Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Výukové prostředí funkcionálního programování pro děti

2023

Jakub Brázdil

Vedoucí práce:
Mgr. Jan Laštovička, Ph.D.

Studijní program: Aplikovaná informatika,
prezenční forma

Bibliografické údaje

Autor:	Jakub Brázdil
Název práce:	Výukové prostředí funkcionálního programování pro děti
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2023
Studijní program:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	Mgr. Jan Laštovička, Ph.D.
Počet stran:	63
Přílohy:	1 CD/DVD
Jazyk práce:	slovenský

Bibliographic info

Author:	Jakub Brázdil
Title:	Environment for teaching functional programming to children
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2023
Study program:	Applied Computer Science, full-time form
Supervisor:	Mgr. Jan Laštovička, Ph.D.
Page count:	63
Supplements:	1 CD/DVD
Thesis language:	Slovak

Anotácia

Jazyků pro výuku imperativního programování je celá řada. Naproti tomu rodič, který by se rozhodl učit své dítě programovat funkcionálně, má možnosti dost omezené. Táto práce popisuje vytváření jednoduchého funkcionálního jazyka a grafického prostředí, kde se děti od útlého věku seznamují se základy funkcionálního programování.

Synopsis

There are many languages for teaching imperative programming. On the other hand, a parent who decides to teach his or her child to program functionally has quite limited options. This paper describes the creation of a simple functional language and graphical environment where children learn the basics of functional programming from an early age.

Kľúčové slová: funkcionálne programovanie; funkcie; lambda; vývojové prostredie, výuka programovania pre deti

Keywords: functional programming, functions, lambda, development enviroment, teaching of programming for children

Ďakujem vedúcemu práce Mgr. Jánovi Laštovičkovi, Ph.D. za cenné rady a odborné vedenie pri vypracovaní bakalárskej práce. Zároveň ďakujem mojej rodine za trpezlivosť a všetkým účastníkom testovania programu.

Čestne vyhlasujem, že som celú prácu vrátane príloh vypracoval/a samostatne a za použitia iba zdrojov spomínaných v texte práce a uvedených v zozname literatúry.

dátum odovzdania práce podpis autora

Obsah

Zoznam obrázkov

1 Úvod

Funkcionálne programovanie je v informatike programovacia paradigma, kde sú počítačové programy vytvárané pomocou skladania a aplikácií funkcií. Zatiaľ čo iné programovacie paradigmy odrážajú koncepty, na ktorých je digitálny počítač postavený, funkcionálne programovanie nás od tohto zmýšľania odkláňa a umožňuje nám riešiť problém inak, bez nutnosti zamýšľania sa nad jeho formou z pohľadu digitálneho počítača.

Výuka programovania pre deti ponúka množstvo výhod, ktoré sa na prvý pohľad nemusia zdať zrejmé. Pri výuke však musíme zvážiť, aké princípy sa snažíme deti naučiť. Jazykov pre výuku imperatívneho programovania je celá rada. Môžeme si na príklad spomenúť na jazyk Karel. Naproti tomu rodič, ktorý by sa rozhodol učiť svoje dieťa programovať funkcionálne, má svoje možnosti do značnej miery obmedzené.

V tejto práci predstavujeme niekoľko dôvodov, prečo si myslíme, že deti by sa mali učiť práve funkcionálnu paradigmu. Použitím tých správnych prostriedkov ich vieme naučiť aj zložitejšie koncepty, ktoré využijú nie len pri práci s funkcionálne orinetovanými programovacími jazykmi. Na základných a stredných školách sa predovšetkým vyučujú procedurálne, imperatívne a niekedy objektovo orientované programovacie paradigmy. Mnohí si však výhody funkcionálneho prístupu k programovaniu uvedomujú a snažia sa ho predstaviť aj mladším generáciám. Prostriedkov, ako učiť deti funkcionálnej paradigme je však podstatne menej, no je dôležité spomenúť, že existujú. Tie popisujeme v tretej kapitole. Mnohí pedagógovia alebo rodičia sa môžu rozhodnúť ísť vlastnou cestou a skúsiť tak naučiť deti princípom funkcionálneho programovania na vlastnú päsť[1]. My tým ostatným pomôžeme, preto cieľom tejto práce bolo vytvorenie jednoduchého prostredia výuky funkcionálneho programovania pre deti.

V úvodnej kapitole predstavujeme funkcionálne programovanie, jeho výhody a nevýhody a predovšetkým aké sú jeho princípy. Objasňujmeme, prečo si myslíme, že výuka funkcionálneho programovania je pre deti prospešná, ako aj výuka programovania všeobecne.

V štvrtej kapitole na to naväzujeme a popisujeme návrh jazyka spoločne s prostredím, ktoré deťom postupne a hravou formou odkrýva princípy funkcionálneho programovania, medzi ktoré na príklad patria funkcie prvej triedy, absencia mutátorov, parciálna aplikácia a iné.

V šiestej kapitole pridávame sériu úrovní, prostredníctvom ktorých vyššie spomenuté princípy predstavíme deťom. Na koniec v ôsmej kapitole toto prostredie testujeme na vzorke dvoch detí.

[1] https://jackcoughonsoftware.blogspot.com/2009/05/teaching-functional-programming-to-kids.html

2 Výuka funkcionálného programovania

> *"In some ways, programming is like painting. You start with a blank canvas and certain basic raw materials. You use a combination of science, art, and craft to determine what to do with them. You sketch out an overall shape, paint the underlying environment, then fill in the details. You constantly step back with a critical eye to view what you've done. Every now and then you'll throw a canvas away and start again"* (Hunt, Thomas, 1990, str. 34) [1].

2.1 Funkcionálne programovanie

Funkcionálne programovanie je v informatike programovacia paradigma, kde sú počítačové programy vytvárané pomocou skladania a aplikácii funkcii. Celý program napísaný touto paradigmou sa dá vyjadriť ako výraz, ktorý odpovedá matematickej funkcii. Počiatky funkcionálneho programovania siahajú až do obdia pred prvými počítačmi. Funkcionálne jazyky sú veľmi výstižné a expresné, no zároveň je všetko dosiahnuté s použitím minimálneho množstva konceptov. Napriek tejto elegancii, funkcionálne jazyky sa netešili veľkej popularite, ba priam boli ingorované developermi [2].

Funkcionálne programovanie je v mnohých ohľadoch jednoduchšia a prehľadnejšia paradigma. Má to za následok to, že táto paradigma vychádza z čisto matematickej disciplíny teórie funkcií. Zatiaľ čo iné paradigmy, odrážajú koncepty, na ktorých je digitálny počítač postavený (ako napríklad imperatívna paradigma, ktorého správanie by sme vedeli zhrnúť vo vete: *"Najprv sprav toto, a potom urob toto."*), tak funkcionálne programovanie nás od tohto odkláňa a umožňuje nám riešiť problém inak, bez nutnosti zamýšlania sa nad jeho formou z pohľadu digitálneho počítača [3].

Funkcionálne programovanie môže byť dané do kontrastu s imperatívnym programovaním aj v negatívnom zmysle. Programy napísané funkcionálnou paradigmou nepoužívajú premenné. Nie je tu teda žiadny vnútorný stav, ktorým by sme vedeli identifikovať priebeh programu v danom okamihu. Vykonávanie inštrukcií v rade za sebou je vo funkcionálnom programovaní bezpredmetné, pretože vykonanie prvého príkazu, nemá žiadny dopad na ten nasledujúci. Môže za to spomínamá absencia vnútorného stavu, ktorá by to sprostredkovala. Funkcionálne programy využívajú funkcie dômyselnejšie ako ich imperatívne náprotivky. Funkcie sú tu brané podobne ako jednoduchšie objekty (na príklad celé čísla). Môžu byť teda argumentmi iných funkcií, alebo ako návratové hodnoty. Namiesto vykonávania inštrukcii za sebou a vykonávania cyklov, funkcionálne jazyky používajú rekurzívne funkcie alebo rekurzívny výpočetný proces [4]. Rekurzívne funkcie sú funkcie, ktoré vo svojom vnútri, obsahujú aplikáciu seba samej. Rekurzívny výpočetný proces je proces, keď behom aplikácie funkcie dôjde znovu k aplikácii tej istej funkcie [5].

