vypočtena – je tedy pevně volena programátorem.

Na rozdíl od rekurzivně založených příkazů řešíme u rekurzivních funkcích pořadí jejich volání, protože jsou povětšinou součástí nějakého složeného výrazu, jenž prostředí musí postupně vyhodnotit v uvedeném sledu a dopočítat se ke konečnému výsledku.

---

[1]) Logický součet v Booleově algebře má následující výsledky: <pravda> s <pravda> jsou {<pravda>}, <pravda> s <nepravda> či naopak jsou též {<pravda>}, a <nepravda> s <nepravda> vždy {<nepravda>}.

[2]) Ačkoliv jsme práci se seznamy ještě neprobrali, lze si představit význam funkce (nový seznam ze seznamů), která byla vytvořena speciálně pro tento příklad. Přijme libovolný počet referencí na seznam, a hodnoty prvků těchto seznamů vloží do nového seznamu, jehož referenci nakonec vrátí.

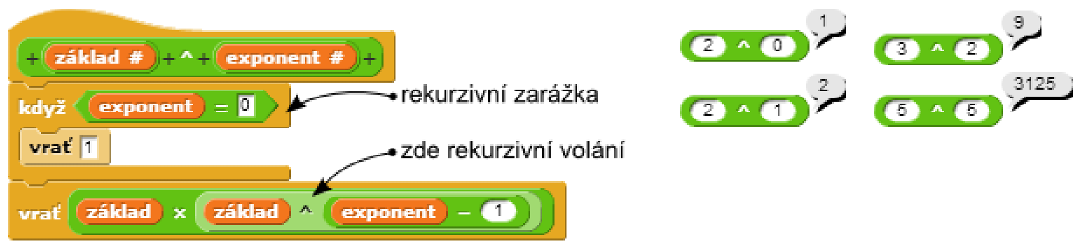
Je podivuhodné, že ačkoliv máme k dispozici mnoho různých matematických operací k realizování sofistikovaných výpočtů, nemůžeme v prostředích jednoduše provést operaci umocňování – tedy zkrácený způsob vícenásobného násobení. Existují sice scénáře umocňující základ na exponent, ale nejsou řešeny rekurzivně.[1])

Napravme tento nedostatek prostředí a demonstrujme si tvorbu první rekurzivně definované umocňující funkce. Scénář je uveden na obrázku 4.1.7, avšak bylo zvoleno nejjednodušší řešení. Název této funkce tvoří znak stříšky (ˆ), jelikož se v mnoha textově založených programovacích jazycích umocňující operace zapisuje tímto znakem.

Aby se funkce správně vyhodnotila, musí být exponent z oboru celých čísel $\mathbb{N}_0$, tedy kladným a zároveň celým číslem, které může být i nulou. Bylo sice možné algoritmus trochu vylepšit, aby exponent mohl být i záporným celým číslem, ale snížila by se tím čitelnost rekurze – a tu nyní potřebujeme správně pochopit.

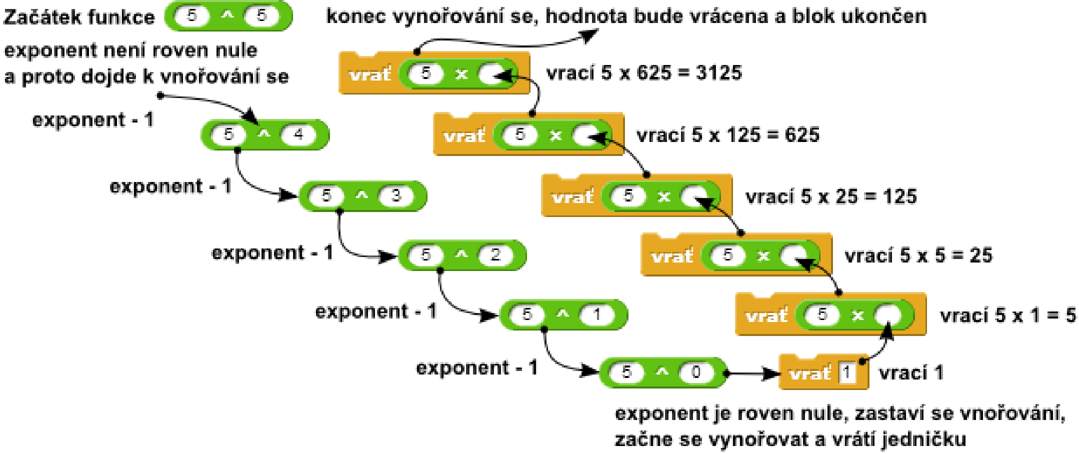
Řídící struktura `[když]` je rekurzivní zarážkou. Ověřuje hodnotu exponentu uloženou v parametru `(exponent)`, kterou s každým dalším rekurzivním voláním snižujeme o jedničku. Jakmile exponent klesne na nulu, je třeba zastavit rekurzi a vrátit jedničku. Následně dojde k vynořování se z rekurzivního zanoření, pakliže k němu vůbec došlo.

Posledním příkazem je `[vrať]` obsahující rekurzivní volání. Zápis lze vyjádřit slovy jako: „*Ještě než vynásobíš základ s exponentem funkcí* `(vynásob)`, *proveď rekurzivní volání sebe sama. První operand (základ) ponechej stejný a druhý operand (exponent) sniž o jedničku. Ihned po vyhodnocení umocňující funkce (sebe sama) užij vrácenou hodnotu k vynásobení se základem funkcí* `(vynásob)` *a vrať výsledek násobení*“. Lze předpokládat, že se bude větný popis kvůli rekurzivnímu volání opakovat několikrát po sobě.



Obrázek 4.1.7 Rekurzivně naprogramovaná umocňující vlastní funkce

Na obr. 4.1.7 vpravo opět vidíme ukázky. U první s argumenty `{2, 0}` k rekurzivnímu volání vůbec nedojde, protože řídící struktura `[když]` zachytí nulový exponent a vrátí jako výsledek jedničku. To je správně, neboť cokoliv na nultou je jedna. U druhé s `{2, 1}` se rekurzivně zanoříme pouze jedenkrát. Případ s `{5, 5}` je rozkreslen na obr. 4.1.8.



Obrázek 4.1.8 Ukázka vyhodnocení rekurzivní funkce krok za krokem

[1]) http://wiki.scratch.mit.edu/wiki/Solving_Exponents

• **Čisté funkce a funkce s vedlejším efektem**

Na předchozím obrázku 4.1.7 definující umocňující funkci jsme viděli, že před příkazem `[vrať]` je uveden ještě jeden příkaz – řídící struktura `[když]`. Z toho plyne, že i funkce mohou být tvořeny libovolnými příkazy. Může se v nich vyskytnout kupříkladu `[jdi na pozici]`, `[oblékni kostým]`, `[zahraj tón]`, `[pero dolů]`, `[čekej]`, `[zeptej se]`, `[nastav]`, `[nahraď prvek]`; a další. Podle užitých příkazů lze určit, jestli jde o funkci *čistou* anebo *s vedlejšími efekty*, přitom je lepší se funkcím s vedlejším efektem vyhnout.

Čisté funkce obsahují pouze ty příkazy, které nemají vliv na stav postavy, scény, či projektu a také na rychlost vykonávání scénáře. Nezasahují do ostatních záležitostí a „hledí si svého“. Proto je možné je bez obav použít v jakékoliv části scénáře.

Naopak funkce *s vedlejším efektem* obsahují příkazy ovlivňující své okolí. To znamená, že před jejich užitím musíme uvážit případný negativní dopad na stav projektu. Tak například použití funkce obsahující příkaz `[jdi na pozici]` ovlivní pozici postavy na scéně, což bude zřejmě v mnoha chvílích nežádoucí. Navíc se užitím tohoto příkazu připravujeme o možnost vykonat funkci na scéně, neboť ta nemá příkazy určené k pohybu a pokus o vykonání funkce skončí chybou. Totéž platí pro `[pero dolů]`.

Pakliže se pokusíme změnit kostým, bude mít taková funkce další vedlejší efekt, jelikož ovlivní postavu (popř. scénu). Příkazem `[čekej]` či `[zahraj tón]` zase budeme zdržovat vyhodnocení funkce, takže ji nebudeme moci využít v případech, kdy trváme na rychlosti. Příkaz `[zeptej se]` zase pozastaví vykonávání do doby, dokud uživatel neodpoví na otázku. Zároveň přepíše předešlou odpověď uloženou v `(odpověď)`. Příkazy `[nastav]` a `[nahraď prvek]` mohou být užity ke změně hodnot parametrů, ale mění-li hodnoty proměnných postavy či projektu, způsobují vedlejší efekt.

Z výše uvedeného vyplývá, že je třeba si při vývoji scénáře funkce dát velký pozor na vznik *vedlejších efektů*. Jak jsme si již uvedli – výrazně snižují využitelnost funkcí. S nimi totiž čelíme riziku, že naruší požadovaný stav postavy, scény, či celého projektu. *Vedlejší efekty* však řešíme pouze u funkcí – příkazy jsou stvořeny k změně stavu.

• **Napojení funkcí na příkazy pomocným příkazem** `[zavolej]`

Programátorům se občas stane, že by potřebovali vrátit hodnotu vlastního příkazu jako výsledek vykonané práce (např. *úspěšný/neúspěšný*). Víme však, že příkazy žádné hodnoty nevracejí. V takovém případě programátoři mění typ bloku z příkazu na funkci a na konkrétní místa vkládají příkaz `[vrať]` anebo jím nahrazují ve scénáři již existující příkaz `[zastav]`. Takovou funkci (dříve v podobě příkazu) užijí třeba v podmínkách k reakci na úspěšnost akce či si uloží výsledek do proměnné pro pozdější účely.

Většinou však bude potřeba takovouto funkci použít jako dříve – tedy jako příkaz. Tímto však narážíme na nevýhodu grafického prostředí, jelikož funkce svým tvarem nelze napojit na předchozí příkazy. Můžeme si ale pomoci tvorbou pomocného příkazu.

Obrázek 4.1.9 ukazuje definici bloku `[zavolej]` s jedním parametrem. Příkaz vůbec nic neprovádí – scénář tělíčka je prázdný. Můžeme jej napojit na předchozí příkazy a jako argument prvního parametru zvolit libovolnou funkci. Před vstupem do příkazu `[zavolej]` se funkce vyhodnotí a vrátí hodnotu, kterou ale v bloku `[zavolej]` nikde nevyužíváme. Dosáhneme tak možnosti zavolat funkci bez obav, že se vrácená hodnota někde použije. Proto je možné tento blok využít v kombinaci s funkcemi, které potřebujeme vykonat, ale o jejichž výsledek se nezajímáme a ani jej nijak nepoužíváme.

Obrázek 4.1.9 Způsob umožňující napojit do scénáře funkci či podmínku příkazem nevyužívající získanou hodnotu v parametru

4.1.2 Vlastní podmínky

Třetím a posledním typem bloků, které nám prostředí Snap zatím umožňuje vytvářet, jsou *podmínky*. *Podmínky* jsou v podstatě *funkcemi*, pouze je prostředí odlišuje tvarem se špičatými konci a při vracení hodnoty vracejí vždy a pouze jen pravdivostní hodnotu. Jejich účelem je něco ověřit a následně vrátit výsledek onoho ověření – tedy {<pravda>} či {<nepravda>}. *Podmínky* dosahují téměř identických možností jako funkce, tudíž je možné vytvořit rekurzivní podmínku a také podmínkou *čistou* či *s vedlejším efektem*.

Povinností programátora při tvorbě podmínky je zajistit, aby se vždy jedna z pravdivostních hodnot vrátila. Existují dva způsoby jak toho dosáhnout. Buďto do příkazu [vrať] vložíme jednu z pravdivostních konstant <pravda> či <nepravda>, anebo do něj vložíme celý podmíněný výraz, jehož výsledek po vyhodnocení v podobě pravdivostní konstanty přijme příkaz [vrať], který jej opět vrátí jako výsledek celé podmínky.

Obrázek 4.1.10 Tři ukázky primitivních podmínek v prostředí Snap

Jelikož prostředí nenabízí mnoho primitivních podmínek, může být těžší si představit, jaké vlastní podmínky bychom asi tak mohli vytvářet. Proto si ukažme na následujícím obrázku 4.1.11 několik podmínek pro inspiraci.

je přetahována? | dosaženo minimálního skóre? | konec hry? | obsahuje písmena s diakritikou?
je velkým písmenem? | je prázdný? | má pero dole? | je kurzor myši na scéně? | je právě noc?
má oblečen kostým želvy? | je dočasným klonem? | je skryta? | je v intervalu až včetně?

Obrázek 4.1.11 Příklady vlastních podmínek, jež lze vytvořit v Snap

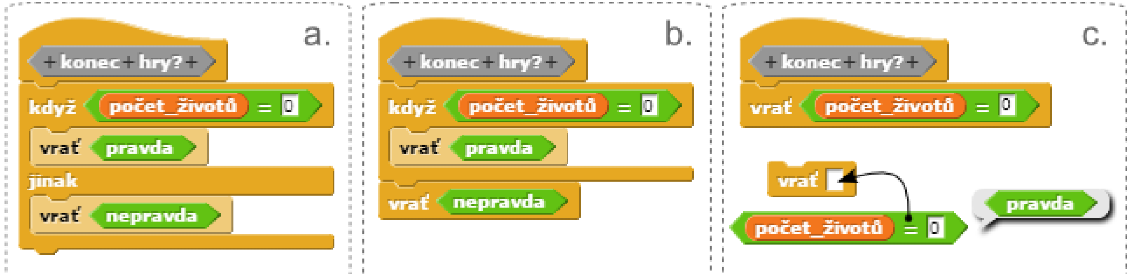
Dodejme, že podmínky <má pero dole?> a <je skryta?> zatím v prostředí nenaprogramujeme. Musíme si počkat na funkci (atribut), která nám k příslušným atributům postavy umožní přistoupit. Podobné je to s podmínkou <je dočasným klonem?>, která bude fungovat jedině tehdy, bude-li si postava uchovávat proměnnou (dočasnýKlon?), jež u dočasných klonů nastavíme v události [když startuji jako klon] na hodnotu {<pravda>}. Podmínka <je dočasným klonem> pak ověří hodnotu proměnné (dočasnýKlon?) a vrátí odpovídající výsledek. V prostředí totiž neexistuje způsob, jak rozlišit dočasný klon od originálu. V budoucnu by nám zřejmě nepomohl ani blok (atribut).

• *Poznámka: Autor práce si oblíbil umisťovat otazník na konce názvů podmínek, přidá-li to na pochopení jejich významu. V žádném případě to není povinností či praktikou.*

Většina projektů obsahuje alespoň jednu podmínku, která musí být naplněna, aby se vykonávání projektu ukončilo. Klasickým příkladem může být dotek postavy s jinou postavou či nějakou barvou, ztráta všech životů, nebo uběhnutí časového limitu. Musí-li programátor podmínky pro ukončení projektu kopírovat do více postav, riskuje zanesení chyby při případné změně těchto podmínek. Proto je vhodné naprogramovat vlastní podmínku obsahující ony podmínky k ukončení projektu, na níž se odkážeme.

Na obrázku 4.1.12 je uvedena definice podmínky <konec hry?> ve třech řešeních. Přestože jsou všechna funkční, jen jedno je z programátorského hlediska jednodušší, čitelnější a efektivnější. Podmínka ověřuje zbývající počet životů postavy, který je uveden v proměnné (počet_životů). Klesla-li hodnota proměnné na nulu, podmínka vrátí {<pravda>} signalizující úplný konec hry, v opačném případě vrátí {<nepravda>}. Rozeberme si rozdíly mezi uvedenými řešeními z obrázku umístěném níže:

a) obsahuje zbytečně [když-jinak] a vyžaduje dvakrát [vrať] (u nichž je zřejmé, že každý vrátí k sobě opačnou pravdivostní konstantu bloky <pravda> a <nepravda>)
b) zjednodušený zápis bez [když-jinak], přesto stále vyžaduje dvakrát [vrať]
c) nejlepší řešení vracející výsledek ověřované podmínky; vystačí si s jedním [vrať]; obejde se bez pravdivostních konstant <pravda> a <nepravda>



Obrázek 4.1.12 Tři způsoby vracení pravdivostních hodnot u podmínek

Procvičme si ještě jednou nejenom vytváření vlastních podmínek, ale též zkrácený zápis vracení výsledku. Ověřením hodnoty (kostým číslo) můžeme zjistit, jestli má postava oblečen základní kostým želvy. Něž abychom neustále kontrolovali jestli je číslo oblečeného kostýmu rovno nule, naprogramujme si vlastní podmínku, která za nás celou práci odvede a svým názvem taktéž sdělí význam svého scénáře v tělíčku. Podmínka <má oblečen kostým želvy?> je i s ukázkou zobrazena na obrázku 4.1.13.

Obrázek 4.1.13 Podmínka ověřující zdali má postava oblečen svůj kostým

V předchozích odstavcích bylo několikrát apelováno na použití zkráceného řešení bez bloků pravdivostních konstant a s jedním příkazem [vrať]. To však neznamená, že jsou ostatní řešení – v minulých případech zavržená – špatná anebo že se vůbec nevyužívají. Podmínka může mít, stejně jako funkce, klidně dva a více příkazů [vrať]. Proto zřejmě nebude překvapením, že mohou existovat scénáře se složitějším větvením bloky [když] či [když-jinak], v nichž podle stanovených podmínek určujeme příkazům [vrať], že v prvním případě vrátí např. <pravda>, v druhém také <pravda>, v jiném <nepravda>, a tak dále. Ostatně přesvědčme se sami u řešení následujícího problému.

Ačkoliv nám prostředí Snap nabízí podmínku <je typu> k odlišení například čísla od textu, nelze s ní ověřit, zdali text obsahuje jednu či více číslic. To potřebujeme např. při žádosti o jméno blokem [zeptej se] k zjištění data jeho svátku. Zadaná odpověď musí být kalendářním jménem – nikoli přezdívkou –, tedy textem s pouze abecedními znaky, mezi něž právě číslice nepatří. Na kontrolu výskytu alespoň jedné číslice kdekoliv v textu si ale můžeme naprogramovat vlastní podmínku <obsahuje číslo?>, jejíž dvě řešení s odlišným způsobem vracení výsledku jsou uvedeny na obrázku 4.1.14.

Obrázek 4.1.14 Vracení pravdivostní hodnoty přes [vrať] dvěma způsoby

Než si rozebereme rozdíly mezi řešeními, popišme si hlavní cíl scénáře. Každé písmeno zadaného textu v parametru (text) je ověřováno podmínkou <je typu>. Narazíme-li na číslo, víme, že text obsahuje alespoň jednu číslici. V tomto případě netřeba procházet další znaky textu a je možné okamžitě vrátit {<pravda>} – tedy že onen text obsahuje číslo. Pakliže se dostaneme až k poslednímu znaku v textu a ani tento nebude číslem, vrátíme {<nepravda>}, neboť text neobsahuje žádnou číslici. Přístup ke konkrétnímu znaku v textu provádíme přes proměnnou (i), jež zvyšujeme o jedna příkazem [změň].

Řešení na levé straně obrázku využívá ještě proměnnou (obsahuje_čísla?), kterou před průchodem znaků textu nastavíme na <nepravda> – předpokládáme, že text žádné číslo neobsahuje. V řídící struktuře [opakuj dokud nenastane] je složený podmíněný výraz, který ukončí cyklus buďto dojdeme-li na poslední znak textu (pak (i) se bude rovnat délce (text)) anebo pokud bude (obsahuje_číslo?) nabývat <pravda>, což se stane pouze je-li porovnávaný znak číslem. Protože kontrolujeme hodnotu proměnné (obsahuje_číslo?), v některých případech zmenšíme počet iterací cyklu. Příkazem [vrať] pak vracíme hodnotu (obsahuje_číslo?) jako výsledek celé podmínky.

Pravá varianta místo [opakuj dokud nenastane] používá [opakuj n krát]. Aby se ale zbytečně neprocházely další znaky v textu, je třeba předčasně ukončit vykonávání cyklu. To nám umožní příkaz [vrať] s hodnotou <pravda>, kterým ukončíme celý scénář podmínky <obsahuje číslo?>. Nevstoupíme-li do větve [když], pak se cyklus po průchodu všech znaků přirozeně ukončí a dojde k vrácení <nepravda>. Toto řešení ukazuje, že ne vždy musíme užít zkráceného způsobu vracení pravdivostní hodnoty.

Hned na první pohled je jasné, že definice podmínky vlevo je méně přehledná oproti její druhé variantě vpravo. Přesto je z programátorského hlediska správnější. Předčasně ukončit vykonávání cyklu příkazem [vrať] (stejně jako u příkazu [zastav]) může být v některých případech pro scénář studující programátory nepřehledné. Správně bychom měli programovat tak, aby se každý cyklus přirozeně ukončil, a až poté lze provést další kroky – například vrátit hodnotu. Obě varianty jsou však funkční. Volba je jen na nás.

- **Speciální případ**

Prostředí Scratch 2 nám umožní vkládat do skutečných parametrů hexagonálního tvaru (typu vyžadujícího podmínku) pouze bloky podmínek – ty se špičatými konci. Navzdory grafickému značení je ve Snap možné do těchto hexagonálních parametrů vložit i bloky funkcí – ty se zakulacenými rohy. Řekněme si proč tuto možnost prostředí nezakazuje.

Vraťme se na okamžik k obecné sjednocující funkci, která byla variantou k primitivní (sečti), a jejíž definici jsme si ukázali na obrázku 4.1.6. Ta dokáže kromě *číselné* a *textové* hodnoty vrátit i hodnotu *pravdivostní*, popřípadě referenci na seznam. Ačkoliv za určitých okolností vrací jednu z pravdivostních konstant <pravda>/<nepravda>, bylo by nesprávné ji vytvořit jako *podmínku*, neboť vrací i jiné datové typy, čímž nesplňuje hlavní pravidlo k tvorbě podmínek o vracení jedině pravdivostní hodnoty. I tak by bylo nesmyslné, kdyby prostředí bránilo programátorům vložit funkce vracející i pravdivostní hodnotu do parametrů hexagonálního tvaru – což nedělá (viz obr. 4.1.15).

Stejný případ nastává u proměnných s pravdivostní hodnotou. Je zbytečné ověřovat podmínkou <je rovno>, jestli je proměnná rovna hodnotě <pravda> či <nepravda>. Stačí ji, tedy blok typu funkce (např. (proměnná)), vložit do hexagonálního parametru, kterému předá jím očekávanou hodnotu pravdivostního datového typu. Tento způsob jsme mimochodem použili už u podmínky <obsahuje číslo?> z obr. 4.1.14, kde pravdivostní proměnná (obsahuje_číslo?) vyplňuje druhý argument podmínky <nebo>.

Obrázek 4.1.15 Vkládání bloků funkcí do hexagonálního parametru

4.1.3 Rozbalovací nabídky parametrů

Skutečné parametry vlastních bloků můžeme vybavit rozbalovací nabídkou s námi určeným *výčtem* možností. Uživatelům bloku tak usnadníme práci při určování argumentu. Připomeňme si některé primitivy s tímto vylepšením na obrázku 4.1.16.

Obrázek 4.1.16 Primitivy s rozbalovací nabídkou u svých parametrů

Rozbalovací nabídka se definuje konkrétnímu parametru a to přímo v jeho editoru. Je-li parametr jednoduché varianty a jedním z typů: *číslo*, *text*, *libovolný* anebo *libovolný (nevyhodnoceno)*; umožní nám prostředí jej vybavit rozbalovací nabídkou. Po splnění výše uvedených podmínek se při kliknutí pravým tlačítkem myši někde v prázdném prostoru editoru parametru zobrazí speciální menu umožňující nadefinovat konkrétní možnosti (lze označit také za výčet možností), jež budou dostupné po rozevření rozbalovací nabídky uživatelům bloku. Ve speciálním menu lze ještě zvolit mezi rozbalovací nabídkou *editovatelnou* a *needitovatelnou*, o čemž až později.

Rozbalovací nabídka slouží k nabídce očekávaných či přímo vyžadovaných hodnot, na které umí blok dle zvolené možnosti různě zareagovat. Mnohdy totiž uživatel nemá tušení, jakými možnostmi jím užívané bloky disponují. Rozbalovací nabídky se týkají pouze skutečných parametrů a pomáhají určit hodnotu parametrů formálních. Zlehčují především práci uživatelům s takovýmto typem bloků a nemají až zas tak velký dopad na způsob programování tělíčka (scénáře) samotného bloku.

• Editovatelné a needitovatelné rozbalovací nabídky

Všimněme si, že jsou některé skutečné parametry s touto nabídkou „zatmavené". Tím je určeno, že argument nelze zapsat ručně v textové podobě – tedy že výběr argumentu lze provést jedině výběrem nějaké možnosti z nabídky. Například příkaz `[nastav]` logicky neumožňuje napsat slovy název proměnné jež chceme nastavit, neboť je lepší si nechat poradit prostředím, které proměnné jsou v oné části scénáře přístupné. Tento typ rozbalovacích nabídek označujme za *needitovatelné*.

Za to příkaz `[zamiř směrem]` sice nabízí čtyři předvybrané směry, kterými můžeme postavu namířit, avšak nechává skutečný parametr nezatmavený, aby si mohli uživatelé nastavit hodnotu směru vlastní. Může, ale také nemusí, využít rozbalovací nabídku. Již je možné předpovědět, že tento typ nabídky nazveme jako *editovatelný*.

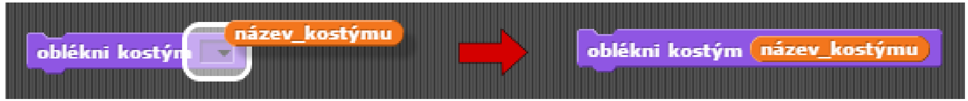
Ve speciálním menu přes možnost „pouze výběr" se zaškrtávátkem lze určit, zdali chceme skutečný parametr *editovatelný* či nikoliv. Rozdíl ukazuje obr. 4.1.17, na kterém jsou tři příkazy se třemi typy povolených parametrů lišící se ve zvolené možnosti „pouze výběr". Všimněme si, že se černá šipka rozbalovací nabídky zobrazí u všech povolených typů skutečných parametrů.

Obrázek 4.1.17 Editovatelné versus needitovatelné nabídky

• Chráněné a nechráněné needitovatelné rozbalovací nabídky

U některých primitivů s needitovatelnou rozbalovací nabídkou můžeme namísto výběrů nějaké možnosti do ní zasadit blok funkce, po jejímž vyhodnocení se vrácená hodnota „vnutí" rozbalovací nabídce. Je tak možné předat hodnotu, která mezi možnostmi nabídky nefiguruje – což může narušit vykonávání scénáře uvnitř bloku. Abychom si porozuměli, pohlédněme na obrázek 4.1.18, kde namísto výběru konkrétního názvu kostýmu se odkazujeme na hodnotu uloženou v proměnné `(název_kostýmu)`. Taková řešení volíme

ve chvíli, kdy je vyžadována určitá variabilita scénáře, neboť se předpokládá opakované oblékání různorodých kostýmů postavě jedním a tentýž scénářem.



Obrázek 4.1.18 Předávání hodnoty do nechráněné rozbalovací nabídky

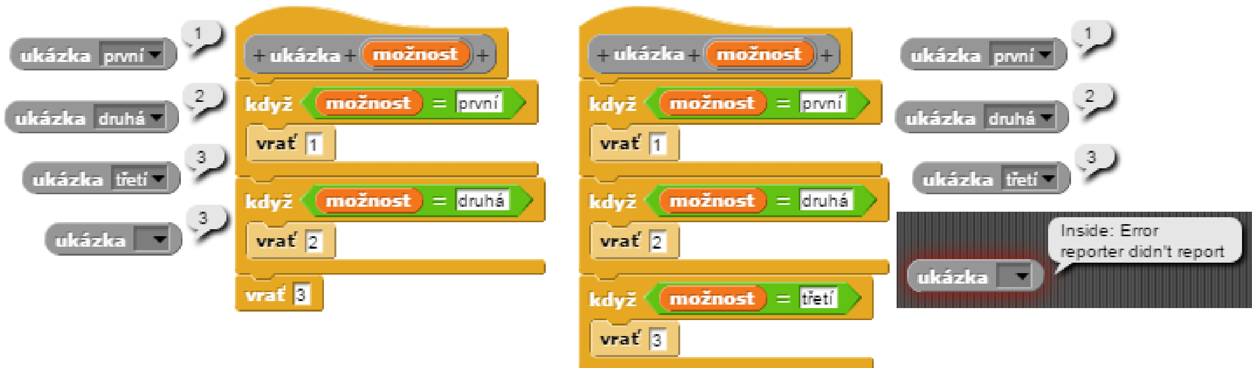
Jak je vidno, rozbalovací nabídka u bloku [oblékni kostým] je needitovatelná a nelze do ní zapsat název kostýmu psaním na klávesnici. Přesto je možné do tohoto parametru vpravit funkci. Takováto rozbalovací nabídka je *nechráněna* proti zadání neplatné hodnoty „zvenku“. Naštěstí [oblékni kostým] při zadání neplatného názvu kostýmu umí prostředí obléci postavě základní kostým želvy a neskončí tak s chybou. My se ale u vlastních bloků na zadání neplatné hodnoty musíme připravit.

V prostředí ještě existují některé primitivy, jejichž needitovatelná rozbalovací nabídka je *chráněna* proti předání neplatné hodnoty, jelikož nepovoluje vložení žádného z bloků. Tyto primitivy ve Snap s *chráněnou* rozbalovací nabídkou jsou na obr. 4.1.19.

Obrázek 4.1.19 Bloky s chráněnou rozbalovací nabídkou

Celou problematiku si vysvětlujeme proto, že rozbalovací nabídky u parametrů vlastních bloků jsou vždy *nechráněné* a tak musíme počítat s tím, že uživatelé našich bloků mohou předat, ať už úmyslně anebo omylem, neplatné argumenty. Proto bychom měli ověřovat hodnotu i poslední možnosti z výčtu u parametru s rozbalovací nabídkou přes řídící strukturu [když]. Obrázek 4.1.20 zobrazuje funkci (ukázka) ve dvou verzích.

Scénář funkce vlevo neošetřuje třetí možnost z rozbalovací nabídky, tudíž není-li (možnost) {první} či {druhá}, funkce vrátí {3}. Jak je vidno na levé straně obrázku, při vyhodnocení (ukázka) bez zvolené možnosti ({}) dojde k vrácení hodnoty, kterou by funkce mimo jiné vrátila i kdyby v rozbalovací nabídce byla jiná funkce s výsledkem odlišným od dostupných možností. Ačkoliv (ukázka) získá jinou než žádanou hodnotu, v konečném důsledku se vždy provede a vrátí {3}, což je potenciálem k zanesení chyby do projektu při jeho vývoji, jíž je velmi obtížné odhalit. Kdyby však byla u této funkce rozbalovací nabídka editovatelná, toto řešení by bylo naprosto správné. Jelikož jsme ji ale nastavili na needitovatelnou, přejeme si získat jen hodnoty z nabídky.



Obrázek 4.1.20 Rizika spojená s nechráněnými rozbalovacími nabídkami

Řešení na pravé straně ošetřuje i poslední možnost z rozbalovací nabídky a řídící strukturou [když] pokrývá hodnotu {třetí}. Úplně vpravo na obrázku vidíme, že voláme-li (ukázka) s neplatnou hodnotou ({}), prostředí Snap zobrazí chybové hlášení o tom,

že funkce nevrátila žádnou hodnotu. Skutečně se na žádný z příkazů [vrať] nedostalo a funkce se tak nemohla správně vyhodnotit. Zde však narušujeme jedno z pravidel funkcí, kterým je povinnost vždy vracet hodnotu. Získáváme tím však zpětnou vazbu od prostředí informujíc nás o nesprávné hodnotě, na kterou funkce neumí reagovat.

Nemožnost nastavit needitovatelné rozbalovací nabídky na *chráněné* přináší programátorům značné problémy. Je tedy na nás, jaký způsob si při vývoji bloků zvolíme. Mimochodem u příkazů tento problém nenastane. Nenavštíví-li se ani jedna z větví [když], příkaz skončí a žádná chyba se nezobrazí (neb nic nevrací). U příkazů si proto ošetření všech možností z needitovatelné rozbalovací nabídky můžeme dovolit.

- **Definování výčtu možností nabídky**

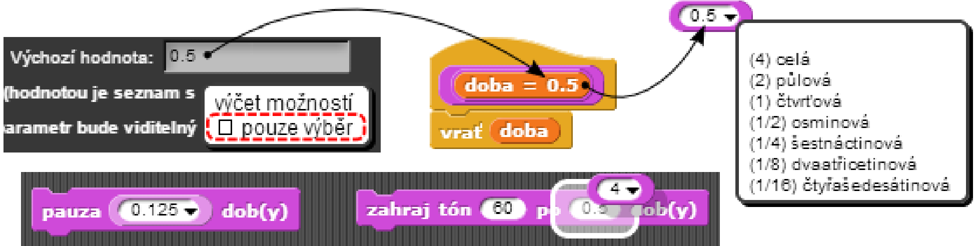
Aby se ve skutečném parametru zobrazila rozbalovací nabídka, je zapotřebí ji stanovit alespoň jednu možnost z celkového výčtu možností. Přes speciální nabídku vyvolanou pravým tlačítkem myši v editoru parametru se po zvolení možnosti „výčet možností" zobrazí dialogové okno, do něhož jednotlivé možnosti k výběru zapisujeme. Co možnost, to text na jeden řádek – a mějme na paměti, že na pořadí záleží.

Definovaným možnostem můžeme ještě v dialogovém oknu přiřadit hodnotu, která se namísto jejich názvu použije, jakmile dojde k jejich výběru z nabídky. Funkcionalita funguje na systému `KLÍČ = HODNOTA` – rozevřená rozbalovací nabídka zobrazuje klíče. Pokud má klíč přidělenou vlastní hodnotu, po zvolení takovéhoto klíče se do skutečného parametru za argument vepíše jeho hodnota. Vzpomeňme opět příkaz [zamiř směrem], který funguje na tomto principu. Zvolíme-li v něm z nabídky třeba {(90) doprava}, pak se za argument místo textové hodnoty vloží hodnota číselná ({90}). K aplikování této funkcionality na nějakou možnost nabídky musíme mezi klíč a jeho hodnotu vložit rovnítko (=) a nesmí být okolo něj mezery. Dost bylo teorie a raději přejděme na ukázky.

- **Příklady**

Při programování hudby je relativně složité vybírat vhodnou dobu pauzy a délku hraného tónu, jelikož se zadávají v číselné podobě. Zvláště obtížné je to v případě, máme-li za úkol přepsat nějakou píseň z notového zápisu, což vyžaduje znalost hudební notace vyučující se většinou na hudební nauce. Programátor bez hudebního vzdělání asi neví, že celou dobu u čtyřdobého taktu vyjádří hodnotou {4}. Proto je možné si pomoci tvorbou vlastní funkce s rozbalovací nabídkou, která nám zprostředkuje existující doby v přesných hodnotách (viz obrázek 4.1.21) pod pomocnými názvy v možnostech.

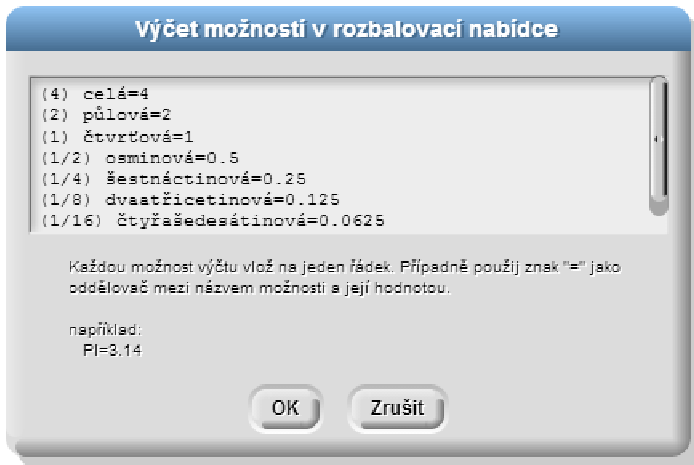
Číselný parametr (doba) obsahuje editovatelnou rozbalovací nabídku se sedmi různými dobami k výběru, které mají určené své hodnoty. Cílem bloku je pouze nabídnout možnosti, proto okamžitě vrací hodnotu (doba) příkazem [vrať]. Chce-li uživatel této funkce použít vlastní dobu zápisem do parametru, nic mu v tom nebrání. Neopomeňme nastavenou výchozí hodnotu parametru, což je důležité u parametrů s rozbalovací nabídkou – zvláště je-li needitovatelnou. Výchozí hodnota však není {(1/2) osminová}, ale {0.5}, neb je hodnotou, jež se po zvolení možnosti z nabídky vybere za argument.



Obrázek 4.1.21 Funkce s rozbalovací nabídkou nabízející hudební doby

Funkci z kategorie zvuky netřeba pojmenovávat. Jejím cílem je pouze dopravit správnou hodnotu, kterou si programátor zvolil přes rozbalovací nabídku s výčtem jednotlivých dob v přijatelných názvech. Uvědomme si, že dvaatřicetinová doba je vyjádřena hodnotou {0.125} a čtyřašedesátinová {0.0625} – to si málokdo pamatuje. Podívejme

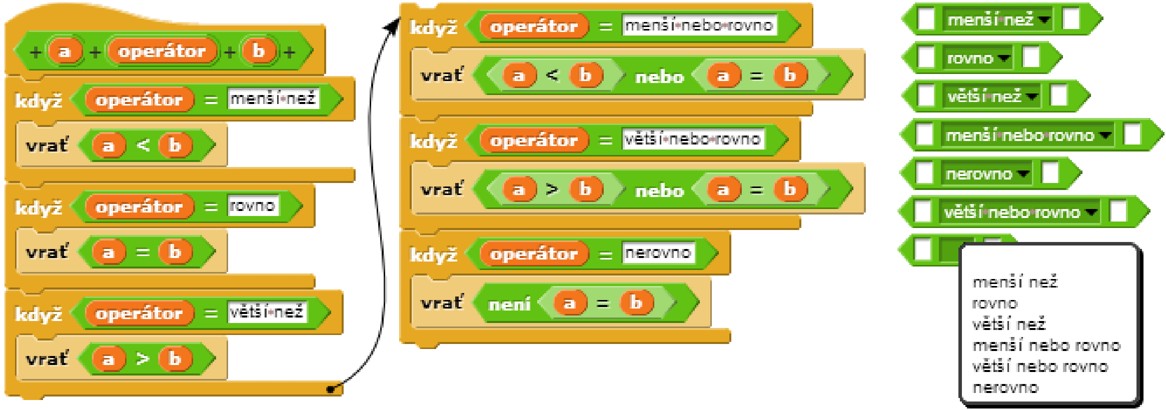
se ještě na obrázek 4.1.22, kde byly v dialogovém okně definovány možnosti nabídky (klíče) společně s jejich číselnými hodnotami, do kterých se po výběru promění.