Niektoré charakteristiky jazykov podporujúce funkcionálne programovanie:

- **funkcie prvej triedy** - funkcie sú brané ako hodnoty a preto môžu byť ukladané do premenných a používané ako argumenty a návratové hodnoty iných funkcí [6],
- **funkcie vyššieho rádu** - funkcie, ktoré sú určené k tomu, aby pracovali s inými funkciami ako s hodnotami, to jest aby iné funkcie prijímali ako svoje argumenty, prípadne aby funkcie vracali ako svoj výsledok [6],
- **čisté funkcie** - funkcie bez vedľajšieho efektu. Vedľajší efekt znamená, že okrem vrátenia hodnoty, výraz spôsobí nejakú vonkajšiu zmenu. Tou zmenou môže byť zmena prostredia, tlačový výstup alebo iný výstup [7],
- **rekurzia** - ako náhrada cyklu, dosiahnutá či už rekurzívnymi funkciami alebo rekurzívnym výpočetným procesom,
- **absencia mutátorov** - mútatory sú nástroje na zmenu hodnôt uložených v premenných a dátových štruktúrach,
- **anonymné funkcie** - funkcie bez názvu [8]. Napríklad v programovacom jazyku *Common Lisp* ich vytvárame pomocou makra lambda[2],
- **currying** - currying je proces transformácie funkcie, ktorá prijíma ako argument viacero argumentov, na funkciu, ktorá prijíma len jeden argument a vracia inú funkciu, ktorá prijíma ďalšie argumenty, jeden po druhom, ktoré by pôvodná funkcia prijala vo zvyšku argumentov [9],
- **parciálna aplikácia** - je jav, kedy dodaním menšieho ako plného počtu argumentov funkcii, ktorá prijíma viacero argumentov, vzniká funkcia s menším počtom parametrov [10],
- **lenivé vyhodnocovanie** - technika, kedy s objektom pracujeme ako s hodnotou, napriek tomu, že jeho presnú hodnotu doposiaľ nepoznáme. Výpočet tejto hodnoty odkladáme až na najneskoršiu možnú dobu. Takéto objekty sú označované aj ako prísľuby [11],
- **syntax jazyka podporujúca vyhodnocovanie výrazov**.

2.2 Prečo programovať funkcionálne

V tejto časti si predstavíme niekoľko výhod funkcionálneho programovania a objasníme jeho opodstatnenie pri vývoji softwaru.

[2] http://www.lispworks.com/documentation/lw51/CLHS/Body/m_lambda.htm

2.2.1 Čisté funkcie

Čistá funkcia je funkcia, ktorá pri rovnakom vstupe vždy vráti rovnaký výstup a nemá žiadny pozorovateľný vedľajší účinok [12]. Takéto funkcie sú predvídateľné a jednoduchšie na odhalenie chyby v programe. Pretože nedochádza k zmene žiadneho stavu mimo danú funkciu a funkcia vždy vykoná presne to čo má a nič viac, je spoľahlivá a znižuje sa tak šanca na výskyt chyby v programe. Ďalšou výhodou je, že kód pozostavený z čistých funkcii je jednoduchší na ladenie. Tým, že nám funkcia vráti vždy rovnaký výsledok pre dané argumenty, výborne sa nám hodí pre unit testing[3] [13].

2.2.2 Ladenie

Funkcionálne jazyky sú jednoduchšie na ladenie. Pretože sa program skladá z čistých funkcií, ktoré nemenia žiadny stav, tak chyba v programe môže nastať len v niektorej z nich. Tým, že sú funkcie deterministické, tak chybu v programe vieme jednoducho simulovať, zadaním tých argumentov, ktoré ju vyvolali. Naopak pri programoch, ktoré sa skladajú z nederministických funkcií, môže byť simulovanie chyby náročné. Počiatok tejto chyby, mohol vzniknúť tesne pred pádom programu, ale mohol vzniknúť aj pred niekoľkými minútami, či hodinami.

2.2.3 Znovupoužiteľnosť programu

Dekompozícia je proces rozkladania zložitejšieho problému na menšie, jednoduchšie problémy. Hlavnou motiváciou tohto procesu je fakt, že zložitejšie problémy sú nepomerne ťažšie na ich vyriešenie ako tie jednoduchšie. Je podstatne jednoduchšie napísať dva programy ktorých dĺžka je 500 riadkov ako jeden o dĺžke 1000 riadkov [14].

Funkcionálne programovanie nám tento proces do značnej miery uľahčuje. Riešenie problému rozkladáme na stále jednoduchšie funkcie, pokým neprídeme po také, ktorých implementácia je banálna [15]. Takéto funkcie sú znovu použiteľné a môžeme ich využiť pri riešení iných problémov, na ktoré narazíme v neskorších fázach vývoja. Ďalšou výhodou týchto funkcií je, že ich úprava sa zjednodušuje a prípadný zásah do ich definície, neponesnie toľko následkov, ako keby sa jednalo o funkciu, ktorá je komplexnejšia.

2.2.4 Viacjadrové procesory

Paralelné programovanie je paradigma konštrukcie programu, obsahujúceho dva a viac procesov (vlákien), vykonávaných paralelne [16]. Jedným z veľkých problémov paralelného programovania je synchronizácia jednotlivých vlákien. Súbežné vlákna pristupujúce k rovnakému mutabilnému stavu vedú k nedeterminizmu.

$$\text{nedeterminizmus} = \text{mutabilný stav} + \text{paralelné spracovanie}$$

[3] https://en.wikipedia.org/wiki/Unit_testing

Nedeterminizmus zo sebou prináša radu problémov. Súbeh (race condition), deadloky, hladovanie po zdrojoch, livelocky. Aby sme dostali deterministické spracovanie, tak sa potrebujeme zbaviť mutabilného stavu, alebo sa aspoň snažiť ho používať čo najmenej. Jeho zbavením sa, znamená programovať funkcionálne [17] (pre úplnosť dodajme, že absencia mutabilného stavu je charakteristická aj pre logické programovanie).

Ako však môžu procesy medzi sebou komunikovať? Napríklad v jazyku Erlang[4] je tento problém riešený pomocou zdielanej asynchronnej signalizácie. Jednotlivé signály sú prijímané asynchrónne a automaticky. Samotný proces nedokáže nič spraviť pre to, aby tomuto signálu zabránil a na tento signál nemusí zareagovať. Všetky prijaté správy sú zhromažďované v zásobníku správ a na vyzdvihnutie signálu sa využíva akcia receive. Po jej vyzdvihnutí proces pokračuje ďalej v činnosti [18].

2.3 Prečo učiť deti funkcionálne programovať

Skôr než si ukážeme, prečo by sa mali deti učiť funkcionálne programovať, musíme sa pozrieť na to, prečo je programovanie benefitom všeobecne.

2.3.1 Prečo programovať

V médiách často krát počujeme, že počet programátorov na pracovnom trhu je nízky a je nutné vychovávať každoročne čoraz viac nových programátorov [19][20]. Jedným z hlavných motivátorov, prečo programovať, ktoré média uvádzajú, je nadpriemerná výška zárobku, dobré uplatnenie a istota na pracovnom trhu a podobne. Tieto vidiny môžu motivovať mnohých nahliadnuť do sveta programovania, no skutočných dôvodov prečo by sa ľudia mali učiť programovať je viacero [21]:

- zvýšenie schopnosti vyjadriť svoje myšlienky a nápady,
- rozvoj kritického myslenia a schopnosti riešiť problémy,
- skvalitnenie výberu vhodných nástrojov na riešenie problému,
- zvýšená flexibilita pri učení sa a adaptácii na nové koncepty.

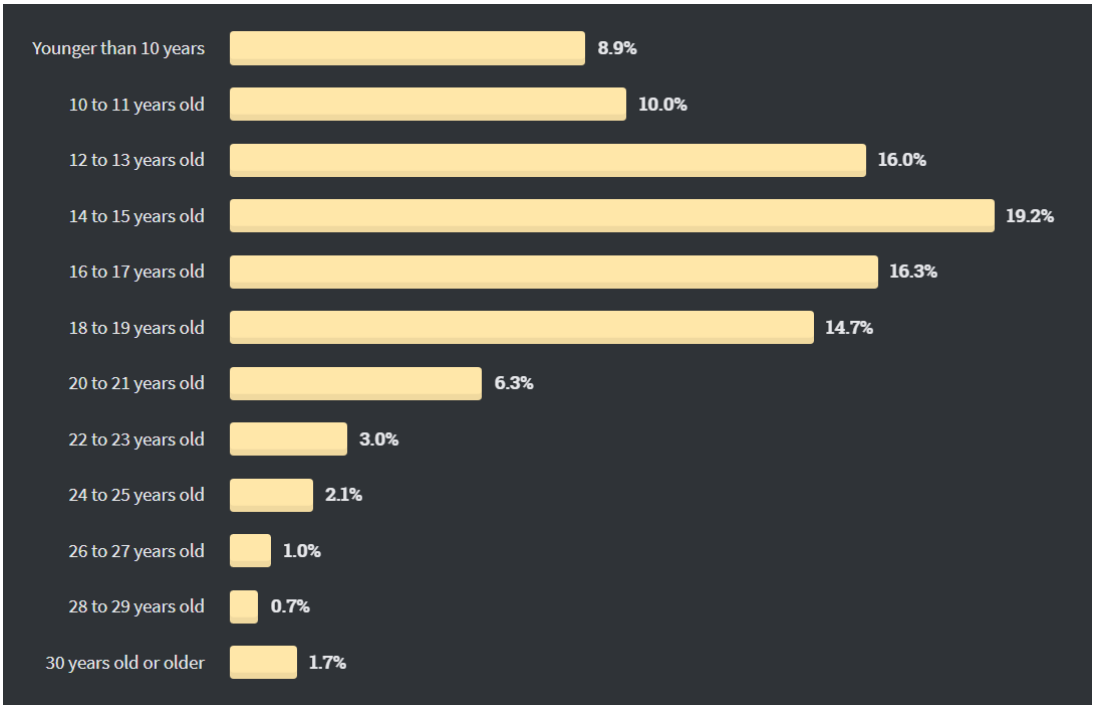
Výhod ktoré nadobudneme vzelávaním sa v obore programovania skutočne nie je málo. Vyššie spomenuté body, môžu byť kľúčové pri raste a rozvoji dieťaťa, no otázne však je, do akej miery.

[4] https://www.erlang.org/

2.3.2 Prečo učiť deti programovať

Programovanie prináša množstvo výhod. Prečo by sme však mali začať programovať od útleho veku?

Webstránka *Stackoverflow* v roku 2020 [22] podnikla prieskum s názvom *Developer Survey*, kde sa jej užívateľov pýtala rôzne otázky, súvisiace s oborom informačných technológii. Mimo iné zisťovala, v akom veku napísali opýtaní respondetni, svoj prvý riadok kódu. Z prieskumu vyšlo, že až 54% opýtaných napísalo prvý riadok kódu do veku 16 rokov. O to zaujímavejšie je, že až 8.9% opýtaných, začalo programovať ešte pred dovŕšením 10 rokov a z počtu 59,700 opýtaných, sa skutočne jedná o nezanedbateľnú skupinu ľudí (obr. 1).



Obr. 1: Výsledky prieskumu *2020 Developer Survey* stránky *Stackoverflow* na otázku, v akom veku, napísali respondenti svoj prvý riadok kódu

Programovanie sa však v takom veku, ako je menej než 10 rokov, môže zdať na prvý pohľad príliš ambiciózne. Kvalitatívna štúdia z roku 2018, vykonaná pod záštitou Európskej komisie však ukázala, že deti prichádzajú do prvého kontaktu s obrazovkami a digitálnymi technológiami už vo veľmi ranom veku, zvyčajne prostredníctvom zariadení rodičov, ktoré na to nie sú uspôsobené [23]. V súčasnosti sa kladie veľký dôraz na digitalizáciu vyučovacích procesov a vznikajú rôzne programy na podporu digitálnych technológii pri výuke [24]. Výukou programovania, učíme deti tieto technológie nie len používať, ale učíme ich aj princípom, na ktorých sú tieto technológie postavené a dávame im možnosť, aby ich v budúcnosti rozvíjali a nezostali iba ich užívateľmi.

Ďalším motivátorom je, že znalosť programovania zvyšuje gramotnosť. *"Čím*

viac komunikácie, spoločenskej organizácie, vládne funkcie a obchod sa uskutočňujú prostredníctvom kódu - a počítačová gramotnosť sa stáva čoraz viac infraštruktúrnou, rovnováha síl sa opäť posúva v prospech tých, ktorí sú zruční v tejto novej technológii gramotnosti." [25] napísala autorka knihy *Coding Literacy* Anette Vee (2013, str. 55), ktorá schopnosť programovania považuje za novodobú súčasť gramotnosti. Faktom je, že deti sa učia jazyky o poznanie jednoduchšie, ako dospelí [26]. Nakoľko sa znalosť programovania stáva gramotnosťou a dostáva sa na rovnakú úroveň, ako znalosť iného jazyka, je užitočné toto okno príležitosti využiť a zaradiť výuku programovania ideálne súbežne, s výukou iného cudzieho jazyka. Programovanie môže tiež slúžiť ako prostriedok, na zvýšenie neverbálnych kognitívnych funkcii[5] u detí [28] a taktiež preukázateľne vedie k rozvoju matematických konceptov a riešenia problémov [29].

[5]Kognícia je súbor mentálnych, poznávacích aktivít, ako je vnímanie, pamätanie, myslenie, rozhodovanie, hodnotenie a riešenie problémov [27].

3 Súčasné možnosti výuky programovania pre deti

Pred vytvorením prostredia, je veľmi dôležité vykonať analýzu dostupných prostriedkov, využívaných pre výuku programovania. Jednak nám poskytnú dôležité informácie, ktoré môžeme využiť pri výuke programovania, ale môžu nás aj inšpirovať pri samotnom vývoji našeho prostredia.

3.1 Jazyky bežne vyučované na školách

V tejto časti si predstavíme jazyky používané pre výuku programovania na základných a stredných školách na Slovensku. Tieto jazyky pre výuku odporúča Štátny pedagogický ústav Ministerstva školstva Slovenskej republiky a uvádza ich vo svojej publikácii *Malé programovacie jazyky* [30]. Je preto veľká pravdepodobnosť, že niektoré (prípadne) všetky sú zaradené do procesu výuky na základných a stredných školách.

3.1.1 Karel

Karel je začiatočnícky programovací jazyk. V roku 1981 ho vytvoril Richard Pattis a využíval ho ako modelový jazyk jeho prednášok na Stanford University v Kalifornii. Jazyk je pomenovaný po českom spisovateľovi Karolovi Čapekovi [31].

Výhodou jazyka Karel je, že na prácu v ňom sa nepredpokladajú žiadne predchádzajúce znalosti z informatiky či matematiky.

Princípom jazyka, je ovládanie robota zvaného *Karel*. Karel sa pohybuje vo svojom svete, ktorý je rozdelený mriežkou. Karlovi môžeme zadávať príkazy, ktoré bude vykonávať vo svojom svete. Karel rozumie iba niekoľkým základným príkazom (vychádzame z jeho implementácie v jazyku Python[6]):

- **move()** - krok vpred o jeden blok v mriežke,
- **turn_left()** - rotácia vľavo o 90 stupňov,
- **pick_beeper()** - zdvihnutie ”beeperu” (pípača) zo zeme a uloženie si ho medzi ostatné beepery (Karel dokáže uschovať nekonečné množstvo beeperov),
- **put_beeper()** - položenie beeperu na blok v mriežke, na ktorom Karel práve stojí.

Karel dokáže rozhodovať, či sa pri ňom nachádza beeper alebo či je pred ním stena. Súčasťou jazyka sú aj konštrukty pre cyklus a vetvenie.