Obrázek 4.1.22 Dialogové okno k definici možností rozbalovací nabídky

Dalším příkladem může být usnadnění si porovnávání dvou hodnot. Na relační operace *je menší*, *je větší*, a *je rovno* máme v prostředích k tomu určené primitivní podmínky <je menší>, <je větší>, a <je rovno>. U operací *je menší nebo rovno*, *je větší nebo rovno*, a *je nerovno* si musíme pomoci složenými výrazy z těchto podmínek v kombinaci s podmínkami <a zároveň>, <nebo>, a <není>. Při sofistikovanějším porovnávání se však scénář stává méně přehledným.

Vytvořme si proto podmínku, která všechny porovnávací operace nabídne k výběru v rozbalovací nabídce. Složené výrazy vyřeší uvnitř a umožní nám zpřehlednit scénář. Její definice je uvedena na obrázku 4.1.23 (blok byl z důvodu velikosti rozdělen napůl).



Obrázek 4.1.23 Vlastní podmínka sjednocující relační operace

V původní verzi naší podmínky byla namísto dlouhých popisků (*menší než*, *rovno*, *větší nebo rovno*...) použita znaménka relačních operátorů (<, =, >=...). Prostředí Snap se však s těmito znaky nedokázalo vypořádat a projekt s uloženou podmínkou nedokázalo otevřít. Proto je vhodné – alespoň a bohužel prozatím – používat celé názvy možností v rozbalovací nabídce namísto zavedených a výřečných zkratek.

Některé čtenáře by nyní mohlo napadnout vytvořit obecnou funkci na výpočet aritmetických operací. Kromě dvou parametrů představující operandy aritmetické operace by třetí parametr měl rozbalovací nabídku s možnostmi {plus/mínus/krát/děleno}. Na první pohled se zdá, že bychom si pomohli, neboť bychom nemuseli složitě obměňovat různé primitivní funkce aritmetických operací v případě úpravy výpočtu. K tomu ale máme v prostředí možnost záměny bloků přes nabídku vyvolanou pravým tlačítkem myši, kde lze právě (sečti), (odečti), (vynásob), a (vyděl) jednoduše zaměňovat.

U funkce s aritmetickými operacemi bychom žádného rozšíření nedosáhli a proto je její tvorba bezpředmětná. U předchozího příkladu s podmínkami jsme si však pomohli.

- **Proměnlivý (závislý) výčet možností**

Pohrajeme-li si možnostmi primitivní funkce `(z)`, zjistíme, že má nějakým způsobem oba parametry s rozbalovací nabídkou propojeny. Podle toho, jestli vybereme v druhém parametru z nabídky možnost `{scéna}` či nějakou postavu (mezi možnostmi reprezentovanou jejím názvem), se změní nabídka možností v prvním parametru. Při zvolené scéně můžeme z první nabídky vybírat mezi `{kostým číslo/název kostýmu}`, zatímco u postavy vybíráme mezi `{souřadnice x/souřadnice y/směr/kostým číslo/název kostýmu/velikost}` (viz obr. 4.1.24). Tuto „provázanost" lze pojmenovat jako *proměnlivý výčet možností*, neboť se nabídka možností mění na základě stavu jiné nabídky.

Obrázek 4.1.24 Proměnlivý výčet možností primitivní funkce `(z)`

Bohužel provázat vícero rozbalovacích nabídek v rámci vlastního bloku prostředí zatím neumožňuje. K tomu si stěží představit způsob jejich propojování, uvážíme-li současně navržený a nepříliš přívětivý způsob definování výčtu možností nabídky v prostředí. Pochybují-li však někteří, že by tato provázanost rozbalovacích nabídek nebyla k užitku, nechť se podívají na příklad z obrázku 4.1.25, na němž je funkce z kategorie `vnímání`.

Funkce umí dle zadaných hodnot vrátit datum v požadovaném formátu v podobě, která se stala standardem v elektronických dokumentech. Ačkoliv ji nebylo nutné pojmenovat, neb je vybavena pěti výřečnými parametry, pro pochopení ji můžeme v této práci nazvat jako `(datum ve formátu)`. *Číselné* parametry `(den)` a `(rok)` s *libovolným* parametrem `(měsíc)` slouží k definici data. Proměnná `(formát)` umožňuje zvolit si pořadí dne, měsíce, a roku ve výsledném zápisu. A na oddělení hodnot máme proměnnou `(oddělovač)`. Všimněme si, jak všechny parametry obsahují rozbalovací nabídku, což je extrémní – avšak velmi užitečnou – demonstrací využitelnosti této funkcionality.

Obrázek 4.1.25 Vlastní funkce s mnoha rozb. nabídkami formátující datum

Nyní si vysvětleme, jak souvisí funkce `(datum ve formátu)` s propojováním rozbalovacích nabídek. Některé měsíce se liší počtem dní. Nejméně, tedy 28, jich nabízí měsíc Únor, ostatní se pak pohybují mezi 30–31 dny. Při výběru konkrétního měsíce bychom

uvítali, kdyby se v první rozbalovací nabídce proměnné (den) určené k výběru dne aktualizoval počet možností dle počtu dnů zvoleného měsíce z druhé nabídky – tedy aby při výběru {Únor/2} bylo možné vybrat pouze mezi hodnotami 1–28, u {Září/9} mezi 1–30, a u {Květen/5} 1–31. Jak ale víme, toto bohužel udělat nemůžeme.

Na tomto příkladu si ještě ukazujeme dvě věci. Druhý parametr s editovatelnou rozbalovací nabídkou je typu *libovolný*, což nám umožňuje zadat měsíc různorodými způsoby. Lze tedy vyjádřit první den měsíce ledna roku 2015 v následujících podobách: „1.1.2015“, „1. leden 2015“, „1. ledna 2015“, anebo třeba „1. január 2015“. Sama rozbalovací nabídka má smíšené možnosti {1--12/Leden--Prosinec}.

Taktéž je na obrázku volána funkce bez zvoleného dne, měsíce, roku a oddělovače, jejímž výsledkem je {00}. To z toho důvodu, že parametry (den) a (rok) jsou číselné a implicitní hodnotou těchto parametrů je právě nula. U parametru (měsíc), který též nemá zvolený argument, je implicitní hodnota {}, což se vzhledem k nezvolenému oddělovači ve výsledku vůbec neprojeví. Tolik jen pro zajímavost.

• Speciální případ – číselný typ parametru může obsahovat text

Většina primitivů pracujících se seznamy obsahuje editovatelnou rozbalovací nabídku, mezi jejímiž možnostmi jsou {1/poslední/náhodný}. Můžeme tak kupříkladu získat náhodný prvek seznamu funkcí (prvek), vložit prvek na náhodnou pozici příkazem [vlož], anebo nahradit náhodný prvek za prvek jiný příkazem [nahraď prvek]. Zřejmě bychom očekávali, že příkaz [zruš] umožní smazat prvek také na náhodné pozici, jako to umí jemu podobné bloky, ale ačkoliv disponuje rozbalovací nabídkou, mezi jejími možnostmi možnost {náhodný} nefiguruje. Možnosti [zruš] jsou navíc trochu odlišné – konkrétně {1/poslední/všechno}. Tato nekonzistentnost v možnostech napříč bloky pracujících se seznamy přetrvává už od prostředí Scratch 1.

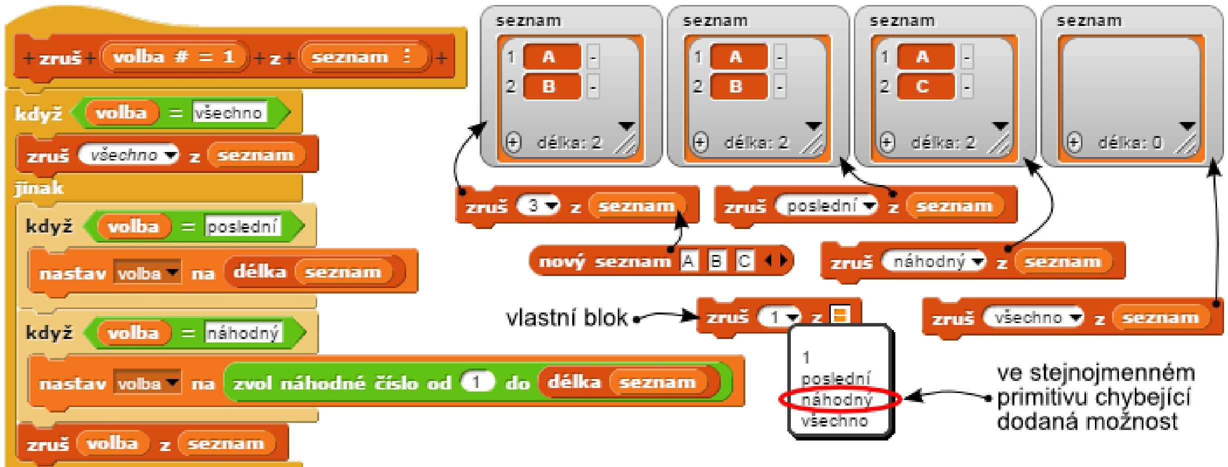
Možnost {všechno} u rozbalovacích nabídek ostatních primitivů nelze samozřejmě očekávat, neboť pracovat se všemi prvky seznamu má smysl jen u mazání. Přesto existují případy, kdy bychom rádi zrušili prvek i na náhodné pozici. Dosud jsme si museli poradit, a to třeba užitím funkce (zvol náhodné číslo), které jsme zadali argumenty {1} a délku seznamu. Vrácená hodnota pak byla předána příkazu [zruš], jenž odstranil prvek ze seznamu na příslušné (náhodné) pozici.

Otázkou zůstává, proč si vše popisujeme pod částí s názvem *speciální případ*. Všimněme si, že jsou parametry s rozbalovací nabídkou u bloků pracujících se seznamy číselného typu, který se „zvenku“ bloku zobrazuje v oválném tvaru. Prostředí v takovém případě hlídá uživatele, aby nezadal jiný než číselný argument. Jenže zvolíme-li jinou možnost než {1} z možností u těchto primitivů, rozbalovací nabídka „vnutí“ číselnému parametru hodnotu v textové podobě – například {náhodný}. S žádným jiným blokem s číselným parametrem obsahující rozbalovací nabídku napříč kategoriemi se do tohoto speciálního případu nedostaneme. Rozbalovací nabídka tak může narušit pravidla prostředí, na což bychom měli myslet i my při vývoji stejně zaměřených parametrů. Že se s takovým případem můžeme setkat ukazuje následující příklad.

Zkusme si udělat vlastní příkaz naprosto totožný k primitivnímu příkazu [zruš], kterémuž rozšíříme rozbalovací nabídku o možnost {náhodný}. Obrázek 4.1.26 obsahuje definicí bloku společně s ukázkami jeho funkcionality.

Ačkoliv je parametr (volba) typu *číslo* (s oválným tvarem), víme již, že přes jeho rozbalovací nabídku můžeme kromě čísla přepravit do (volba) i text. Pokud (volba) nabývá možnosti {všechno}, zrušíme všechny prvky v seznamu a blok se ukončí. Jinak se ptáme na další dvě možnosti. Bude-li (volba) {poslední}, přenastavíme hodnotu této proměnné na délku zadaného seznamu. Pokud bude hodnota (volba) {náhodný}, přenastavíme (volba) již zmíněným způsobem funkcí (zvol náhodné číslo), kterou získáme náhodné číslo v rozsahu délky seznamu. Nakonec příkazem [zruš] odstraníme ze seznamu prvek na pozici odpovídající nové hodnotě (volba) (poslední či náhodný).

Tato kamufláž je natolik neprůhledná, že nerozeznáme [zruš] od [zruš]. V definici našeho příkazu [zruš] se všude (třikrát) vyskytuje jeho primitivní verze [zruš]. Kdežto v ukázkách na pravé straně obrázku je všude užit náš vlastní blok [zruš].



Obrázek 4.1.26 Vlastní příkaz vylepšující primitivní [zruš] o další možnost

Zbývá na závěr zmínit, že v našem příkazu, nahrazující stejnojmenné primitivum, je rozbalovací nabídka též editovatelná. Můžeme tak stále určovat pozici odstraňovaného prvku přímo – a i užitím klávesnice. Pokud bude tedy (volba) nabývat například {3}, hodnota této proměnné se příkazem [nastav] nezmění, jelikož se nevstoupí do větví řídících podmínek [když]. Příkaz [zruš] tak přijme neupravený (původní) argument.

SNAP PROJEKT: Definice vlastních bloků s rozbalovací nabídkou #4

4.1.4 Imitace konstant

Jeden ze základních „přestupků“ proti dobrým programátorským mravům je hodnoty, které se nemění, nechat ve scénáři zapsané „jen tak“. Ba co hůř – ještě je opakovat na více místech ve scénáři. Existuje-li ve scénáři například hodnota {5}, jak máme vědět, co představuje. Budeme se nepochybně ptát, proč zrovna pět a ne třeba osm. Takovéto hodnoty se občas označují termínem *magická čísla*, neboť se chovají „magicky“ – nevíme co dělají a když je změníme, výsledek celého scénáře se nepředvídatelně změní. Často si ani sám autor scénáře po delší době nevzpomene, kde k takovým hodnotám přišel. Pokud jim ale dáme jméno (např. „počet životů“), v tu ránu je jejich význam jasný.

Víme-li, že se nějaká hodnota nebude ve scénáři měnit či potřebujeme-li její změně zabránit, je nasnadě vytvořit *konstantu*, jejíž název zřejmě vychází z latinského *constantia* – stálost, trvalost, nezměnitelnost. Většina programovacích jazyků tvorbu konstant podporuje, jejichž užitím dávají programátoři jasně najevo, že stanovená hodnota se bude pouze používat (třeba k výpočtu), avšak nikdy ne měnit – což ani jazyk nedovolí.

S konstantami se setkal téměř každý z nás. Určitě známé matematickou konstantu π (Ludolfovo číslo), z fyziky gravitační konstantu G, z chemie prvky periodické tabulky, a nebo čtvrťovou pomlku z hudební notace. Všem výše zmíněným byla kdysi v historii přiřazena hodnota, jež se už nikdy nezmění – $\pi = 3,14159\ldots$, $G = (6,67 \pm 0,01) \cdot 10^{11}$, u např. vodíku jde o značku H, a čtvrťová pomlka má hodnotu $\frac{1}{4}$. Proto je lze označovat za konstanty a prostředí by takto definované hodnoty nemělo umožňovat změnit.

Definujeme-li ale tyto hodnoty jako proměnné (a je jedno jestli budou proměnnými postavy, projektu, anebo scénáře), můžeme je upravit příkazy [nastav] a [změň], což porušuje základní výjimečnost konstanty – její neměnitelnost. Bohužel v prostředích konstanty přes nějaké k tomu určené tlačítko – tak jako máme jedno na tvorbu proměnných – vytvářet nemůžeme. Přesto si můžeme imitovat konstanty tvorbou vlastních bezparametrických funkcí, jež vrátí vždy jednu a týž hodnotu. Nesmí se však v příkazu [vrať] vyskytnout funkce, která by mohla v různých chvílích vracet různé hodnoty.

Takových funkcí vydávajících se za konstanty může být celá řada. Prohlédněme si následující obrázek 4.1.27 s různými ukázkami a definice jedné z nich.

Obrázek 4.1.27 Ukázky imitace konstant funkcemi vracející jejich hodnoty

Jelikož je hodnota imitované (neúplné, nepravé) konstanty uvedená uvnitř funkce, pak je chráněna proti jakékoliv úpravě bloky [nastav] a [změň]. Chceme-li hodnotu konstanty upravit, pak musíme její funkci editovat a změnit onu hodnotu v příkazu [vrať].

Také si povšimněme, že všechny funkce z obrázku 4.1.27 mají název napsán velkými písmeny. Jedná se o nepsané pravidlo k pojmenování konstant, což zvyšuje přehlednost scénáře. Snáze tak odlišíme proměnnou od konstanty. Při víceslovném názvu konstanty je vhodné použít podtržítko jako oddělovač slov. Další výhodou je možnost přiřazení kategorie vlastním funkcím, čímž dáváme najevo zaměření konstant.

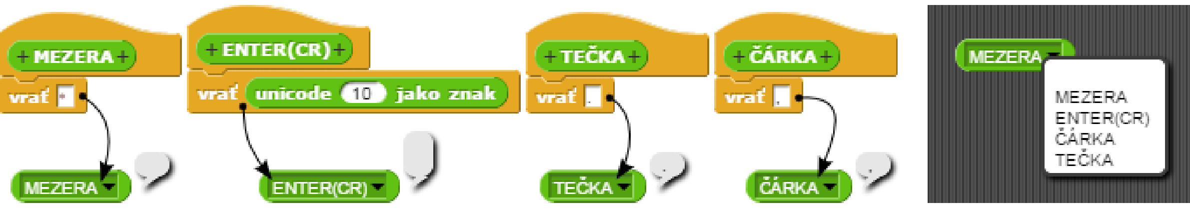
- **Více konstant v jednom bloku s rozbalovací nabídkou**

Některé konstanty mají k sobě svým zaměřením velmi blízko. Příkladem jsou matematické konstanty – Ludolfovo číslo (π), Eulerovo číslo (e), zlatý řez (φ), a kladné/záporné nekonečno ($+\infty/-\infty$). Asi již víme, jak bychom z nich udělali konstanty. Stačí vytvořit funkci (například (PI)), a do příkazu [vrať] vepsat konkrétní (fixní) hodnotu ($3,14$).

Než abychom vytvářeli toliko konstantních funkcí se stejným zaměřením, bude jednodušší si vytvořit jednu funkci s rozbalovací nabídkou, mezi jejímiž možnostmi budou všechny zmiňované matematické konstanty. Funkce pak sama uvnitř zjistí, kterou hodnotu má vrátit. Abychom se ale pořád netočili okolo matematiky, ukažme si v dalším příkladu konstanty užívané při psaní či skládání textu.

Na obrázku 4.1.28 vidíme konstanty definované funkcemi (MEZERA), (ENTER(CR)), (TEČKA), a (ČÁRKA). Přestože postava mluvením v bublině tyto znaky viditelně zobrazí, při jejich zapisování do argumentů bojujeme s horší čitelností – tedy za předpokladu, že nemáme přiblížené bloky. Ačkoliv prostředí zobrazuje mezeru vertikálně vystředěnou tečkou, stále je přehlédnutelná. Vložení znaku ukončující řádek zase vyžaduje orientaci v unicodové tabulce. A téměř do nebe volající je nepatrný rozdíl mezi tečkou a čárkou, který na obrázku ani nerozeznáme – natož v samotném prostředí. Takovéto nedostatky jsou důvodem ke vzniku konstant. Konkrétní hodnotu stačí definovat v příkazu [vrať] a názvem funkce vyjádřit vracenou hodnotu. Klidně je možné vytvořit další konstanty pro ostatní interpunkční znaménka, ale to už je pro potřeby demonstrace zbytečné.

Abychom konstanty spadající do oblasti textových znaků nemuseli „lovit“ v paletě bloků, protože při definici dalších znakových konstant by paleta celkem narostla, je lepší vytvořit jeden blok s rozbalovací nabídkou, jejímiž možnostmi si o konkrétní hodnotu zažádáme. Uvnitř bloku poté přes podmínky zjistíme, o kterou hodnotu máme zájem a tu vrátíme. Funkce s rozbalovací nabídkou bez definice je vidět na tomtéž obrázku.



Obrázek 4.1.28 Zaměřením podobné konstanty lze namísto do čtyřech bloků umístit do jednoho s rozbalovací nabídkou

SNAP PROJEKT: Definice imitovaných konstant tvořených funkcemi #5

4.2 Seznamy

Prostředí Scratch 2 umožňuje programátorům využívat seznamy velmi omezeně. Provedeme-li srovnání s prostředím Snap, zjistíme, že jde o naprosto odlišný koncept práce se seznamy, jež ve svém důsledku otevírá dveře novým možnostem při vývoji projektu.

Tento rozdíl je však klíčové pochopit. Už jednou jsme si jej vysvětlili v části podsekce 3.1.3 o datových typech. Není-li si tímto rozdílem čtenář jist, měl by se do této části vrátit a připomenout si celou problematiku včetně způsobu tvorby seznamů funkcí (nový seznam), která na ně vrací referenci, jež užijeme k různým účelům.

4.2.1 Seznam jako typ parametru

Chceme-li pracovat se seznamem v prostředí Scratch 2, používáme k tomu určené primitivy (např. [přidej], (prvek), [zruš] aj.). Abychom upravili těmito bloky požadované seznamy, určujeme jejich názvy v rozbalovacích nabídkách. V Snap však ukládáme bezejmenné seznamy do proměnných příkazem [nastav] a protože prostředí nekontroluje jejich obsah, nemá ani ponětí, v kolika z nich si udržujeme reference na ně.

Proto parametry primitivů pracujících se seznamy v prostředí Snap nemají rozbalovací nabídky, ale jsou typu *seznam*, čímž dávají jasně najevo, že očekávají referenci na již existující seznam. Tyto bloky s typem *seznam* jsou na obrázku 4.2.1.

Obrázek 4.2.1 Bloky pracující se seznamy mající *seznam* jako typ parametru

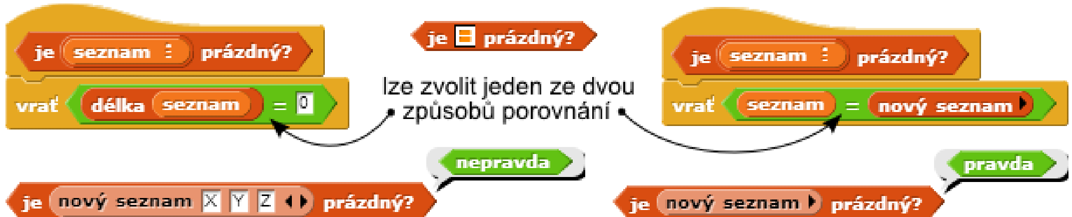
Při tvorbě vlastních bloků, jenž pro svou činnost potřebují referenci na seznam, bychom měli konkrétním parametrům zvolit tento typ. Obrázek 4.2.2 ukazuje několik vlastních bloků pracujících se seznamy, které by programátor mohl ve svých projektech využít. Jsou důkazem, že parametr typu *seznam* má své opodstatnění. Podobných bloků pracujících se seznamy je určitě celá řada. Čtenáře možná napadnou nějaké další příklady.

Obrázek 4.2.2 Příklady vlastních bloků, které pracují se seznamy a využívají k jejich získání k tomu určený typ *seznam* u svých parametrů

Naprogramujme si dva užitečné bloky na ukázku – jednu podmínku a jednu funkci.

- **Podmínka** <je prázdný>

Velmi často potřebujeme ověřit, zdali seznam obsahuje alespoň jeden prvek. Než abychom stále používali funkce <je rovno> či <je větší> v kombinaci s (délka), můžeme si vytvořit vlastní podmínku <je prázdný> obsahující porovnání uvnitř. Svým názvem a zvolenou kategorií je mnohem výřečnější a zlepší tak čitelnost scénáře (viz obr. 4.2.3).



Obrázek 4.2.3 Podmínka ověřující (dvěma způsoby) prázdnost seznamu

Seznamy v Snap můžeme porovnat i bez funkce (délka). Podmínka <je rovno> vždy porovnává u seznamů nejprve počet jejich prvků a v případě shody ještě jejich hodnoty. Lze tak porovnat dodaný seznam s novým – v rámci podmínky – vytvořeným prázdným

seznamem funkcí (nový seznam) (viz obr. 4.2.3 vpravo). Zde je však vhodné zmínit, že pokaždé vytváříme v paměti počítače prázdný seznam – a to v podstatě zbytečně, neboť můžeme porovnávat seznamy prvním způsobem přes (délka).

• **Funkce** (pozice v)

Tážeme-li se seznamu podmínkou <obsahuje> zdali v sobě ukrývá prvek s hledanou hodnotou, pak sice získáme odpověď v podobě pravdivostní hodnoty, avšak stále nebudeme vědět, na jaké pozici se v seznamu nachází. Mnohdy totiž potřebujeme k takovému prvku ihned přistoupit a jeho hodnotu někde použít.

Ve většině programovacích jazycích se tento problém řeší funkcí (pozice v), která se snaží nalézt hledaný prvek v seznamu. Bude-li v hledání úspěšná, vrátí jeho pozici. Jestliže prvek nenalezne, vrátí speciální hodnotu {-1}. Při ověřování existence prvku v seznamu pak tuto hodnotu musíme zohlednit.

V prostředí Scratch 2 si můžeme naprogramovat vlastní příkaz (např. [zjisti pozici prvku]) s parametrem představující hledaný prvek a výsledek ukládat někam do proměnné postavy. Jedná se ale o naprosto složité řešení a ještě potřebujeme proměnnou navíc, kterou ještě může více scénářů s tímto blokem v nevhodnou chvíli přepsat.

Naštěstí si v prostředí Snap funkci (pozice v) můžeme vytvořit, což znázorňuje obrázek 4.2.4 i s ukázkami a příkladem použití. Existencí (pozice v) však podmínku <obsahuje> nenahrazujeme. Stále ji použijeme, když nepotřebujeme znát pozici prvku.

Obrázek 4.2.4 Funkce vracející pozici hledané hodnoty v seznamu s příklady

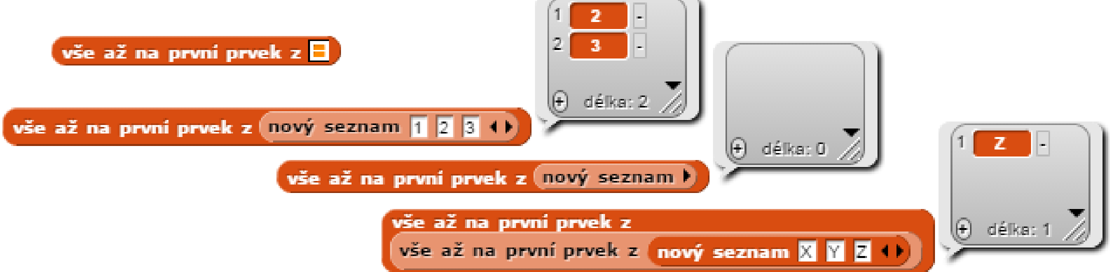
4.2.2 Přidané funkce pro práci se seznamy

Prostředí nabízí další nové funkce (vše až na první prvek) a (vlož do popředí) pro práci se seznamy. Mají velmi specifické chování, které si vzápětí popíšeme. Oceníme je především v rekurzivních blocích vyžadující postupné odebírání či rozšiřování prvků.

• **Funkce** (vše až na první prvek)

Tato funkce s jedním parametrem typu *seznam* vrátí kopii seznamu bez jeho prvního prvku. Předáme-li funkci (vše až na první prvek) seznam s prvky {A, B, C}, vrátí nám seznam nový jen s prvky {B, C}. Zavoláme-li nad tímto vráceným seznamem opět (vše až na první prvek), pak vracený nový seznam bude obsahovat jen jeden prvek {C}. No a když provede zase to samé, vrátí se již seznam prázdný, neboť byl odebrán poslední prvek {C}. Odebírání přes (vše až na první prvek) z prázdného seznamu vždy vyústí ve vrácení nového – taktéž prázdného – seznamu.

Název (vše až na první prvek) byl zvolen proto, že vrací seznam se všemi prvky dodaného seznamu, až na ten první. Zde by někteří mohli namítat, že k rušení prvního prvku seznamu můžeme použít příkaz [zruš]. Ten modifikuje seznam původní, kdežto funkce (vše až na první prvek) modifikuje pouze vracený seznam – ne originál.



Obrázek 4.2.5 Ukázka funkcionality funkce (vše až na první prvek)

Funkci využijeme především v rekurzivně založených algoritmech. Přesto najde využití i v těch iteračních. Následující obrázek 4.2.6 ukazuje, jak pomocí ní procházíme seznam s prvky {A, B, C}, jež je před svým průchodem uchován v (seznam). Jelikož postupně přes (vše až na první prvek) „odřezáváme“ první prvky, musíme se odkazovat na jejich hodnotu ještě než je ztratíme. Proto u funkce (prvek) figuruje argument {1}. Nakonec do (seznam) pokaždé uložíme novou kopii seznamu o jeden prvek kratší. Tento způsob průchodu nevyžaduje počítadlo, což uvítáme u rekurzivních vlastních bloků.

Obrázek 4.2.6 Procházení seznamu novou funkcí bez využití počítadla

Nutno podotknout, že u tohoto řešení ztrácíme možnost obnovit seznam do původního stavu po jeho průchodu. Dochází totiž k postupnému odřezávání (resp. zapomínání) prvků předešlých a referenci na „původní“ seznam si v žádné vedlejší proměnné neuchováváme. Toto řešení si můžeme dovolit jen v některých případech – kupříkladu ve scénářích vlastních bloků, které referenci na seznam získaly zvenčí –, přesto funguje.

Konečně k ukázce využívající funkci (vše až na první prvek) v rekurzi. Funkce (zamíchej prvky) z obrázku 4.2.7 vyžaduje referenci na dva seznamy. Prvky toho prvního ((vmíchávaný)) vmíchá náhodně mezi prvky seznamu druhého ((míchaný)).

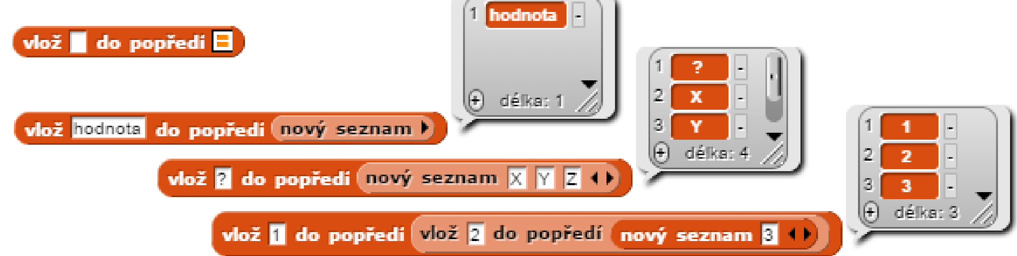
Obrázek 4.2.7 Funkce míchající prvky seznamu prvního mezi prvky druhého

Rekurzivní volání ukončuje zarážka kontrolující prázdnost seznamu. Vmícháváme-li seznam s alespoň jedním prvkem, pak přes příkaz [vlož] vkládáme vždy první prvek vmíchávaného seznamu na náhodnou pozici do (míchaný). Rekurzivní volání (zamíchej prvky) obsahuje (vše až na první prvek), takže při každém rekurzivním vnoření bude naše funkce pracovat se seznamem o jeden prvek kratším pod parametrem (vmíchávaný). Z toho důvodu se opět ve funkci (prvek) odkazujeme na první prvek seznamu argumentem {1}.

• **Funkce** (vlož do popředí)

Druhá funkce funguje na opačném principu, tedy namísto odebírání prvků vkládá prvek nový na úplný začátek seznamu. Použití funkce je zobrazeno na dalším obrázku 4.2.8. Dodáme-li (vlož do popředí) seznam s prvky {X, Y, Z}, pak vrátí nový seznam s hodnotami {?, X, Y, Z}, přičemž {?} budiž vkládanou hodnotou. Tuto funkci užijeme hlavně v rekurzivních blocích a často i v kombinaci s (vše až na první prvek).

Pozorný čtenář bude jistě namítat, že docílí – alespoň na první pohled – stejného výsledku blokem [vlož]. I zde je stejná paralela k rozdílu mezi (vše až na první prvek) a [zruš]. Rozdíl mezi nimi je nejen v tom, že jeden je funkcí a druhý příkazem, ale hlavně [vlož] upravuje seznam původní, zatímco (vlož do popředí) vrací nový modifikovaný seznam. Onen seznam původní tak zůstane netknutý.



Obrázek 4.2.8 Ukázka funkcionality funkce (vlož do popředí)

Vysvětleme si poslední ukázku na obrázku 4.2.8, kde jsou v sobě vnořené funkce (vlož do popředí). Jelikož se funkce v prostředích vyhodnocují od té nejvíce zanořené, nejprve se vytvoří seznam funkcí (nový seznam) s prvkem {3}. Reference na něj přijme funkce (vlož do popředí), která vytvoří nový seznam se stejným prvkem a ještě vloží {2} do popředí. Reference na tento již jednou rozšiřovaný seznam přijme opět (vlož do popředí), která jej zkopíruje a přidá mu na počátek hodnotu {1}. Ve výsledku se vrátí nový seznam se třemi prvky v podobě {1, 2, 3}.

Závěrem si ukažme obě primitivní funkce v neustále zmiňované rekurzivní ukázce, a to vlastní funkci vracející nový seznam s otočeným pořadím prvků dodaného seznamu. Byla nazvána (pozpátku) a její definice je zobrazena na obrázku 4.2.9 První parametr (otáčený) představuje seznam, jež máme v úmyslu otočit. Abychom však neupravovali tento seznam původní, potřebujeme vložit prvky v opačném pořadí do seznamu jiného, který si uchováme pod parametrem (otočený).

Funkce (pozpátku) rekurzivně volá sama sebe do té doby, než dojde funkcí (vše až na první prvek) k úplnému vyprázdnění seznamu (otáčený). Zde zarážka ukončí rekurzivní volání a započne vynořování se, při kterém dojde k přidávání prvků v opačném pořadí přes (vlož do popředí) – nejprve poslední, poté předposlední, pak předpředposlední atd. – do otočeného (otočený). Vespod obrázku je sestavení rekurze.

Obrázek 4.2.9 Vlastní funkce vracející otočenou kopii seznamu původního

Bohužel vlivem rekurze nemůžeme vytvořit prázdný seznam uchovávající otočené prvky uvnitř definice naší funkce. Z toho důvodu musíme jako uživatelé (pozpátku) druhému

parametru předat vždy prázdný seznam, do kterého budeme otáčet prvky. Jak zakrýt tento parametr, aby nás zbytečně neobtěžoval, si ukážeme v podsekci 4.3.3.

4.2.3 Vícenásobná varianta parametru

V předchozí kapitole se podsekce 3.1.4 věnovala parametrům bloků. V ní bylo uvedeno, že v Snap mohou být parametry bloků různých variant. Jednou z nich je *vícenásobná* varianta. Takový parametr bude vybaven zvenku černými šipkami umožňující mu zvolit proměnný počet argumentů. Zvolené hodnoty se pak vloží do automaticky prostředím vytvořeného nového seznamu a předají parametru. Počet argumentů může být n_0.

S vícenásobným parametrem se můžeme setkat u primitiv `(nový seznam)`, `[proměnné scénáře]`, `[spusť]`, `(zavolej)`, `[zahaj]`, `(spoj)` a `(JavaScript funkce)`.

Tuto variantu, přidávající černé šipky, lze přiřadit i parametrům vlastních bloků bez ohledu na jejich typ. Ačkoliv jsme si ještě všechny typy parametrů nevysvětlili, obrázek 4.2.10 zobrazuje tyto typy na vlastním nepojmenovaném pomocném příkazu s jedním parametrem ve vícenásobné variantě nalevo.

Obrázek 4.2.10 Vícenásobný parametr u všech dvanácti dostupných typů

Vzpomeňme na funkci `(nový seznam)`, kterou můžeme dle dodaných argumentů rovnou naplnit seznam. Již máme zkušenost, že počet argumentů může být n – a to včetně nuly pro vytvoření prázdného seznamu. Otázkou tedy zůstává, jak se na tyto variabilní argumenty uvnitř bloku odkážeme ve vlastních blocích, když jednou jich můžeme dostat pět, jindy tři, a někdy žádný.

Na pravé straně obrázku je pomocná funkce `(ukázka)`, jejíž jediný parametr `(vícenásobný)` je *libovolný* a vícenásobné varianty. Prostředí dodané variabilní argumenty vždy automaticky převede do seznamu, který se předá dovnitř bloku tomuto parametru. Před vstupem do funkce `(ukázka)`, jež vrací hodnotu parametru `(vícenásobný)`, tak dojde vždy k vytvoření seznamu naplněného hodnotami argumentů. Pohlédneme-li opět na obrázek, pak vidíme ukázku s nula argumenty (vyústí v prázdný seznam) a tři nevyplněné argumenty (vrátí se seznam s třemi prázdnými prvky).

V jakých dalších příkladech je možné vícenásobnou variantu parametru využít se snaží nastínit následující obrázek 4.2.11, jež obsahuje bloky různého typu a kategorií.

Obrázek 4.2.11 Tři primitivní a pak různé vlastní bloky s vícen. parametrem

Podmínka `<dotýká se barvy>` má u barev vícenásobný parametr a umožňuje kontrolovat dotek s více barvami. Byla ale uměle vytvořena autorem této práce (v grafickém editoru), neboť ten počítá s tím, že časem budeme moci zvolit i takovýto typ parametru, jakmile Snap zavede další datový typ *barva*. Zatím tedy nelze vytvořit.

Dále všechny zaobalující funkce (, , a) mají černé šipky k přidání či odebrání parametrů. To zmiňujeme proto, že na obrázku je ukázána pouze . No a nakonec primitivní příkaz `[zahaj]` je speciálním (stejně jako jemu podobné `[spusť]` a `(zavolej)`), jenž po zvolení alespoň jednoho argumentu černými šipkami doplní nápis „s argumenty“. Vysvětlíme si je všechny až v jiné sekci. Tolik k restům z obrázku 4.2.11.

- **Příklady**

K sečtení více čísel musíme do sebe několikrát vnořit funkci (sečti), což je ale způsob zbytečně složitý. Jakékoliv pozdější úpravy či rozšíření výpočtu si žádá manipulaci se všemi vnořenými funkcemi. Zkusme si zjednodušit práci a naprogramujme vlastní funkci (sečti) s jediným parametrem typu *seznam*. Ta postupně projde a sečte číselné hodnoty prvků předaného seznamu a nakonec vrátí výsledný součet. Definice funkce je zobrazena na následujícím obrázku 4.2.12.

Obrázek 4.2.12 Funkce vyžadující seznam jehož prvky sečte a vrátí výsledek

Na obrázku vidíme, že funkce vrátí {0}, když nedostane žádnou referenci na seznam. To proto, že používáme ve scénáři (délka) namísto (délka), která by při ověřování délky prázdného seznamu způsobila chybu. Pak vypočítáváme součet hodnot v dodaném bezejmenně vytvořeném seznamu s prvky {10, 20, 30, 40, 50}. No a nakonec můžeme ještě předat referenci na seznam třeba přes proměnnou (seznam_čísel).

Sice jsme si existencí vlastní funkce (sečti) usnadnili sčítaní více než dvou hodnot, ale pořád se musíme starat o dodání nějakého seznamu s čísly. Navíc protože je parametr typu *seznam*, není zřejmé, jakých hodnot by měl nabývat. Pojďme vylepšit tuto funkci vybavením parametru (seznam_čísel) vícenásobnou variantou namísto jednoduché a změnou jeho typu z *seznam* na *číslo*. Výsledný rozdíl je na obr. 4.2.13.

Obrázek 4.2.13 Stejná funkce ke sčítání čísel s vícenásobným parametrem

Podíváme-li se na ukázky vpravo, všimneme si, že byl parametr zvenku bloku vybaven černými šipkami. Nyní je zřejmé, že očekáváme čísla, neboť typ tohoto parametru je oválného tvaru. Taktéž se už nemusíme starat o nějaký seznam, jelikož za argumenty doplníme hodnoty k součtu ručně. Scénář funkce zůstal naprosto stejný. Jediná změna nastala v symbolu parametru, který se ze tří svislých teček (symbolizující seznam) změnil na tři vodorovné tečky (symbolizující vícenásobnou variantu parametru).