Cieľom programov napísaných v jazyku Karel je navigovanie robota cez jeho svet, pričom jeho úlohov je zbieranie beeperov a následne ich rozmiestňovať podľa

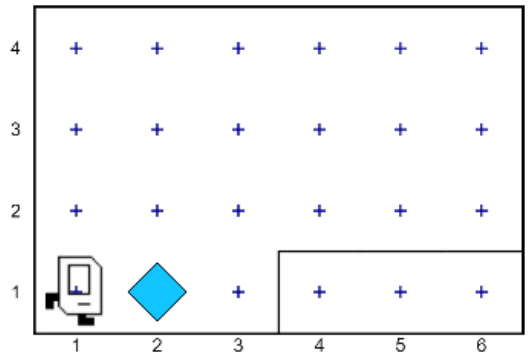
[6] https://compedu.stanford.edu/karel-reader/docs/python/en/chapter2.html

vopred určenej predlohy. Dôležité je, aby Karel nenarazil do steny a aby nezodvihol beeper tam, kde nie je (obr. 2).

Príkazy sa v jazyku Karel vykonávajú sekvenčne za sebou, a pareto je postavený na princípe procedurálnej programovacej paradigmy.

```
1 def main():
2   move()
3   pick_beeper()
4   move()
```

Zdrojový kód 1: Zdrojový kód jednoduchého programu jazyka Karel v Pythone



Obr. 2: Grafické rozhranie implementácie jazyka Karel v Pythone

3.1.2 SGP Baltík 3

Je detský programovací jazyk vyvinutý spoločnosťou *SGP Systems*. Výhodou tohto jazyka je, že príkazy sa zapisujú prostredníctvom ikon. Pri nesprávnom použití je v programe okamžite vidieť chybu. Baltík existuje aj vo verzii 4 zvanej *SGP Baltie 4 C#* kde môže užívateľ priamo pracovať so zdrojovým kódom. Hlavným hrdinom ktorý sprevádza deti pri riešení úloh v tomto jazyku je čarodejník Baltík.

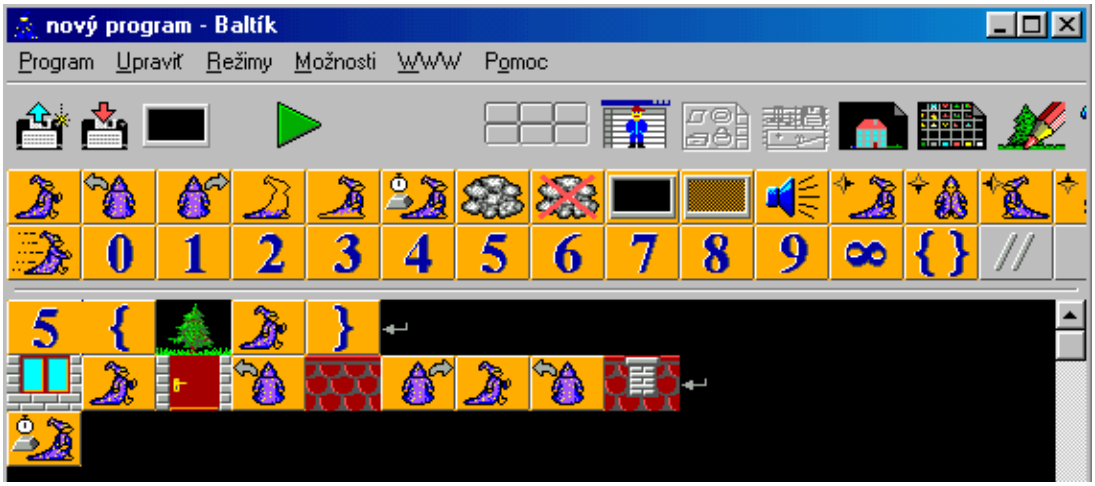
Prostredie dokopy s jazykom tvorí multimediálne kresliace nástroje a funguje v 3 režimoch [32]:

- režim **Skladať scénu** - tu si dieťa skladá obrázky a tvorí tak scény,
- režim **Čarovať scénu** - v tomto režime dieťa ovláda Baltíka pomocou príkazov ako otočenie vpravo, vľavo či krok vopred a môže rôzne interagovať s vytvorenou scénou,
- režim **Programovať** - v tomto režime deti môžu vytvoriť jednoduché programy, animácie a rozprávky. Tento režim delíme na režimy:

– režim **začiatočník** - obmedzené množstvo príkazov,

– režim **pokročilý** - v tomto režime sú dostupné všetky príkazy.

Princíp Baltíka (obr. 3) je však v mnohom podobný jazyku Karel a to v tom, že ovládanie čarodejníka je na základe programovu vytvorených procedurálnou programovacou paradigmou.



Obr. 3: Príklad jednoduchého programu v jazyku Baltík

3.1.3 Scratch

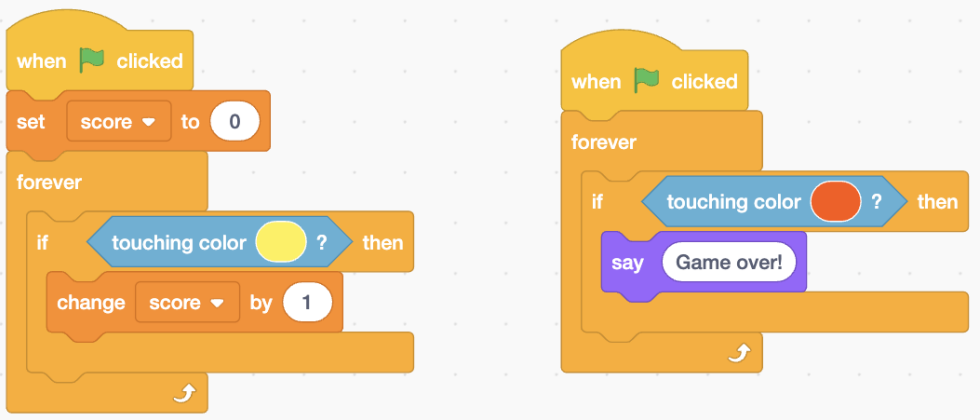
Scratch je jednoduché vizuálne prostredie slúžiace na vytváranie animácii, príbehov a hier. Jeho autorom je výskumné laboratórium *MIT Media Lab* z *Massachusettskej technickej univerzity*. Cieľová skupina tohto jazyka sú žiaci vo veku 8 až 16 rokov. Je založené na troch princípoch [33]:

- **flexibilnosť** - je daná spoluprácou s firmou *Lego*. Programovanie v Scratchi je inšpirované skladaním lega a má ho do určitej miery reflektovať. Program vzniká skladaním blokov (obr. 4). Jednotlivé bloky do seba zapadajú len v tom prípade, ak toto spojenie dáva sémantický zmysel. Na základe tejto skutočnosti sa Scratch označuje aj za jazyk ktorého paradigma je založená blokoch no jeho hlavná paradigma je paradigma riadená udalosťami,

- **zmysluplnosť** - zmyslom Scratchu, je aby v ňom deti vedeli vyjadriť akýkoľvek svoj nápad či myšlienku. Prostredie preto ponúka množstvo personalizácie a rozmanitosti, aby bola výuka v Scratchi efektívna,

- **sociálnosť** - súčasťou Scratchu je možnosť zdielať svoj projekt na webstránke[7] kde si projekt môžu pozrieť ostatní užívatelia.

[7] https://scratch.mit.edu/



Obr. 4: Príklad jednoduchého programu v jazyku Scratch [34]

3.2 Funkcionálne vývojové prostredia

Existuje veľké množstvo nástrojov pre výuku programovania impertívnou paradigmou programovania. Aké možnosti však máme, pokiaľ chceme deti vyučovať funkcionálnu programovaciu paradigmu? Odpoveď na túto otázku je o niečo zložitejšia.

3.2.1 Shelly

Shelly je moderný programovací jazyk inšpirovaný jazykom *Logo*. Je dostupný prostredníctvom prehliadača s možnosťou zdielania programov v ňom vytvorených. Napriek jeho inšpirácii, syntax jazyka Logo takmer vôbec neprípomina no myšlienka je veľmi podobná. K dispozícii máme korytnačku, ktorá kreslí čiary a tvary rozličných farieb, za použitia rôznych štýlov. Charakteristiky jazyka Shelly sú:

- všetky prvky jazyka sú **výrazy**,
- **funkcie prvej triedy**,
- **absencia mutátorov**.

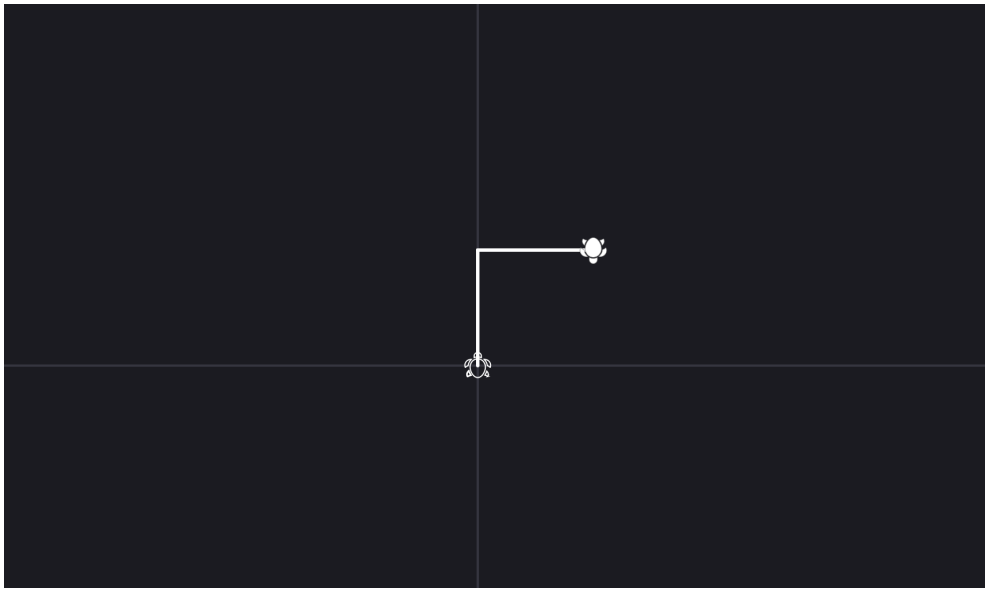
Inštrukcie jazyka sú hodnoty. Implementácia inštrukcií je riešená prostredníctvom *párov* zložených z vypočítanej hodnoty a zoznamu inštrukcií pre korytnačku. Celý program nám tak na konci vráti vypočítanú hodnotu a inštrukcie pre korytnačku [35].

V programe (zdrojový kód 2) využívame výraz *forwardRight* presne dva krát. Hodnota celého výrazu tak obsahuje 4 inštrukcie, ktorými sú inštrukcie z tela funkcie *forwardRight*.

```
let forwardRight = (
  fw 100
  right 90
)

forwardRight
forwardRight
```

Zdrojový kód 2: Ukážka jednoduchého programu v jazyku Shelly (obr. 5)



Obr. 5: Program v jazyku Shelly z ukážky zdrojový kód 2

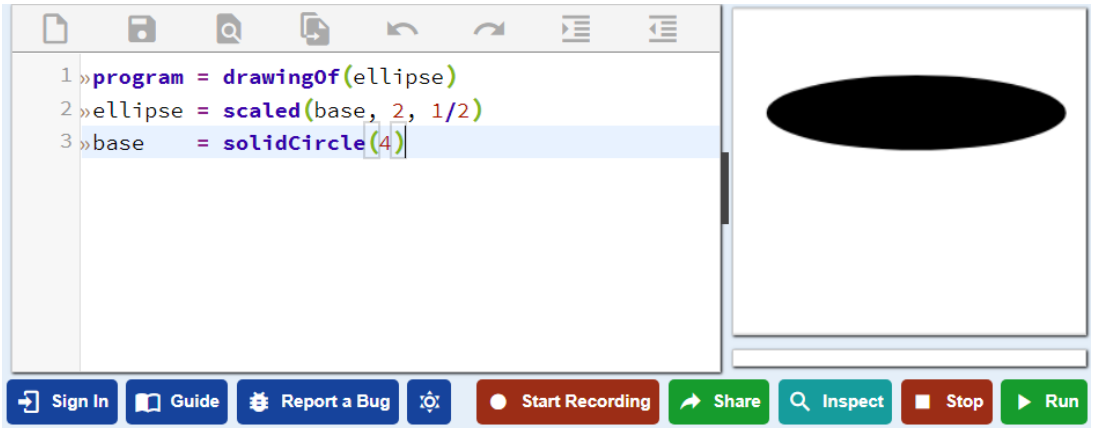
3.2.2 CodeWorld

CodeWorld je zjednodušené prostredie pre výuku jazyka *Haskell*. Ponúka jednoduché matematické modely tvorbu geometrických útvarov. Užívatelia sú v tomto prostredí schopní vytvárať unikátne kresby, animácie a dokonca hry. Tak ako Shelly, toto prostredie je dostupné prostredníctvom prehliadača s možnosťou zdielania programov v ňom vytvorených. Existuje niekoľko variánt tohto prostredia [36]:

- **CodeWorld** - štandardná verzia prostredia (obr. 6), ktorá používa edukačnú variantu jazyka Haskell a knižnice, podporujúce matematické inštrukcie.
- **CodeWorld Haskell** - je postavený na štandardnom jazyku Haskell a nie jeho edukačnej verzii. Slúži na vytváranie programov, ktoré môžu byť zobrazené na web stránke pomocou GHCJS (kompilátor z Haskell do JavaSc-

ript, ktorý používa GHC API) alebo kompilované do API od CodeWorld. Je prirodzeným následovníkom po štandardnej verzii prostredia CodeWorld,

- **CodeWorld Blocks** - zjednodušná verzia CodeWorld slúžiaca pre výuku mladších ročníkov. Jej princíp je veľmi podobný jazyku Scratch, kedy programy skladáme pomocou puzzle *Drag & Drop* gestom. Táto verzia je však stále vo vývoji a neodporúča sa jej používanie.



Obr. 6: Ukážka jednoduchého programu v štandardnej verzii CodeWorld

4 Ako sme učili deti funkcionálne programovať

Predtým, než sme sa rozhodli, ako bude naše prostredie vyzerať, stanovili sme si našu cieľovú skupinu.

4.1 Cieľová skupina

Pri tvorbe sme vychádzali zo školských osnov, ktoré sú aktuálne platné v Slovenskej republike.

Informatika sa začína na školách vyučovať v prvom stupni základnej školy a to v treťom ročníku. Na konci štvrtého ročníka by mal byť žiak schopný analyzovať problém, interaktívne zostaviť riešenie pomocou postupnosti príkazov a poskytnúť tak jeho algoritmické riešenie[8]. Mnohé deti prichádzajú do kontaktu s technológiami podstatne skôr ako na hodine informatiky.

Funkcionálne programovanie má silný matematický základ a ak použijeme tie správne prostriedky a abstrakciu, vieme vysvetliť jeho princípy aj bez predošlej znalosti programovania. Matematika sa na školách vyučuje už od prvého ročníka základnej školy[9] a teda je predpoklad, že učiť základy funkcionálneho programovania, sme schopní už po získaní základných znalostí matematiky.

Problémom však zostáva gramotnosť žiakov. Predpokladom na programovanie v bežných programovacích jazykoch (samozrejme nie vizuálnych) je znalosť čítania a písania. Našou snahou bolo, aby vstupné predpoklady boli čo najnižšie a znalosť čítania a písania by nemala byť bariérou pre prácu v našom prostredí.

Ak spojíme dokopy vyššie spomenuté, tak dostávame celkom konkrétnu charakteristiku prostredia a jazyka, ktorý sme vytvorili:

- prostredie by malo byť na používanie čo **najjednoduchšie** (ideálne by ovládanie malo prebiehať "iba" pomocou počítačovej myšy a jedného tlačidla),
- jazyk našeho prostredia bude **vizuálny**,
- využitie vhodnej **abstrakcie**,
- obmedzená znalosť čítania a písania je na prácu v prostredí dostatočná.

4.2 Jazyk našeho prostredia

Ako prvý krok pri tvorbe našeho jazyka, sme museli vymyslieť vhodnú abstrakciu, ktorú budeme využívať pre výuku.