- **Předání reference na existující seznam vícenásobnému parametru**

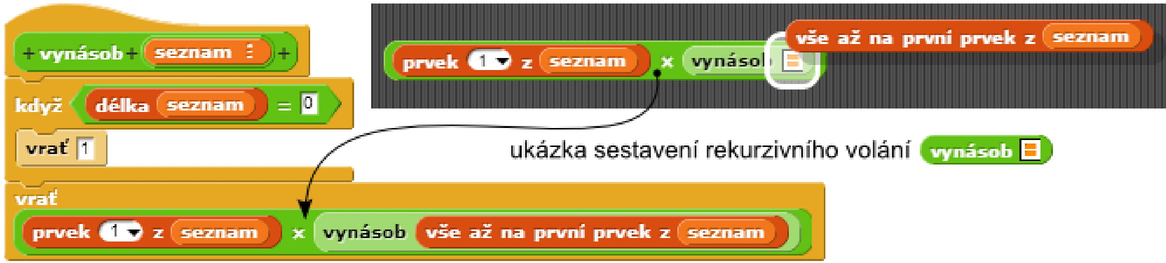
Zůstaňme stále u vylepšené funkce (sečti). Jeden může namítat, že není tak výhodná, jako její předchůdkyně z obrázku 4.2.12, neboť ji nelze předat odkaz na seznam. To ale není pravda. Přeci jsme si řekli, že obě funkce (z obrázků 4.2.12 a 4.2.13) jsou totožné. Obě pracují se seznamy a oběma je také možné referenci na seznam předat.

Pro předání reference na seznam vícenásobnému parametru je zapotřebí onu referenci upustit na černé šipky parametru. Prostředí bude indikovat tuto skutečnost

červeným ohraničením. Vícenásobný parametr se pak proměnní v popisek „prvky seznamu:“ a za ním bude blok, který jsme na šipky upustili. Obrázek 4.2.14 napoví víc.

Obrázek 4.2.14 Způsob jak předat vícenásobnému parametru referenci na již existující seznam

A protože tento způsob předání reference využijeme u rekurzivně založených algoritmů, byla by škoda, kdybychom si žádný takový neukázali. Pro změnu si namísto (sečti) naprogramujme vlastní funkci (vynásob) ve dvou verzích. Ta první je na obrázku 4.2.15 a její jediný parametr je typu *seznam* jednoduché varianty. Obrázek také ukazuje, jak se rekurzivní volání sestavilo, což je důležité pro pochopení následujícího popisu.



Obrázek 4.2.15 Rekurzivní násobící funkce s parametrem jednoduché varianty typu *seznam*

Abychom nemuseli znovu shánět referenci na seznam naplněný čísly určených k součinu, změňme typ parametru (seznam) z *seznam* na *číslo* a zvolme mu vícenásobnou variantu. Jelikož se změnila hlavička naší funkce (vynásob), musíme na to zareagovat v oblasti rekurzivního volání. Proto jsme v příkazu [vrať] museli sestavit bloky jiným způsobem, což demonstruje obr. 4.2.16. Z tohoto řešení těžíme především možnost vkládat číselné hodnoty namísto obstarávání seznamu s hodnotami v něm vložených.

Obrázek 4.2.16 Upravená rekurzivní (vynásob) jiného typu a varianty

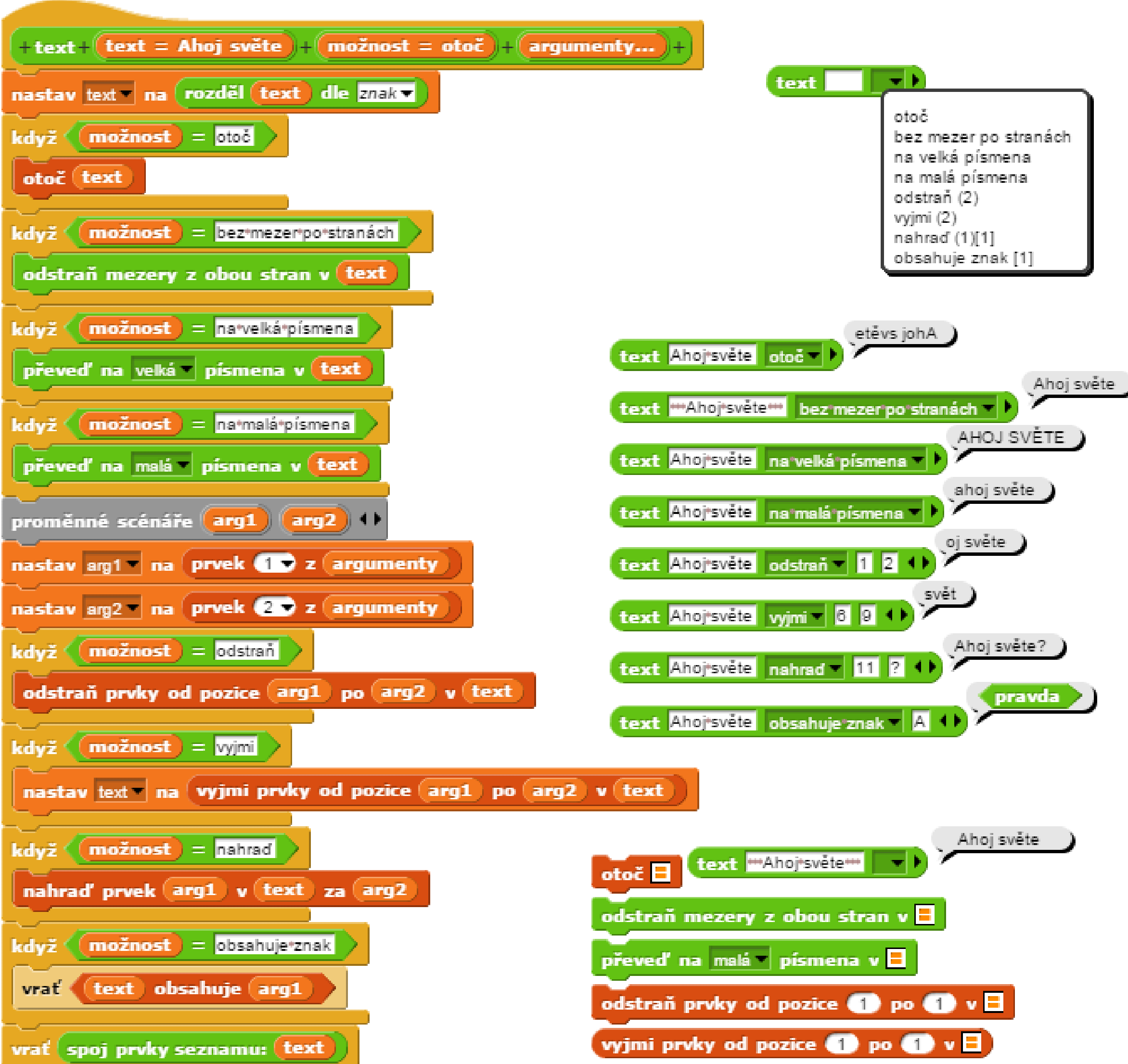
• Vícenásobný parametr jako opravdu variabilní z jiného pohledu

Manipulace s texty je ve všech prostředích velmi omezená. V podstatě můžeme texty jen spojovat funkcí (spoj), přistupovat k jejich jednotlivým písmenům funkcí (písmeno), a zjišťovat jejich délku funkcí (délka). Nahradit některá písmena za jiná, vložit písmeno na určitou pozici, odstranit některá, otočit text pozpátku, či ověřit výskyt nějakého písmena v něm; nelze provést jednoduchým způsobem. Vždy musíme přetvořit upravený text od jeho začátku a aplikovat kýžené úpravy před jeho dokončením.

V programovacích jazycích je text vytvořen jako seznam znaků, neboť se se znaky v seznamu snáze pracuje. I tak je tomu ve všech prostředích na nižší úrovni v jazycích, ve kterých jsou naprogramována. V prostředí Snap máme k dispozici funkci (rozděl),

která zadaný text umí rozdělit do seznamu. Jakmile tak učiníme, otevřou se nám další možnosti pro práci s textem, neboť najednou užijeme i bloky z kategorie `seznamy`.

Následující funkce `(text)` byla vytvořena v prostředí Snap. Přijímá textovou hodnotu, kterou okamžitě funkcí `(rozděl)` rozdělí znak po znaku přes zvolenou možnost `{znak}`. Tento seznam pak uchováváme v parametru `(text)` reprezentující zadaný text, který je však v tu chvíli rozložen v seznamu. Dále disponuje parametrem `(možnost)` s needitovatelnou rozbalovací nabídkou, mezi jejímiž možnostmi nabízíme různé operace proveditelné nad textem. Jde o možnosti `{otoč/bez mezer po stranách/na velká písmena/na malá písmena/odstraň/vyjmi/nahraď/obsahuje znak}`. Jelikož každá operace vyžaduje odlišný počet argumentů ke svému provedení, potřebujeme vícenásobný parametr, jenž se v bloku nazývá `(argumenty)`. Ještě než si popíšeme konkrétní příklady, podívejme se na definici této funkce na obr. 4.2.17. Aby byl scénář funkce `(text)` přehlednější, využívá několik pomocných bloků (umístěných vpravo dole).



Obrázek 4.2.17 Funkce `(text)` nabízející operace nad textovými hodnotami

Příklad si ukazujeme nejenom proto, abychom si uvědomili jak snadno lze přidělat do prostředí operace manipulující s textem, ale hlavně kvůli vícenásobnému parametru.

Tak třeba operace {otoč} otáčející text (tj. všechny jeho písmena v seznamu) nevyžaduje žádný argument, neboť k otočení netřeba žádných upřesňujících informací. Před zavoláním funkce (text) s touto možností tak nemusíme černými šipkami přidat ani jeden argument. To samé platí pro některé z dalších možností u funkce (text).

Pak ovšem existují operace, které potřebují alespoň jeden argument. V kontextu s (text) jde o operaci ověřující výskyt nějakého znaku v textu. V rozbalovací nabídce je za touto možností {obsahuje znak} uvedeno [1], značící potřebu jednoho libovolného argumentu. Černými šipkami tak přidáme jeden argument při používání této operace, na jehož výskyt v textu se tážeme.

Operace odstraňující či vyjímající některé části z textu zase potřebují parametry dva – a to číselné. Proto v možnostech {vyjmi} a {odstraň} je (2) značící potřebu dvou číselných hodnot stanovujících rozmezí vyjímané/mazané části z textu.

Nakonec jediná možnost {nahraď} potřebuje argumenty dvojího typu. První musí být číslem určující pozici písmene v textu, jež bude nahrazeno. Druhý argument pak představuje nahrazující znak. Proto je v nabídce u možnosti uvedeno (1)[1].

Z výše uvedeného si můžeme udělat představu o využitelnosti vícenásobné varianty parametru. Dle konkrétních potřeb bloku (zvolených možností) se odvíjí počet argumentů nutných k úspěšnému provedení některé z operací nad textem.

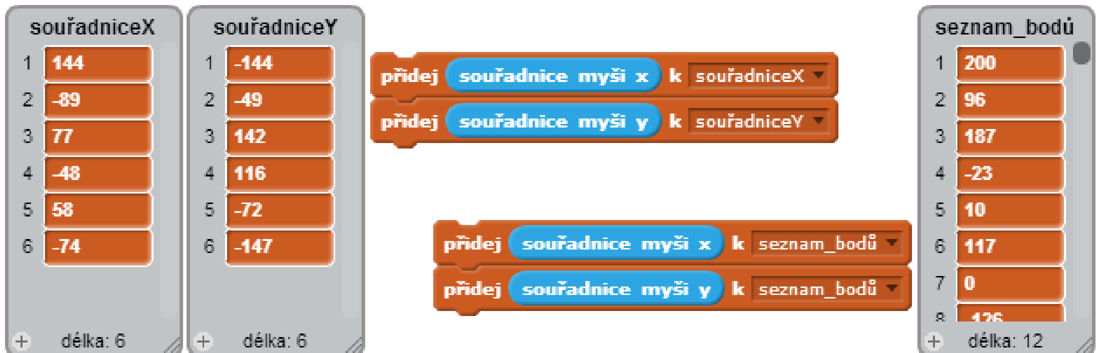
SNAP PROJEKT: Definice bloků s parametry vícenásobné varianty #6

4.2.4 Vnořené seznamy

Již víme, že vzhledem k prvotřídnosti seznamů je možné uchovat seznam v jiném (hlavním) seznamu. Takové seznamy uvnitř jiných seznamů nazývejme *vnořenými*. Stačí jen referenci na vnořený seznam vložit do seznamu hlavního. Ukažme si, kde bychom vnořené seznamy mohli využít.

V prostředí Scratch 2 k uchování několika dvourozměrných bodů můžeme postupovat dvěma způsoby. Každý dvourozměrný bod je složen z x-ové a y-ové souřadnice na scéně. Protože při posunu postavy potřebujeme k oběma hodnotám přistoupit zvlášť, je možné si je uchovávat ve dvou seznamech. Jeden bude sloužit k uchování souřadnice x a druhý k uchování souřadnice y. Prvky na stejných pozicích v obou těchto seznamech pak představují definici jednoho dvourozměrného bodu. Více viz obr. 4.2.18 vlevo.

Nelíbí-li se nám dva seznamy, můžeme hodnoty souřadnic x a y vkládat do seznamu jednoho (s názvem „seznam bodů“). Problémem ovšem je, že není přímo zřejmé, zdali ukládáme body dvourozměrné či trojrozměrné, neboť jsou všechny hodnoty uvedené v seznamu za sebou a nejsou nijak odděleny (viz obr. 4.2.18 vpravo).



Obrázek 4.2.18 Dva způsoby jak ukládat dvourozměrný bod do seznamů

Následující projekt je hrou, ve kterém spojujeme body reprezentované černými puntíky na scéně. Spojením bodů ve správném pořadí, neboť jsou očíslovány, vynikne schovaný obrazec. Každé kliknutí na scénu přesune postavu s kostýmem tužky na místo kurzoru

myši a vykreslí čáru vedoucí od její předchozí pozice k současné. Zároveň se uloží tento bod do dvou seznamů (souřadniceX) a (souřadniceY) tak, jak je ukázáno v předchozím obrázku nalevo. Jakmile nabudeme dojmu, že jsme všechny černé puntíky spojili, stiskněme mezerník. Tužka vymaže čáry pera a začne postupně navštěvovat všechny uložené body v seznamech. Bude tak opakovat náš postup spojování bodů. Cílem hry je ukázat, jak se dvourozměrné body v prostředí Scratch 2 dají uchovat a jak k jejich hodnotám přistupujeme. Jestli správně napojíme body či nikoliv je nepodstatné.

SCRATCH 2 PROJEKT: Hra - spoj body (1. verze) #7

Existuje ještě třetí způsob v prostředí Scratch 2, jak dvourozměrné body do jednoho seznamu uložit. Každý prvek bude obsahovat všechny potřebné hodnoty – tedy souřadnice x a y pohromadě, které však oddělíme nějakým oddělovačem. Kdybychom tak neučili, čísla by se spojila do jednoho a už bychom hodnoty nikdy nezískali v původní podobě zpět. Obrázek 4.2.19 znázorňuje popisovaný způsob uchování bodů.

Obrázek 4.2.19 Třetí způsob jak uchovat bod v jednom prvku seznamu

Nevýhodou tohoto řešení je však potřeba naprogramování scénáře, který spojené informace o bodu rozloží. Tedy aby například bod reprezentovaný textem {125;123} rozdělil buď na {125} (x) či na {123} (y). Navíc jelikož v prostředí Scratch 2 nemůžeme snadno pracovat s textem, bude i obtížné měnit hodnotu některé ze souřadnic bodu uloženém v seznamu. Z {125;123} neuděláme např. {125;-9} jednoduchým způsobem. Ačkoliv je bod reprezentován jedním prvkem seznamu, těžko pracujeme s hodnotami zvlášť.

Konečně se dostáváme k prostředí Snap, ve kterém lze dvourozměrný bod reprezentovat vlastním seznamem. Co bod, to seznam s dvěma prvky – souřadnicemi x a y. Všechny body (tedy seznamy) pak uložíme do jednoho hlavního seznamu. Stanou se tak seznamy vnořenými. Obrázek 4.2.20 ukazuje tento způsob ve sledovači. Kromě toho je na něm vytvořen příkaz [přidej bod] s dvěma parametry přidávající nový seznam (dvourozměrný bod) s hodnotami parametrů (x) a (y) do hlavního seznamu bodů.

Obrázek 4.2.20 Seznam reprezentující bod vložený do seznamu bodů

Procházení bodů v prostředí Snap lze mimochodem provést velmi jednoduše. Stačí si vytvořit vlastní příkaz [posuň se na bod] s parametrem očekávající referenci na

seznam. Tento příkaz sám získá souřadnici x a y ze seznamu dvourozměrného bodu. Definice a ukázky jsou zobrazeny na nadcházejícím obrázku 4.2.21.

Obrázek 4.2.21 Jednoduchý způsob jak posunou postavu na bod ze seznamu

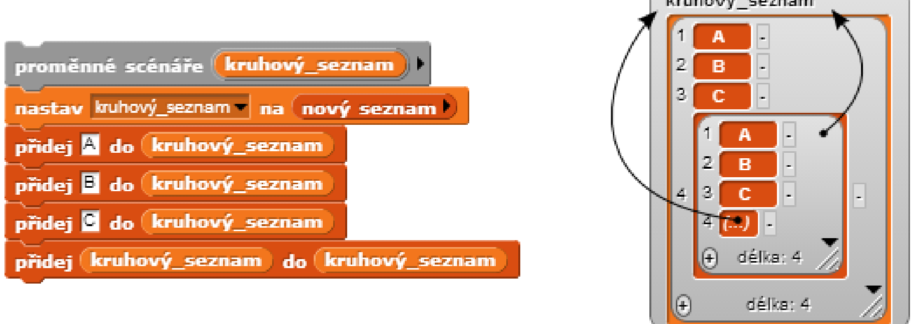
Oproti Scratch 2 můžeme nyní snadno manipulovat se všemi hodnotami bodu. Také jsou tyto hodnoty (x a y) rozděleny do dvou prvků v seznamu. Vnořené seznamy (body) jsou jasně odlišitelné od ostatních. Předchozí projekt s hrou byl vylepšen v prostředí Snap o toto řešení – ukládání, přidávání, a přistupování k dvourozměrným bodům.

`SNAP PROJEKT: Hra - spoj body (2. verze)` #8

4.2.5 Kruhové seznamy

Do seznamů vkládáme různé hodnoty, které se uchovávají jakožto jeho prvky. Protože v Snap pracujeme s referencemi na seznamy, můžeme do nějakého seznamu vložit referenci na něj samotný. Přidáme-li referenci „sebe sama“ na poslední pozici seznamu, vytvoříme tzv. *kruhový seznam*. Jde o vylepšení obyčejného seznamu. Kruhový se nazývá proto, že nemá konec – poslední prvek míří na sebe sama – a tak při průchodu tohoto seznamu jej v podstatě můžeme procházíme donekonečna.

Na obrázku 4.2.22 je tvorba kruhového seznamu v prostředí Snap znázorněna. Nejprve byl vytvořen seznam prázdný a reference na něj uchována v proměnné `(kruhový_seznam)`. Poté jsme třikrát vložili různé prvky a nakonec referenci na sebe sama. Vzniká jakési nekonečné vnoření do sebe sama, což se snaží ukázat i sledovač, který namísto aby nekonečné krát zobrazoval vnoření raději použije popisek „*(...)*“.



Obrázek 4.2.22 Způsob tvorby a znázornění kruhového seznamu v Snap

Abychom nemuseli při tvorbě kruhového seznamu neustále používat zmíněné způsoby vyžadující příkaz `[přidej]`, můžeme si udělat vlastní funkci `(nový kruhový seznam)`, která bude podobná primitivní funkci `(nový seznam)`. Její definice s ukázkami je zobrazena na obrázku 4.2.23. Vícenásobný parametr `(prvky)` je typu *libovolný* a obsahuje různé hodnoty, jež se do nového seznamu přidají. Nakonec přes `[přidej]` v rámci naší funkce vložíme referenci na seznam na poslední pozici. Vrácený seznam bude kruhovým.

Obrázek 4.2.23 Funkce tvořící kruhový seznam s vícenásobným parametrem

Kruhové seznamy musíme procházet speciálním způsobem, který je zobrazen na obrázku 4.2.24. Funkcí (vše až na první prvek) se vždy posuneme o další prvek vpřed. Kdy se posuneme o další prvek v kruhovém seznamu však určíme jen my. Proto by bylo vhodné tuto část scénáře vložit do vlastního příkazu s názvem například [další prvek kruhového seznamu]. Po vykonání tohoto bloku se posuneme v kruhovém seznamu na další prvek, k němuž přistoupíme funkcí (prvek) s argumentem {1}. Jakmile dorazíme na referenci mířící na sebe sama, níže uvedený scénář umí do tohoto seznamu vstoupit a opět jej začít procházet od počátku. Tento autorem navržený systém průchodu funguje na principu dávkování, kdy si ve vhodné chvíli požádáme o posun na další prvek kruhové seznamu.

Obrázek 4.2.24 Jeden ze způsobů průchodu kruhovým seznamem

V ukázce o způsobu procházení kruhového seznamu je třeba vybírat a používat názvy proměnných, v nichž si referenci na kruhové seznamy uchováváme. Máme-li v projektu vícero kruhových seznamů, pak pro každý zvlášť musíme zbytečně vytvořit příkaz [další v řadě seznamu1], [další v řadě seznamu2], [další v řadě seznamu3] atd. Lze ovšem naprogramovat obecný blok k procházení kruhového seznamu, jehož definice je zobrazena na obr. 4.2.25. K jeho pochopení je však vyžadováno prostudovat budoucí podsekci 4.4.4 popisující typ parametru *libovolný (nevyhodnoceno)*.

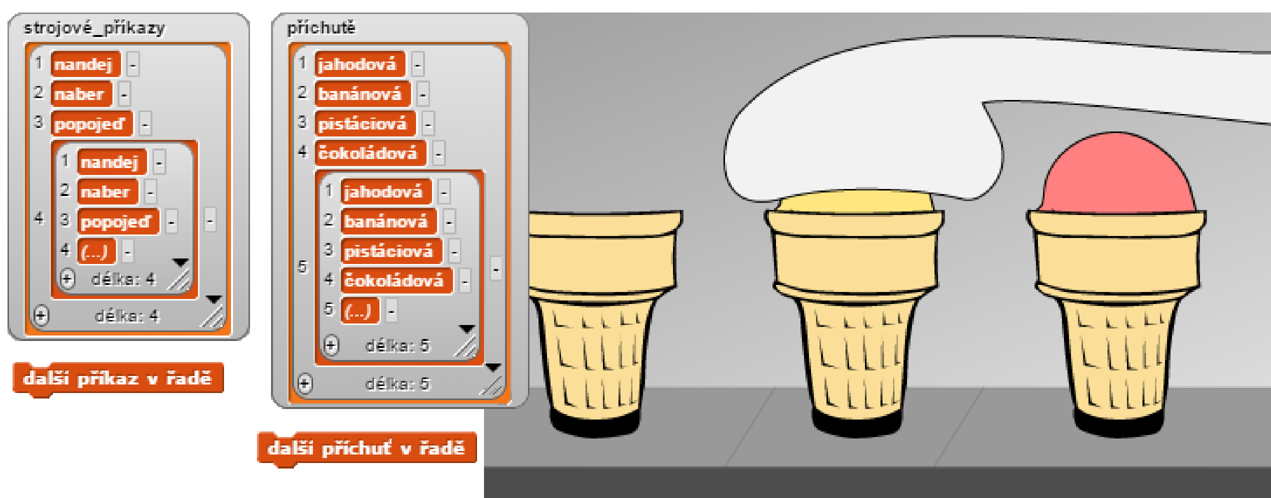
Obrázek 4.2.25 Definice příkazu k průchodu libovolného kruhového seznamu

- **Příklad**

Konečně po sáhodlouhých vysvětleních jak pracovat s kruhovými seznamy se dostáváme k jejich užití v příkladu. Následující projekt obsahuje dva kruhové seznamy. Tento projekt se nehraje, funguje totiž automaticky. Jedná se o stroj na zmrzlinu, který posouvá na pásu prázdné kornouty, do kterých stroj plní vždy jeden kopeček zmrzliny různé příchuti. Tato výrobní linka neustále opakuje tyto kroky:

1) popojeď s pásem – na určenou pozici najede kornout bez zmrzliny
2) naber další kopeček – lžíce stroje sama nabere kopeček zmrzliny určité příchuti
3) nandej kopeček zmrzliny – lžíce nandá kopeček do kornoutu

Proto byly tyto kroky v textové podobě uchovány v kruhovém seznamu. Střídají se k tomu vytvořeným příkazem `[další příkaz v řadě]`. Kromě toho se neustále střídají příchutě zmrzliny. Ty jsou též uloženy v kruhovém seznamu v textové podobě. Příkazem `[další příchuť v řadě]` se pak posuneme na následující příchuť v seznamu.



Obrázek 4.2.26 Ukázka kruhových seznamů z projektu „stroj na zmrzlinu“

Kruhové seznamy byly zvoleny proto, že neustále dokola procházíme jejich obsah. Nepotřebujeme žádné počítadlo na postupné procházení prvků seznamu. Stačí si ve vhodnou chvíli přeskočit na další prvek k tomu vytvořenými vlastními příkazy. Všechny kroky a příchutě se totiž opakují dokola v pevně stanoveném pořadí.

`SNAP PROJEKT: Stroj na zmrzlinu` #9

4.2.6 Kopírování seznamů

Protože Scratch 2 pracuje jen s fixními seznamy vytvořenými uživatelem přes rozhraní prostředí, o nějakém kopírování a vzniku seznamů programově kdykoliv za běhu scénáře můžeme pouze snít. To ovšem neplatí u seznamů v prostředí Snap.

Jelikož ale nedisponuje funkcí na kopírování seznamů, musíme si ji naprogramovat sami. Už v dodatku C pojednávající o referencích jsme zmínili pojmy *mělká* a *hluboká* kopie. Chceme-li zkopírovat seznam kompletně, musíme provést onu kopii *hlubokou*, což spočívá v tvorbě nového seznamu na jiném místě v paměti počítače funkcí `(nový seznam)` a překopírování všech prvků seznamu původního do toho nového.

Algoritmus kopírovací funkce skýtá určitá „nebezpečí“, a proto se tématu kopírování blíže věnujeme i v této podsekci. Ukažme si zatím jednoduchou kopírovací funkci `(kopíruj)` vracející referenci na hlubokou kopii originálu na obrázku 4.2.27 vlevo. Vespod je ještě ukázka tvorby a porovnání referencí kopírovaného seznamu s originálem. Na ní nás může zarazit, že podmínka `<je rovno>` vrátila `{<pravda>}`. Tato podmínka u seznamu totiž porovnává délku seznamů a v případě shody ještě porovná hodnoty těchto prvků. Ty jsou pro oba seznamy z ukázky identické – `{A, B, C}`.

Obrázek 4.2.27 Dva způsoby kopírování seznamů a porovnání referencí

Kopírování obyčejných seznamů je bezproblémové. Avšak kdybychom kopírovali seznam kruhový či seznam s vnořenými seznamy, jejichž některé prvky obsahují reference na seznamy odlišné, neprovedla by se úplná hluboká kopie. Uvedeným kopírovacím algoritmem kopírujeme pouze hodnoty prvků první úrovně seznamu, a tak zkopírujeme i reference na případné vnořené seznamy. Kopírování kruhového seznamu by se sice na první pohled provedlo úspěšně, ale jeho poslední prvek (mířící na sebe sama) by stále mířil na originální (kopírovaný) kruhový seznam převzatou původní referencí na něj.

Pojďme celý problém vyřešit úpravou funkce `(kopíruj)`. Pomocná proměnná `(prvek)` bude nabývat vždy hodnoty aktuálně procházeného prvku originálního seznamu z `(originál)`. Jestliže je `(prvek)` referencí na seznam, nastávají dvě možnosti:

a) buď jde o referenci na sebe sama, tudíž o kruhový seznam; anebo
b) jde o vnořený seznam obyčejný či kruhový (avšak vnořený kruhový)

Je-li kopírovaný seznam kruhovým, musíme vytvořit z jeho kopie také kruhový seznam, a to přidáním reference na sebe sama – tedy vložit `(kopie)` do seznamu `(kopie)` příkazem `[přidej]`. V opačném případě je seznam vnořeným, který musíme také zkopírovat, což provedeme rekurzivním voláním. Správná kopírovací verze je na obr. 4.2.27 vpravo.

4.2.7 Tvorba datových struktur vnořenými seznamy

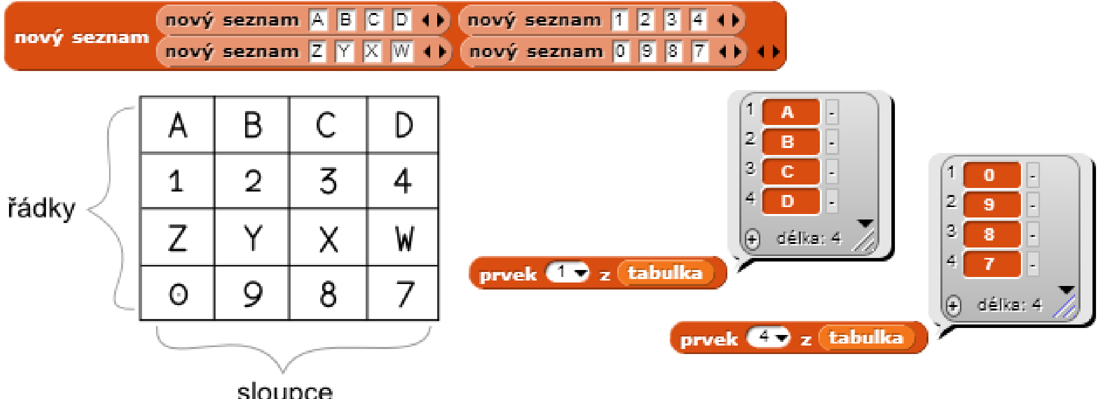
Určitým vnořováním seznamů lze uchovat informace v takové podobě, že se při správné manipulaci s nimi podobají některým známým datovým strukturám. V této podsekci si vytvoříme dvě datové struktury postavené na seznamech a to *tabulku* – řádky, sloupce a buňky obsahující –, a poté *slovník* – strukturu fungující na principu `KLÍČ-HODNOTA`.

Struktury jsou jen seznamy v seznamech. Záleží však na tom, v jakém pořadí a jak hluboko jsou do sebe vnořeny. Proto prostředí Scratch 2 nemá vhodné podmínky pro práci se sofistikovanějšími datovými strukturami, neboť je nutné vytvářet kvanta pojmenovaných seznamů. Zato v prostředí Snap stačí pojmenovat celou strukturu – uložením hlavního seznamu do proměnné – a další vnořené seznamy (či záznamy) pojmenovávat nemusíme. K jednotlivým z nich pak přistupujeme pomocí funkce `(prvek)`.

Manipulace s vnořenými seznamy, převážně přístup a přeřazování prvků, je v prostředí Snap velmi těžkopádná. Je zapotřebí provést hodně práce k dosažení malé změny. Proto se v projektech, které si čtenář otevře, vyskytuje mnoho pomocných bloků z kategorie `seznamy` provádějící potřebné – v prostředí složitě naprogramované – operace nad konkrétními datovými strukturami.

- **Tabulka**

Už bylo nastíněno, že se tabulka skládá z řádků a sloupců. V místě kde dojde k protnutí nějakého řádku a sloupce rozeznáváme buňky. Obrázek 4.2.28 jednu takovou ukazuje.



Obrázek 4.2.28 Tvorba tabulky v prostředí Snap a přístup k řádkům

Na obrázku jsme vnořili čtyři seznamy do hlavního seznamu. Ten první s prvky `{A, B, C, D}` představuje první řádek uvedené tabulky. To platí analogicky pro ostatní. Poté jsme funkcí `(prvek)` získali z hlavní tabulky jednotlivé řádky (první a čtvrtý) a jelikož jsme řádky definovali seznamy, pak se také tyto seznamy funkcí `(prvek)` vrátily.

Bohužel nemáme možnost získat sloupec v seznamu, neboť jsou jednotlivé hodnoty sloupce uchovány ve čtyřech různých vnořených seznamech. Podívejme se alespoň na způsob, jak přistoupit k hodnotě nějaké buňky v tabulce (viz obr. 4.2.29).

Obrázek 4.2.29 Přístup k buňkám z třetího sloupce tabulky

Pokus o průchod celé tabulky, respektive vyřčení blokem `[povídej]` všech jeho prvků, se však značně zkomplikuje. Vyžaduje totiž řešení o dvou vnořených cyklech. Následující obrázek 4.2.30 je toho důkazem.

Obrázek 4.2.30 Scénář procházející všechny buňky tabulky

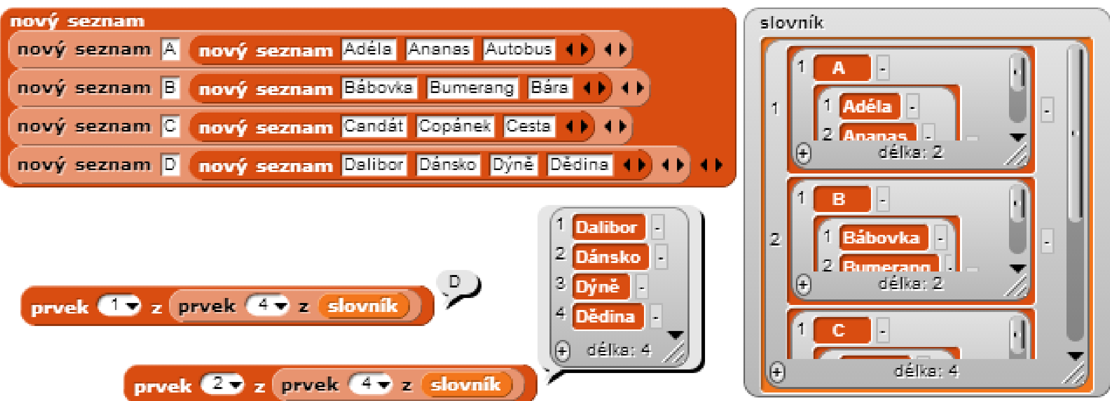
Následující projekt obsahuje hru s dvoubarevnými políčky v tabulce 4x4. Při spuštění hry projekt tyto políčka zamíchá náhodným způsobem. Cílem hry je barevně sjednotit všechna políčka. Kliknutím na některé z nich dojde k prohození barvy všech ostatních políček v jeho řádku a sloupci. Na tomto projektu cvičíme práci s tabulkou – prohazování hodnot buněk, průchod řádků a sloupců, kontrolu barev všech buněk apod.

Jelikož projekt obsahuje mnoho pomocných příkazů, nebudeme si jej dále rozepisovat. Snad jen zmíníme, že tabulka je uložena v proměnné (políčka) a má-li hráč problém hru vyřešit, nechť použije nápovědu obsahující pozice políček, na která má v určeném pořadí kliknout k úspěšnému dokončení hry.

SNAP PROJEKT: Hra - barevná políčka #10

- **Slovník**

Nakonec si ukažme ještě jednu strukturu – *slovník*. Na následujícím obrázku 4.2.31 vkládáme do hlavního seznamu několik dalších seznamů s dvěma prvky. Prvním prvkem je vždy písmeno abecedy. Druhým prvkem je pak seznam se slovy začínajícími na ono písmeno. Vytvořili jsme tak slovník simulací vztahu KLÍČ-HODNOTA. Poznamenejme, že slovník nemusí obsahovat vždy písmena abecedy a jimi začínající slova.



Obrázek 4.2.31 Definice slovníku v prostředí Snap

Všichni známe hru „slovní fotbal“, ve které každý hráč vyřkne určité slovo, které ještě ostatními hráči nebylo řečeno. Následující hráč musí vymyslet takové slovo, které začíná na poslední písmeno předchozího vyřčeného slova. Hráč ze hry vypadne tehdy, nezná-li vhodné slovo a hra skončí, jakmile zbude poslední hráč.

Taková hra je vytvořena v dalším projektu, ve kterém jsou čtyři fotbalisté hrající slovní fotbal na hřišti. Hráči, který je na řadě, je vždy předcházejícím hráčem kopnutý míč. Pořadí hráčů určuje kruhový seznam, jenž je naplněn jejich jmény (jmény postav). Jakmile nějaký hráč vypadne, zmizí ze scény, jeho jméno se odstraní z kruhového seznamu, a ostatní hráči pokračují ve hře předáváním si míče.

Jelikož fotbalisté potřebují určitou množinu slov, musel autor projektu naimportovat do prostředí textový soubor s jím vytvořenou databází slov. Přítomnými scénáři databázi rozdělil do struktury *slovník*. Každý hráč si pak zkopíruje kopírovací funkcí onen slovník a vyhodí z něj několik slov pro každé písmeno – to proto, aby hráči znali odlišná a ve výjimečných případech shodná slova.

SNAP PROJEKT: Hra - slovní fotbal #11

4.3 Zaobalené procedury

V prostředí Snap můžeme část scénáře (přesněji složené příkazy, funkce, a podmínky) „zaobalit“ do zaobalujících funkcí (, , a), jež na bloky v nich zaobalené (zaobalenou proceduru) vrací referenci. Zaobalenou proceduru nikdy nevykonají, neboť to je účelem, který lze vyjádřit slovy: „*Uchovej nějaké bloky na později a vykonej si je ve správnou chvíli*“. K vykonání zaobalených bloků slouží primitivní bloky [spusť] a (zavolej). O zaobalujících funkcích jsme napsali mnoho už v předchozí podsekci 3.1.3 o datových typech, kterou je vhodné si v tuto chvíli zopakovat.

Vše s nimi spojené si v této sekci vysvětlíme. Musíme však vzhledem k náročnosti problematiky postupovat po krůčcích. Ještě před příklady si vyjmenujme jejich užití:

- uchování scénáře a jeho použití na více místech bez nutnosti tvorby vlastního bloku
- změna chování odkazováním se na zaobalené bloky bez nutnosti úprav scénáře
- zakrytí potřebných parametrů vlastních bloků při programování koncové rekurze
- ovlivnění algoritmu uvnitř bloku zaobalenými bloky, jenž předáme parametru

• Funkcionalita a způsob zaobalování

Při běhu námi sestaveného scénáře se všechny jeho bloky postupně vykonávají. Pakliže sestavíme nějakou sekvenci příkazů, provedou se jeden po druhém. Složíme-li více funkcí či podmínek do sebe, pak se postupně zavolají v určitém pořadí a vrátí svůj výsledek. Na obrázku 4.3.1 jsou tři bloky ([povídej], (vyděl), <turbo mód>), které po kliknutí na ně v oblasti scénářů byly prostředím automaticky vyhodnoceny.

Obrázek 4.3.1 Tři vyhodnocené bloky prostředím po kliknutí na ně

To pro nás není žádným překvapením. Takovou funkcionalitu pro vykonání bloku používáme běžně. Příkaz se vykoná a funkce s podmínkou vrátí výsledek v jejich bublině. Cílem ukázky je připomenout, že dochází k okamžitému vyhodnocení bloků.

Nyní zkusme tyto bloky zaobalit k tomu určenými zaobalujícími funkcemi. Příkaz [povídej] do , funkci (vyděl) do , a podmínku <turbo mód> do . Po žádosti o jejich vykonání vrátí referenci na sebe sama, což demonstruje obrázek 4.3.2.

Obrázek 4.3.2 Tři bloky zaobalené do zaobalujících funkcí

Zaobalující funkce je totiž chrání před vykonáním a to vrácením reference na sebe sama. Abychom zaobalené příkazy, funkce, a podmínky vyhodnotili, musíme je vložit buď do příkazu [spusť] (očekávající) anebo do (zavolej) (očekávající či). Více napoví obrázek 4.3.3.

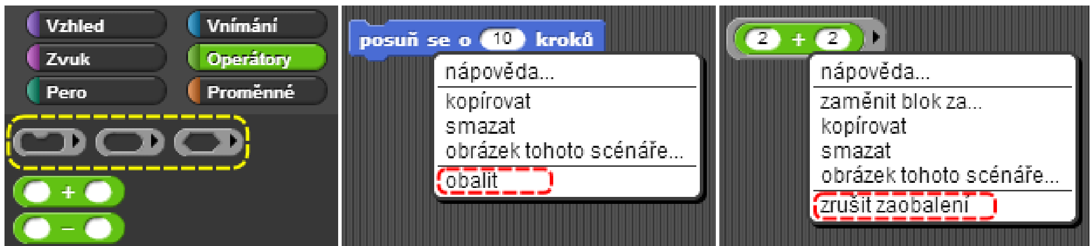
Obrázek 4.3.3 Vykonání zaobalených bloků bloky [spusť] a (zavolej)

K funkcionalitě zaobalení ještě zmiňme, že můžeme zaobalit všechny typy bloků kromě bloků událostí. Dále je možné zaobalit bloky v libovolném stavu – s vyplněnými argumenty (číslem, textem, proměnnou, či jiným blokem) i s nevyplněnými. To ostatně

ukazuje následující obrázek 4.3.4. Těm parametrům, kterým nebyl v zaobalené proceduře zvolen argument, je prostředím automaticky dodán v podobě nuly. Konečně doplňme, že do lze vždy „vhodit“ a naopak. Například funkce (zavolej) očekává , ale protože i podmínky jsou určitým druhem funkcí a je třeba je zavolat, můžeme zaobalenou podmínku v do (zavolej) bez problému vložit.

Obrázek 4.3.4 Ukázky zaobalených funkcí v vyvolaných (zavolej)

Tímto známe základní funkcionalitu zaobalování. Ještě si ukažme, jak bloky zaobalovat. Buď můžeme zaobalující funkce vzít z kategorie operátory a manuálně do nich vpravit bloky, nebo lze přes nabídku vyvolanou kliknutím pravého tlačítka myši na blok či část scénáře zvolit možnost „obalit“ (popř. opačnou „zrušit zaobalení“), čímž požádáme prostředí, aby nám bloky samo zaobalilo. To přitom zvolí vhodnou zaobalující funkci dle typu zaobalovaného bloku či skupiny bloků.



Obrázek 4.3.5 Umístění zaobalujících funkcí a zaobalení přes rozhraní

4.3.1 Zaobalená procedura namísto vlastního bloku

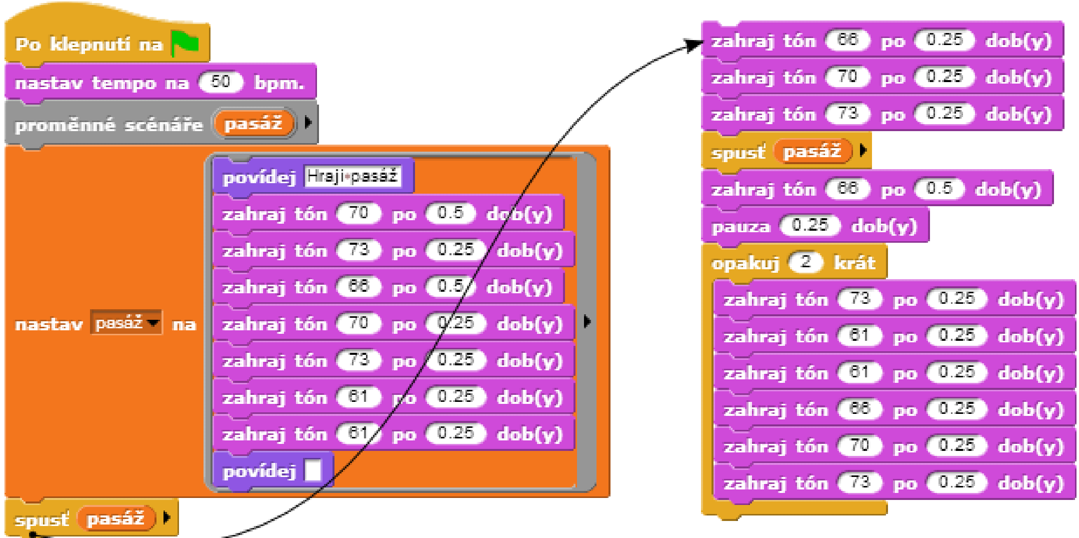
Hudbu lze programovat v obou prostředích a občas se při skládání písní můžeme dostat k problému opakujících se pasáží. Klasickým příkladem budiž refrén, který se vyskytuje za každou slokou písně. Nejhorší způsob jak takový problém s opakující se pasáží vyřešit je kopírovat naprogramované pasáže na různá místa ve scénáři. Jeden se pak těžko vyzná, kde každá pasáž ve scénáři začíná a kde končí. Navíc jakákoliv úprava zkopírovaných pasáží do různých částí scénáře je rizikem k zanesení případných chyb.

Ve Scratch 2 můžeme tento problém vyřešit vlastním blokem [pasáž], do kterého si pasáž vložíme a na než se v určitých částech písně odkážeme. Obrázek 4.3.6 ukazuje část scénáře z dalšího projektu hrající lidovou píseň „*Jede, jede poštovský panáček*“.

Obrázek 4.3.6 Ukázka písně s oddělenou pasáží do [pasáž] v Scratch 2

SCRATCH 2 PROJEKT: Píseň – „Jede, jede poštovský panáček" #12

Na první pohled jde o efektivní řešení. Než se ovšem vyvedeme z omylu, podívejme se na řešení se zaobalenou procedurou v prostředí Snap. Ačkoliv bychom v něm mohli také vytvořit vlastní příkaz [pasáž], nemusíme zbytečně plnit paletu bloků z kategorie zvuky (u Scratch 2 jsme zaplnili blokem [pasáž] kategorii bloky) a můžeme opakované příkazy pasáže zaobalit do . Jelikož zaobalující funkce vrací referenci na sebe sama a svůj obsah nevykoná, uchováme si tuto referenci do pomocné proměnné scénáře (pasáž) – jak ostatně ukazuje obrázek 4.3.7. Obrázek byl kvůli své délce rozdělen na dvě části spojené černou šipkou.



Obrázek 4.3.7 Ukázka písně s zaobalenou pasáží do v Snap

Prozraďme si jednu klíčovou informaci. Jakmile zaobalíme nějaký blok či skupinu bloků, vznikne tak bezejmenný vlastní blok bez zvoleného nápisu a kategorie. Dosud jsme toto popisovali jako zaobalenou proceduru. Tudíž zaobalená pasáž je ve své podstatě to samé co [pasáž] či [pasáž]. K vykonání zaobalené procedury použijeme příkaz [spusť], protože na rozdíl od vlastních bloků nemají název, na který bychom se mohli odkázat. Proto (pasáž) vykonáme příkazem [spusť], jenž jest dvakrát na předchozím obrázku.

SNAP PROJEKT: Píseň – „Jede, jede poštovský panáček" #13

Použitím zaobalené pasáže ve Snap jsme tedy ušetřili jeden blok, který již nemusíme vytvářet jako vlastní. Nicméně to může vyvolat dojem, že nejde o zas tak velkou výhodu. Zvažme ovšem jinou píseň, která obsahuje několik opakujících se pasáží. Lidová písnička „*Já jsem muzikant*" má takové opakující se pasáže rovnou tři.

Původním řešením ve Scratch bychom museli vytvořit tři pomocné bloky pro opakující se pasáže. Navíc bude-li píseň samotná ve vlastním bloku, pak budeme potřebovat dohromady příkazy čtyři. Do předchozího projektu z Scratch 2 byla tato píseň přidána se svými bloky. Podíváme-li se nyní do kategorie bloky, stěží se v ní vyznáme.

SCRATCH 2 PROJEKT: Písně – „Poštovský panáček" a „Já jsem muzikant" #14

Pokud přejdeme do prostředí Snap a použijeme zaobalování pro každou pasáž této písně, pak si vystačíme pouze s jedním blokem, ve kterém bude celá píseň naprogramována. Následující projekt, jež je ekvivalentní k projektu předešlému z Scratch 2, ve vlastním příkazu [píseň: Já jsem muzikant] používá proměnné scénáře (první_pasáž), (druhá_pasáž) a (třetí_pasáž). Do těchto si uchováváme zaobalené opakující se pasáže, které příkazem [spusť] ve vhodných částech písně vykonáme. Vystačíme si tak s jedním namísto čtyřech bloků oproti Scratch 2.

SNAP PROJEKT: Písně – „Poštovský panáček" a „Já jsem muzikant" #15

Jelikož vlastním blokům přidáváme parametry a zaobalující funkce jsou ve své podstatě vlastními bloky bez názvu a kategorie, můžeme i zaobalujícím funkcím přidat tyto parametry. Proto mají na svém konci černé šipky (např.). Jakmile klikneme na černou šipku, zaobalující funkce získá nápis „*názvy parametrů*“ a přidá automaticky parametr pojmenovaný `(#1)`. Analogicky další parametry budou číslovány od dvojky. Kliknutím na parametr jej můžeme přejmenovat.

V čem se však stanovení parametrů u vlastních bloků od zaobalujících funkcí liší je absence volby typu parametru. Zatímco vlastní blok může vyžadovat například číselný, textový, pravdivostní, či další z celkem dvanácti typů; zaobalující funkce typ parametru vůbec neurčuje. To samé platí pro varianty parametrů.

Argumenty parametrům zaobalujících funkcí předáváme v blocích `[spusť]`, `(zavolej)`, či `[zahaj]`, které mají také černé šipky. Při stanovení alespoň jednoho argumentu dostanou přídavný nápis „*s argumenty*“. Tento speciální vícenásobný parametr je vždy typu *libovolný*. To proto, že zmíněné bloky nemohou tušit, jaké typy parametrů bychom u zaobalujících funkcí potřebovali – nabízí proto možnost zadat *libovolné* argumenty. Vše o parametrech zaobalujících funkcí a volení argumentů je na obr. 4.3.8.

Máme-li potíže s rozdílem mezi „zaobalující funkce“ a „zaobalená procedura“, pak zřejmě pomůže následující vysvětlení. Zaobalující funkci (třeba) lze vnímat jako hlavičku vlastního bloku (přestože nemá nápisy ani kategorii, může stanovit parametry), kdežto do zaobalující funkce vložené bloky či skupinu bloků můžeme vnímat jako tělíčko vlastního bloku, tedy nějakou proceduru zaobalenou zaobalující funkcí.

Obrázek 4.3.8 Rozdíly mezi vlastním blokem a zaobalující funkcí

Procvičme si tento princip na následujícím projektu. Ve sklepě je myš, která prokousává pytel s moukou. Klikneme-li na žárovku v horní části scény, rozsvítí se na sekundu a půl. Myš se lekne a uteče. Až žárovka zhasne, myš se pomalu vrátí a bude v požírání mouky pokračovat. Rozsvícení a zhasnutí je definováno obyčejnou procedurou s posloupností příkazů `[oblékni kostým]`, `[rozešli všem]`, a `[hraj zvuk]`. Sice bychom si mohli tuto proceduru uchovat ve vlastním příkazu, ale potřebujeme si nyní procvičit definici parametrů a zvolení argumentů u zaobalených procedur. Proto na obrázku 4.3.9 je tento typ řešení uchovávající si zaobalenou proceduru v proměnné `(procedura)`, kterou vykonává příkazem `[spusť]` dvakrát po sobě s rozdílem jedné a půl sekundy. Zároveň volíme jiné argumenty, kterými jednou rozsvítíme a podruhé zhasneme ve sklepě.

Obrázek 4.3.9 Scénář rozsvěcující a zhasínající žárovku z projektu s ukázkou

SNAP PROJEKT: Hra - zkontroluj sklep #16

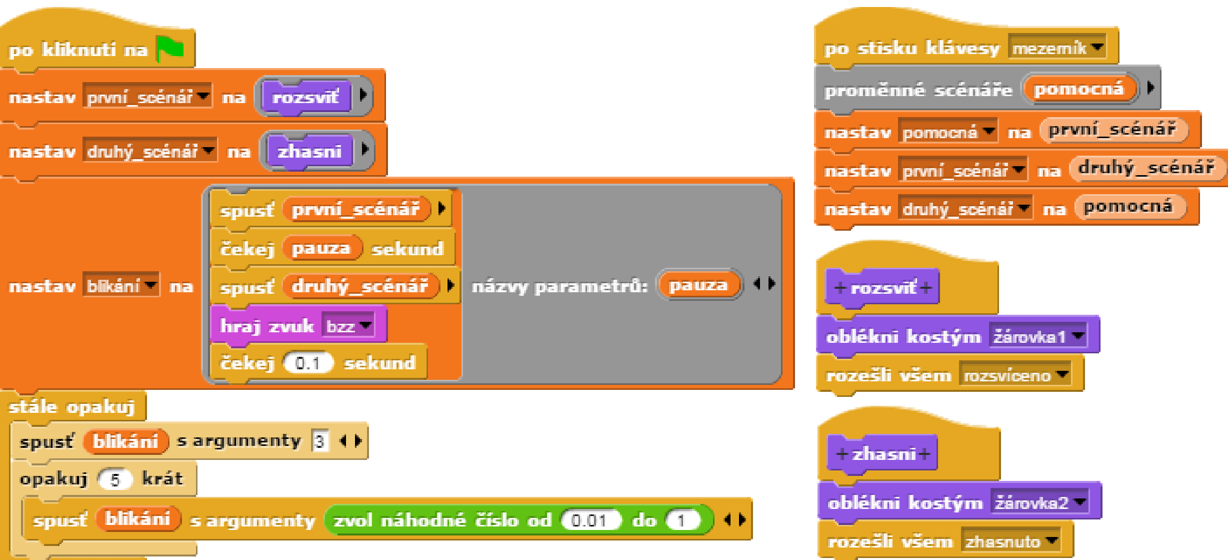
4.3.2 Zaobalená procedura použita ke změně chování

Protože je možné uchovat referenci na bezejmennou zaobalenou proceduru přes jednu ze tří zaobalujících funkcí do proměnné, můžeme naprogramovat scénář takovým způsobem, že se budeme vždy odkazovat bloky [spusť] či (zavolej) na obsah (zaobalenou proceduru) v nějaké proměnné. Ta bude za určitých okolností svou hodnotu měnit či střídat v jinou. Ačkoliv scénář zůstane stejný, změnou hodnoty proměnné na jinou zaobalenou proceduru vykonáme odlišnou část scénáře, která ve výsledku změní chování celého projektu. Pojďme si toto demonstrovat v příkladu uvedeném na obrázku 4.3.10.

Upravme předchozí projekt se žárovkou tak, že odstraníme myš a pytel s moukou. Žárovka však nepravidelně problikává. Buď svítí několik sekund a občas pohasne, nebo naopak několik sekund nesvítí a občas zablikne. To, zda je převážně zhasnutá či rozsvícená určujeme zaobalenými příkazy [zhasni] a [rozsviť]. Reference na ně si totiž uchováváme v proměnných (první_scénář) a (druhý_scénář), které si mezi těmito prohazujeme pomocným scénářem po stisknutí mezerníku.

V proměnné (blikání) je zaobalená procedura s jedním parametrem, která žárovku problikne na zadanou dobu určenou parametrem (pauza). Odkazuje se přitom na zaobalené procedury uchované v (první_scénář) a (druhý_scénář). Jejich prohozením stiskem mezerníku ovlivníme výsledek procedury (blikání).

Konečně pod příkazy [nastav] je hlavní scénář, starající se o neustálé problikávání žárovky. Ta vždy svítí či je zhasnutá na tři sekundy (příkazem [spusť] s argumentem {3} předávající se (blikání) do parametru (pauza)) a poté pětkrát v náhodných intervalech opět problikne (stejným způsobem přes [spusť]). Ačkoliv se tento hlavní scénář v [opakuj dokola] nemění, můžeme ovlivnit výsledek jeho provádění záměnou zaobalených procedur v proměnných, na jejichž hodnoty se odkazuje.



Obrázek 4.3.10 Scénáře nutné k pochopení upraveného projektu se žárovkou

SNAP PROJEKT: Hra - problikávací žárovka #17

V souvislosti se změnou chování si ukážeme ještě jeden složitější projekt (viz obrázek 4.3.11), v němž hrajeme za modrou rybu, která prochází evolucí. Ve vodním světě je ohrožena nepřátelskou oranžovou rybou do té doby, než vyroste natolik, aby ji oranžová ryba nemohla ublížit. Zelená ryba je neškodná, takže se nemusíme bát jí dotknout. Poslední postavou v projektu je korýš, který slouží jako počáteční potrava.

Abychom se dostali na další evoluční úroveň, musíme vždy sežrat tři živočichy. Které z nich to mají být, určuje úroveň hry. V první úrovni musíme sežrat třikrát koryše, v druhé třikrát neškodnou rybu, a v poslední neškodnou či nepřátelskou rybu. Od druhé úrovně nejsme nepřátelskou rybou ohroženi, avšak koryš nás už nenakrmí. Dotkneme-li se nepřátelské ryby v první úrovni, sežere nás a hru prohrajeme. Dostaneme-li se však na vrchol potravinového řetězce, hra bude úspěšně ukončena.

Obrázek 4.3.11 Ukázka ukládání zaobalených procedur v seznamu

Projekt si ukazujeme proto, že se některé části hlavního scénáře mění na základě úrovně hry, jejíž podmínky jsou definovány v obyčejném seznamu. Co úroveň, to seznam se zaobalenou podmínkou a v druhém prvku zaobaleným příkazem. Ty určují, které ryby naše modrá ryba dle úrovně sežere a jak na tuto skutečnost zareagujeme. Hlavně si ale ukazujeme poprvé funkci `(zavolej)` v projektu.

`SNAP PROJEKT: Hra - evoluce ryb` #18

4.3.3 Rekurze u zaobalených procedur a skrývání některých parametrů koncové rekurze

Při programování rekurzivních bloků se odkazujeme v rekurzivním volání na týž blok. Jednodušeji – k sestavení rekurze musíme mít k dispozici nějaký blok, v němž je opakující se část scénáře definována. Bezejmenné procedury stvořené zaobalujícími funkcemi , , a ; ale žádný název nemají a tak se samy na sebe nemohou v oblasti obsahující rekurzivní volání odkázat.

Víme však, že referenci na zaobalené procedury můžeme uchovávat v proměnných. Proto abychom mohli programovat rekurzi v těchto procedurách, musíme se odkázat na název proměnné, ve které se reference na proceduru uložila. Vzorové řešení programování rekurze zaobalených procedur je na obrázku 4.3.12. Obsahuje několik pomocných bloků k vytvoření rekurzivního scénáře. Zaobalující funkci vložíme do proměnné `(rekurzivní_procedura)`, na niž se uvnitř odkážeme v příkazu `[spusť]`, čímž zajistíme rekurzivní volání sebe sama. Za příkazem `[nastav]` pak stačí provést zaobalenou proceduru příkazem `[spusť]` k započatí rekurze.

Obrázek 4.3.12 Vzor k naprogramování rekurzivní zaobalené procedury

- **Skrývání některých parametrů koncové rekurze**

Rekurzivně naprogramované vlastní bloky občas potřebují pomocné parametry k uchování mezivýsledků. Zvenku bloků však tyto parametry obtěžují jejich uživatele, neboť musejí vložit potřebné argumenty k jejich úspěšnému vyhodnocení. I to lze odstranit.

Zapátrejme v paměti na dřívější funkci (pozpátku) z obrázku 4.2.9 v podsekci 4.2.2 o přidaných funkcích pro práci se seznamy. Ta potřebovala dodat vždy za druhý parametr prázdný seznam, do něhož vkládala prvky seznamu původního v otočeném pořadí. Uživatel této funkce ale nemusí tušit, jaký seznam má do druhého parametru vložit – a zdali vůbec. Proto by bylo lepší si tento druhý parametr skrýt, neboť umíme předpovědět, jaké hodnoty musí nabývat (vždy reference na prázdný seznam). Výsledné řešení je ukázáno na nadcházejícím obrázku 4.3.13, které si vzápětí vysvětlíme.

Obrázek 4.3.13 Upravená funkce (pozpátku) zakrývající druhý parametr

Scénář původní funkce (pozpátku) byl zaobalen do a uchován do pomocné proměnné scénáře (funkce). V důsledku toho se muselo upravit i rekurzivní volání, kde přes funkci (zavolej) v příkazu [vrať] voláme onu zaobalenou proceduru uchovanou v (funkce). To vyžadovalo přidat do přes černé šipky dva parametry, jež byly pojmenovány totožně s parametry původní nedokonalé verze (pozpátku).

Nakonec pod příkazem [nastav] musíme ještě zaobalenou proceduru spustit, protože by se sama nevykonala. Voláme proto v příkazu [vrať] přes (zavolej) její obsah s dodaným seznamem (seznam) a potřebným prázdným seznamem z (nový seznam).

Jelikož už netřeba druhého parametru u funkce (pozpátku), byl proto odstraněn z její hlavičky. Ta tak má nyní jen jeden parametr vyžadující seznam, jehož prvky vrátí naše funkce v opačném pořadí v odlišném seznamu.

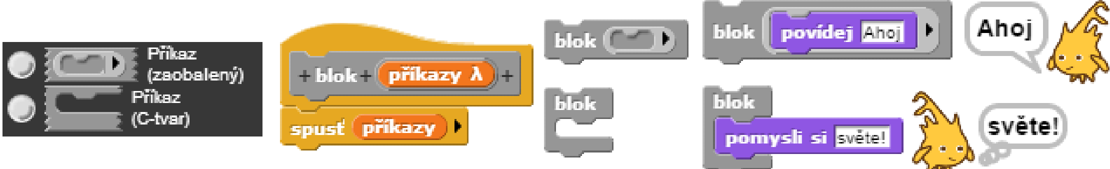
4.4 Ostatní dosud neužité typy a varianty parametrů

Ačkoliv jsme v celé bakalářské práci několikrát zmínili všechny typy parametrů vlastních bloků, většinu z nich jsme si ještě neměli možnost ukázat v příkladech. To napravíme v této sekci, která obsahuje přehršel různých ukázek, příkladů a projektů.

4.4.1 Parametry vyžadující bloky příkazů

Začněme těmi typy, jež vyžadují od svých uživatelů dodání libovolných příkazů, a které jsou zobrazeny na následujícím obrázku 4.4.1 ve vzorové ukázce.

První typ *příkaz (zaobalený)* „zvenku" bloku zobrazuje sám sebe v podobě „vystrčené" zaobalující funkce . Jelikož máme přístup k této zaobalující funkci, můžeme též definovat přes černé šipky oné proceduře nějaké parametry. Rekurzi u typu ale nenaprogramujeme, neboť nevidíme na formální parametr uchovávající si referenci na tuto zaobalenou proceduru. Dodané příkazy vyhodnotíme ve vhodnou chvíli uvnitř bloku přes [spusť], nebo je předáme jinému bloku, anebo vrátíme příkazem [vrať].



Obrázek 4.4.1 Vzhled a práce s typy parametrů vyžadující příkazy

Druhý typ *C-tvar* též vyžaduje libovolné příkazy, ale nezobrazuje svou podobu v . Jeho vzhled je totožný k vzhledu bloků řídících struktur, jakými jsou například [když-jinak], [opakuj dokud nenastane] a další – tedy ty s „výřezem" na příkazy. Blok s jedním tímto typem parametru pak připomíná tvar písmena *C* (viz obr. 4.4.1).

C-tvar se od *příkaz (zaobalený)*() liší pouze v zobrazení zvenku bloku – jinak jde o principiálně stejný typ parametru. Odstiňuje akorát od uživatele teorii zaobalování, neboť zaobalující funkci očekávající příkazy nezobrazuje. U tohoto typu nemůžeme určit parametry zaobalené procedury – nevidíme na černé šipky zaobal. funkce.

Mezi těmito je ještě drobný rozdíl. Jestliže je nastavíme jako *vícenásobné*, pak argumenty typu jsou přidávány za sebe – tedy , ... *C-tvar* však ve své vícenásobné variantě každý další argument zobrazuje přímo pod ten předchozí.

• Parametry typu – „příkaz (zaobalený)"

Kreslení n-úhelníku není žádná věda. Jenže někdy bychom rádi kreslenému n-úhelníku přidali nějaký efekt. Například aby měl proměnlivou barvu či tloušťku čáry, aby se vykreslil s přerušovanou čárou a podobně. To umí následující projekt v prostředí Scratch 2, který vykresluje nějaký n-úhelník příkazem [nakresli n-úhelník].

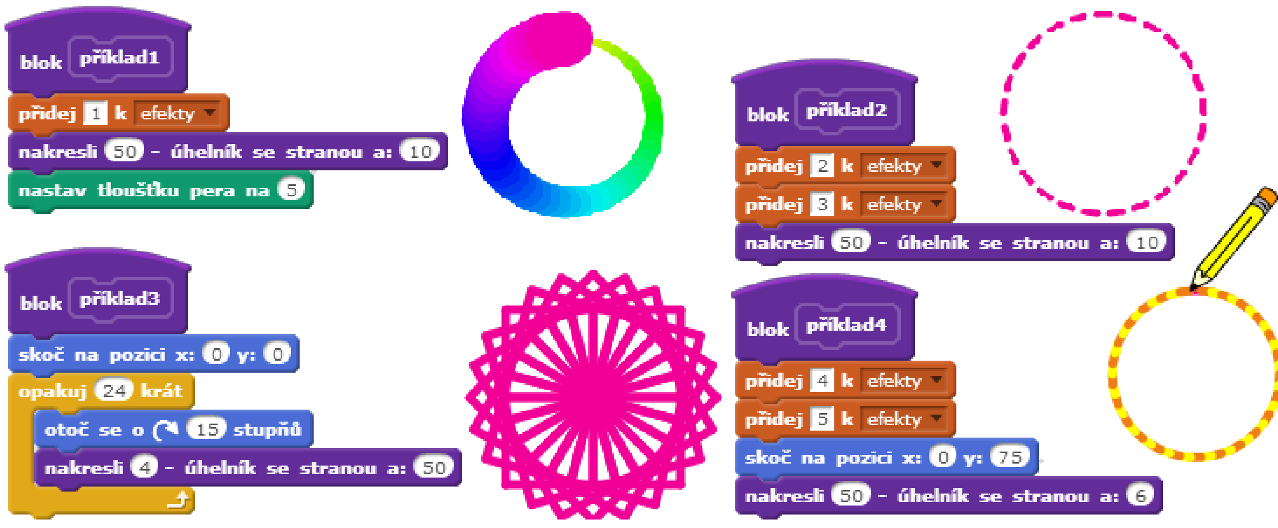
Různé efekty lze aplikovat ve chvíli, kdy při kreslení navštívíme další vrchol kresleného n-úhelníku. Abychom mohli efekty obměňovat, bylo třeba si vytvořit pomocný příkaz [proveď efekt] s jedním parametrem číselného typu, kterým zvolíme konkrétní efekt. Tento příkaz – kromě toho, že je součástí [nakresli n-úhelník] – větvením [když] rozlišuje, jaký efekt chceme v tu či onu chvíli použít. Efekt je vždy proveden změnou stavu postavy, aby došlo k ovlivnění jejího následného způsobu kreslení. Měníme barvu či tloušťku jejího pera, přikazujeme ji otisknout se, atd.

Někdy ale potřebujeme efekty střídat s každým navštíveným vrcholem n-úhelníku, neboť to nám umožní vykreslit ještě hezčí obrazce. Například se může jednat o střídání příkazů [pero nahoru] a [pero dolů], což by kupříkladu kruh vykreslilo s čárkovanou čarou. Proto si v projektu uchováváme efekty do seznamu v číselných hodnotách.

Další obrázek 4.4.2 ukazuje definici příkazů [nakresli n-úhelník] a [proveď efekt] v prostředí Scratch 2. Zejména [proveď efekt] je vhodné si prostudovat.

Obrázek 4.4.2 Definice [nakresli n-úhelník] a [proveď efekt]

Následující obrázek 4.4.3 pak ukazuje čtyři příklady, které jsou v projektu ze Scratch 2 vykresleny. Příklady do prázdného seznamu (efekty) přidávají čísla konkrétního efektu. Některé aplikují efekty dva, jiný žádný. Při kreslení n-úhelníku blokem [nakresli n-úhelník] pak postupně procházíme s každým dalším navštíveným vrcholem kresleného n-úhelníku následující prvek seznamu (efekt), čímž ovlivníme způsob jeho vykreslení.



Obrázek 4.4.3 Příklady vykreslující čtyři obrazce přes [nakresli n-úhelník] ▮

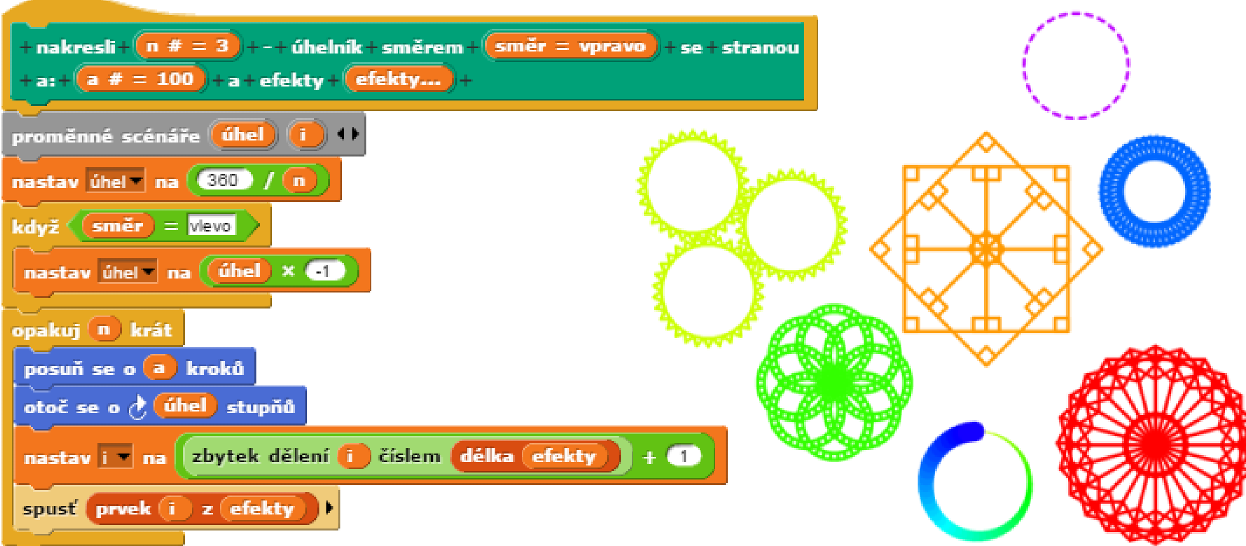
SCRATCH 2 PROJEKT: Kreslení n-úhelníků #19

Jenže řešení v Scratch 2 naráží na určité limity. Kdybychom chtěli vykreslit modrožlutý n-úhelník, zelenočervený, černofialový, anebo jakýkoliv jiný, museli bychom umístit do [proveď efekt] za efekty čtyři a pět z obrázku 4.4.2 další řídící struktury [když] obsahující [nastav barvu pera], jež by se však lišily pouze zvolenou barvou. Takovýchto problému se najde celá řada. Ideální by bylo, kdybychom prostě bloku

[nakresli n-úhelník] mohli dodat konkrétní efekty rovnou a nepracovat s žádným seznamem ani pomocným příkazem [proveď efekt].

V prostředí Snap můžeme problém vyřešit naprosto elegantně. Příkaz [nakresli n-úhelník] je uveden na obrázku 4.4.4. Oproti verzi z Scratch 2 obsahuje parametr (směr) s možnostmi rozbalovací nabídky {vpravo/vlevo} pro určení směru kreslení (využijeme později). Hlavně však disponuje parametrem (efekty), jenž je typu *příkaz (zaobalený)*, který se zvenku bloku znázorňuje jako ⊂⊃▸. Do něj můžeme vložit příkaz, který zastane efekt, a bude spuštěn příkazem [spusť] v okamžiku příchodu do dalšího vrcholu kresleného n-úhelníku. Navíc je (efekty) vícenásobné varianty, takže můžeme přidat více těchto zaobalujících funkcí černými šipkami, do nichž vložíme různé příkazy, které se budou postupně střídat – jako u seznamu efektů předchozí verze Scratch 2.

V podstatě můžeme ovlivňovat vykonávání [nakresli n-úhelník] dodáním různých příkazů a to v různém počtu. Smysl typu parametru *příkaz (zaobalený)* v kontextu tohoto bloku lze vyjádřit slovy: „*Kreslení n-úhelníku obstarám sám, jen si řekni jak to chceš (jaké efekty) a zbytek nech na mne.*“.



Obrázek 4.4.4 Definice [nakresli n-úhelník] s ukázkami v Snap

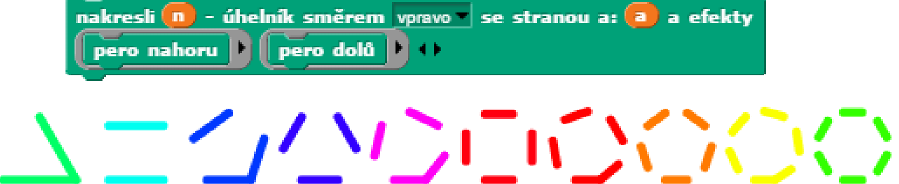
Předchozí obrázek kromě definice našeho [nakresli n-úhelník] ukazuje i různé obrazce tímto blokem vykreslené. Stačilo pouze dodat ve správném pořadí správné příkazy do ⊂⊃▸ parametru (efekty). Než navštívíme projekt, uveďme několik příkladů, na kterých bude zřejmý rozdíl mezi [nakresli n-úhelník] a [nakresli n-úhelník].

Následující příklad 4.4.5 vykreslil devět n-úhelníků počínaje trojúhelníkem a konče dvanáctiúhelníkem. Do ⊂⊃▸ jsme dodali vlastní příkaz [udělej značku], který dá postavě pero dolu, posune ji o minimální krok (čímž udělá značku na scéně) a nakonec ji zvedne pero. Jelikož se efekty aplikují při návštěvě každého vrcholu n-úhelníku, vidíme na obrázku tečky namísto dlouhých čar.

Obrázek 4.4.5 Kreslení n-úhelníků ovlivněné dodáním jednoho příkazu

Nyní zkusme přidat další argument vícenásobnému parametru (efekty). Obrázek 4.4.6 vykresluje n-úhelníky přerušovanými čarami. Postava doputováním do dalšího vrcholu změní polohu svého pera. Jednou jej zvedne a podruhé položí, takže dojde ke střídavému

kreslení. Výsledek je lépe vidět spíše u kruhů, které si takto můžeme udělat čárkované. Barevné rozdíly však s efekty nemají nic společného. Scénář, jímž byly vykresleny, měl za `[nakresli n-úhelník]` příkaz `[změň barvu pera]`, aby je od sebe barevně oddělil.



Obrázek 4.4.6 Kreslení n-úhelníků se střídajícími se dvěma příkazy

Tímto lze při kreslení n-úhelníků v Snap docílit nesběrného množství efektů. Otázkou zůstává, jak by to bylo v prostředích, kdybychom chtěli při kreslení nějakého n-úhelníku kreslit jiný či ještě další n-úhelníky – tj. kreslit n-úhelníky rekurzivně.

V prostředí Scratch 2 bychom museli do příkazu `[proveď efekt]` přidat další řídící struktury `[když]`, v nichž bychom definovali „vnořený“ `[nakresli n-úhelník]`. Jenže bychom nemohli upravovat jednoduše parametry této ve své podstatě rekurzivně volané proceduře. Projekt by musel být zásadně rozšířen a byl by mnohem méně přehledný.

Zato v prostředí Snap lze vložit do za argument opět `[nakresli n-úhelník]`, jímž vykreslíme další n-úhelníky při každé návštěvě jednoho z vrcholů. Můžeme takto rozvětvovat kreslení do propracovanějších a krásnějších obrazců. Obrázek 4.4.7 obsahuje příklad rekurzivního kreslení. V každém navštíveném vrcholu se vykreslí směrem vlevo malý trojúhelník. Vzniknou tak různé hvězdice relativně jednoduchým způsobem.

Obrázek 4.4.7 Kreslení n-úhelníků rekurzivně vnořením stejného příkazu

Vespod předchozího obrázku ještě vidíme tak trochu odlišné n-úhelníky. Takového výsledku dosáhneme, když ve vnořeném příkazu `[nakresli n-úhelník]` zvolíme místo `{vlevo}` možnost `{vpravo}`. Trojúhelníčky se totiž začnou kreslit namísto ven směrem dovnitř. Vnořený `[nakresli n-úhelník]` avšak musí mít jeden argument , jinak by prostředí zahlásilo chybu. Jedná se o problém, který musí vývojáři opravit.

Konečně se podívejme do následujícího projektu, v němž je pro čtenáře připravena sada obrazců. Ty projekt umí vykreslit. Má-li uživatel zájem, může se poučit z těchto.

`SNAP PROJEKT: Kreslení n-úhelníků` #20

• **Parametr typu „C-tvar“**

U parametrů vyžadující zaobalené příkazy volíme typ *C-tvar* jedině tehdy, když očekáváme více na sebe napojených příkazů (delší část scénáře). Pakliže bychom ponechali parametru typ *příkaz (zaobalený)* ( ), po vložení většího úseku příkazů by se výška bloku s parametrem tohoto typu enormně natáhla – a to nemluvě o přehlednosti. Právě *C-tvar* se hodí tam, kam vložíme delší úsek scénáře. Tento typ automaticky zaobaluje příkazy do a odstiňuje tuto skutečnost od uživatele bloku.

Na následujícím obrázku 4.4.8 je definována funkce (proveď měření), která provede dodané (zaobalené) příkazy n-krát, přičemž při každém provedení příkazů změří čas potřebný k jejich vykonání. Nakonec časy zprůměruje a vrátí výsledek. S větším počtem opakovaných měření se zvyšuje přesnost výsledné zprůměrované hodnoty představující potřebný čas k vykonání zadaných příkazů.

Číselný parametr (n) představuje počet měření. Parametr (příkazy) typu *C-tvar* dodané příkazy automaticky zaobalí, čímž je budeme moci vykonat ve vhodnou chvíli. Tou je okamžik těsně po uchování si času před měřením do proměnné (startovní_čas) funkcí (aktuální) s možností {čas v milisekundách}. Jakmile se provedou dodané příkazy blokem [spusť], uchováme znovu aktuální čas do proměnné (konečný_čas). Do proměnné (mezivýsledek) přičítáme potřebný čas každého měření v sekundách, kterou nakonec použijeme k výpočtu aritmetického průměru všech sečtených časů.