[8] https://www.minedu.sk/data/att/22036.pdf
[9] https://www.minedu.sk/data/att/11634.pdf

4.2.1 Abstrakcia

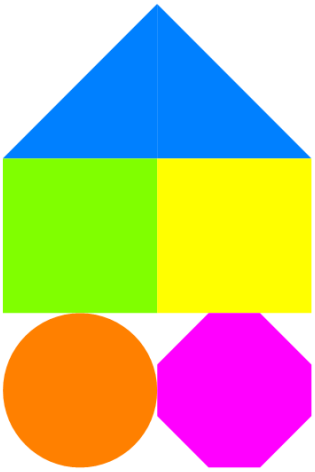
Rôzne prostredia pre výuku programovania využívajú abstrakciu zadávania príkazov. Či už je to ovládanie robota alebo grafického pera, ich princíp zostáva rovnaký a tým je vykonávanie príkazov sekvenčne za sebou. Funkcionálna paradigma je však špecifická a založená na inom princípe. Beh funkcionálneho programu je pozostáva z vyhodnocovania výrazov. Naše prostredie preto podporuje skladanie výrazov a ich okamžié vyhodnocovanie. Inšpirovali sme sa hrou skladania drevených kociek (obr. 7)[10].

Obr. 7: Hra skladania drevených kociek

4.2.2 Cieľ prostredia

Našu verziu skladania kociek sme jemne upravili a prevediedli do 2D podoby (obr. 8). Všetky základné útvary majú rovnakú veľkosť a sú definované svojím tvarom. Cieľom prostredia je replikovať existujúce útvary pomocou tých, ktoré máme v danom okamihu dostupné.

[10] https://www.ikea.com/sk/sk/p/underhalla-40-dielny-drevena-stavebnica-viacfarebny-00506684/



Obr. 8: Príklad zloženého útvaru v našom prostredí

4.2.3 Hodnoty jazyka

Jazyk obsahuje tri typy hodnôt:

- tvar
- farba
- f lambda

Tvar

Tvar delíme na dva typy:

- jednoduchý tvar
- zložený tvar

Existuje 6 druhov jednoduchých tvarov:

- štvorec
- kruh
- trojuholník
- šesťuholník
- hviezda
- neviditeľný tvar (zaberá miesto na plátne, no **nevykreslí sa**)

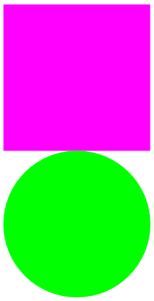
Jednoduchý tvar môžeme rôzne upravovať, to znamená rotovať, zväčšovať, zmenšovať alebo zafarbiť.

Jedná sa o nemennú hodnotu. Pokiaľ chceme získať jednoduchý tvar odlišných vlastností aký máme, je nutné vytvoriť nový. Absencia mutátorov je charakteristickou vlastnosťou funkcionálnych jazykov.

Jednoduché tvary môžeme skladať do väčších, zložených tvarov. Typov zloženia poznáme tri:

-  vertikálne spojenie
-  horizontálne spojenie
-  prekrytie

Vertikálne spojenie (obr. 9) je typ zloženia, kde sa tvary, z ktorých toto zloženie pozostáva, vykresľujú za sebou vertikálne, v poradí, v ktorom bolo toto zloženie vytvorené.

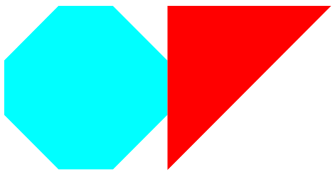


Obr. 9: Príklad zloženého tvaru typu vertikálne spojenie

Vertikálne spojenie centruje menší z objektov na stred (obr. 10) a je nutné na to pri jeho vytváraní myslieť. Preto ak chceme vytvoriť zložený tvar tohto typu so zarovnaním objektov napríklad vľavo (samozrejme iba vizuálne), hodí sa nám jednoduchý tvar typu neviditeľný tvar.

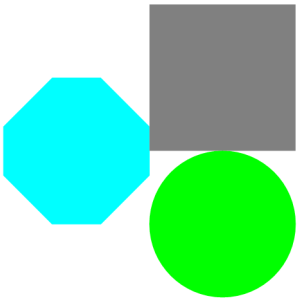
Obr. 10: Príklad centrovania menšieho z tvarov pri type vertikálne spojenie

Horizontálne spojenie (obr. 11) je typ zloženia, kde sa tvary, z ktorých toto zloženie pozostáva, vykresľujú za sebou horizontálne, v poradí, v ktorom bolo toto zloženie vytvorené.



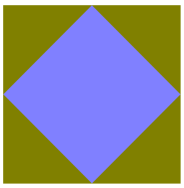
Obr. 11: Príklad zloženého tvaru typu horizontálne spojenie

Podobne ako vertikálne spojenie aj horizontálne spojenie centruje menší z tvarov na stred (obr. 12).



Obr. 12: Príklad centrovania menšieho tvaru pri type horizontálne spojenie

Prekrytie (obr. 13) je typ zloženia, kde sa tvary, z ktorých toto zloženie pozostáva, vykresľujú nad sebou, v poradí, v ktorom bolo toto zloženie vytvorené.



Obr. 13: Príklad zloženého tvaru typu prekrytie

Farba

S farbou pracujeme ako s každou inou hodnotou. Je nemenná a slúži na prefarbenie tvarov. Môžeme ju taktiež vyhodnocovať.

Lambda

Lambda je hodnota, ktorá slúži na prácu s inými hodnotami našeho jazyka. Môže ich spájať, otáčať, zmenšovať a rôzne upravovať. Ak by sme mali použiť

abstrakciu, tak každú lambdu si vieme predstaviť ako krabičku, ktorá má určitý počet parametrov (sú zobrazované ako guličky s nápisom ”arg”), ktoré môžeme spájať s inými krabičkami. Ona s nimi vykoná nejakú akciu, na základe ktorej dostaneme novú hodnotu. Táto hodnota je výsledkom aplikácie lambdy na jej argumenty. Tento výsledok záleží na jednotlivých krabičkách, ktoré sú spojené s guličkami danej lambdy.

V jazyku používame aj názov funkcia. Funkcie predstavujú lambdy, ktoré sú zabudované do našeho jazyka. Ide o *primitíva*. Ostatné lambdy sú lambdy vytvorené užívateľom, a nazývame ich *užívateľsky definované*. Interne sú oba typy rovnaké.

Primitív poznáme osem:

-  vertikálne spojenie
-  horizontálne spojenie
-  prekrytie
-  zmenšenie
-  zväčšenie
-  otočenie vľavo
-  otočenie vpravo
-  prefarbenie

Vertikálne spojenie akceptuje 2 argumenty typu *tvar* a výsledkom je *zložený tvar* typu *vertikálne spojenie*.

Horizontálne spojenie akceptuje 2 argumenty typu *tvar* a výsledkom je *zložený tvar* typu *horizontálne spojenie*.

Prekrytie akceptuje 2 argumenty typu *tvar* a výsledkom je *zložený tvar* typu *prekrytie*.

Zmenšenie akceptuje 1 argument typu *tvar* a výsledkom je tvar zmenšený o 50%.

Zväčšenie akceptuje 1 argument typu *tvar* a výsledkom je tvar zväčšený o 50%.

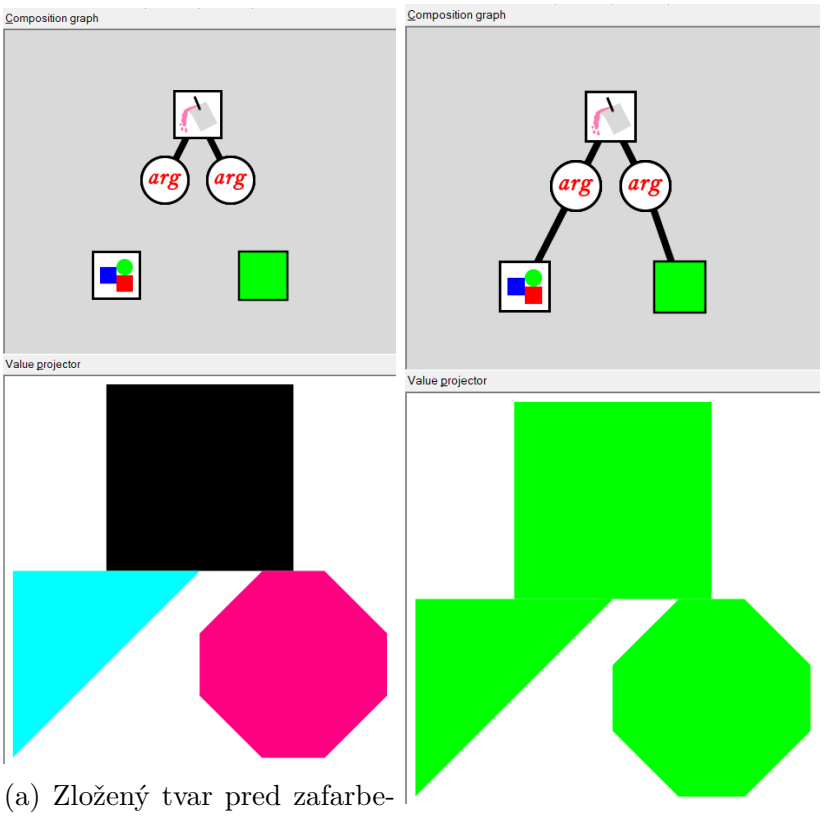
Rotácia vľavo akceptuje 1 argument typu *tvar* a výsledkom je tvar otočený o 90 stupňov vľavo podľa jeho stredu.