Obrázek 4.4.8 Definice a ukázky způsobu užití funkce (proveď měření)

Funkci využijeme u těch částí scénářů, které se provádějí pomaleji než očekáváme – tedy chceme-li dbát na efektivitu celého projektu při jeho vývoji. Obrázek 4.4.9 obsahuje ukázkový příklad, ve kterém desetkrát měříme potřebný čas k naplnění seznamu tisíci prvky s hodnotou {věc} (na hodnotě však nezáleží). Funkce provedla dodané příkazy n-krát a zjistila, že naplnění seznamu tisíci prvky trvá přibližně šest sekund. Na pravé straně obrázku je scénář vylepšen vsazením do příkazu [bez obnovy obrazovky], jehož vyhodnocení funkce spočítala na méně než desetinu sekundy. Rozdíl je markantní. Dokážeme si tak udělat představu jak o výhodě příkazu [bez obnovy obrazovky] tak i o využitelnosti funkce (proveď měření), bez které bychom rozdíly nevypočítali.

Obrázek 4.4.9 Měření náročnosti plnění seznamů funkcí (proveď měření)

4.4.2 Parametry vyžadující bloky funkcí či podmínek

Parametry vlastních bloků mohou krom bloků příkazů žádat i bloky funkcí a podmínek. Zde paralela s parametrem typu je více než zřejmá. Stačí zvolit parametrům jiný typ v řádce parametru editoru a to buď na *funkce* () či *podmínka* ().

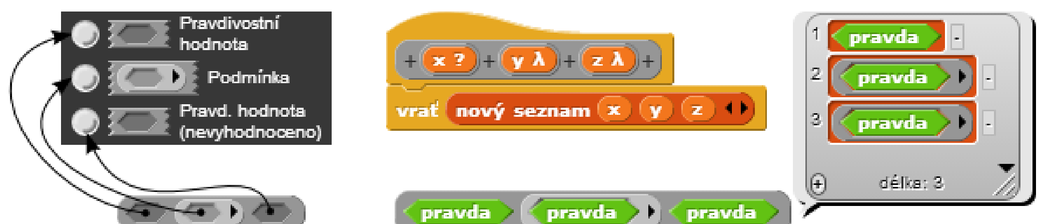
Rozdíly mezi *libovolný*, *funkce*, *libovolný (nevyhodnoceno)* a také mezi *pravdivostní hodnota*, *podmínka*, *pravd. hodnota (nevyhodnoceno)*; nemusí být zřejmé. Rozlišíme je.

První obrázek 4.4.10 ukazuje tři podobné typy stejného sloupce editoru parametru. Pro potřeby demonstrace byla vytvořena funkce se třemi parametry `(x)`, `(y)`, a `(z)`; jež hodnoty parametrů vrátí v seznamu. Parametr `(x)` je typu *libovolný*, `(y)` typu *funkce*, a analogicky `(z)` je *libovolný (nevyhodnoceno)*. Všimněme si vespod nalevo, že *libovolný* a *libovolný (nevyhod.)* jsou identického vzhledu, zatímco *funkce* vypadá „zvenku" bloku jako . Předáme-li do všech parametrů funkci `(sečti)` s argumenty `{5, 5}`, pak:

- *libovolný* typ `(sečti)` vyhodnotí a její výsledek `{10}` předá parametru `(x)`
- funkci `(sečti)` jsme vložili do (*funkce*), a tak se uchová do `(y)` a nevyhodnotí
- *libovolný (nevyhodnoceno)* „potají" automaticky zaobalí `(sečti)` do do `(z)`

Obrázek 4.4.10 Popis rozdílů tří podobně zaměřených typů parametrů

Na stejném principu fungují typy *pravdivostní hodnota*, *podmínka*, *pravd. hodnota (nevyhodnoceno)*, jak ostatně ukazuje následující obrázek 4.4.11. Vše výše popsané můžeme aplikovat i na parametry s typy očekávající bloky podmínek.



Obrázek 4.4.11 Popis rozdílů tří podobně zaměřených typů parametrů

• Parametr typu – „funkce (zaobalená)"

Když v prostředí Scratch 2 potřebujeme pozměnit hodnoty **všech** prvků seznamu, musíme si naprogramovat složitý scénář procházející celý seznam a nahrazující hodnoty prvků příkazem `[nahraď prvek]`. Máme-li seznam čísel, pak je občas žádané nad každým z nich provést matematickou operaci – jednou přičíst hodnotu k prvku, jindy jej vynásobit, apod. Pracujeme-li se seznamem písmen, pak zase můžeme chtít například posunout jednotlivá písmena v abecedě, nebo je změnit z velkých na malá a opačně, či je jakkoliv upravit. Zkrátka – případů, ve kterých potřebujeme ovlivnit prvky seznamu různými funkcemi je nespočetné množství. To vyžaduje neustálé opakování scénáře procházející prvky seznamu, ve kterém změníme vždy jen požadovanou funkci.

Mnoho programovacích jazyků řeší tento problém funkcí zvanou „map", která nad každým prvkem zvoleného seznamu zavolá dodanou funkci či složené funkce. V prostředí Snap ji lze taktéž vytvořit, což učiníme na obrázku 4.4.12. Nazvěme ji `(zavolej nad každým prvkem)`. Její parametr `(funkce)` je typu *funkce* (zaobalující), která námi dopravený blok funkce nevykoná a uchová na později.

Obrázek 4.4.12 Vlastní funkce volající dodanou funkci nad prvky seznamu

Tato rekurzivně naprogramovaná funkce funguje tak, že do nového seznamu postupně přidá všechny prvky přes (vlož do popředí). Každý aktuálně přidávaný prvek ještě před vložením do nového seznamu upraví aplikací dodané funkce uchované v (funkce). Výsledkem je vždy nový seznam a tak původní seznam není znehodnocen. Scénář jsme mimochodem vložili do [bez obnovy obrazovky], abychom významně urychlili vyhodnocení této funkce. Následující obrázek 4.4.13 ukazuje její užití ve třech příkladech.

Obrázek 4.4.13 Funkcionalita funkce (zavolej nad každým prvkem)

Zatímco v Scratch 2 potřebujeme ke stejnému výsledku tři dlouhé scénáře lišíc se jen dodanými funkcemi ((sečti), (vynásob), a vnořené (unicode), (unicode jako znak) a (sečti)), v prostředí Snap touto funkcí provedeme všechny úpravy pohodlně jedním blokem – výsledek funkce totiž můžeme uložit do proměnné příkazem [nastav].

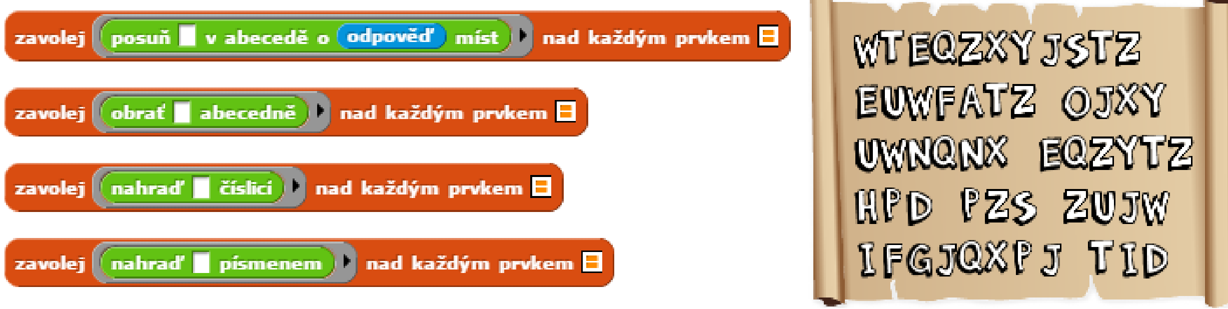
Vysvětleme si ještě třetí ukázku z obrázku 4.4.13, jelikož může být obtížnější pochopit jak funguje posun písmen v abecedě. Například prvek {A} vložíme do (unicode) vracející pořadové číslo onoho znaku v unikódové tabulce ({65}). K tomuto číslu funkcí (sečti) přičteme hodnotu posunu v abecedě. Nakonec funkcí (unicode jako znak) získáme znak odpovídající upravené hodnotě ({67}), kterou je znak {C}.

Kdykoliv budeme potřebovat aplikovat nějakou funkci nad všemi prvky seznamu, stačí si vyexportovat z projektu tento blok, naimportovat jej do jiného, a použít v příslušném scénáři. Stejně tak bylo provedeno i v následujícím projektu šifrující různými způsoby zprávu, který používá (zavolej nad každým prvkem) ve čtyřech postavách.

Při spuštění projektu je celá zpráva zašifrovaná, což návštěvník bez přečtení této části práce nezjistí. Než ale do projektu vstoupíme, bylo by vhodné prohlédnou následující obrázek 4.4.14 a přečíst si níže popsaný způsob, jakým byla zpráva zašifrována.

SNAP PROJEKT: Hra - rozlušti zašifrovanou zprávu #21

Z podsekce 4.2.3 o vícenásobné variantě parametru víme, že chceme-li provádět sofistikovanější úpravy s textem, musíme jej převést do prvků seznamu písmeno po písmeni – například funkcí (rozděl). Totéž bylo provedeno v projektu se zprávou, abychom mohli aplikovat různé šifry nad jednotlivými písmeny zprávy pomocí funkce (zavolej nad každým prvkem).



Obrázek 4.4.14 Způsob jak provést různé šifry nad zprávou a obrázek hry

Projekt také obsahuje různá tlačítka, kterými lze aplikovat šifry na zprávu. Po kliknutí na ně se přes `(zavolej nad každým prvkem)` konkrétní šifrovací funkce provede nad zprávou uchovanou v proměnné `(zpráva)`. Můžeme tak logicky vytvořit pomyslný řetěz šifer a tím provést lepší „kamufláž“ původní zprávy, jež bude velmi obtížné dešifrovat. Projekt má funkci `(NEZAŠIFROVANÁ ZPRÁVA)` vracející text původní neupravené zprávy.

Popišme si, jaké a jak fungují jednotlivé šifry dostupné v projektu:

- `(posuň v abecedě)` – posune písmeno v abecedě o zadaný počet („Ceasarova šifra“)
- `(obrať abecedně)` – např. `{A}` zamění za `{Z}`, `{Z => A}`, `{B => X}`, `{X => B}`
- `(nahraď číslicí)` – některá písmena zamění za číslici 0–9: `{A => 4}`, `{B => 5}`, `{E => 3}`, `{G => 6}`, `{I => 1}`, `{O => 0}`, `{P => 7}`, `{Q => 9}`, `{S => 8}`
- `(nahraď písmenem)` – opak `(nahraď číslicí)` zaměňující číslice 0–9 za písmena

Zprávu je ještě možné otočit funkcí `(pozpátku)`. Pokud máme zájem, lze si vytvořit vlastní zprávu k tomu určeným tlačítkem. Aby však projekt správně vykreslil jednotlivé znaky, je důležité, aby každé písmeno bylo velkým písmenem. Jelikož vykreslujeme text na scénu otisknutím jednotlivých kostýmů k tomu určenou postavou, někdy se slovo rozdělí ve špatné části. Tomu je možné předejít přidání mezery do původní zprávy.

Když nyní vstoupíme do projektu, původní zpráva bude zašifrována několika šiframi. Abychom ji správně rozluštili, musíme aplikovat šifry v úplně opačném pořadí a s opačnými hodnotami. Jak rozšifrovat zašifrovanou zprávu se dozvíme v poznámkách k projektu v prostředí, které nás přímo navedou krok za krokem.

I tento projekt bychom vytvořili v prostředí Scratch 2. Museli bychom však vytvořit čtyři scénáře procházející seznam a aplikující konkrétní šifru v každé postavě zvlášť.

• Parametr typu – „podmínka (zaobalená)“

Dalším typem parametru je vyžadující bloky podmínek. Dáváme tak uživatelům bloku s tímto typem najevo, že mohou za určitých *podmínek* pozměnit výsledek bloku.

Máme-li seznam nějakých hodnot, potřebujeme je mnohdy seřadit. Způsob řazení je však potřeba přizpůsobit kontextu a přestože převážně používáme relační operátory `<je menší>` a `<je větší>` k porovnání jednotlivých prvků, občas musíme podmínky porovnání definovat konkrétněji.

Vzpomeňme na popisovaný problém v předchozí části ohledně nutnosti opakovat scénář procházející prvky seznamu, u něž změníme pouze funkci aplikovanou na každý prvek. To samé platí i v tomto případě. Při různém způsobu porovnání bychom museli celý řadící scénář zkopírovat, abychom měnili podmínky určující kritéria řazení. Řešení je opět možné dosáhnout v Snap dodáním bloků podmínek do parametru typu .

Proto byla naprogramována funkce `(seřaď prvky dle)`, jež dle podmínky v seřadí prvky seznamu. Její „bublinkový“ řadící algoritmus si představovat nebudeme, neboť to je nad rámec této práce. Podívejme se jak vypadá a funguje na obrázku 4.4.15.

Obrázek 4.4.15 Funkce (seřaď prvky dle) bez definice v příkladech

Zatím jsme porovnávali čísla v seznamu, u nichž si vystačíme s obyčejným <je menší> a <je větší>. Pokud bychom chtěli seřadit třeba postavy, musíme zvolit konkrétní atribut či atributy, jež budou klíčovými při porovnání postav.

Na obrázku 4.4.16 je několik postav různorodé velikosti. Největší je drak, následně jednorožec, pes, o trochu menší kočka, a nejmenší je netopýr. Pokud je chceme porovnat podle jejich velikosti (atribut *velikost*), musíme podmínku v bloku (seřaď prvky dle) upravit. Přestože v současné verzi Snap 4.0 chybí funkce (atribut), lze použít alespoň funkci (z) se zvolenou možností {velikost}. Volit název postavy rozbalovací nabídkou nemusíme, protože je prostředí samo dodá – doplní chybějící argumenty zaobalené procedury o názvy porovnávaných postav. Seřazení postav je od největší po nejmenší.

Obrázek 4.4.16 Způsob řazení postav naší funkcí dle jejich velikosti

Nyní zkusme porovnat postavu dle více kritérií. Zvířatům z předchozího příkladu byla přidána vlastní proměnná (počet_životů). Kočka jich má devět – ostatní po jednom. Zaveďme takové řazení, kdy upřednostňujeme řazení dle počtu životů. Mají-li postavy stejný počet, musíme je porovnat dle dalšího kritéria, jímž je opět velikost. V případě, že mají počet životů rozdílný (což upřednostňujeme), pak postavy seřadíme podle tohoto atributu. Obrázek 4.4.17 napoví. Řazení „katapultovalo“ kočku do popředí, protože má nejvíce životů, což je naše primární řadící kritérium. Zbytek byl řazen velikostně.

Obrázek 4.4.17 Rozšířený způsob řazení dle počtu životů i velikosti

Příklad ukazujeme, jelikož přidáváme zaobalené proceduře parametry (#1) a (#2). Odkazujeme se totiž na oba porovnávané prvky (postavy) na více místech odlišným způsobem. Neučiníme-li tak, prostředí doplní do argumentů názvy ve špatném pořadí.

4.4.3 Parametry typu „objekt“ a „pravdivostní hodnota“

Přestože se v této kapitole vůbec nevěnujeme objektově orientovanému programování (tvorbě vlastních tříd či prototypů, klonování postav, a využívání k tomu určených primitivních bloků) z důvodu nedostatečně podpory v současné verzi prostředí Snap 4.0, můžeme si alespoň ukázat i v nynější verzi funkční příklady využívající parametr typu *objekt* u vlastních bloků v souvislosti s dálkovým ovládáním a komunikací postav.

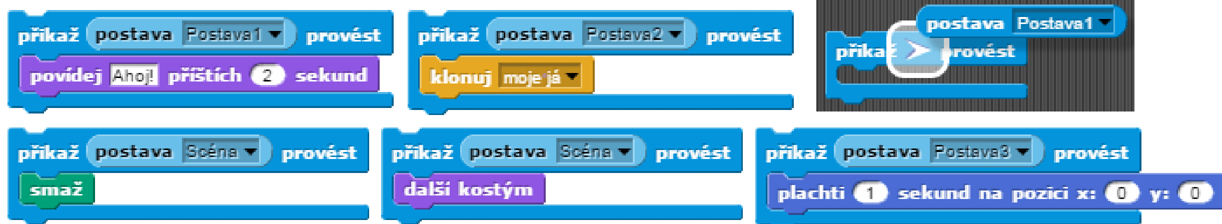
Na obrázku 3.1.15 jsme si ukázali jak postava může na dálku jinou postavu buďto ovládat anebo se ji na něco zeptat. Sestavení bloků k nepřímé komunikaci s postavou je ale jaksi nemotorné. Bude proto mnohem jednodušší, vytvoříme-li si pomocné bloky, jež všechnu práci provedou – [přikaž postavě provést] a (zeptej se postavy).

Obrázek 4.4.18 ukazuje vlastní blok [přikaž postavě provést], jehož první parametr (příkazy) je typu *objekt*. Tím značíme očekávání – když ne reference – alespoň názvu postavy v textové podobě. Druhý parametr (příkazy) je typu *C-tvar* automaticky zaobalující dodané příkazy. Tělíčko bloku obsahuje ono nepříliš intuitivní sestavení bloků přikazující nějaké postavě provést konkrétní příkazy. Přejděme k ukázkám.

Obrázek 4.4.18 [přikaž postavě provést] k nepřímému přikazování

Proberme postupně všechny ukázky z následujícího obrázku 4.4.19. Ještě předtím však zmiňme, že z důvodu absence funkce (postava) v současné verzi 4.0 prostředí Snap autor této práce stejnojmennou funkci vytvořil a použil v následujících příkladech. Místo referencí na postavy vrací jejich název a možnosti v rozbalovací nabídce jsou napevno (nerozšiřují se např. po přidání postavy). Ve verzi 4.1 ji nahradíme tou primitivní.

První příklad přikáže postavě „Postava1“ povídat {Ahoj!} dvě sekundy. Příklad hned vedle se pokusí postavě „Postava2“ přikázat klonovat sebe sama přes {moje já}. Výsledkem je však klon postavy, ve které jsme [přikaž postavě provést] provedli. To není chyba našeho bloku – to tak v prostředí funguje u nepřímého ovládání postavy.



Obrázek 4.4.19 Příklady užití příkazu [přikaž postavě provést]

Přes [přikaž postavě provést] je možné ovládat i scénu. Ukázky vespod ji přikazují smazat otisky postav a čáry per příkazem [smaž] anebo změnit kostým – což v kontextu s dálkovým ovládáním funguje stejně jako [změň pozadí] u každé postavy v Scratch 2. Konečně ještě přikážeme postavě „Postava3“ posun přes [plachti na pozici].

Dost bylo přikazování. Vrhněme se na vlastní blok zjednodušující nám dotazování se postavy. Funkce (zeptej se postavy) je uvedena i s příklady na 4.4.20. První parametr (postava) je samozřejmě typu *objekt* a druhý (dotaz) typu *funkce (zaobalená)* (), do kterého vložíme dotazující se blok funkce. Chceme-li se zeptat postavy přes (zeptej se postavy) na nějakou podmínku, můžeme opět zaměnit za .

Z ukázek můžeme vypozorovat, že dotazování se na souřadnici x funkcí (souřadnice x) vyústilo v odlišné výsledky u postav „Postava1“ a „Postava2“, neboť jsou na odlišných pozicích. U scény se můžeme zeptat například na pořadí oblečeného kostýmu (jejího pozadí) funkcí (kostým číslo). Obrázek nakonec ukazuje optání se postav na dotek s okrajem scény přes <dotýká se> se zvolenou možností {okraje}.

Obrázek 4.4.20 Definice a příklady (zeptej se postavy) k dotazování se

Vzpomeňme však na možnosti primitivní funkce (postava) z prostředí BYOB, mezi nimiž se vyskytuje i možnost {všechny postavy} vracející seznam s referencemi na postavy projektu (vyjma scény). Naše bloky [přikaž postavě provést] a (zeptej se postavy) s referencí na seznam však pracovat neumí. Nicméně stačí parametr (postava) změnit z *jednoduché* na *vícenásobnou* variantu a trochu upravit jejich tělíčka.

Začněme upravenou (zeptej se postavy), jež jest na obrázku 4.4.21. Její definice je zbytečně rozsáhlá a z toho důvodu si ji ukazovat nebudeme. Jelikož se dotazujeme všech postav – získáme n odpovědí – pak musí funkce vracet seznam s výsledky, které jsou vázány na pořadí postav v seznamu získaném z (postava) s {všechny postavy}.

Obrázek 4.4.21 Vylepšení (zeptej se postavy) dotazující se více postav

Při úpravě [přikaž postavě provést] k podpoře komunikace se všemi postavami ale narazíme na menší zádrhel. Vykonáváme totiž dodané příkazy parametru (příkazy) blokem [spusť], což vede k synchronnímu přikazování. Kdybychom chtěli všemi postavami něco říci, povídali by postupně jedna po druhé. Abychom všem najednou přikázali něco provést, musíme namísto [spusť] použít příkaz [zahaj], který příkazy či část scénáře v ( ) vykoná v odděleném procesu. Více informací o [zahaj] si řekneme až v podsekci 4.5.2. Zatím se podívejme na obrázek 4.4.22 obsahující řešení problému.

Obrázek 4.4.22 Vylepšený [přikaž postavě provést] přikazující více postavám

Bylo by avšak hloupé zbavovat se možnosti volby, zdali chceme přikazovat všem postavám postupně či najednou. Proto byl přidán do upravené [přikaž postavě provést] třetí parametr (provést_najednou), jež je typem *pravdivostní hodnota*. Bude-li nabývat hodnoty <pravda>, pak použijeme při procházení postav v seznamu blok [zahaj]. V opačném případě – tj. chceme-li přikazovat postavám postupně – použijeme [spusť].

Popišme si příklady z předchozího obrázku 4.4.22. První všem postavám blokem [povídej příštích] určuje povědět {Ahoj!} na dvě sekundy. Aby pozdravily všechny postavy, musíme do parametru (provést_najednou) předat jako argument <pravda> či jinou podmínku s tímto výsledkem.

Druhý příklad přikazuje postavám donekonečna se otáčet s využitím cyklu [opakuj dokola]. Otáčela by se však jenom jedna postava, neboť by scénář „uvízl“ kvůli cyklu [opakuj dokola] a k přikázání otáčení dalším postavám by tak vůbec nedošlo. Řešení opět představuje asynchronní přikazování postavám a to dodáním <pravda> do (provést_najednou).

Poslední ukázka přikazuje postavám prozradit pozici na scéně. Zde si povšimněme, že se dotazujeme na atributy postavy funkcemi (souřadnice x) a (souřadnice y), ale vůbec nemusíme použít funkci (zeptej se postavy). Prostředí samo vykoná zmíněné funkce nad konkrétní (zvolenou) postavou. Tento příklad má dva menší chytáky. Zvažovat přikazování postupné či najednou nemá vůbec smysl, protože příkaz [povídej] není [povídej příštích] a provede se téměř okamžitě. Dále kdybychom chtěli přikázat postavám provést jiné příkazy a **postupně**, musíme do (provést_najednou) předat <nepravda>. V příkladu na obrázku jsme však do parametru za argument nic nevložili a je tudíž prázdný. Implicitní hodnota pravdivostních datových typů je totiž <nepravda> a tak jej nemusíme vyplňovat. Je však zadobře zamyslet se nad tím, jak si později čtení takového bloku vyložíme. Bude nám zřejmě připadat, že jsme <pravda> či <nepravda> do bloku zapomněli vložit – takže dávejme pozor na toto vynechávání.

SNAP PROJEKT: Definice [přikaž postavě provést] (zeptej se postavy) #22

4.4.4 Obsah automaticky zaobalující typy parametrů

V prostředí Snap existují ještě tři poslední typy parametrů, které se „zvenku“ bloku neukazují jako zaobalující funkce (, , anebo), a před vstupem do jejich bloku svůj obsah namísto vykonání zaobalí do příslušné zaobalující funkce. Odstiňují problematiku zaobalování (uživatel nemusí znát teorii) a zároveň neumožňují stanovat zaobalené proceduře parametry, na rozdíl od černých šípek u zaobalujících funkcí.

Tyto typy jsou zobrazeny na obrázku 4.4.23. Typ *C-tvar* jsme si už představili, tudíž se zaměříme jen na *libovolný (nevyhodnoceno)* a *pravd. hodnota (nevyhodnoceno)*.

Obrázek 4.4.23 Tři automaticky obsah zaobalující typy parametru

- **Parametr typu „libovolný (nevyhodnoceno)“**

Z obrázku 4.4.10 už víme, jak typ *libovolný (nevyhodnoceno)* vypadá. Lze do něj zapsat číslo či text, a vložit blok funkce či podmínky; jež jsou automaticky zaobaleny do .

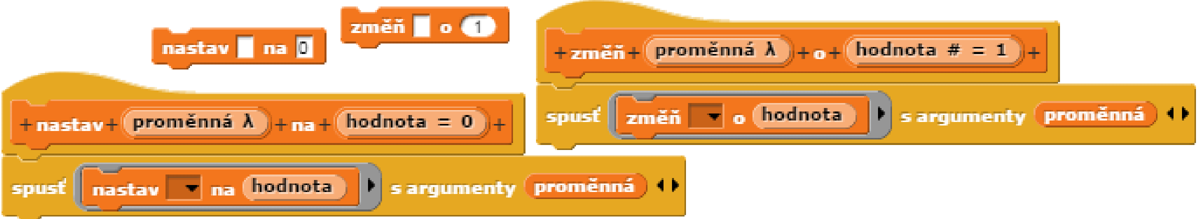
Jelikož je také zaobalující funkcí, můžeme do tohoto parametru předat jí zaobalené příkazy. Dojde však k dvojitému zaobalení – argument parametr *libovolný (nevyhodnoceno)* zaobalí do . Podmínka <je typu> však zázračně detekuje vnitřní možností {zaobalený příkaz}.

V souvislosti s rozbalovací nabídkou jsme v sekci 4.1.3 zmínili *chráněné* a *nechráněné* nabídky. Do chráněných nabídek nemůžeme normálně vložit nějakou funkci (třeba (proměnná)). Existuje však trik, kterým toto chování můžeme obejít.

Vložíme-li například příkaz [nastav] s chráněnou rozbalovací nabídkou do příkazu [spusť], vynecháme výběr možnosti z jeho rozbalovací nabídky, a nakonec dodáme zaobalenou proměnnou do jakožto argument [spusť]; pak prostředí do chráněné

nabídky proměnnou vpraví „násilím“ a nastaví či změní její hodnotu. Jde o již popsané chování, kdy jsou nevyplněné argumenty bloku v zaobalené proceduře doplněny argumenty parametrů zaobalující funkce. Povězme si ale, jak toto souvisí s typem parametru *libovolný (nevyhodnoceno)*.

Na obrázku 4.4.24 jsou **vlastní** bloky [nastav] a [změň], které přijmou proměnnou (třeba (skóre)) a nastaví ji speciálním trikem hodnotu na (hodnota). Už jsme si pověděli, že abychom tímto trikem upravili hodnotu proměnné, musíme ji zaobalit do . Právě proto zde využijeme typ parametru *libovolný (nevyhodnoceno)*, kterým je (proměnná), jež každou dodanou proměnnou za nás automaticky zaobalí.



Obrázek 4.4.24 Definice vlastních bloků [nastav] a [změň]

Jaký je rozdíl mezi primitivními a vlastními příkazy [nastav] a [změň] je ukázáno na obrázku 4.4.25, na němž si ještě předvádíme manipulaci s našimi vlastními příkazy.

Obrázek 4.4.25 Odlišnost a práce s [nastav] a [změň] oproti primitivům

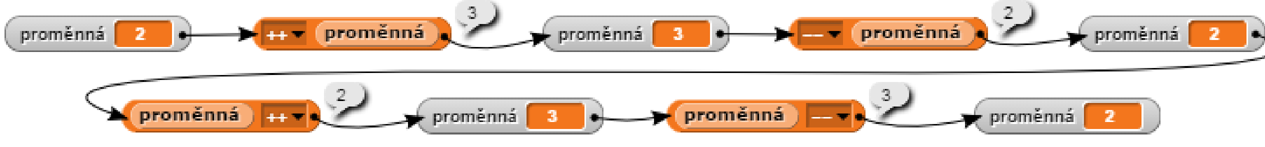
Fakt, že můžeme namísto výběru proměnné z rozbalovací nabídky použít přímo funkci proměnné (např. (pom)) a vsadit ji do vlastních [nastav] a [změň] není zas tak velkou výhodou. Pokud ale upravíme parametr (proměnná) na vícenásobnou variantu, budeme moci nastavit vícero proměnných na stejnou hodnotu jedním příkazem – jak ostatně demonstruje obrázek 4.4.26. To s primitivními bloky [nastav] a [změň] nedokážeme. Představme si, že bychom nyní (x), (y), a (z) museli ručně zaobalovat. To však za nás naštěstí udělá *libovolný (nevyhodnoceno)*, jež se hodil užít do tohoto příkladu.

nastav na cokoliv | nastav x y z na cokoliv | změň o 1 | změň x y z o 1

Obrázek 4.4.26 Vlastní [nastav] a [změň] s vícenásobným parametrem

Mimoto lze speciální trik využít i k implementaci tzv. *unárních operátorů*, kterými měníme dle volby číselnou hodnotu proměnné o jedničku. Dělí se na *prefixové* a *postfixové*. Například *prefixový* inkrement se značkou ++ přičte jedničku k zvolené proměnné ještě před jejím použití. *Postfixový* -- nejprve hodnotu proměnné použije a poté ji sníží.

V prostředí Snap *unární operátory* definujeme funkcemi z kategorie proměnné. Definice *prefixových* a *postfixových* unárních operátorů si ale neukážeme – nalezneme je v projektu. Jak přesně vypadají a fungují se snaží vysvětlit následující obrázek 4.4.27.



Obrázek 4.4.27 Demonstrace funkčnosti vlastních unárních operátorů

SNAP PROJEKT: Definice [nastav], [změň], a unárních operátorů #23

- **Parametr typu „pravdivostní hodnota (nevyhodnoceno)“ a „text“**

Leckdy potřebujeme od uživatele dodat blok podmínky, který potřebujeme zaobalit a vykonat v určitou chvíli funkcí (zavolej), ale nechceme jej zatěžovat s teorií zaobalování, případně těm zkušenějším nechceme umožnit zaobalené proceduře přidat nějaké parametry – tak, jako by to bylo možné u typu .

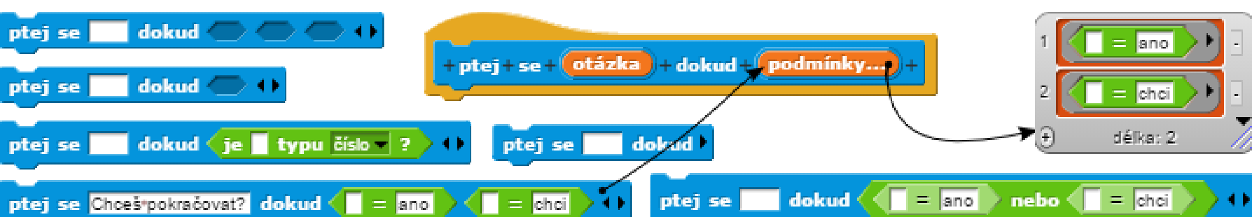
Proto existuje v prostředí typ parametru *pravd. hodnota (nevyhod.)*, který vypadá stejně jako typ *pravdivostní hodnota*. Liší se ale v tom, že namísto aby podmínku zavolal ji raději automaticky zaobalí a uchová k použití na později. Přejděme k příkladu.

Téměř vždy se při tázání uživatele na nějakou otázku blokem [zeptej se] nespokojíme s libovolnou odpovědí. Občas vyžadujeme konkrétní datový typ (například číslo), jindy hodnotu v určeném rozsahu či z nabízených možností, někdy kombinujeme vše dohromady. Bylo by pohodlné, kdyby [zeptej se] uměl sám ověřovat zadanou odpověď uživatele. Při neplatné odpovědi by se na stejnou otázku zeptal podruhé, potřetí, po n-té – a to do té doby, než by uživatelova odpověď odpovídala stanoveným podmínkám.

Vytvořme si příkaz [ptej se dokud], který, kromě svého prvního *text*ového parametru představující otázku na uživatele, bude disponovat ještě parametrem *pravd. hodnota (nevyhod.)* vícenásobné varianty. Vsazením bloků podmínek tak dokážeme jediným příkazem čekat na správnou odpověď (viz obr. 4.4.28). Připomeňme, že podmínky jsou automaticky zaobaleny a tak neotravujeme uživatele zaobalováním.

Černými šipkami opět ovlivníme počet požadovaných podmínek. Příkaz [ptej se dokud] lze vykonat i bez podmínek – pak bude fungovat stejně jako [zeptej se]. Argument nevyplněného argumentu podmínky prostředí samo doplní odpovědí uživatele.

Poslední ukázka se ptá na otázku {Chceš pokračovat?} a požaduje odpověď odpovídající jen hodnotám {ano} nebo {chci}. Zadáme-li podmínky každou zvlášť, blok nebude fungovat podle našich představ. Uvnitř definice [ptej se dokud] totiž používáme <a zároveň> – ne <nebo>. Problém vyřešíme tak, že obě podmínky do podmínky <nebo> vsadíme a tu předáme samostatně bloku [ptej se dokud] (viz obrázek).



Obrázek 4.4.28 Ukázky příkazu [ptej se dokud] a definice jeho hlavičky

Zastavme se u důvodu, proč potřebujeme typ *pravd. hodnota (nevyhod.)* a nevystačíme si pouze s typem *pravdivostní hodnota*. Parametr typu *pravdivostní hodnota* podmínku vykoná těsně před vstupem do bloku s tímto typem parametru, tudíž se podmínka provede nejen v nechtěnou chvílí, ale zároveň bude parametr nabývat pouze pravdivostních hodnot <pravda>/<nepravda>. My však potřebujeme podmínky zaobalit (nevyhodnotit), abychom je mohli opakovaně použít v případě, že uživatel odpoví špatně, jakmile bude vyzván k zadání nové odpovědi.

Sílu tohoto příkazu [ptej se dokud] můžeme pocítit na následujícím projektu. Nacházíme se v matematické třídě, kde učitel ověřuje naše různé matematické znalosti a požaduje přesné odpovědi. V prostředí Scratch 2 bychom museli vytvořit několik vlastních příkazů ptající se na různé otázky a očekávající přesné odpovědi.

Oproti Scratch 2 stačí v prostředí Snap umístit náš příkaz [ptej se dokud] pod sebe a ony všechnu práci s ověřováním odpovědí provedou za nás. Projekt také ukazuje, že lze elegantně předat odpovědi v seznamu parametru vícenásobné varianty.

SNAP PROJEKT: Hra - zkouška v hodině matematiky #24

4.4.5 Parametr přesahující varianty

• Přesahující varianta parametru

Dosud jsme si ještě nevysvětlili poslední variantu parametru nazývající se *přesahující*. Parametr s touto variantou je viditelný „zvenku" bloku svému volajícímu scénáři, který onen blok využívá. Záměrně zmiňujme blok, neboť přesahující parametry využijeme jak u příkazu tak i u funkcí. V následujících ukázkách však užijeme jenom příkazy.

Přesahující parametr není žádného typu – má pouze svou variantu. Uvnitř bloku, kde se vyskytuje v podobě formálního parametru, je vybaven symbolem šipky (↑), ale „zvenku" bloku, jakožto skutečný parametr, vypadá jako obyčejná proměnná.

Jakmile za blok s touto variantou parametru napojíme další příkazy a vytvoříme delší scénář, můžeme onen přesahující parametr klidně (stejně jako kteroukoliv proměnnou) poupravit příkazy [nastav] či [změň]. Nemůžeme se proto spolehnout na jeho hodnotu. Kliknutím levého tlačítka myši na tento parametr zvenku bloku jej přejmenujeme, stejně jako je tomu například u proměnných scénáře.

Když už zmiňujeme proměnné scénáře, ukažme si na obrázku 4.4.29 první využití parametru s přesahující variantou. Při práci s blokem [proměnné scénáře] dokážeme definovat jednu a více proměnných, ale všechny jsou implicitně = {0}. V programovacích jazycích je běžně možné krom *deklarace* proměnné provést i její *inicializaci* – tedy zároveň s definováním jejího názvu jí přiřadit i počáteční hodnotu.

Vytvořme si proto příkaz [proměnná scénáře][1]), jehož první parametr (a) je přesahující varianty a „vydává" se za proměnnou scénáře. Druhým parametrem (hodnota) pak hodnotu parametru (a) nastavíme. Lze tak deklarovat i inicializovat zároveň.

Obrázek 4.4.29 Definice [proměnná scénáře] s přesahujícím parametrem

Následující obrázek 4.4.30 pak ukazuje nalevo rozdíl mezi doposud užívaným způsobem tvorby proměnné scénáře (i), nastavujícím její hodnotu na {1}, a současným pomocí [proměnná scénáře]. Ušetříme jeden příkaz – [nastav], takže je scénář přehlednější.

Napravo téhož obrázku je ještě možné vidět, že můžeme nastavit přesahující parametr (v tomto případě jde o (seznam)) i třeba na referenci seznamu, jelikož parametr (hodnota) uvnitř příkazu [proměnná scénáře] je typu *libovolný* a umožňuje tak vykonat libovolnou funkci či podmínku, na jejíž výsledek se nastaví přesahující parametr.

Obrázek 4.4.30 Další užití [proměnná scénáře] vůči [proměnné scénáře]

• Tvorba vlastních cyklů s využitím parametru přesahující varianty

Ačkoliv prostředí nabízí několik cyklů k řízenému vykonávání scénáře, programovací jazyky disponují povětšinou dalšími druhy cyklů. Dobrou zprávou je, že většinu z nich lze do prostředí Snap s využitím parametru typu *C-tvar* doprogramovat. Jejich představení a vysvětlení je však mimo rozsah této práce. Z toho důvodu si ukážeme pouze zjednodušenou verzi cyklu „for" (známý z jazyka Pascal), jenž definujeme blokem [opakuj pro], a cyklus procházející všechny prvky seznamu [pro každý prvek]. Zdůrazněme

[1]) Všimněme si rozdílu mezi [proměnná scénáře] a [proměnné scénáře].

však, že přesahující parametr není nutný k tvorbě vlastních cyklů. Jsou i takové, které se bez nich obejdou.

- **Cyklus** [opakuj pro] **jako alternativa k** [opakuj n krát]

Tento cyklus je podobný cyklu [opakuj n krát] s tím rozdílem, že je možné přistoupit k jeho počítadlu a navíc můžeme zvolit startovní a konečnou hodnotu tohoto počítadla. Namísto určení opakování n-krát stačí tedy nastavit interval $\langle od, do \rangle$, který sám stanoví počet opakování. Definice tohoto cyklu je uvedena na obrázku 4.4.31.

Obrázek 4.4.31 Definice cyklu [opakuj pro] s přesahujícím parametrem

Počítadlo představuje parametr (i), který je *číselným* typem a *přesahující* variantou. Připomeňme, že kliknutím na tento parametr „zvenku" bloku jej můžeme přejmenovat.

Parametr (i) nastavíme na hodnotu parametru (začátek), neboť ten určuje počátek intervalu opakování. V podmínce primitivního cyklu [opakuj dokud nenastane] pak kontrolujeme, zdali počítadlo (i) nepřekročilo požadovaný interval, jehož konec je reprezentován parametrem (konec). Začátek i konec musejí být celočíselné, aby cyklus fungoval správně. Takto jednoduše jsme postavili nový cyklus na cyklu původním.

Obrázek 4.4.32 zobrazuje funkcionalitu tohoto cyklu. Nejprve je vhodné si uvědomit, že příkazem [nastav] máme možnost ovlivnit hodnotu parametru (i). Uděláme-li tak, cyklus se bude provádět buď méně, více, či nekonečno krát. Tomu však můžeme zabránit v definici [opakuj pro], kde bychom krátkou úpravou scénáře namísto [změň] použili [nastav], abychom zajistili před dalším opakováním cyklu nastavení hodnoty zpět na číselnou. Museli bychom si však pamatovat další pomocnou proměnnou scénáře.

Obrázek 4.4.32 Různé ukázky cyklu [opakuj pro] nutné k jeho pochopení

Další příklad ukazuje, jak můžeme postavě přikázat vyřknout čísla od jedné do desíti příkazem [povídej] využívajícím přesahující parametr (i). Poslední ukázka z obrázku obsahuje dva vnořené cykly [opakuj pro]. Aby nedošlo ke kolizím názvů přesahujících parametrů (prostředí by sice nezahlásilo chybu, avšak cykly by si přepisovaly hodnoty těchto parametrů), byly přejmenovány na (první) a (druhý). Každá iterace prvního cyklu vyústí ve vykonání cyklu vnořeného, jehož každá iterace přikáže postavě říci obě hodnoty oddělené tečkami. V bublině se postupně objeví každých půl sekundy {1..2}, {1..3}, {1..4}, {1..5}, {2..2}, {2..3} ... {3..4}, {3..5}.

Nový cyklus [opakuj pro] užijeme i ke snadnějšímu (přehlednějšímu) procházení datových struktur. Vzpomeňme na *tabulku*, kterou jsme vytvářeli v sekci se seznamy. Onen scénář, který procházel všechny prvky *tabulky*, nalezneme na 4.2.30. Následující obrázek 4.4.33 převzal zobrazenou tabulku z již zmíněného obrázku a ukazuje průchod všech jejích prvků po řádcích s využitím dvou vnořených cyklů [opakuj pro]. První cyklus prochází řádky, zatímco druhý prochází sloupec konkrétního řádku.

Obrázek 4.4.33 Průchod struktury *tabulka* vlastním cyklem [opakuj pro]

• Cyklus [pro každý prvek] pracující s prvky seznamu

Abychom pořád nemuseli při programování průchodu všech prvků seznamu užívat cykly [opakuj n krát], [opakuj dokud nenastane], anebo vlastní [opakuj pro] (s nimiž je scénář méně přehledný), vytvořme si další cyklus určený pouze k průchodu seznamu.

Na obrázku 4.4.34 je uvedena definice tohoto cyklu s názvem [pro každý prvek], který záměrně spadá do kategorie seznamy, neboť pracuje pouze s nimi. Cyklus začne procházet seznam od prvního prvku, následně provede příkazy (vykoná obsah svého těla), a poté se přesune na další prvek v pořadí. Takto pokračuje do té doby, než navštíví poslední prvek dodaného seznamu. Oproti funkci (zavolej nad každým prvkem) neměníme hodnoty prvků seznamu, jen k nim přistupujeme přes (prvek).

Prvním parametrem [pro každý prvek] jest (prvek) *přesahující varianty*, který nabude hodnoty vždy aktuálně procházeného prvku seznamu. Dalším je (seznam) očekávající referenci na seznam, jenž budeme procházet. Třetím je (příkazy) typu *C-tvar* obsahující dodané příkazy, jež provedeme příkazem [spusť] ve správnou chvíli.

```
+pro+každý+(prvek ↑)+v+(seznam ⋮)+(příkazy λ)+
opakuj dokud nenastane <(délka (seznam)) = 0>
  nastav prvek na (prvek 1 z seznam)
  spusť (příkazy)
  nastav seznam na (vše až na první prvek z seznam)

pro každý (prvek) v ☰

pro každý (prvek) v (nový seznam A B C)
  povídej (prvek) příštích 0.5 sekund

pro každý (řádek) v (tabulka)
  pro každý (sloupec) v (řádek)
    povídej (sloupec) příštích 0.5 sekund
```

Obrázek 4.4.34 Definice a ukázky vlastního cyklu [pro každý prvek]

Na pravé straně zmiňovaného obrázku je ukázán nejenom blok [pro každý prvek], ale také dvě ukázky jeho využití. První předává bloku referenci na seznam s prvky {A, B, C}, jenž ale vzápětí zapomeneme. Cyklus jej projde a přes příkaz [povídej] využívající přesahující parametr (prvek) postava vyřkne všechny tři písmena postupně.

Druhá ukázka obsahuje vnořené cykly [pro každý prvek]. Už několikrát jsme prošli datovou strukturu *tabulka*, která obsahuje řádky a sloupce, v místě jejichž protnutí vznikají jednotlivé buňky. Doposud jsme si popsali dva způsoby průchodu *tabulky* – a to na obrázcích 4.2.30 (využívající [opakuj n krát]) a 4.4.33 (se scénář zjednodušujícím [opakuj pro]). Podíváme-li se opět druhou ukázku obrázku 4.4.34, pak vnořené cykly [pro každý prvek] dosáhnou naprosto stejného výsledku. Jedná se o nejpřehlednější a nejjednodušší způsob, jak přikázat postavě vyřknout všechny buňky *tabulky*.

SNAP PROJEKT: Definice cyklů včetně [opakuj pro] a [pro každý prvek] #25

4.5 Bloky (JavaScript funkce) a [zahaj], přepis vybraných bloků do textové podoby

4.5.1 Funkce (JavaScript funkce)

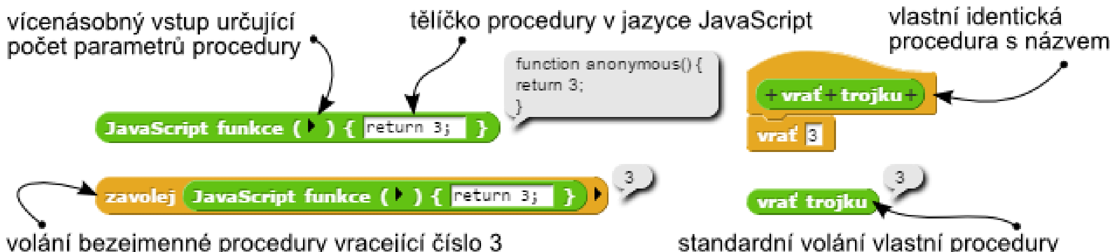
Funkcí (JavaScript funkce) můžeme přesáhnout základní rámec prostředí a vykonat libovolný kód jazyka JavaScript, ve kterém je i prostředí Snap naprogramováno. Skrze ni vytváříme bezejmenné procedury, stejně je tomu u zaobalených funkcí , , . Oproti nim ale do těl procedur píšeme JavaScript kód v textové podobě, kdežto u zaobalujících funkcí využíváme k definici procedur grafické bloky prostředí. Její nevýhodou je vyžadovaná znalost jazyka JavaScript (a to i těmi, jenž projekt jen prohlížejí), dále snadné zanesení chyby, chybějící zbarvování syntaxe; a absence ladícího nástroje.

Ještě než si ale ukážeme využití této funkce v praxi, naučme se s ní pracovat, neboť se jedná o složitější proces. Vrhněme se předně na procvičující a bez účelový příklad.

Na obrázku 4.5.1 jsme si vytvořili dvě procedury. Jednu klasickým způsobem přes editor bloků – tj. s vlastním názvem a kategorií – a druhou přes funkci (JavaScript funkce). Obě procedury dělají totéž, rozdíl je však ve způsobu jejich tvorby a volání. Zaměřme nyní naši pozornost na funkci (JavaScript funkce).

Prvním parametrem funkce (JavaScript funkce) je vícenásobný parametr k určení parametrů procedury. Tento proces je nám znám u již zmiňovaných zaobalujících funkcí. Parametry jsou pouze v textové podobě a odkazujeme se na ně stejným způsobem v těle metody. Tělo se definuje vždy v druhém parametru této funkce. Procedura, kterou nyní vytváříme dvěma způsoby, nemá žádný parametr a okamžitě vrací trojku.

Všimněme si, že ve vlastním bloku používáme k navrácení hodnoty příkaz [vrať], kdežto na úrovni JavaScriptu musíme použít klíčové slůvko `return`, jemuž JS rozumí. U textově založených programovacích jazyků je pravidlem, že příkazy a volání procedur ukončujeme vždy středníkem. I proto se ve funkci objevuje „`return 3;`“.



Obrázek 4.5.1 Ukázka manipulace s funkcí (JavaScript funkce)

Na obrázku je též vrácená hodnota funkcí (JavaScript funkce) v bublině, konkrétně:

```
function anonymous(){
   return 3;
}
```

Pro mnohé bude určitě překvapením, že vrácený obsah v bublině ve skutečnosti není textem, přestože se jeví jím být. To ostatně můžeme otestovat podmínkou <je typu>, která s možností {text} vrátí <nepravda>. Bohužel nemůžeme zjistit, jestli proměnná nabývá bezejmenné procedury vytvořenou funkcí (JavaScript funkce) či nikoliv.

Pro vykonání bezejmenné procedury ve správný čas používáme stejně jako u zaobalených funkcí bloky [spusť] a (zavolej). Protože v ukázkové proceduře vracíme číslo tři, byl použit na obrázku blok (zavolej) k jejímu vyhodnocení v prostředí.

Závěrem popišme definování procedur, které žádnou hodnotu nevracejí. Pakliže bezejmenná procedura definována v těle bloku (JavaScript funkce) neobsahuje klíčové slovo `return`, nic nevrací a proto ji ani nevyhodnocujme blokem (zavolej). V takovém případě je jediným správným řešením užití bloku [spusť], neboť (JavaScript funkce) vrátila proceduru rovnající se obyčejnému příkazu.

• *Poznámka: Nebuďme zmateni tím, že v hlavičce procedury vrácené funkcí* `(JavaScript funkce)` *je vždy slovo „function“. Jazyk JavaScript zná jen funkce (i když jsou příkazy).*

• Využití knihoven jazyka JavaScript

První užitečný příklad bude o řešení nedostatků námi vytvářené umocňující funkce, na niž byla demonstrována rekurze ve funkcích a která je zobrazena v sekci 4.1 na obrázku 4.1.7. Nefungovala za předpokladu, že bylo exponentem záporné celé číslo anebo číslo s desetinnou čárkou. Problém s matematickými operacemi řeší programátoři ve většině programovacích jazyků zabudovanou matematickou knihovnou, která jejich matematické potřeby uspokojí – a stejně jako je tomu i u jazyka JavaScript.

Naším cílem tedy bude upravit tělíčko již zmíněné umocňující funkce tak, abychom se odkázali na zabudovanou matematickou knihovnu jazyka JavaScript, nechali ji vhodnou funkci vypočíst výsledek mocnění a ten vrátili příkazem `(zavolej)`. Výše zmíněné problémy s rekurzivně definovanou umocňující funkcí tak odpadnou, neboť složitý algoritmus přesměrujeme na léty prověřenou knihovnu. Rozšíříme si tak prostředí využitím zabudované knihovny `Math`, která má umocňující funkci `pow` se dvěma parametry.[1])

Na obrázku 4.5.2 je zobrazené řešení. Naše vlastní umocňující funkce vytvořená uvnitř prostředí stále využívá parametry `(základ)` a `(exponent)`, které předává funkci `(JavaScript funkce)` jakožto argumenty v příkazu `(zavolej)`. Ty se do `(JavaScript funkce)` přesunou pod parametry `a` a `b`, protože tak byly definovány ve vícenásobném parametru (a odlišně, aby nedošlo k záměně pojmů). Nakonec v tělíčku této bezejmenné procedury najdeme jednoduchý příkaz `return Math.pow(a,b);`, což můžeme přeložit jako „*Jdi do matematické knihovny, zavolej funkci* `pow` *s určitými hodnotami, a nakonec vrať výsledek výpočtu.*“. Jde o relativně elegantní, ale hlavně bezchybné řešení.

Obrázek 4.5.2 Tvorba umocňující funkce s využitím `(JavaScript funkce)`

• Přístup do zdrojových souborů prostředí Snap

Byla by škoda, kdybychom nezkusili rozšířit možnosti prostředí Snap přístupem do jeho zdrojových kódů přes funkci `(JavaScript funkce)`. Zkusme si doplnit blok `[nastav způsob otáčení]`, kterým na rozdíl od Snap disponuje prostředí Scratch 2. Vytvořme blok s identickým názvem a kategorií, jenž umožní ve svém parametru s needitovatelnou rozbalovací nabídkou zvolit způsob otáčení postavy.

Hotový blok je obrázku 4.5.3. Prostředí Snap definuje ve svých zdrojových kódech tři způsoby otáčení postavy číselnou hodnotou – zákaz otáčení nulou, otáčení dokola jedničkou, a otáčení jen vlevo/vpravo dvojkou. Proto bezejmenná procedura zasahující do zdrojových souborů má parametr `x` představující číselně vyjádřený způsob otáčení. Její tělo obsahuje jednoduchý příkaz `this.rotationStyle = parseInt(x);`, což opět přeložme jako „*Způsob otáčení této postavy budiž roven hodnotě* `x`*, kterou převedeme z textové hodnoty na číselnou.*“.

Vzpomeňme na zaobalující funkce a definování jejich parametrů. Nemáme možnost určit, jakého typu ony parametry budou (zdali mají být číselné, textové, logické, či jiné hodnoty). To si už musíme hlídat sami pořadím zadávaných argumentů. Z toho důvodu bylo nutné použít funkci `parseInt`, která se pokusí hodnotu argumentu převést na číslo. Bez ní by totiž naše funkce nefungovala správně. Návrhem bloku jsme si jisti, že převod na číslo bude vždy úspěšný a tak je – alespoň na první pohled – práce hotova.

---

1) `Math` je zkratkou pro `mathematics` a `pow` zase pro `power`, což znamená `mocnina`.

Obrázek 4.5.3 Tvorba [nastav způsob otáčení] s (JavaScript funkce)

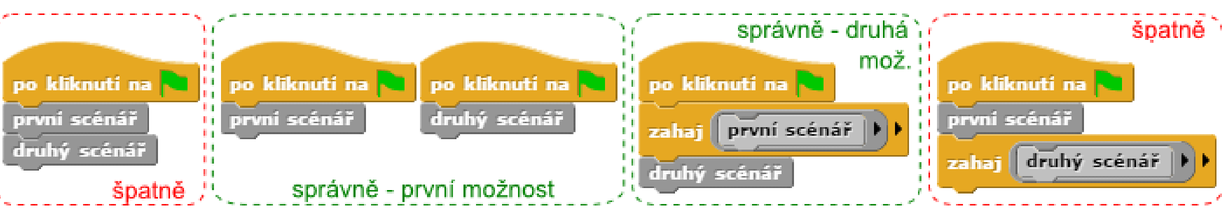
I tento příklad má jeden nedostatek. Ačkoliv se v logice prostředí změní hodnota způsobu otáčení a postava na to správně zareaguje, tlačítka umožňující změnu způsobu otáčení (umístěna v horním panelu nad paletou scénářů) se neaktualizují a nezobrazí tak aktuální způsob otáčení postavy. Abychom tomuto problému předešli, museli bychom rozšířit obsah tělíčka naší bezejmenné procedury v bloku (JavaScript funkce), aby při změně rotace došlo i k aktualizování těchto tlačítek. Pro potřeby demonstrace je to však zatěžující „povinnost“ snižující přehlednost řešení. Skvěle ale demonstruje různé nástrahy, kterým musíme čelit při rozšiřování zdrojových kódů Snap.

SNAP PROJEKT: Definice bloků užívajících funkci (JavaScript funkce) #26

4.5.2 Příkaz [zahaj] zahajující scénář v novém procesu

Jsou chvíle, kdy potřebujeme zahájit více scénářů ve stejný okamžik – jinak řečeno najednou, synchronizovaně, či paralelně. Kromě jednoho způsobu, jež je možné využít ve všech prostředích, má ještě prostředí Snap k tomu určený primitivní blok [zahaj]. Vyžaduje část scénáře, tedy proceduru, kterou zaobalí a spustí (resp. zahájí) v odděleném procesu. Procedura se začne provádět nezávisle na původním scénáři, ve kterém se příkaz [zahaj] spustil. Podívejme se na obrázek 4.5.4 obsahující různé způsoby (jak špatné, tak správné) k zahájení více scénářů ve stejný čas. Popišme si jej podrobně.

První červený rámeček ukazuje špatné řešení. Scénáře (reprezentované pomocnými bloky [první scénář] a [druhý scénář]) neprovedeme paralelně (najednou), nýbrž sériově, jelikož jsou umístěny za sebou. Druhý rámeček v pořadí ukazuje už zmíněný a funkční způsob spouštějící oba scénáře synchronizovaně. Po kliknutí na tlačítko se zelenou vlaječkou se dvakrát spustí blok události [po kliknutí na start]. Ačkoliv jde o stejnou událost, odchytáváme ji zmíněným blokem dvakrát a provádíme jím odlišné scénáře. Ani tento způsob není ideální. Nelze použít, když potřebujeme další část scénáře provést paralelně někde v polovině jiného scénáře. To by šlo vyřešit rozesíláním zpráv, ale s nimi přichází i některé další nevýhody – proto na ně raději zanevřeme.



Obrázek 4.5.4 Špatné a správné způsoby užití příkazu [zahaj]

Třetí rámeček ukazuje konečně správné řešení. [první scénář] bude spuštěn příkazem [zahaj], který je na začátku scénáře. Hned na to se spustí blok [druhý scénář] ve stejném scénáři (procesu). Oba bloky se tak budou vykonávat najednou. Příkaz [zahaj] totiž na nic nečeká, spustí scénář bezejmenné zaobalené procedury a pokračuje ve vykonávání dalších příkazů (v našem případě k vykonání bloku [druhý scénář]).

Na obrázku 4.5.4 je ještě na konci v červeném rámečku ukázka upraveného a špatného řešení, jež je dobré uvážit. Jelikož se příkaz [zahaj] přesunul pod [první scénář], spustí se až po jeho vykonání. Proto se oba scénáře nikdy nevykonají nastejno. Příkaz [zahaj], přestože spustí [druhý scénář] v novém procesu, musí čekat na dokončení [první scénář]. Proto je toto řešení nefunkční – alespoň dle našeho zadaní.

Dost bylo o předchozí ukázce. Podívejme se na užitečnější příklad. Víme, že existují příkazy, které se provádějí nějakou námi zadanou dobu, o kterou zdrží vykonávání bloků následujících. Mezi takové řadíme: [plachti na pozici], [povídej příštích], [pomysli si příštích], [pauza], [zahraj tón], a [čekej]. Nezapomeňme zmínit ještě [zeptej se], který zastaví scénář dokud uživatel neodpoví na otázku. Obrázek 4.5.5 zobrazuje čtyři scénáře, jejichž společným cílem je ve stejný okamžik posunout postavu, nechat ji promluvit a zároveň zahrát tón. Jen jeden z nich je však správný.

Obrázek 4.5.5 Špatné a správné užití vícero [zahaj] v jednom scénáři

První špatně složený scénář vykoná příkazy postupně, což nechceme. Druhý scénář je umístěn v zaobalené proceduře příkazu [zahaj], který ho spustí v odděleném procesu, avšak příkazy scénáře se stále vykonají postupně a nikoliv najednou. Třetí špatně složený scénář už je téměř správně. Procedura obsahující příkazy [plachti na pozici] a [povídej příštích] se spustí příkazem [zahaj] společně s následujícím příkazem [zahraj tón], avšak ony dva zmíněné příkazy se nespustí vzájemně a tak se postava nejrpve pohne a zahraje tón, a až po dvou sekundách řekne „Ahoj!“ – což je mimochodem v úplně jiném pořadí, než jsme si původně stanovili.

Zbývá nám popsat poslední – správně řešený – scénář. Blok [plachti na pozici] se spustí příkazem [zahaj], po němž hned následuje tentýž příkaz zahajující v odlišném procesu [povídej příštích], a po tomto druhém příkazu [zahaj] se zároveň provede [zahraj tón]. Všechny příkazy poběží najednou ve stejný čas (dva z nich v odlišných procesech). Jakmile se příkazy ukončí, procesy společně s nimi zaniknou. Blok [zahraj tón] už nemusíme dávat do příkazu [zahaj], ale může existovat případ, kdy bychom chtěli spustit nějaký další příkaz po tomto příkazu a nechtěli čekat dvě sekundy, než se tón přehraje. V tom případě bychom [zahraj tón] též vložili do zaobalené procedury [zahaj], který by tak byl ve scénáři třikrát za sebou.

Nezbývá než dodat, že příkaz [zahaj] v sobě může mít samozřejmě nespočet dalších příkazů [zahaj] – scénář spuštěný v novém procesu spustí další scénář v odlišném procesu. Také zmiňme, že příkazem [zahaj] si při kreslení želví grafiky nepomůžeme. Na vykreslení obrazce nebude mít užití tohoto bloku vliv. Obrazec se zřejmě špatně vy-

kreslí, protože různé scénáře spuštěné v různých procesech budou manipulovat s pozicí postavy. Přejděme už ale konečně k projektu demonstrující příkaz [zahaj].

Postava v projektu tancuje na lidovou píseň „Tancuj, tancuj, vykrúcaj“, jejíž text také zpívá a hraje v tónech. Tancuje, zpívá, a hraje pouze dvě sloky této písně, neboť více netřeba. Právě v tomto případě využíváme příkaz [zahaj] k současnému pohybu postavy, přeříkávání textu, a přehrávání tónů písně.

SNAP PROJEKT: Užití [zahaj] na písni „Tancuj, tancuj, vykrúcaj" #27

Do (nerozdělený_text) byl importován text písně ze souboru, který je rozdělen podle řádků funkcí (rozděl). Vrácený seznam touto funkcí je uložen do (seznam_textů) a je mimochodem seznamem kruhovým. Způsoby tance jsou v bezejmenných procedurách jako prvky dalšího kruhového seznamu uloženým v (seznam_tanců). Poslední kruhový seznam obsahuje názvy částí písně, které určují přehrávanou pasáž písně „Tancuj, tancuj, vykrúcaj“. Tyto tři kruhové seznamy jsou nastaveny v bloku [připrav postavu].

V projektu dále využíváme blok [nastav způsob otáčení], který jsme si ukázali v předchozí části věnované funkci (JavaScript funkce). Pak [hrej píseň Tancuj, tancuj, vykrúcaj] a <procesy dokončeny?>. Blok <procesy dokončeny?> využíváme ke sjednocení zpívání, tancování, a hraní tónů, neboť prostředí Snap nemá optimalizovanou manipulaci s kruhovými seznamy.

Obrázek 4.5.6 ukazuje scénář, který řídí zpívání, tancování, a přehrávání tónů postavou. Neustále postava zpívá část písně, tancuje kus nějakého tance, a přehrává její další úsek. Pozornému čtenáři neušlo, že jsme si identické řešení ukázali v předchozím obrázku 4.5.5. V projektu těžíme především z kruhových seznamů. Hlavně však můžeme vykonávat různé bloky, které se provádějí nějakou dobu, současně. Nemusíme ani rozesílat a odchytávat zprávy. Kdyby se totiž v projektu vyskytovaly další postavy, obdrželi by informaci o rozeslané zprávě – a to nechceme. Vše je nyní vyřešeno v jednom scénáři za využití příkazu [zahaj].

Obrázek 4.5.6 Scénář a snímek z projektu „Tancuj, tancuj, vykrúcaj“

4.5.3 Přepis bloků do textové podoby

Prostředí Snap má nástroj převádějící bloky do textové podoby. Blokům určeným pro převod musíme určit, jakým textovým zápisem mají být přesně vyjádřeny. Je tak možné převést scénář složený z grafický komponent (bloků) v prostředí do jiného textově založeného programovacího jazyka. Převedený scénář na text je vhodné uložit do proměnné, jejíž obsah přes nabídku ve sledovači vyvolanou pravým tlačítkem myši vyexportujeme do nové záložky prohlížeče, odkud můžeme text zkopírovat anebo uložit do souboru.

Jelikož je prostředí Snap skriptovací prostředí, lze předpokládat, že jeho scénáře budeme převádět též do skriptovacích programovacích jazyků. Nabízí se například JavaScript, Python, Lua, Perl, anebo Smalltalk. Přesto je možné převést bloky i do dalších programovacích jazyků, jakými jsou třeba procedurální C či funkcionální Scheme.

Podle jednoho z vývojářů je však hlavním cílovým jazykem k převodu scénáře právě JavaScript, na němž je prostředí postaveno. U projektů se scénáři náročnými na provádění příkazů či výpočetní rychlost je totiž vhodné scénáře převést do textu a vykonat kód v jazyce JavaScript funkcí `(JavaScript funkce)`, čímž dojde k zlepšení výkonu.[47]

Nástroj je potřeba pro každý projekt aktivovat zvlášť v nastavení prostředí a to zaškrtnutím možnosti „*Podpora přepisu bloků do textu*“, jež přidá do kategorie `ostatní` čtyři doplňující bloky `[převeď blok]`, `[převeď String]`, `[převeď část]`, a `(kód z)` (viz obrázek 4.5.7). Autoři nazývají tuto možnost složitě jako *podpora kodifikace*.

Obrázek 4.5.7 Čtyři bloky určené k přepisu bloků a jejich částí do textu

Se zapnutým nástrojem pro převod bloků do textové podoby se většina částí umístěných na blocích promění v aktivní prvky, jež zobrazí po kliknutí na ně pravým tlačítkem myši nabídku s možnostmi. Převod do textu lze totiž provést i přes speciální dialogová okna, která skrze nabídky otevřeme. My se však zaměříme na převod přidanými bloky.

- **Příkaz** `[převeď blok]` **k převodu bloku do kódu a příkaz** `[převeď String]`

Víme-li, že budeme potřebovat převést nějaký blok v prostředí, musíme mu určit jeho textovou podobu, jež bude užita při převodu. Obrázek 4.5.8 ukazuje příkaz, místo něhož se může vyskytnout jiný libovolný blok z prostředí. Všimněme si (už i na obrázku 4.5.7), že první parametr bloku `[převeď blok]` je typu *funkce*, který vložené bloky zaobalí do bezejmenné procedury namísto aby je vykonal. Totéž platí u `(kód z)`, která přijme blok nebo část scénáře, následně se pokusí obsah zaobalené procedury převést do textu, a vrátí výsledek. Převést je možné samozřejmě i bloky typu funkce a podmínky, jen musíme zajistit, aby byly zaobaleny správnou zaobalující funkcí (příkazy do , funkce do , a podmínky do), což právě u těchto bloků snadno přehlédneme. Doplňme, že si Snap pamatuje textovou podobu bloku a tak jej stačí převést jen jednou.

Obrázek 4.5.8 Převod přes `[převeď blok]` a přístup k textu přes `(kód z)`

Některé bloky mají také své parametry, jež budou nabývat hodnot dle zvolených argumentů. Při převodu do textu je tedy nutné převést i hodnoty těchto argumentů. Na ně se v `[převeď blok]` odkazujeme zápisem `<#n>`, kde n je pořadí argumentu.

- *Poznámka: K zápisu různých znaků (`<`, `#`, `>` a dalších) budou muset někteří přepnout v operačním systému klávesnici z české jazykové sady na anglickou.*

Dále na obrázku 4.5.9 převádíme primitivní blok `[povídej]` do jazyka JavaScript. Příkaz *alert* (angl. „výstraha“) zobrazí v prohlížeči dialogové okno s hlášením, které je tomuto příkazu zadáno v závorkách – jazyk JavaScript parametry i argumenty ohraničuje kulatými závorkami a příkazy ukončuje středníkem. Podoba přepisu `alert(<#1>);` tak způsobí, že se blok převede do `alert(argument);`, kde za `argument` prostředí dosadí hodnotu argumentu. A proto na obrázku funkce `(kód z)` vrací `{alert(Zpráva);}`, ale i `{alert(123);}`. Výsledek bude v případě jiného argumentu logicky odlišný.

Obrázek 4.5.9 Převod primitivního bloku s argumentem a přístup k podobě

Přesto – (kód z) vrací jen text. Abychom příkaz *alert* vykonali v jazyce JavaScript, musíme jej vložit do druhého parametru funkce (JavaScript funkce). Vrácený výsledek, tedy bezejmennou proceduru s hlavičkou function anonymous()..., lze vykonat příkazem [spusť]. Až poté zobrazí internetový prohlížeč dialogové okno se zprávou odpovídající námi zadané hodnotě {123}. JavaScript užívá function i pro příkazy.

Blok [převeď String] máme k převodu *textového* datového typu, jenž je v počítačové terminologii nazýván tradičně jako „String“. Ve většině programovacích jazycích se text musí obklopit uvozovkami, apostrofy, popřípadě jinými znaky.

Vraťme se ještě k předchozímu příkladu z obr. 4.5.9 a zkusme zaměnit u (kód z) umístěném v (JavaScript funkce) argument {123} za {Zpráva}. Pokus o vykonání bezejmenné procedury by poté skončilo chybou Zpráva is not defined. To proto, že se JavaScript snažil nalézt proměnnou tohoto názvu, avšak neúspěšně. Aby vnímal Zpráva jako text, musíme jej obklopit uvozovkami. To by ale za nás mělo dělat prostředí automaticky, čehož dosáhneme použitím bloku [převeď String]. Poté budeme moci příkazem [spusť] s funkcí (JavaScript funkce) provést alert(Zpráva); bezchybně.

Prostředí nám dává možnost v bloku [převeď String] definovat znak *před* textovým argumentem a znak *po* něm následující. Proto nesmíme zapomenou odkázat se na dodaný argument zkratkou <#1>. Ukázkou zápisu buďiž: '<#1>'. Poté například slovo bude převedeno na 'slovo'. Řešení pro jazyk JavaScript ukazuje obrázek 4.5.10.

Obrázek 4.5.10 Převod textové hodnoty přes [převeď String]

Předchozí obrázek ukazuje ještě jednu věc. Argument {123} v bloku [povídej] funkce (kód z) neohraničila uvozovkami a vrátila text {alert(123);}. To proto, že prostředí Snap samo detekovalo hodnotu {123} jako číselnou (netextovou) a tak na ní neprovádělo žádnou dodatečnou úpravu. U argumentu {Zpráva} ale (kód z) uvozovky dodala.

Součástí scénářů jsou i komentáře, které je nedobré ignorovat. Prostředí však jejich převod neumožňuje, o což si autor této práce sám zažádal na GitHub#373. Představa je taková, že blok [převeď String] místo nápisu „String“ bude obsahovat rozbalovací nabídku s možnostmi {String/komentář}. Převod obsahu grafického komentáře by se definoval obdobně jako u přepisu textového datového typu, tedy například: //<#1>.

• Převod definice celého vlastního bloku příkazem [převeď blok]

Definice primitivních bloků jsou nám v prostředí nepřístupné, neb jsou naprogramovány na nižší úrovni v jazyce JavaScript. Proto nám stačí přes [převeď blok] určit textovou podobu jen při žádosti o jejich provedení či volání. Příkaz [převeď blok] ale ve své rozbalovací nabídce obsahuje vyjma {kód} i možnost {hlavička}, která slouží k určení textové podoby hlavičky vlastního bloku včetně ohraničení kódu tělíčka.

Konkrétní ukázku uvidíme na obrázku 4.5.11, kde je vytvořen vlastní příkaz [test] se třemi parametry ((x), (y), a (z)), v jehož tělíčku je třikrát blok [povídej] využívající hodnoty uvedených parametrů. Na levé straně onoho obrázku je pak podpora pro přepis těchto bloků do textu. Převod bloku [povídej] do {kódu} již známe z předešlé ukázky. Protože však příkaz [test] jazyk JavaScript nezná, musíme jej ho naučit.

Obrázek 4.5.11 Převod vlastního bloku [test] – tělíčka i hlavičky

Nejprve [test] převádíme do {hlavičky}. Ačkoliv je příkazem, JavaScript vyžaduje k jeho definici klíčové slovíčko function. Za tímto slovem následuje název příkazu, poté názvy parametrů oddělené čárkou, jež ještě musíme obehnat kulatými závorkami, a nakonec složené závorky definující hranice tělíčka. Příkazy v tělíčku vlastního bloku však do definice hlavičky nepíšeme – o to se prostředí postará samo. Musíme jen určit místo, na které scénář tělíčka v textové podobě vloží. Z toho důvodu se mezi složenými závorkami objevuje speciální slovo <body> (z angl. „tělo“).

Přestože se už JavaScript naučil náš příkaz [test] definicí jeho hlavičky v textové podobě, ještě jsme ho nenaučili zapsat žádost o jeho vykonání. To řeší třetí a poslední příkaz v ukázce převádějící [test] do {kódu}. Textová podoba obsahuje název příkazu shodný s definicí v hlavičce a dále odkazy na tři argumenty oddělené čárkou. Též jsou obehnány závorkami – přesně jak pravidla jazyka JavaScript vyžadují.

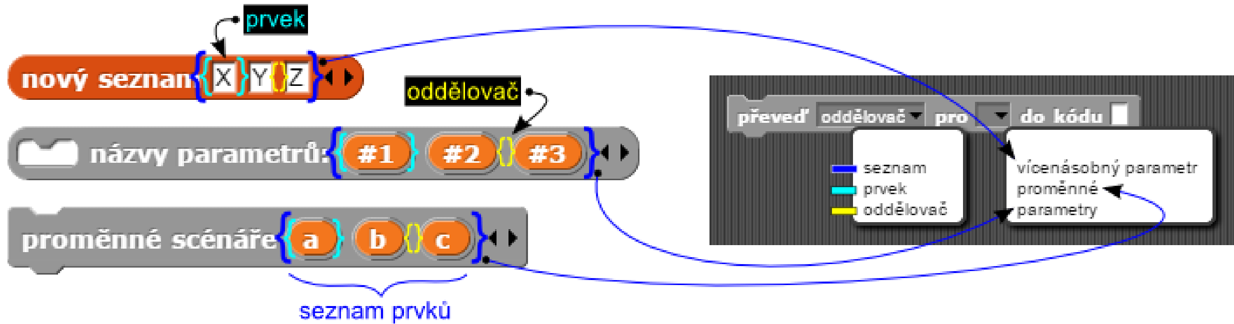
Ukázku zakončeme popisem textu, který vrátí funkce (kód z). Kdyby se v zaobalené proceduře vyskytoval příkaz [test] vícekrát, jeho definice (tj. hlavička a tělíčko) by se vygenerovala stále jedenkrát – jinak by jsme se snažili JavaScript nesmyslně naučit dva stejně pojmenované příkazy. Na posledním řádku vráceného textu je pak žádost o provedení našeho příkazu [test] s argumenty {Ahoj, Světe!, 42}. Všimněme si, že textovým hodnotám prostředí samo dodalo uvozovky. Pokud nyní zkusíme vykonat tento úryvek textu funkcí (JavaScript funkce) a příkazem [spusť], pak webový prohlížeč třikrát zobrazí dialogové okno s {Ahoj}, poté {Světe!}, a {42}.

- **Příkaz [převeď část] a převod zaobalujících funkcí**

Argumenty parametru vícenásobné varianty, zaobalené procedury, a proměnné scénáře běžně používáme i v jiných programovacích jazycích. Při převodu bloků s těmito prvky (částmi) je třeba se o jejich textovou podobu postarat k tomu určeným příkazem [převeď část]. Někdy je potřeba obklopit nějakými znaky převáděnou část, občas obklopit konkrétní jeden argument, parametr, či proměnnou, a jindy vmísit oddělovač mezi jednotlivé z nich. Tento složitý popis si raději ukážeme na příkladu s obrázkem, protože práce s tímto příkazem není vůbec jednoduchá.

Příkaz [převeď část] obsahuje dvě rozbalovací nabídky. Tou **druhou** volíme, jakou část bloku budeme převádět – zdali argumenty vícenásobného parametru, parametry zaobalené procedury, či proměnné scénáře – viz obr. 4.5.12. No a první nabídkou určujeme, zdali převedeme skupinu části, prvky části, a nebo oddělovač prvků části.

Uvažme, že bychom chtěli převést argumenty vícenásobného parametru (na obr. 4.5.12 {X}, {Y}, a {Z} ve funkci (nový seznam)) do této podoby: [{X}; {Y}; {Z}]. V tom případě budeme muset užít blok [převeď část] třikrát. Abychom hodnoty obklopili hranatými závorkami [...] před a za skupinou argumentů, zvolíme z možností nabídek {seznam} a {vícenásobný parametr} – tu ponecháme i pro další dvě použití [převeď část]. Pro obklopení každého prvku zvlášť (např. {X}) složenými závorkami {...} musíme namísto {seznam} v rozbalovací nabídce zvolit {prvek}. No a konečně doplnění středníku (;) mezi jednotlivé argumenty provedeme výběrem {oddělovač}. U jediné {oddělovač} nepoužíváme <#1> – nepřidáváme znak před ani za oddělovač.



Obrázek 4.5.12 Pomocný obrázek k pochopení příkazu [převeď část]

Poslední obrázek využívá blok [převeď část] k převodu oddělovače mezi parametry zaobalené procedury. Ještě jsme si ale neukázali, jak zaobalující funkce přepíšeme, čehož je možné dosáhnout pouze mazaným způsobem. Zaobalující funkci (, , či) musíme dvojitě zaobalit – tedy zaobalit třeba do . Prvním parametrem zaobalujících funkcí jsou vložené bloky a druhým (<#2>) parametry procedury.

Obrázek 4.5.13 Způsob umožňující převést zaobalující funkci

• Mazání a přepisování textových podob

Chceme-li se zbavit stanovené textové podoby některého jednou již převedeného bloku, stačí jej převést znova, ale žádnou textovou podobu mu nedefinovat ({}). Někdy však tento způsob překvapivě nefunguje a je proto třeba využít dialogové okno, ve kterém textovou podobu odstraníme. Připomeňme, že příslušné dialogové okno zobrazíme výběrem možnosti z nabídky zobrazenou kliknutím pravého tlačítka myši na konkrétní část (název bloku, argumenty parametru vícenásobné varianty, proměnné scénáře, parametry zaobalené procedury, atd.) převáděného bloku.

Pakliže budeme převádět bloky do více programovacích jazyků, musíme počítat s tím, že před převodem do dalšího jazyka musíme odlišně definovat textové podoby převáděných bloků, jinak by je (kód z) převedla do podoby jazyka předešlého.