Rotácia vpravo akceptuje 1 argument typu *tvar* a výsledkom je tvar otočený o 90 stupňov vpravo podľa jeho stredu.

Prefarbenie akceptuje 2 argumenty. Prvý typu *tvar* a druhý typu *farba*. Výsledkom je tvar, ktorého obsah je celý prefarbený príslušnou farbou.

4.2.4 Transformácia hodnôt

S hodnotami nášeho jazyka môžeme rôzne narábať. Problémom však je, že interne naše hodnoty nenesú v sebe žiadnu inú informáciu okrem svojho typu a prípadne dodatočných dát. Preto ak aplikujeme napríklad lambdu zmenšenia na jednoduchý tvar, výsledkom je ten istý tvar s menšou veľkosťou. V našom jazyku sa preto nachádza ďalší typ hodnoty a tým je **transformácia**. Transformácia je hodnota ako každá iná a slúži na priradie dodatočných vlastností k hodnotám našeho prostredia. Každá transformácia v sebe nesie hodnotu, na ktorú bola aplikovaná a potom hodnotu a typ tejto transformácie. Z pohľadu užívateľa sa ale jedná o abstrakciu a nemusí priamo pracovať s transformáciami. Takýto prístup má však svoj následok, s ktorým musíme počítať. Tento následok vidíme napríklad pri prefarbení zloženého tvaru, kedy dôjde k prefarbeniu všetkých jeho tvarov, z ktorých sa skladá (obr. 14).



(a) Zložený tvar pred zafarbením

(b) Zložený tvar po zafarbení

Obr. 14: Zložený tvar pred a po jeho zafarbení.

4.2.5 Reprezentácia hodnôt v prostredí

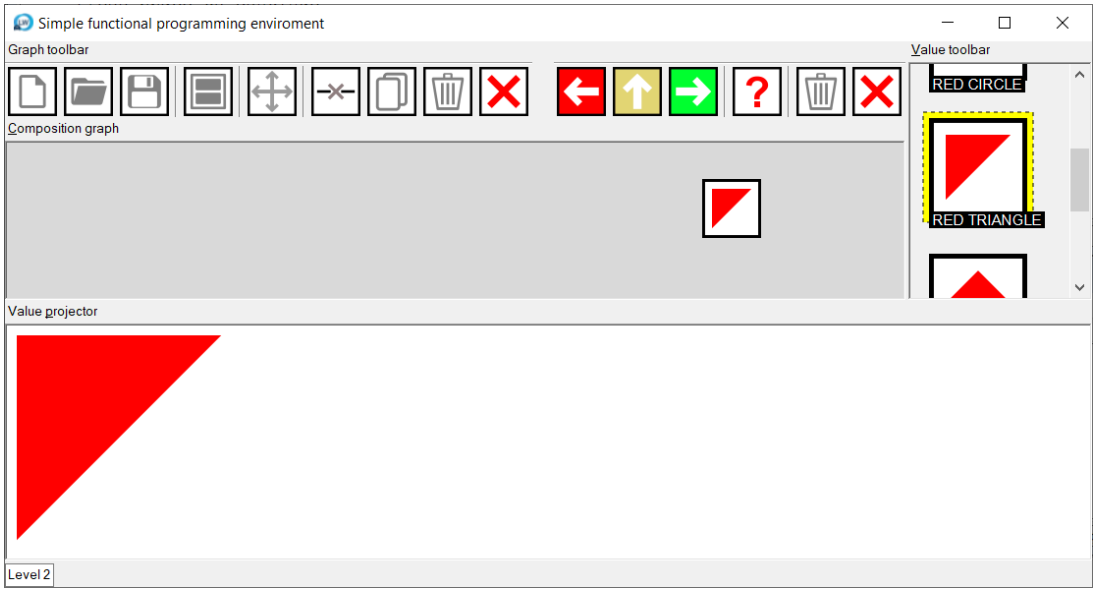
Pretože sa jedná o grafický jazyk, hodnoty našeho jazyka sme museli v prostredí nejak reprezentovať. Ich zobrazenie záleží od miesta a situácie, kedy hodnotu zobrazujeme. Zobrazenie môžeme rozdeliť na 2 prípady:

- **bežné zobrazenie hodnoty v prostredí** - je zobrazenie hodnôt v grafe a toolbare (viacej o grafe a toolbare v sekcii Užívateľská príručka),
- **projekcia hodnoty** - je zobrazenie hodnoty po jej vyhodnotení v nástroji zvanom projektor.

Jednoduchý tvar

V grafe a toolbare jednoduchý tvar reprezentujeme ako krabičku v ktorej sa nachádza jednoduchý tvar (obr. 15). Tento tvar sa vykresľuje so všetkými prílušnými transformáciami okrem zmeny veľkosti, takže sa zobrazuje so správnou farbou a rotáciou ale nie veľkosťou, pretože by sa nemusel zobraziť vôbec, alebo naopak jeho veľkosť by bola väčšia ako veľkosť samotnej krabičky.

V projektore jednoduchý tvar zobrazujeme priamo so všetkými transformáciami a bez žiadnych obmedzení (obr. 15).

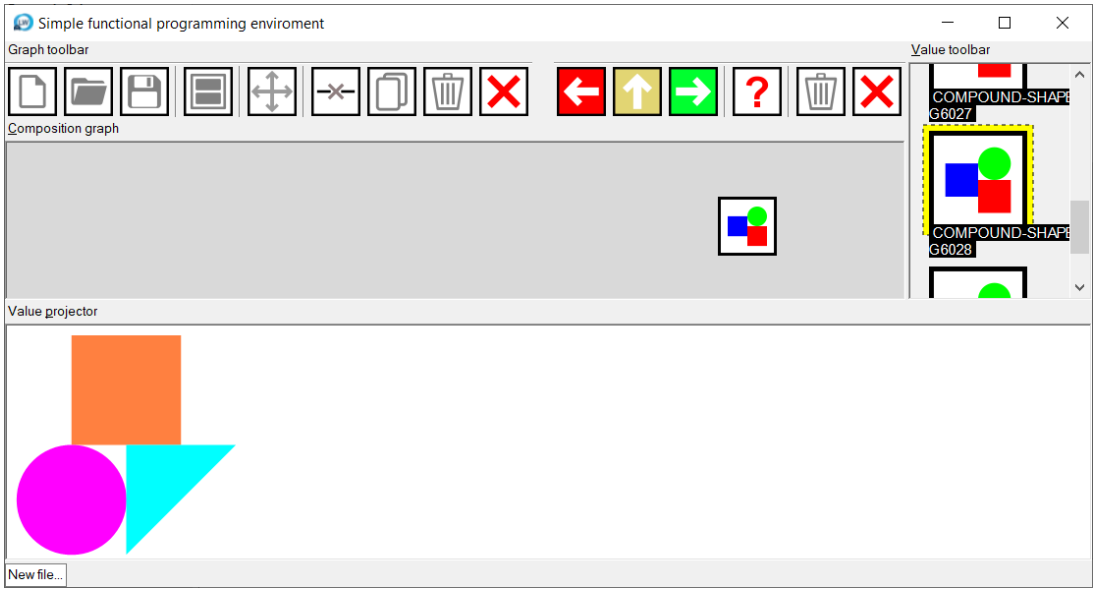


Obr. 15: Porovnanie reprezentácie jednoduchého tvaru

Zložený tvar

V grafe a toolbare je reprezentácia zloženého tvaru do určitej miery limitujúca. Zobrazujeme ho ako krabičku, v ktorej vnútri sa nachádza obrázok reprezentujúci zložený útvar (obr. 16).