• Příklad v projektu

Autor práce řešil otázku, jaký programovací jazyk pro převod scénáře do textové podoby zvolit v ukázkovém projektu. Přeci jenom nechtěl čtenáře obtěžovat s instalátory různých jazyků a zároveň zohledňoval jejich složitost. Proto vybral on-line editor želví grafiky[1]), jenž byl vytvořen v rámci diplomové práce studentem Bc. Martinem Korbelem na Masarykově Universitě v Brně. Jazyk sice rozeznává příkazy v angličtině, přesto má jednoduchou syntaxi a ve většině si rozumí s bloky existujícími v prostředí Snap.

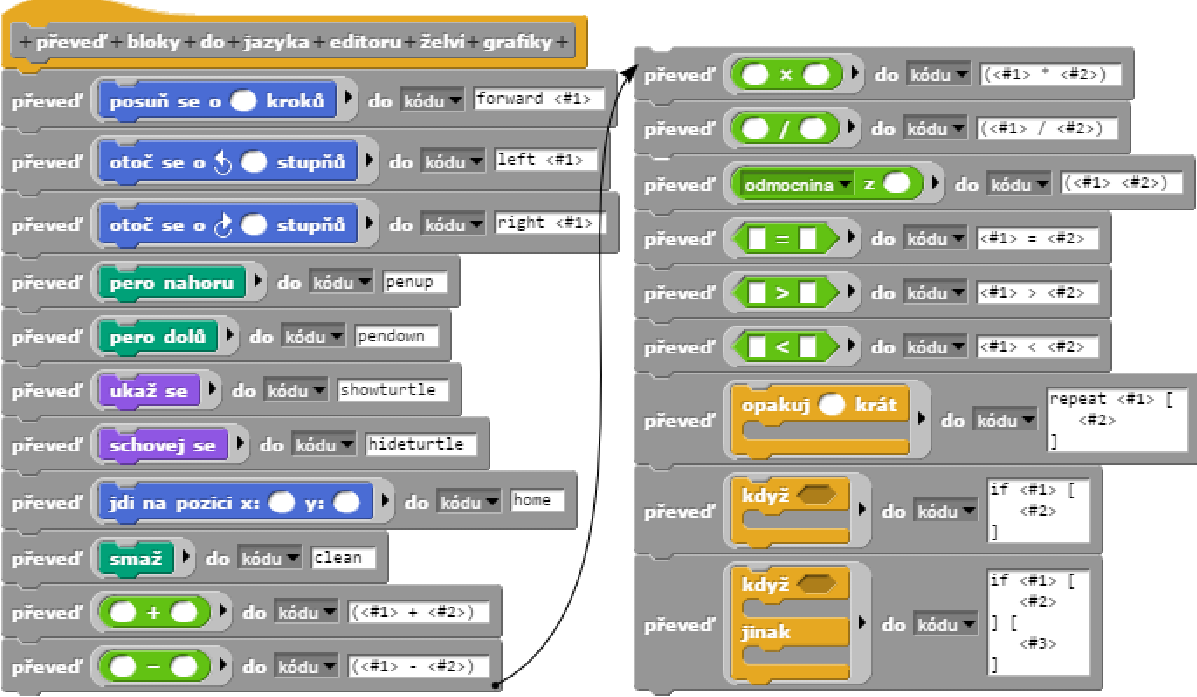
Editor želví grafiky obsahuje několik ukázek vykreslující různé obrazce. Dvě z nich byly užity i v následujícím projektu. To znamená, že autor této práce sestavil scénáře bloky přítomnými v prostředí Snap přesně podle uvedených algoritmů dvou vybraných obrazců z editoru želví grafiky. Ty poté převedl do textové podoby, jejíž podporu musel do prostředí doprogramovat. Převedený text vložil do editoru želví grafiky a porovnal výsledek vykreslený mezi oběma prostředími. Lze konstatovat, že převod byl úspěšný.

Pojďme si to hlavní vysvětlit a ukázat na rozsáhlých obrázcích. Projekt umí vykreslit několik n-úhelníků v řadě a také sofistikovanější spirálu. Jelikož je definice spirály příliš složitá, ukážeme si pouze příklad s kreslením pěti n-úhelníků.

`SNAP PROJEKT: Přepis scénáře do on-line editoru želví grafiky` #28

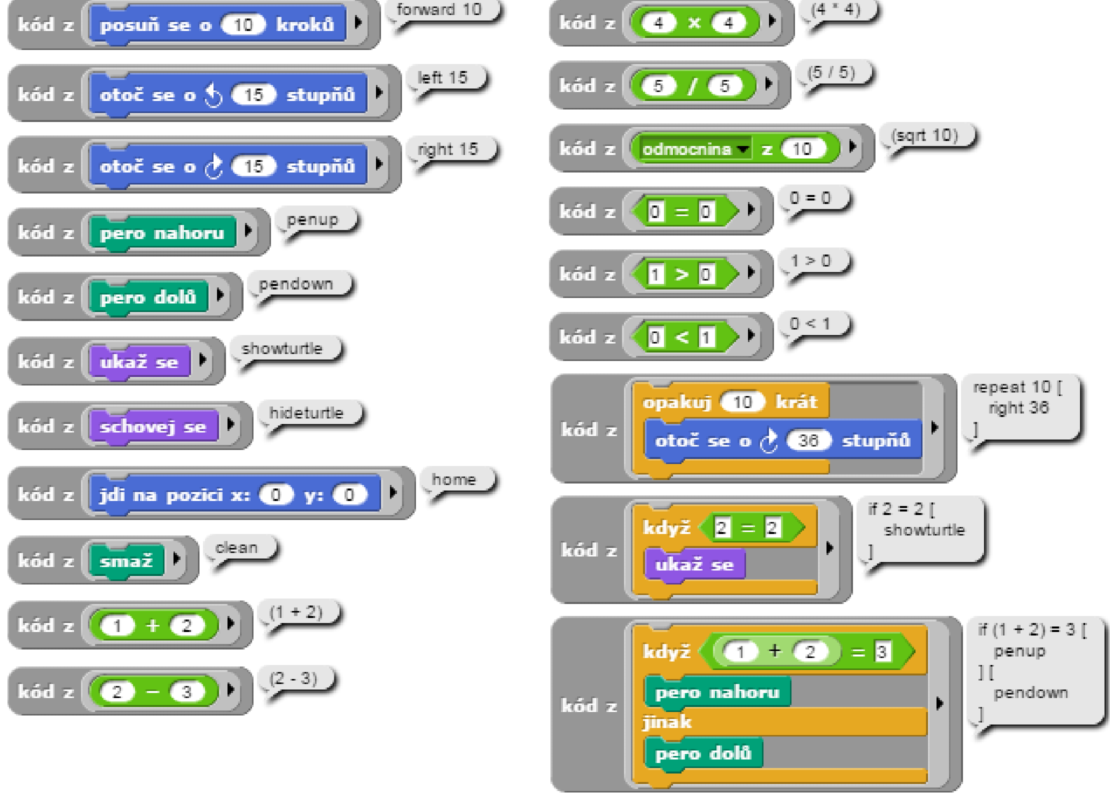
Nejprve si však musíme ukázat, jak syntaxi editoru želví grafiky sjednotit s bloky v prostředí Snap. Na obrázku 4.5.14 je definice bloku [převeď bloky do jazyka editoru želví grafiky], ve kterém používáme většinu příkazů, funkcí, a podmínek rozeznávaných oběma jazyky. Jejich převod je nutný, jinak bychom scénář z Snap do jazyka editoru želví grafiky nepřevedli.

[1]) http://www.fi.muni.cz/~xpelanek/logo/



Obrázek 4.5.14 Převod bloků do užívané syntaxe editoru želví grafiky

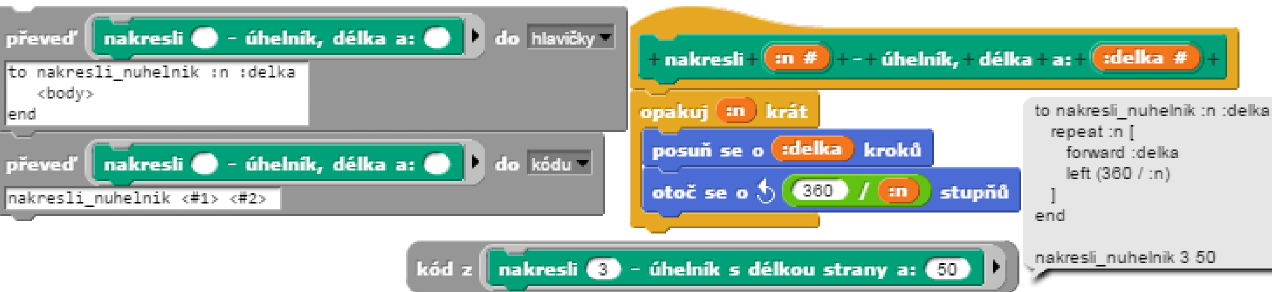
Převod řídících struktur nás může zaskočit svou složitostí. U například primitivního příkazu `[když-jinak]` je první parametr typu *podmínky* očekávající blok se špičatými konci. Druhý a třetí parametr jsou typu *C-tvar* očekávající dodané příkazy. Proto se při převodu tohoto bloku na obr. 4.5.14 odkazujeme na všechny tři argumenty (<#1>, <#2>, <#3>) v různých částech textové podoby. Raději si ukažme ony převedené bloky na obrázku 4.5.15 a porovnejme jejich textovou podobu s grafickým zápisem ve funkci `(kód z)`. Jsou mimo jiné uvedené ve stejném pořadí v jakém byly převedeny.



Obrázek 4.5.15 Textové podoby převedených bloků pro editor želví grafiky

Za povšimnutí stojí příkaz [jdi na pozici], jenž sice ve funkci (kód z) vystupuje s argumenty {0, 0}, přesto je jeho textovou podobou funkcí vrácené {home}. To proto, že když chceme želvu v editoru želví grafiky přesunout doprostřed plátna, stačí napsat pouze příkaz home bez uvedení přesných souřadnic. Při převodu tohoto bloku tak nebylo potřeba převádět argumenty x-ové a y-ové osy zkratkami <#1> a <#2>.

Přestože jsme převedli všechny potřebné bloky do editoru želví grafiky, ještě musíme vytvořit vlastní příkaz na kreslení n-úhelníku. Editor želví grafiky umisťuje před názvy parametrů dvojtečku (:), proto v definici bloku [nakresli n-uhelnik] na obrázku 4.5.16 musíme tomuto pravidlu parametry (n) a (délka) přizpůsobit. U textové podoby hlavičky si také všimněme slov to a end definující hranice bloku. Na procvičení připomeňme, že na místo <body> vkládá prostředí Snap vložit textovou podobu tělíčka.



Obrázek 4.5.16 Převod vlastního bloku [nakresli n-uhelnik] z projektu

Konečně se dostáváme ke kreslení n-úhelníků v prostředí Snap a k převádění kreslícího scénáře do jazyka editoru želví grafiky. Vše je uvedeno na obrázku 4.5.17. Nejprve byl scénář vložen do zaobalující funkce a reference na něj uchována na více použití do (příklad). Následně se do proměnné (kód) uložila textová podoba kreslícího scénáře funkcí (kód z). Nakonec prostředí Snap vykreslí obrazce příkazem [spusť] na scéně.

Obrázek 4.5.17 Užití [nakresli n-uhelnik] vykreslující pět n-úhelníků

Na předchozím obrázku 4.5.17 je ještě zobrazen sledovač proměnné `(kód)`, který obsahuje text převedených bloků z funkce `(kód z)`. Klikneme-li pravým tlačítkem myši na tento sledovač, můžeme z nabídky výběrem možnosti „*export...*“ obsah proměnné zobrazit (exportovat) na nové záložce internetového prohlížeče.

Protože na nové záložce prohlížeče je možné vyexportovaný text označit, můžeme jej zkusit vložit do on-line editoru želví grafiky. Navštívíme-li webovou stránku na které je uložen, stačí vložit zkopírovaný vyexportovaný text do příslušného editačního okénka a kliknout na tlačítko *start* s modrou šipkou na vrchní části aplikace vykreslující obrazec. On-line editor želví grafiky následně vykreslí obrazce totožné k obrazcům vykresleným prostředím Snap – ostatně porovnejme obr. 4.5.17 s dalším obr. 4.5.18.

Obrázek 4.5.18 Převedený scénář z Snap vložený a vykonaný v želvím editoru grafiky vykreslující identický obrazec

Jelikož jsou obrazce identické, převod scénáře do jiného textově založeného jazyka byl úspěšně dokončen. Nezapomeňme ještě na druhý příklad vykreslující složitější spirálu příkazem `[nakresli spirálu]`, kterou vykreslí totožně obě programovací prostředí.

Kapitola 5
Závěr bakalářské práce

Přínosem bakalářské práce je představení prostředí Snap všem uživatelům předchozích prostředí (Scratch 1, Scratch 2, a BYOB), kteří nebyli spokojeni s jejich dosavadními možnostmi a hledali nějakou, k nim vhodnou, alternativu. Čtenář byl seznámen s postupným vývojem všech prostředí, dále mu bylo poskytnuto jejich porovnání na základě zvolených kritérií, a nakonec mu byly představeny další možnosti programování zpřístupněné návrhem prostředí Snap na příkladech a algoritmech za podpory doplňujících, autorem vyvinutých, projektů. Autor též provedl několik aktivit nad rámec zadání této bakalářské práce.

Na konci roku 2013 vystoupil před tehdejšími studenty předmětu Didaktika programovaní (se zkratkou KIN/DPLA), pod vedením vedoucího práce, na Pedagogické fakultě Jihočeské univerzity v Českých Budějovicích, kterým představil prostředí Snap v návaznosti na jimi probírané prostředí Scratch 2. Prezentace je uložena na datovém nosiči CD-ROM pod názvem `prezentace_DPLA.pdf`.

Dále navázal e-mailovou komunikaci s českým překladatelem Snap – panem Michalem Mocem – a urgoval jej o aktualizaci lokalizace a poupravení dropných nesrovnalostí v překladu. Vzhledem k vzniklým okolnostem se však rozhodl nečekat na žádané změny a vzal si již dostupnou českou lokalizaci z prostředí, kterou posléze upravil a hojně rozšířil k obrazu svému. Mimo jiné s ní fotil algoritmy (scénáře) v prostředí. Soubor je též na nosiči CD-ROM pod názvem `lang-cs.js` v adresáři `upravena_cestina`.

Při vývoji projektů v prostředí Snap také narazil na několik nedostatků a zásadních chyb, které hlásil na k tomu určeném repositáři GitHub[1]). Vystupoval pod přezdívkou `@studej` a kromě hlášení chyb se také pokusil navrhnout nějaké změny a vylepšení. Do současnosti např. nahlásil devatenáct programátorských chyb uznaných vývojáři.

Prostředí Snap se, i přes své nedávné oficiální vydání ve verzi 4.0 (červen 2015), stále rozvíjí. Z toho důvodu může dojít k určitým změnám, rozšířením, či zrušení některých funkcionalit. Přesto lze na tuto práci navázat jinou prací, zvláště pak vezmeme-li v potaz již započatý vývoj spolu s oznámenými rozšířeními další verze Snap 4.1. Tato práce tak může být pomyslným mostem k snadnějšímu přechodu z prostředí Scratch 2 do prostředí Snap a ona navazující práce by se již mohla soustředit na složitější – dosud nepopsané – oblasti programování, koncepty anebo techniky; přidané verzí 4.1.

[1]) https://github.com/jmoenig/Snap--Build-Your-Own-Blocks

Dodatek A
Základní informace k vlastním blokům

A.1 Tvorba a editace, přidání nápisů a parametrů

• Tvorba bloku a přesun do jeho editoru

Tvorba *vlastního bloku* v prostředí Snap je trochu odlišná od prostředí Scratch 2. Přesto však vychází ze svého předchůdce BYOB. Tvorba *vlastního bloku* se provádí přes dialogové okno, které můžeme vyvolat dvěma způsoby:

- tlačítkem „*Vytvoř blok*“ vespod kategorie `proměnné`
- zvolením možnosti „*vytvořit blok...*“ z nabídky vyvolané kliknutím pravého tlačítka myši kdekoliv v oblasti scénářů postavy či scény.

Dialogové okno „*Vytvoř blok*“, jež je zobrazeno na levé straně obrázku A.1.1, umožňuje nastavit základní údaje bloku. Volíme v něm *kategorii* bloku, jeho *nápisy* (popř. *parametry*), *typ* a také zdali bude k dispozici *všem* či *pouze aktuální* postavě (příp. scéně).

Obrázek A.1.1 Okno základ. nastavení (vlevo) a okno editoru (vpravo) bloku

Po odsouhlasení nastavení kliknutím na tlačítko „*OK*“ se toto okno zavře a otevře jiné („*Editor bloku*“), jenž slouží k editaci bloku. Do tohoto okna lze přetahovat různé bloky odkudkoliv z prostředí – dokonce i z jednoho editoru bloků do druhého. *Editor bloku* mimochodem vyvoláme zvolením možnosti „*upravit...*“ v nabídce vyvolanou kliknutím pravého tlačítka myši na *vlastní blok* (i kdybychom na něj klikli v paletě bloků).

Hlavní rozdíl oproti Scratch 2 spočívá ve schované definici bloku, která není součástí scénářů postavy – a tak ani nepřekáží mezi ostatními scénáři. Navíc – jelikož můžeme vytvořit *vlastní blok* i *pro všechny postavy*, pak je logické, že definice takovéhoto bloku musí být uvedena někde mimo oblast scénářů, tedy v editoru bloku.

Hlavička *vlastního bloku* v *editoru* přebírá barvu dle zvolené kategorie a kopíruje tvar typu bloku. Podíváme-li se opět na obrázek A.1.1, pak v editoru bloku vidíme hlavičku tmavě červenou spadající do kategorie `seznamy` a tvaru příkazu – přesně tak, jak bylo zvoleno v prvním okně umístěném na levé straně obrázku. *Parametry* jsou vždy oranžové a zakulacené. Scénář navazuje na hlavičku stejně jako ve Scratch 2.

• Rychlejší způsob přidání parametrů

Čtenáře určitě zajímá, co znamená zápis `%parametr1` na obr. A.1.1. Abychom nemuseli parametry přidávat ručně, resp. známe-li názvy a počet potřebných parametrů už při

tvorbě bloku ještě před vstupem do jeho editoru, můžeme je vytvořit rychlejší formou v kombinaci se znakem procenta (%). Takto vytvořenému parametru se vždy automaticky přidělý typ *libovolný* a proto potřebujeme-li mu určit typ jiný, musíme na jeho název v *hlavičce* vlastního bloku kliknout a přes *editor parametru* vybrat odlišný typ.

- **Přidání nápisu či parametru**

Přidat nápis či parametr můžeme i přímo v editoru bloku a to kliknutím na znaménko plus (+) mezi jednotlivými nápisy a parametrů. Navíc lze přidat další nápis či parametr před či za jiný nápis anebo parametr, neboť se znaménko plus objevuje z obou stran. Po kliknutí na něj se otevře další dialogové okno, ve kterém si kromě názvu vybereme mezi variantami „*Nápis*" a „*Parametr*". V podstatě jde o editor nápisu či parametru.

Zvolením možnosti „*Nápis*" se vpravo objeví nabídku rozbalující černá šipka, skrze kterou přidáme nápisu nějakou – prostředím rozeznávanou – doplňující ikonku. V případě volby možnosti „*Parametr*" se černá šipka přemístí a kliknutím na ni okno rozšíříme. Objeví se nabízené typy parametrů a nějaká další nastavení (viz obr. 3.1.9).

Obrázek A.1.2 Editor nápisu či parametru a černé šipky s možnostmi

A.2 Změna kategorie a typu, mazání bloku

- **Změna kategorie a typu bloku**

Kliknutím na hlavičku bloku v editoru se vyvolá další nabídka umožňující mu změnit kategorii a typ. Zde je však potřeba zmínit jednu podstatnou informaci. Jestliže se blok již vyskytuje někde ve scénáři, nebude možné jeho typ pozměnit. To proto, že prostředí těžko předpoví, jak zareagovat na změnu jeho tvaru. Programátor se musí ujistit, že se takový blok nevyskytuje **nikde** v oblasti scénářů a pak mu změna typu povolena.

Při vytvoření vlastní funkce či podmínky prostředí automaticky dodá do tělíčka tohoto bloku potřebný příkaz `[vrať]`. Když ale nejprve vytvoříme vlastní blok jako příkaz a poté jeho typ změníme – například na funkci – pak blok `[vrať]` již prostředím nebude automaticky do těla bloku vložen. To proto, že blok už nějaký ten scénář může mít přidělený. V takovém případě si musíme pro příkaz `[vrať]` ručně sáhnout do palety kategorie `ovládání`, který do těla vlastního bloku umístíme.

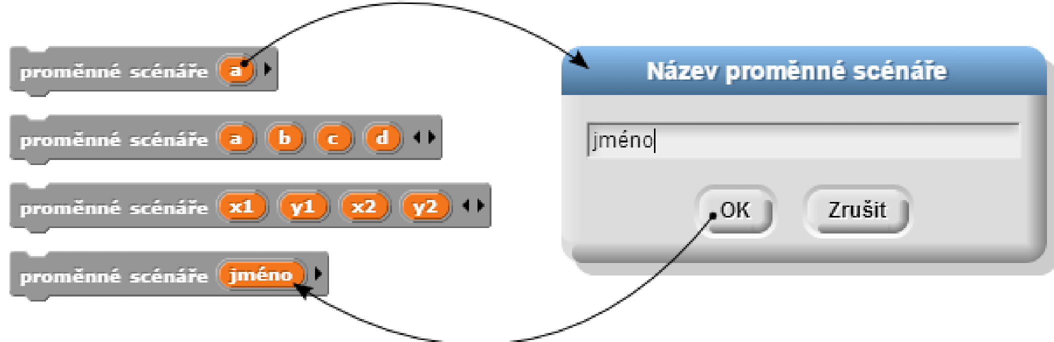
- **Mazání bloků**

Blok smažeme zvolením možnosti „*smazat definici bloku*" přes vyvolanou nabídku kliknutím pravého tlačítka. Nevadí, pakliže se blok vyskytuje ve scénáři – bude z prostředí kompletně odstraněn. Oproti BYOB se nás alespoň Snap zeptá dialogovým oknem, zdali si opravdu přejeme vlastní blok odstranit z projektu.

Dodatek B
O proměnných scénáře

B.1 Vytvoření a přejmenovávání

Proměnné scénáře ke scénářům přiřazujeme příkazem [proměnné scénáře], který má na svém konci černé šipky. Kliknutím na ně přidáváme anebo ubíráme počet proměnných, které chceme vytvořit. Prostředí je automaticky pojmenovává od prvního písmene anglické abecedy. Různé ukázky si prohlédněme na obrázku B.1.1.



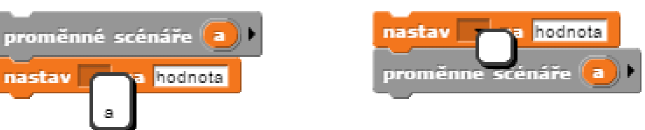
Obrázek B.1.1 Tvorba proměnných scénáře a jejich přejmenování

Stejný obrázek ještě ukazuje způsob, jak proměnnou scénáře přejmenovat a to kliknutím na její název přes příslušné dialogové okno. Jsou to jen *proměnné scénáře*, které lze na rozdíl od proměnných postavy a projektu ve Snap přejmenovat. Řadit jejich pořadí dle libosti není možné a v takovém případě je musíme všechny pracně přejmenovávat.

B.2 O přístupnosti a sledovačích

• Přístupnost k proměnným scénáře

Obrázek B.2.1 ukazuje přístupnost k proměnným scénáře. Scénář vlevo nejprve vytvoří proměnnou (a) přes [proměnné scénáře], čímž se ve výběru následujícímu příkazu [nastav] proměnná (a) zpřístupní v rozbalovací nabídce k nastavení, jelikož jsou oba příkazy ve stejném scénáři. Bez užití příkazu [nastav] by (a) nabývala implicitně {0}.



Obrázek B.2.1 Přístupnost (vlevo) a nepřístupnost k proměnné scénáře

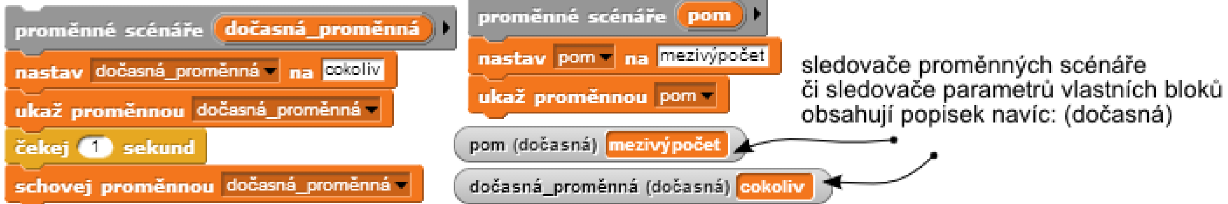
V pravém scénáři téhož obrázku mají stejné příkazy jen prohozené pořadí. Proměnná (a) ještě neexistuje a proto ji příkaz [nastav] „nevidí“ ve své nabídce. To znamená, že proměnné scénáře jsou vždy přístupné navazujícím blokům teprve od místa vzniku.

• Zobrazování a skrývání sledovačů proměnných scénáře (i parametrů)

Sledovač proměnné scénáře je možné zobrazit příkazy [schovej proměnnou] a [ukaž proměnnou]. V ukázce následujícího obrázku B.2.2 jsou vytvořeny proměnné (dočasná proměnná) a (pom). Scénář první proměnné ukáže její sledovač na sekundu a poté jej schová. Druhý scénář proměnnou (pom) také zobrazí, ale o její schování se již nestará.

Jelikož byl do prvního příkladu umístěn příkaz [schovej proměnnou] s možností {dočasná_proměnná}, můžeme její sledovač bez problému skrýt. Jenže u druhého příkladu s (pom) dochází ke ztrátě odkazu na tuto proměnnou. Proto i kdybychom kdekoliv vykonali [schovej proměnnou] s možností {pom}, sledovač této proměnné by se

neskryl – proměnná už totiž neexistuje. Na skrytí všech sledovačů lze však použít nouzovou schopnost příkazu [schovej proměnnou], a to zvolením prázdné možnosti {}.



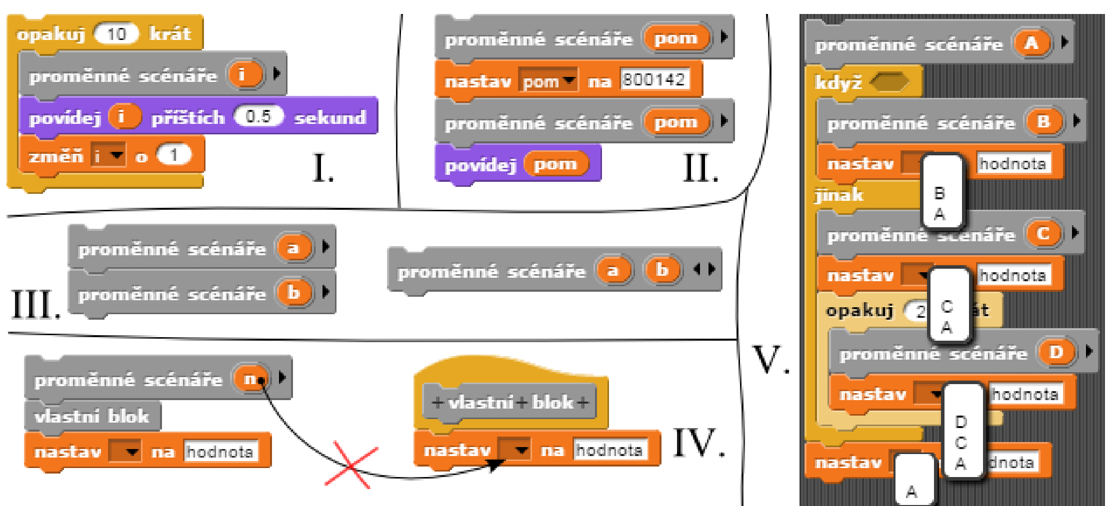
Obrázek B.2.2 Zobrazování a skrytí sledovačů proměnných scénáře

B.3 Procvičující ukázky

Proměnné scénáře bude programátor používat v hojném počtu. Klasicky příklad použití se týká tvorby počítadla pro cyklus. Ve Scratch 2 k tomu vytváříme proměnnou postavy či projektu, kdežto ve Snap těmito proměnnými vše elegantně vyřešíme (viz obr. B.3.1).

Obrázek B.3.1 Pomocné počítadlo (i) ve Scratch 2 (vlevo) a Snap (vpravo)

Další procvičující ukázky s proměnnými scénáře jsou na obrázku B.3.2 s popisem níže.



Obrázek B.3.2 Procvičující ukázky užití bloku [proměnné scénáře]

I. demonstruje chybné užití proměnné scénáře jakožto počítadla pro cyklus [opakuj n krát]. Scénář je špatný, protože se (i) každým průchodem cyklu znovu vytvoří s počáteční (tj. implicitní) hodnotou {0}. Příkazem [změň] ji nemáme šanci změnit na žádanou hodnotu {10}. Řešením je přesunout [proměnné scénáře] nad cyklus.

II. dokazuje, že proměnné se stejným názvem nekolidují (nezpůsobí chybu). Nejprve je (pom) nastavena na {800142} a poté znovu vytvořena s implicit. hodnotou {0}.

III. ukazuje dvě sobě si rovné varianty tvorby proměnných a to *zvlášť* či *najednou*.

IV. proměnná (n) je přístupna jen svému scénáři a tak nepřesahuje do [vlastní blok].

V. zobrazuje rozdíly v přístupu k proměnným vytvořených na odlišných *úrovních*.

Dodatek C
O referencích na data

C.1 Obecný popis (vymezení pojmů)

Reference představuje „odkaz“ anebo „cestičku“ do nějakého místa v paměti počítače, na kterém jsou uloženy informace (data) určitého datového typu. Jedná se samozřejmě o abstraktní pojem, neboť přímo do paměti počítače referencí nemíříme. O to se stará programovací jazyk, který referenci poskytuje – v Snap jde o jazyk JavaScript.

Reference si udržujeme na všechny datové typy kromě těch *primitivních*, mezi něž se řadí *číslo*, *text*, a *pravdivostní hodnota*. Ostatní typy jsou *referenčními*. Liší se v tom, že ty *referenční* jsou oproti natolik primitivním *primitivním* typům složitějšími strukturami skládajíc se z vícero informací (dat), jež nedokážeme vyjádřit jednou hodnotou, a na něž se odkazujeme do paměti počítače *referencemi*. Referenci samozřejmě musíme nejprve nějak získat – a to vždy k tomu určenými funkcemi. Zopakujme si nejenom názvy všech *referenčních* datových typů, ale též funkce vracející reference na jejich data:

- `seznam` – seznam vytváříme konstruující funkcí `(nový seznam)`
- `kostým` – funkcí `(šatník)` získáme referenci na seznam, jehož prvky jsou *kostýmy*
- `zvuk` – funkcí `(jukebox)` získáme referenci na seznam, jehož prvky jsou *zvuky*
- `postava` – funkcí `(postava)` získáme referenci na konkrétní *postavu*, *scénu*, či na seznam se *všemi postavami*, jehož prvky jsou *postavami*.
- `bezejmenný příkaz` – zaobalující funkcí získáme referenci na bezejmennou proceduru tvořenou příkazem či složenými příkazy (bloky příkazů)
- `bezejmenná funkce` – zaobalující funkcí získáme referenci na bezejmennou proceduru tvořenou funkcí či vnořenými funkcemi (bloky funkcí)
- `bezejmenná podmínka` – zaobalující funkcí získáme referenci na bezejmennou proceduru tvořenou podmínkou či vnořenými podmínkami (bloky podmínek)

Ujasněme si jednu věc. Není možné tvrdit, že si uchováváme například referenci seznamu. Seznam přeci žádnou referenci nevlastní. Ten si pamatuje jen počet jeho prvků a jejich hodnoty. Reference je tak vždycky **na** nějaká data. Proto udržujeme referenci na seznam, na kostým, na bezejmennou proceduru, a tak dále. Avšak nikdy ne referenci zvuku, referenci postavy, referenci seznamu atd.

Vzpomeňme též na pojem *prvotřídnost* u datových typů. I reference jsou prvotřídními. Proto je možné je zmíněnými funkcemi vytvořit bezejmenně kdekoliv ve scénáři, uchovat je v proměnných či jakožto prvek seznamu, předat je jako argument nějakému parametru bloku, či je vrátit jako výsledek nějaké funkce. Též je vhodné zmínit, že pokud si získanou referenci rozhodneme neuchovat, pak o ni nadobro přijdeme. Je tedy přímo na nás, jak se získanou referencí naložíme.

C.2 Uchování a sdílení reference

Vysvětleme si rozdíl mezi ukládáním *primitivních* a *referenčních* datových typů. Na následujícím obrázku C.2.1 jsou vytvořené dvě proměnné scénáře `(číslo1)` a `(číslo2)`. Sledovače proměnných na pravé straně ukazují jejich stavy (hodnoty) s každým dalším provedeným příkazem scénáře. V době svého vytvoření tyto proměnné scénáře nabývají implicitní hodnoty `{0}`. Nadcházející příkaz `[nastav]` upraví hodnotu `(číslo1)` na hodnotu `{5}`. Poté `(číslo2)` nastavíme na hodnotu `(číslo1)` – také na `{5}`, neboť této hodnoty nabývá právě `(číslo1)`. Nakonec upravíme hodnotu `(číslo1)` na `{99}`.

Z uvedených sledovačů vyčteme, že při převzetí hodnoty z `(číslo1)` do `(číslo2)` hodnota nebyla sdílena. Do `(číslo2)` se sice uložilo stejné číslo (hodnota `{5}`), kteréhož nabývala `(číslo1)`, avšak bylo vytvořeno samostatně. Proto při změně `(číslo1)` z `{5}` na `{99}` k změně hodnoty `(číslo2)` nedošlo. Jedná se o takzvanou *hlubokou kopii*, kdy se v paměti počítače vytvoří data se stejnými hodnotami na rozličných místech.

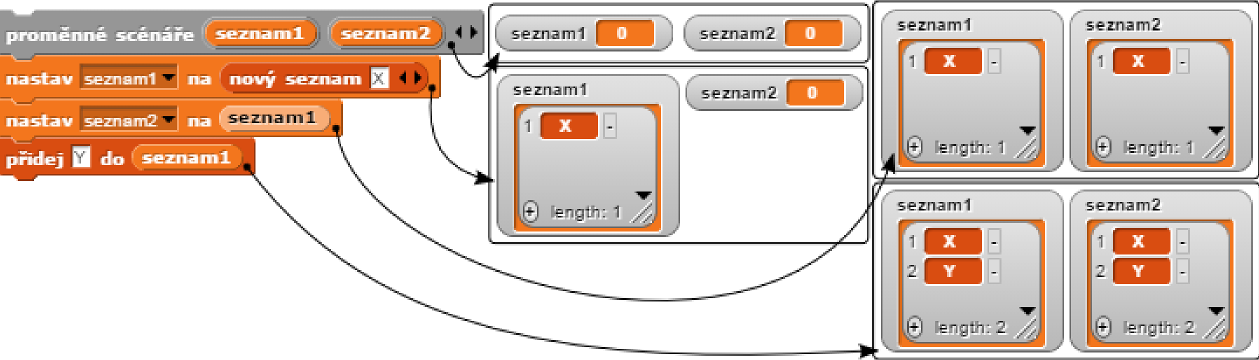
• *Poznámka: Ve sledovačích proměnných na obrázku C.2.1 a taktéž na příštím obrázku C.2.2 by měl být popisek „dočasná“, protože se jedná o proměnné scénáře, jež po ukončení scénáře zaniknou. Kvůli úspoře místa však tento popisek nebyl do obrázků zahrnut.*

Obrázek C.2.1 Uchování a sdílení čísla – primitivního datového typu

U *referenčních* typů však popsaný způsob předávání hodnoty z proměnné do proměnné funguje dosti odlišně. Na obrázku C.2.2 provádíme významem totožný scénář, akorát pracujeme se seznamy (a referencemi na ně) namísto práce s čísly.

Nejprve vytvoříme proměnné (seznam1) a (seznam2) (se vztahem ekvivalentním k předchozím (číslo1) a (číslo2)). Hodnoty těchto proměnných jsou opět implicitní, tedy {0}. Dále vytváříme seznam funkcí (nový seznam), jež vytvoří seznam v paměti počítače, přidá mu první prvek s hodnotou {X} a vrátí na něj referenci, kterou si uchováme do proměnné (seznam1) – jak ostatně ukazuje její sledovač.

Následně nastavujeme proměnnou (seznam2) na hodnotu proměnné (seznam1). Jelikož ale (seznam1) nabývá reference, dojde k jejímu sdílení, a tak (seznam2) bude nabývat stejné hodnoty – reference na totožný seznam s prvkem {X}. Sledovače (seznam1) a (seznam2) jsou tedy totožné, neboť obsahují referenci na stejná data. Vytvořila se tzv. *mělká kopie* – namísto tvorby nového seznamu s totožnými prvky na odlišném místě v paměti a jinou referencí se pouze odkazujeme na již existující seznam.



Obrázek C.2.2 Uchování a sdílení seznamů – referenčního datového typu

Posledním příkazem scénáře je [přidej], jemuž předáváme referenci na seznam uchovanou v (seznam1), do kteréhož se přidá další prvek s hodnotou {Y}. Změna se projeví u sledovačů obou proměnných. Musíme pochopit, že (seznam1) a (seznam2) obsahují totožné reference, resp. míří na stejný seznam v paměti počítače. Kdybychom u příkazu [přidej] namísto (seznam1) použili (seznam2), výsledek by byl úplně stejný.

Kdybychom nyní proměnnou (seznam1) nastavili na jinou hodnotu – třeba {cokoliv}, proměnná (seznam2) by stále hleděla na seznam s prvky {X, Y}. Když jsme nastavovali (seznam2) na hodnotu (seznam1), nedošlo mezi nimi k přímému vztahu ve smyslu „*Jakmile se změní* (seznam1), *já* (seznam2), *se také změním.*“. Proměnná (seznam2) v době nastavení pouze převzala hodnotu (seznam1) – sdílela stejnou referenci na jeden a týž seznam. Co se tedy stane s (seznam1) později už (seznam2) nezajímá.

Pokud bychom chtěli namísto sdílení reference na seznam raději zkopírovat všechny prvky seznamu z (seznam1) do nového seznamu uchovaném v (seznam2) – tedy provést *hlubokou kopii* –, museli bychom doprogramovat vlastní scénář.

Závěrem znovu připomeňme, že ačkoliv jsme dosud hovořili o sdílení referencí pouze mezi proměnnými, lze je sdílet (předávat) i parametrům bloků jakožto argument, vracet je jako výsledek funkce, ukládat do seznamu a tak dále. Reference jsou totiž prvotřídní.

C.3 Porovnání referencí

Zbývá nám probrat porovnávání referencí. Výjimečně musíme ověřit, zdali dvě reference (uložené třeba v proměnných) míří na stejná data do paměti počítače. K porovnávání referenci nepoužíváme podmínku <je rovno>, nýbrž podmínku <je totožný k>. Připomeňme, že na zjištění datového typu – tj. hodnoty, na niž míříme referencí do paměti počítače – máme podmínku <je typu>. Zkusme si nyní porovnat reference na *seznam* a *zaobalenou proceduru* vytvořenou zaobalující funkcí .

Obrázek C.3.1 obsahuje proměnné (seznam1) a (seznam2). Jak jsme si již popsali v předchozí oblasti o sdílení referencí, (seznam2) sdílí referenci na prázdný seznam vytvoření funkcí (nový seznam) uchovanou v (seznam1). Na pravé straně obrázku pak provádíme dvě porovnání referencí podmínkou <je totožný k>.

První porovnání (nový seznam) s (nový seznam) vrací {<nepravda>}, protože si reference nejsou rovny. To proto, že vytváříme bezejmenně dva odlišné seznamy, které jsou v paměti počítače uloženy na různých místech. Když se pak prostředí podmínkou <je totožný k> optá jazyka JavaScript, jestli míří obě získané reference na stejné místo do paměti, dostane negativní odpověď. Zato druhý případ s (seznam1) a (seznam2) vrací {<pravda>}, neboť porovnáváme jednu a tutéž referenci uloženou (sdílenou) dvěma proměnnými.

Obrázek C.3.1 Ukázka porovnávání referencí podmínkou <je totožný k>

To samé platí i v druhém příkladě níže se zaobalenou procedurou. Pokud se snažíme porovnat dvě reference a , které míří na odlišná místa do paměti – neboť se obě zaobalené procedury vytvářely na sobě nezávisle – vrátí se výsledek {<nepravda>}. Kdežto porovnáme-li (procedura2) sdílející převzatou referenci na zaobalenou proceduru získanou funkcí uchovanou v proměnné (procedura1), pak <je totožný k> vrací {<pravda>} při porovnání (procedura1) s (procedura2).

Zdůrazněme však, že podmínku <je rovno> má smysl v souvislosti s referencemi a referenčními datovými typy stále používat. Chceme-li porovnat totožnost třeba seznamů, již víme, že použijeme <je totožný k>. Máme-li však jistotu – a nebo pokud nás nezajímá –, že oba porovnávané seznamy nejsou uloženy na totožném místě v paměti počítače (reference k nim jsou rozdílné), pak velmi často porovnáváme shodnost jejich prvků, což provedeme právě podmínkou <je rovno>. Ta nejprve porovná velikost seznamů, a pokud mají stejný počet prvků, pak ještě porovná každý prvek prvního seznamu s prvky seznamu druhého.

Dodatek D
Důležité informace pro práci se seznamy

D.1 Operace nad seznamy

Ačkoliv se může na zdát, že provádění operací nad seznamy v prostředí Snap neobsahuje navzdory odlišnostem od obou verzí Scratch žádná úskalí, v žádném případě tomu tak není. Bezchybnému provádění operací nad seznamy se věnuje tato část.

- **Přidávání prvků do seznamu**

Do již vytvořeného seznamu přidáváme další prvky buď příkazem `[přidej]` nebo funkcí `(vlož do popředí)`. Zaměřme se nyní na příkaz `[přidej]` a zvažme příklad, kdy potřebujeme přidat hodnotu do seznamu vytvořeného pouze v rámci scénáře. Obrázek D.1.1 na levé straně ukazuje neplatné řešení, při kterém se snažíme přidat hodnotu `{cokoliv}` do vytvořené proměnné scénáře `(seznam)` příkazem `[přidej]`.

Vytvořená proměnná `(seznam)` nabývá implicitní hodnoty `{0}` a jakmile dojde na příkaz `[přidej]`, bude prostředí přikázáno vložit hodnotu `{cokoliv}` do hodnoty `{0}`, což je nesmysl. Jelikož `[přidej]` nedostal referenci na seznam, prostředí zahlásí chybu.

Proto nejprve musíme do proměnné `(seznam)` vložit referenci na seznam – například jeho novým vytvořením přes příkaz `[nastav]` – a až poté je možné přidat hodnotu příkazem `[přidej]`. Správné řešení je zobrazeno na pravé straně téhož obrázku.

Obrázek D.1.1 Vyžadování reference na seznam v proměnné při přidávání

Čtenáře mohlo zarazit, proč jsme nevyužili vícenásobný parametr (černé šipky) funkce `(nový seznam)` a hodnotu `{cokoliv}` nepřidali při jeho tvorbě v souvislosti s příkazem `[nastav]`. To je funkční řešení, ale ne vždy známe přidávanou hodnotu předem a příkaz `[přidej]` se často vyskytne až někde dále ve scénáři až po vytvoření seznamu.

Zmínili jsme-li vícenásobný parametr, pak si pojďme ukázat další – z hlediska programátorských zásad – špatný způsob plnění seznamu. Cílem budiž tvorba seznamu s několika náhodnými čísly. Obrázek D.1.2 zobrazuje odstrašující řešení.

Obrázek D.1.2 Odstrašující užití vícenásobného parametru `(nový seznam)`

Na první pohled se zdá, že jsme si využitím vícenásobného parametru ušetřili práci. Netřeba užít příkaz `[přidej]` několikrát po sobě k přidání čtyř náhodných čísel. Reference na vytvořený seznam byla uložena opět do `(seznam)`. Toto řešení je však špatné ze dvou důvodů: 1) chceme-li vytvořit seznam ne se čtyřmi, ale třeba s desíti náhodnými čísly, pak musíme tolikrát vložit do funkce `(nový seznam)` funkci `(zvol náhodné číslo)`; 2) pokud potřebujeme změnit rozsah generovaného čísla, pak ho musíme upravit u každé užité funkce `(zvol náhodné číslo)` zvlášť. Toto odstrašující řešení si ukazujeme, jelikož k němu může čtenář inklinovat ve snaze vyhnout se příkazu `[přidej]`.

Vícenásobný vstup (černé šipky) funkce `(nový seznam)` nemusíme upřednostňovat nad příkazem `[přidej]` pokaždé. Proto raději `[přidej]` použijme v tomto ukázkovém příkladu. Z programátorského hlediska správnější scénář je uveden na obrázku D.1.3.

Obrázek D.1.3 Naplnění seznamu čtyřmi náhodnými čísly přes [přidej]

- **Odstraňování všech prvků ze seznamu**

Myslí se přesuňme od přidávání prvků k jejich mazání. Protentokrát máme v proměnné (seznam) referenci na seznam s n prvky, které potřebujeme do jednoho odstranit. Na levé straně obrázku D.1.4 je uveden klasický způsob jak smazat všechny prvky seznamu příkazem [zruš] se zvolenou možností {všechno}. Seznam pak bude vyprázdněn.

Obrázek D.1.4 Dva způsoby odstraňování všech prvků seznamu

Zato na pravé straně téhož obrázku je do proměnné (seznam) uložena reference na nově vytvořený seznam funkcí (nový seznam), což znamená, že reference na původní seznam byla nahrazena referencí jinou. Došlo tak ke ztrátě seznamu předešlého a (seznam) nyní obsahuje referenci na nový – prázdný – seznam. Teoreticky lze tento způsob chápat jako techniku k odstranění všech prvků, neboť vyměňujeme naplněný seznam za nový a úplně prázdný. Přesto bez ohledu na použitý způsob – buď příkazem [zruš] anebo [nastav] v kombinaci s (nový seznam) – bude vždy seznam bez jediného prvku.

Nyní nám může vyvstávat otázka, k čemu vlastně druhý uvedený způsob zaměňující původní seznam za nový a prázdný na obrázku D.1.4 využijeme. V některých případech potřebujeme všechny prvky seznamu odstranit a ihned nato či o chvíli později přidat další prvky do zrovna vyprázdněného seznamu. Podívejme se na levou stranu obrázku D.1.5, kde mažeme všechny prvky seznamu příkazem [zruš] a ihned za ním přidáváme čtyři další prvky s hodnotami proměnných (a), (b), (c), a (d). Tento způsob vyžaduje příkaz [přidej] čtyřikrát – a co teprve kdybychom potřebovali přidat prvků ještě více.

Obrázek D.1.5 Dva způsoby odstranění prvků seznamu a následnému plnění

Zde se právě vyplatí referenci na seznam původní s prvky, kterých se potřebujeme zbavit, nahradit nově vytvořeným seznamem funkcí (nový seznam), jež rovnou naplníme za pomocí vícenásobného parametru (s využitím černých šipek). Celý proces odstranění všech prvků a přidání čtyřech nových tak zvládneme místo pěti příkazy pouze jedním.

Navzdory samé chvále však užití tohoto alternativního způsobu záměny naplněného seznamu za nový a prázdný skýtá jednu záludnost představující značné riziko. Obrázek D.1.6 uchovává referenci na seznam do proměnné (seznam1). Dále (seznam2) přebírá referenci uchovanou v (seznam1). Reference obou proměnných jsou znázorněny vpravo.

Obrázek D.1.6 Znázornění referencí (seznam1) a (seznam2) na seznam

Zkusíme-li ale nyní zaměnit seznam původní za nový již uvedeným alternativním způsobem u (seznam1), v paměti počítače zůstane stále seznam původní, na něhož hledí (seznam2), a tak namísto jedno seznamu budeme mít v projektu dva – jeden prázdný a druhý s třemi prvky {1, 2, 3}, což jasně ukazuje obrázek D.1.7. Sice by nebyl problém nastavit opět (seznam2) na stejnou referenci mířící na nově vytvořený prázdný seznam v (seznam1), ale jak víme, kde všude byla reference uchována – třeba neplánovaně vlastním blokem do proměnné všech postav, na což se velmi snadno opomene při mazání tímto alternativním způsobem.

Obrázek D.1.7 Znázornění referencí na dva seznamy při špatném mazání

Ponaučením budiž, že alternativní způsob použijeme jenom ve chvíli, kdy jsme si jisti, že neexistuje více referencí na zaměňovaný seznam. Bude-li existovat pouze (seznam1), pak je záměna bezpečná. Existuje-li ale více proměnných či parametrů vlastních bloků hledících na stejný seznam, bude lepší se z důvodu bezpečnosti uchýlit k použití standardního způsobu vyprazdňování seznamu, kterým je užití příkazu [zruš] s možností {všechno}, jak ostatně ukazuje obrázek D.1.8.

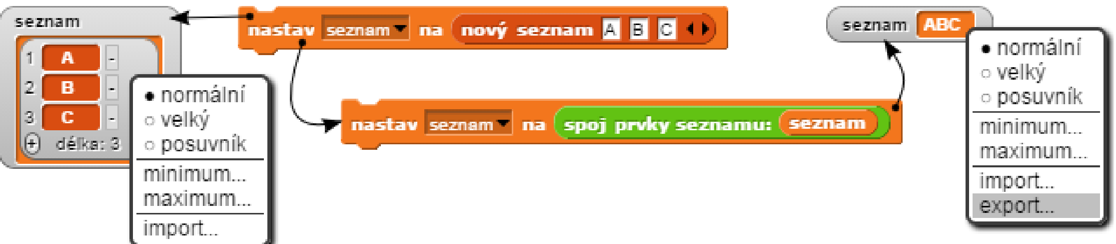
Obrázek D.1.8 Jistý způsob odstranění všech prvků přes [zruš]

D.2 Export a import seznamů

Zastavme se ještě u importování a exportování seznamů v prostředí Snap. Výklad však bude srozumitelnější, začneme-li nejprve s exportem seznamů a poté až s importem.

Exportovat obsah proměnné můžeme volbou možnosti „*export...*“ z nabídky sledovače zobrazenou kliknutím pravého tlačítka na jeho plochu. Exportovat lze všechny datové typy kromě reference na seznam. Jelikož tedy seznamy nemůžeme exportovat, musíme se postarat o převod jejich prvků – a to do textové podoby, kterou uložíme do proměnné a jejíž obsah poté exportujeme ze sledovače.

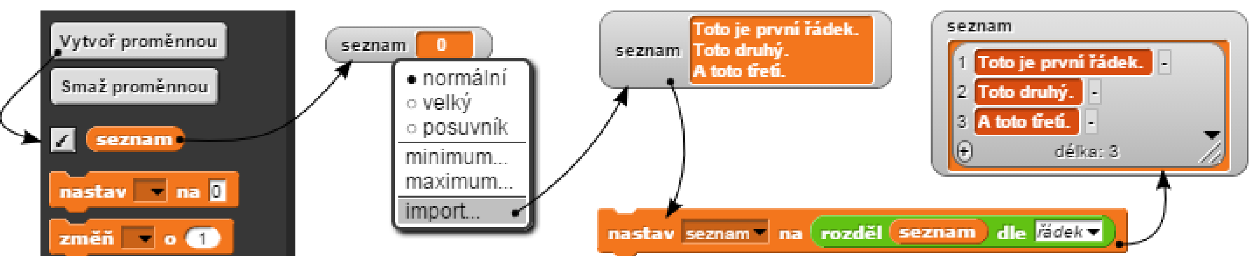
Následující obrázek D.2.1 ukazuje zjednodušeně jak exportovat seznam s prvky {A, B, C} a s využitím primitivní funkce (spoj), které jsme na černé šipky upustili proměnnou (seznam). Jejímu vícenásobnému parametru se tak předala reference na seznam a došlo ke spojení všech tří prvků z {A, B, C} na {ABC} (viz sledovač). Poté bylo možné seznam převedený do textu vyexportovat z proměnné. Prostředí automaticky otevře novou záložku internetového prohlížeče, ze kterého je možné vyexportovaný obsah zkopírovat, popřípadě uložit celý soubor přes rozhraní prohlížeče.



Obrázek D.2.1 Převod seznamu na text a jeho export ze sledovače

Když už víme o způsobu umožňující export seznamů, ukažme si jak je importovat. V prostředí Scratch 2 importujeme textové soubory do proměnné opět přes její sledovač. Scratch 2 rozdělí obsah textového souboru dle řádku a každý takový řádek se stane prvkem seznamu. Jak ale víme z předchozí části, export seznamu v Snap závisí pouze na nás a ne vždy ukládáme prvek na jeden řádek. Prvky při exportu můžeme například oddělit nějakým znakem a umístit je za sebou. Proto i při importu seznamů se musíme postarat v prostředí Snap o zpětné rozdělení celého importovaného textu na části.

Podívejme se na poslední obrázek D.2.2, na kterém je proměnná (seznam). Při jejím vytvoření získá implicitní hodnotu {0}. Kliknutím pravého tlačítka myši na sledovač této proměnné lze vybrat z nabídky možnost „*import...*“, která otevře dialogové okno k výběru textového souboru z našeho počítače. Do proměnné se obsah textového souboru však načte jako text a proto jej musíme rozdělit k tomu určeným primitivem (rozděl). Samozřejmě záleží na způsobu uchování exportovaného seznamu, který jsme importovali do proměnné. Víme-li například, že byl každý prvek seznamu „zakódován“ na jednom řádku v textovém souboru, pak musíme zvolit u primitivní funkce (rozděl) možnost {řádek}. Funkce vrátí obsah v seznamu – tj. řádky z textového souboru.



Obrázek D.2.2 Import textu či „zakódovaného“ seznamu a jejich „dekódování“ přes funkci (rozděl)

U obrázku D.2.1, v němž jsme „zakódovali“ prvky seznamu {A, B, C} na {ABC}, „dekódování“ provedeme tak, že u funkce (rozděl) nevybereme žádnou z možností (předáme za argument prázdný text {}). V tomto případě funkce rozdělí dodaný text {ABC} písmeno po písmeni a vrátí seznam v námi požadované původní podobě {A, B, C}.

Dodatek E

Přehled přidaných bloků k sadě ze Scratch 1

Abychom si udělali představu o – nejenom celkovém počtu – nabízených primitivů prostředí, ukažme si je všechny v přehledech. Předpokládá se, že čtenář zná všechny primitivy z prostředí Scratch 1, na něž přehledy navazují. Přidané primitivy jsou označeny znaménkem plus (+) a odebrané[1]) znaménkem mínus (-).

- **Rozdíl v nabízených primitivech BYOB vůči Scratch 1**

Následující přehled zobrazuje přidané primitivy v BYOB (+22) vůči prostředí Scratch 1:

+ [spusť]
+ (zavolej)
+ [zahaj]
+ [vrať]
+ [zastav blok]
+ [započni ladění]
+ (započni ladění)
+ (postava)
+ (atribut)
+ <pravda>
+ <nepravda>
+ (ascii)
+ (ascii jako znak)
+ <je typu>
+ [scénář]
+ (výraz)
+ (klonuj)
+ [smaž]
+ [proměnné scénáře]
+ (nový seznam)
+ (jako text)
+ (kopíruj)

- **Rozdíl v Scratch 2 vůči Scratch 1**

Další přehled zobrazuje rozdíl v nabízených primitivech Scratch 2 (+14/-3) k Scratch 1:

+ [nastav způsob otáčení]
+ [změň pozadí]
+ [po změně pozadí]
+ [když větší než]
+ [zastav]
+ [když startuji jako klon]
+ [klonuj]
+ [zruš tento klon]
+ (video)
+ [přepni video]
+ [nastav průhlednost videa]
+ (aktuální)
+ (dnů od roku 2000)
+ (jméno uživatele)
- [zastav scénář]
- [zastav vše]
- <hlasitý>

- **Rozdíl v Snap vůči BYOB**

Poslední přehled zobrazuje rozdíly primitivů Snap (+26/-8) vůči primitivům BYOB:

+ (poslední zpráva)
+ [když startuji jako klon]
+ [klonuj]
+ [zruš tento klon]
+ [zastav]
+ [zastav]
+ [pozastav vše]
+ (http://)
+ <turbo mód>
+ [nastav turbo mód]
+ (aktuální)
+
+
+
+ (rozděl)
+ (unicode)
+ (unicode jako znak)
+ <je totožný k>
+ (JavaScript funkce)
+ (vlož do popředí)
+ (vše až na první prvek)
+ [bez obnovy obrazovky]
+ [převeď blok]
+ [převeď část]
+ [převeď String]
+ (kód z)
- [zastav scénář]
- [zastav vše]
- [zastav blok]
- <hlasitý>
- (ascii)
- (ascii jako znak)
- [scénář]
- (výraz)
- (jako text)
- (kopíruj)

Dodejme, že poslední přehled o primitivech prostředí Snap:

- nezahrnuje bloky z vývojového módu, neboť nejsou určeny k běžnému užití
- nezapočítává bloky [započni ladění], (započni ladění), (postava), (atribut), (klonuj) a [smaž]; jenž se objeví až ve verzi 4.1 (již započítány u BYOB)
- vynechává další plánované bloky, jejichž implementace je nejistá či v nedohlednu
- příkaz [zastav] je **správně** uveden v přehledu dvakrát

- *Poznámka: Prostředí Snap vůči prostředí Scratch 1 přidává zatím 34 primitivů. Přesto však některými primitivy z prostředí Scratch 2 prostředí Snap nedisponuje.*

[1]) Současná prostředí Scratch 2 a Snap totiž odebírají z různých důvodů (převážně kvůli lepšímu řešení) některé původní primitivy svých předchozích verzí. Uvádíme je proto, abychom je v prostředích nehledali.

Dodatek F

Upravená čeština a její instalace do Snap

Původní překlad prostředí Snap do češtiny provedl Michal Moc, kterémuž dík za velký kus dobře odvedené práce. Přesto autor této práce chtěl mít některé oblasti odlišně přeložené, neboť nabyl dojmu, že mohou být výřečnější. Proto sám tuto oficiální češtinu poupravil. Změnil několik překladů s cílem sjednotit identické popisky Snap a Scratch 1. Následně doplnil chybějící překlad dalším blokům a různým nápisům grafického uživatelského rozhraní. Ne všechny popisky uživ. rozhraní jsou však přeloženy.

Upravená čeština není oficiálním překladem, ani podmínkou k prohlížení projektů. Pokud však chceme vidět scénáře ve stejné podobě, v jaké je vidíme na obrázcích této práce, je třeba si upravenou češtinu do prostředí nainstalovat. Způsob instalace se liší podle toho, jestli chceme pracovat v on-line či off-line verzi. Závěrem ještě upozorněme, že upravená čeština s dalším vývojem prostředí nebude časem aktuální.

F.1 Instalace upravené češtiny do on-line verze

Má-li čtenář zájem prohlížet si sdílené projekty na účtu autora této práce, bude muset nejprve upravenou češtinu speciálním způsobem do on-line verze prostředí Snap doinstalovat a poté ručně otevírat projekty. Postupujme následujícím způsobem:

1) otevřeme on-line prostředí Snap a přepněme jej do anglického jazyka
2) přes nabídku vyvolanou kliknutím na tlačítko s ikonkou mráčku v kombinaci s klávesou `Shift` vyberme možnost „*open shared project from cloud...*“ (viz obr. F.1.1)
3) za název uživatelského účtu autora projektu doplňme `bpitvpfjcu2014` a potvrďme
4) za název projektu doplňme `UPRAVENA_CESTINA` a potvrďme
5) jakmile se projekt načte, spusťme projekt kliknutím na tlačítko se zeleným praporkem či ručně vykonejme vlastní blok `[nainstaluj češtinu]`.

Obrázek F.1.1 Otevírání sdílených projektů v on-line verzi prostředí Snap

F.2 Instalace upravené češtiny do off-line verze

Ti, jenž nemají přístup k internetu a rádi by projekty s upravenou češtinou prohlíželi, si musí opatřit prostředí Snap v off-line verzi. Prostředí používá pro každý jazyk vlastní lokalizační soubor a český překlad definuje soubor `lang-cs.js`, který čtenář nalezne na přiloženém CD-ROM v adresáři `upravena_cestina`. Postup instalace je následující:

1) stáhněme zdrojové soubory z on-line verze prostředí Snap výběrem možnosti menu zobrazeného kliknutím na logo Snap vlevo nahoře
2) stažený archivovaný soubor rozbalme do libovolného adresáře počítače
3) lokalizační soubor `lang-cs.js` z CD-ROM přesuňme do adresáře s právě rozbaleným obsahem zdrojových souborů Snap a přepišme jím soubor původní
4) spusťme prostředí otevřením souboru `snap.html` ze stejného adresáře v libovolném webovém prohlížeči. Případně přepněme na češtinu není-li zvolena automaticky.

Dodatek G

Přehled projektů s odkazy do prostředí

Pro čtenáře tištěné verze je nutné uvést internetové odkazy na všechny doplňující projekty, které opíše a doplní správné číslo a nebo název vybraného projektu dle prostředí.

Má-li čtenář nainstalovanou upravenou češtinu do on-line verze prostředí Snap, musí otevírat projekty způsobem uvedeným v předchozím dodatku F na obrázku F.1.1. Také ať otevírá projekty tohoto prostředí ve webovém prohlížeči Google Chrome. Pokud se však některé projekty z prostředí Snap přes internetovou adresu v prohlížeči Google Chrome neotevřou, pak je musí otevřít manuálně přes způsob z obrázku F.1.1.

• *Poznámka: Všechny projekty se Snap ve spouští se zapnutým prezentačním módem. Některé z nich však nic neukazují a obsahují třeba jen ukázkové bloky. A tak má-li projekt schované pozadí scény, nechť uživatel nečeká a zamíří rovnou do scénářů postav.*

• Předloha adresy k projektům z prostředí Scratch 2 a jejich číselný kód

`http://scratch.mit.edu/projects/SEM_ZADEJTE_ČÍSLO_PROJEKTU/`

```
#1 = 38621740
#7 = 21953974
#12 = 21214693
#14 = 34506628
#19 = 20641706
```

• Předloha adresy k projektům z prostředí Snap

`http://snap.berkeley.edu/snapsource/snap.html`
`#present:Username=bpitvpfjcu2014&ProjectName=SEM_ZADEJTE_NÁZEV_PROJEKTU`

Za název projektu dosaďme vždy slovo `PROJEKT` za něž vložme pořadové číslo projektu. Například projekt číslo `#4` bude mít v prostředí název `PROJEKT4`.

Reference

- **Elektronické knihy, sborníky, články a akademické publikace**

[1] BRENNAN, Karen, Andrés MONROY-HERNÁNDEZ a Mitchel RESNICK. *Making projects, making friends: Online community as catalyst for interactive media creation* [online]. 2010 [cit. 2015-06-22]. Dostupné z: http://web.media.mit.edu/~mres/papers/NDYD-final.pdf

[2] BRENNAN, Karen a Mitchel RESNICK. *New frameworks for studying and assessing the development of computational thinking* [online]. Vancouver, Canada: AERA, 2012 [cit. 2015-06-20]. Dostupné z: http://web.media.mit.edu/~kbrennan/files/Brennan_Resnick_AERA2012_CT.pdf

[3] HARVEY, Brian a Jens MÖNIG. *Bringing "No Ceiling" to Scratch: an One Language Serve Kids and Computer Scientists?* [online]. Paris: Constructionism, 2010 [cit. 2015-06-20]. Dostupné z: https://www.cs.berkeley.edu/~bh/BYOB.pdf

[4] HARVEY, Brian a Jens MÖNIG. *BYOB Reference Manual: Version 3.1 DRAFT* [online]. 2011 [cit. 2015-06-20]. Dostupné z: http://snap.berkeley.edu/BYOBManual.pdf

[5] HARVEY, Brian a Jens MÖNIG. *Snap! Reference Manual: Version 4.0* [online]. 2014 [cit. 2015-06-20]. Dostupné z: http://snap.berkeley.edu/SnapManual.pdf

[6] HARVEY, Brian. *The Beauty and Joy of Computing: Computer Science for Everyone* [online]. Athens, Greece: Constructionism, 2012 [cit. 2015-06-20]. ISBN 978-960-88298-4-8. Dostupné z: https://www.cs.berkeley.edu/~bh/BJC.pdf

[7] MALAN, David J. a Henry H. LEITNER. *Scratch for Budding Computer Scientists* [online]. 2007 [cit. 2015-06-20]. Dostupné z: http://cs.harvard.edu/malan/publications/fp079-malan.pdf

[8] MALONEY, John, Kylie PEPPLER, Yasmin B. KAFAI, Mitchel RESNICK a Natalie RUSK. *Programming by Choice: Urban Youth Learning Programming with Scratch* [online]. Portland: SIGCSE conference, 2008 [cit. 2015-06-20]. Dostupné z: http://web.media.mit.edu/~mres/papers/sigcse-08.pdf

[9] MALONEY, John, Mitchel RESNICK, Natalie RUSK, Brian SILVERMAN a Evelyn EASTMOND. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education* [online]. 2010, 10(4): 1-15 [cit. 2015-06-20]. DOI: 10.1145/1868358.1868363. ISSN 19466226. Dostupné z: http://web.media.mit.edu/~jmaloney/papers/ScratchLangAndEnvironment.pdf

[10] MÖNIG, Jens a Brian HARVEY. *Snap! Connectivity Strategy* [online]. 2012 [cit. 2015-06-20]. Dostupné z: http://snap.berkeley.edu/snapsource/Snap!%20Connectivity%20Strategy.pdf

[11] RESNICK, Mitchel. *All I Really Need to Know (About Creative Thinking) I Learned (By Studying How Children Learn) in Kindergarten* [online]. Washington DC: ACM Creativity & Cognition conference, 2007 [cit. 2015-06-20]. Dostupné z: http://web.media.mit.edu/~mres/papers/CC2007-handout.pdf

[12] RESNICK, Mitchel, Brian SILVERMAN, Yasmin KAFAI, John MALONEY, Andrés MONROY-HERNÁNDEZ, Natalie RUSK, Evelyn EASTMOND, Karen BRENNAN, Amon MILLNER, et al. Scratch: Programming for all. *Communications of the ACM* [online]. 2009, 52(11): 60- [cit. 2015-06-20]. DOI: 10.1145/1592761.1592779. ISSN 00010782. Dostupné z: http://web.media.mit.edu/~mres/papers/Scratch-CACM-final.pdf

[13] RESNICK, Mitchel a Karen BRENNAN (ed.). *ScratchEd: Working with teachers to develop design-based approaches to the cultivation of computational thinking* [online]. 2010 [cit. 2015-06-20]. Dostupné z: http://web.media.mit.edu/~mres/proposals/NSF-ScratchEd.pdf

[14] RESNICK, Mitchel, John MALONEY a Natalie RUSK. *Scratch 2.0: Cultivating Creativity and Collaboration in the Cloud* [online]. 2010 [cit. 2015-06-22]. Dostupné z: http://web.media.mit.edu/~mres/proposals/Scratch-CreativeIT.pdf

[15] RESNICK, Mitchel. Point of View: Reviving Papert's Dream. *Educational Technology* [online]. 2012, 52(4) [cit. 2015-06-20]. Dostupné z: http://web.media.mit.edu/~mres/papers/educational-technology-2012.pdf

[16] RESNICK, Mitchel, Yasmin KAFAI a John MAEDA. *A Networked, Media-Rich Programming Environment to Enhance Technological Fluency at After-School Centers in Economically-Disadvantaged Communities* [online]. 2003 [cit. 2015-06-22]. Dostupné z: http://web.media.mit.edu/~mres/papers/scratch-proposal.pdf

[17] RUSK, Natalie. *Programming concepts and skills supported in Scratch* [online]. [2008] [cit. 2015-06-20]. Dostupné z: http://scratched.gse.harvard.edu/sites/default/files/scratchprogrammingconcepts-v14.pdf

[18] UTTING, Ian, Stephen COOPER, Michael KÖLLING, John MALONEY a Mitchel RESNICK. Alice, Greenfoot, and Scratch – A Discussion. *ACM Transactions on Computing Education* [online]. 2010, **10**(4): 1-11 [cit. 2015-06-20]. DOI: 10.1145/1868358.1868364. ISSN 19466226. Dostupné z: http://web.media.mit.edu/~jmaloney/papers/AliceGreenfootScratch.pdf

- **Internetové portály, webová sídla, webové stránky, a multimediální obsah**

[19] About. BRENNAN, Karen a OHO INTERACTIVE. *ScratchEd* [online]. 2009 [cit. 2015-06-20]. Dostupné z: http://scratched.gse.harvard.edu/about

[20] Apple and Adobe Flash controversy. In: *Wikipedia: the free encyclopedia* [online]. St. Petersburg (Florida): Wikipedia Foundation, 21. 11. 2012, last modified on 13. 06. 2015 [cit. 2015-06-18]. Dostupné z: http://en.wikipedia.org/wiki/Apple_and_Adobe_Flash_controversy

[21] BLACKMORE, Clinton. *Enchanting* [online]. [2012] [cit. 2015-06-20]. Dostupné z: http://enchanting.robotclub.ab.ca/

[22] BHARVEY. BYOB3-trailer on Scratch. In: *Scratch project player* [online]. 19. 04. 2010, last modified on 19. 04. 2010 [cit. 2015-06-20]. Dostupné z: http://scratch.mit.edu/projects/995971/

[23] Brian Harvey & Jens Mönig (Scratch Conference 2013) In: *Youtube* [online]. 30. 07. 2013 [vid. 2014-04-05]. Dostupné z: http://www.youtube.com/watch?v=rXN81Hsj_A4 Kanál uživatele Citilab Cornella

[24] Build Your Own Blocks (Scratch Modification). In: *Scratch Wiki* [online]. Massachusetts: MIT Media Lab, 29. 12. 2011, last modified on 01. 01. 2015 [cit. 2015-06-18]. Dostupné z: http://wiki.scratch.mit.edu/wiki/BYOB

[25] FEDERICI, Stefano. *Blocklanguages* [online]. [cit. 2015-06-20]. Dostupné z: http://www.blocklanguages.org/

[26] Getting Ready for Scratch 2.0 In: *Youtube* [online]. 03. 05. 2013 [vid. 2014-04-05]. Dostupné z: https://www.youtube.com/watch?v=5u8ACXeNMiA Kanál uživatele TeamScratchEd

[27] HTML5 Player. In: *Scratch Wiki* [online]. Massachusetts: MIT Media Lab, 20. 08. 2013, last modified on 05. 05. 2015 [cit. 2015-06-18]. Dostupné z: http://wiki.scratch.mit.edu/wiki/HTML5_Player

[28] ISO 8601. In: *Wikipedia: the free encyclopedia* [online]. St. Petersburg (Florida): Wikipedia Foundation, 16. 11. 2001, last modified on 12. 06. 2015 [cit. 2015-06-18]. Dostupné z: http://en.wikipedia.org/wiki/ISO_8601

[29] Lambda calculus. In: *Wikipedia: the free encyclopedia* [online]. St. Petersburg (Florida): Wikipedia Foundation, 05. 11. 2001, last modified on 07. 06. 2015 [cit. 2015-06-18]. Dostupné z: https://en.wikipedia.org/wiki/Lambda_calculus

[30] List of educational programming languages. In: *Wikipedia: the free encyclopedia* [online]. St. Petersburg (Florida): Wikipedia Foundation, 10. 01. 2004, last modified on 24. 04. 2015 [cit. 2015-06-18]. Dostupné z: http://en.wikipedia.org/wiki/List_of_educational_programming_languages

[31] List of Scratch Modifications. In: *Scratch Wiki* [online]. Massachusetts: MIT Media Lab, 17. 09. 2010, last modified on 20. 06. 2015 [cit. 2015-06-18]. Dostupné z: http://wiki.scratch.mit.edu/wiki/List_of_Scratch_Modifications

[32] Making the Transition from Blocks to Text with Snap!. *Scratch – Connecting Worlds* [online]. 2013 [cit. 2015-06-20]. Dostupné z: http://scratch2013bcn.org/node/288

[33] Manually install on Android devices. ADOBE SYSTEMS INCORPORATED. *Flash Player Help* [online]. 2015 [cit. 2015-06-20]. Dostupné z: https://helpx.adobe.com/flash-player/kb/installing-flash-player-android-devices.html

[34] Příspěvek #113. BHARVEY. *Discuss Scratch: Snap! Team development discussion* [online]. 29. 05. 2013 [cit. 2015-06-20]. Dostupné z: https://scratch.mit.edu/discuss/post/42084/

[35] SCRATCH. Scratch 2.0: 4 new features. In: *Blogger* [online]. 2012-15-10 [cit. 2015-06-20]. Dostupné z: http://blog.scratch.mit.edu/2012/10/scratch-20-4-new-features.html

[36] Scratch. In: *Scratch Wiki* [online]. Massachusetts: MIT Media Lab, 05. 05. 2010, last modified on 06. 06. 2015 [cit. 2015-06-18]. Dostupné z: http://wiki.scratch.mit.edu/wiki/Scratch

[37] Scratch 2.0. In: *Scratch Wiki* [online]. Massachusetts: MIT Media Lab, 05. 05. 2010, last modified on 23. 05. 2015 [cit. 2015-06-18]. Dostupné z: http://wiki.scratch.mit.edu/wiki/Scratch_2.0

[38] Scratch Modification . In: *Scratch Wiki* [online]. Massachusetts: MIT Media Lab, 18. 04. 2010, last modified on 27. 03. 2015 [cit. 2015-06-18]. Dostupné z: http://wiki.scratch.mit.edu/wiki/Modification

[39] Scratch Statistics. *Scratch – Vymysli, programuj, poděl se* [online]. [cit. 2015-06-20]. Dostupné z: https://scratch.mit.edu/statistics/

[40] Scratch Versions. In: *Scratch Wiki* [online]. Massachusetts: MIT Media Lab, 10. 05. 2010, last modified on 04. 04. 2015 [cit. 2015-06-18]. Dostupné z: http://wiki.scratch.mit.edu/wiki/Scratch_Versions

[41] Snapin8r. HARDMATH123. *ComfortablyNumbered* [online]. 2013 [cit. 2015-06-20]. Dostupné z: http://hardmath123.github.io/Snapin8r/

[42] *Snap! (Build Your Own Blocks) 4.0* [online]. [2011] [cit. 2015-06-20]. Dostupné z: http://snap.berkeley.edu/

[43] Recursion. In: *Scratch Wiki* [online]. Massachusetts: MIT Media Lab, 15. 07. 2010, last modified on 02. 01. 2015 [cit. 2015-06-18]. Dostupné z: http://wiki.scratch.mit.edu/wiki/Recursion

[44] Retro Converter. BLOB8108. *Kurt Tools* [online]. [cit. 2015-06-20]. Dostupné z: http://kurt.herokuapp.com/20to14

[45] User:Bharvey. In: *Scratch Wiki* [online]. Massachusetts: MIT Media Lab, 08. 06. 2010, last modified on 02. 01. 2015 [cit. 2015-06-18]. Dostupné z: http://wiki.scratch.mit.edu/wiki/User:Bharvey

• **Hlášení a řešení v repositáři Snap na webové službě GitHub**

• *Poznámka: Názvy hlášení či řešení na GitHub mohou být změněny. Netřeba se však bát, že by citovaný záznam nebylo možné vyhledat. Níže uvedené odkazy totiž zůstavají neměnné.*

[46] BRIANHARVEY. Suggestion: Use 'is identical to' for case-sensitivity?: Issue #522. In: *GitHub* [on-line]. Jul 22, 2014 [cit. 2014-12-02]. Dostupné z: https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/issues/522

[47] CYCOMACHEAD. Codification Improvements: Issue #373. In: *GitHub* [on-line]. Apr 2, 2014 [cit. 2015-05-23]. Dostupné z: https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/issues/373

[48] CYCOMACHEAD. Feature: Camera + Mic Inputs for Costumes and Sounds: Issue #460. In: *GitHub* [on-line]. May 29, 2014 [cit. 2014-12-02]. Dostupné z: https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/issues/460

[49] CYCOMACHEAD. Text Tool For Costume Editor: Issue #122. In: *GitHub* [on-line]. Aug 23, 2013 [cit. 2014-12-02]. Dostupné z: https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/issues/122

[50] DDGARCIA. Colors: 'We need set pen saturation to (num)': Issue #327. In: *GitHub* [on-line]. Feb 14, 2013 [cit. 2014-12-02]. Dostupné z: https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/issues/327

[51] GUBOLIN. add 'add comment' to block context menus (fix #520): Pull Request #554. In: *GitHub* [on-line]. Aug 4, 2014 [cit. 2014-12-02]. Dostupné z: https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/pull/554

[52] GUBOLIN. leave a mark on pen down (fix #209): Pull Request #605. In: *GitHub* [on-line]. Oct 4, 2014 [cit. 2014-12-02]. Dostupné z: https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/pull/605

[53] GUBOLIN. Shared variables: Issue #629. In: *GitHub* [on-line]. Nov 1, 2014 [cit. 2014-12-02]. Dostupné z: https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/issues/629

[54] LUIS140219. Add Events category: Issue #709. In: *GitHub* [on-line]. Jan 27, 2015 [cit. 2015-06-02]. Dostupné z: https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/issues/709

[55] LUIS140219. Fixes #709: Issue #741. In: *GitHub* [on-line]. Mar 11, 2015 [cit. 2015-06-02]. Dostupné z: https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/issues/741

[56] MARWAHAHA. Default costumes should be visible in costume editor: Issue #420. In: *GitHub* [on-line]. May 7, 2014 [cit. 2014-12-02]. Dostupné z: https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/issues/420

[57] MMSEQUEIRA. NaN in a variable is treated differently from NaN reported by the division operator: Issue #132. In: *GitHub* [on-line]. Sep 2, 2013 [cit. 2014-12-02]. Dostupné z: https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/issues/132

[58] SHOW0K. Feature request : add a custom block category or something to preselect some blocks: Issue #776. In: *GitHub* [on-line]. Apr 20, 2015 [cit. 2015-06-02]. Dostupné z: https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/issues/776