V projektore zložený tvar zobrazujeme priamo so všetkými transformáciami a bez žiadnych obmedzení (obr. 16).

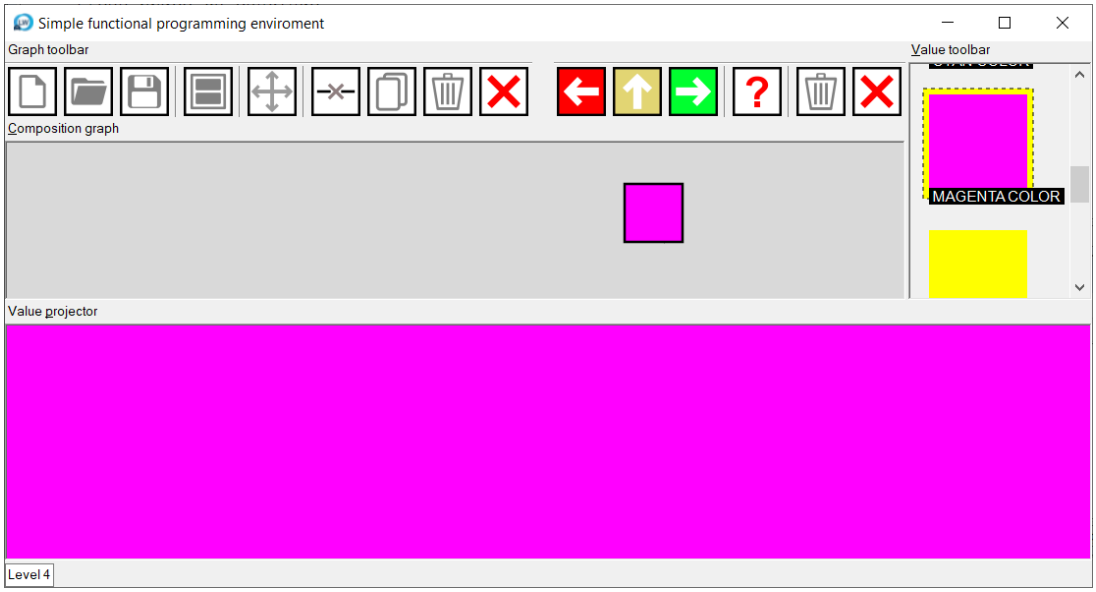


Obr. 16: Porovnanie reprezentácie zloženého tvaru

Farba

V grafe a toolbare farbu reprezentujeme ako krabičku v ktorej vnútri sa nachádza príslušná farba (obr. 17).

V projektore sa pri zobrazovaní farby prefarbí celý projektor príslušnou farbou (obr. 17).



Obr. 17: Porovnanie reprezentácie *farby*

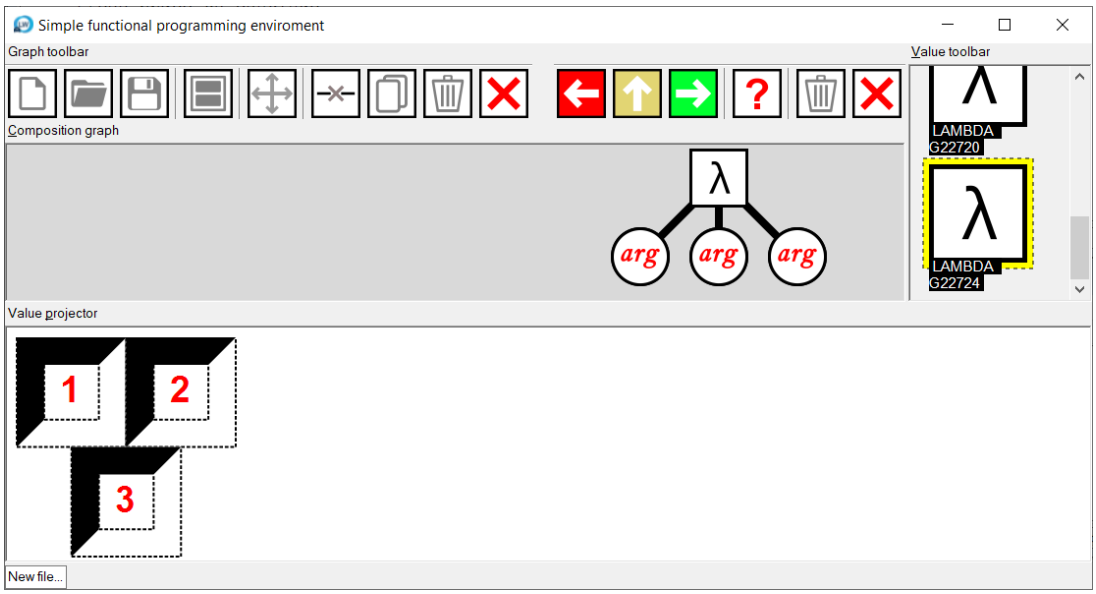
Lambda

Lambda je jediný typ hodnoty, ktorej reprezentácia je odlišná vo všetkých troch častiach prostredia.

V toolbare sa lambda zobrazuje ako krabička, v ktorej vnútri sa nachádza ikona reprezentujúca lambdu (obr. 18). Ak sa však jedná o primitívum tak sa v jej vnútri nachádza ikona, ktorá priamo popisuje, akú akciu daná lambda s jej argumentami vykonáva.

V grafe sa lambda zobrazuje ako krabička ktorá je spojená s guličkami s názvom ”arg” (obr. 18). Sú to jej parametre. Tieto parametre nemôžeme od nej oddeliť. Môžeme ich spájať s inými krabičkami a spojením sa tak stanú jej argumentami.

Zobrazovanie lambdy v projektore je však zložitejšie. Bližšie tento proces popisujeme v sekcii vyhodnocovací proces.



Obr. 18: Porovnanie reprezentácie *lambdy*

4.2.6 Spájanie hodnôt

Pred popísaním vyhodnocovacieho procesu musíme vysvetliť, ako prebieha spájanie hodnôt v grafe. Pojem spájanie hodnôt je značne zjednodušený a reprezentuje viazanie argumentov na parametre lambdy. Užívateľovi stačí vedieť, že na to, aby sa nejaká hodnota (krabička) stala argumentom lambdy, je nutné ju spojiť s parametrom lambdy (guličkou s nápisom ”arg”). V grafe je možné spájať iba krabičky s guličkami. Spájanie prebieha kliknutím na krabičku a za stáleho držania tlačidla *MOUSE1* na myši, je nutné potiahnuť k požadovanému cielu.

V prostredí môžeme spájať krabičky s projektorom a toolbarom. Pri spojení s projektorom dochádza k projekcii hodnoty v krabičke a pri spojení s toolbarom

dochádza k jej pridaniu do toolbaru. Predtým však hodnota vo vnútri krabičky podstúpi vyhodnocovací proces.

4.2.7 Vyhodnocovací proces

Je proces, ktorému podliehajú všetky hodnoty v krabičkách pri ich projekcii alebo pridaní do toolbaru. Vyhodnotenie sa pri rôznych typoch hodnôt líši a taktiež je dôležité v tom, kedy nastáva.

Konkrétne môže nastať v jednom z dvoch prípadov:

(a) **projekcia hodnoty** v projektore,

(b) **pridávanie hodnoty** do toolbaru.

Projekcia hodnoty sa skladá z dvoch krokov:

1. získanie hodnoty vyhodnocovacím procesom,
2. projekcia získanej hodnoty.

Na to aby sme definovali vyhodnocovací proces, je nutné najprv popísať čo je to *výraz*. Za výraz pokladáme:

- hodnotu typu
  - jednoduchý tvar
  - zložený tvar
  - farba
  - lambda
- zložený výraz

Zložený výraz je akékoľvek zoskupenie hodnôt, ktoré sú vzájomne pospájané (jazyk nám dovoluje spojiť iba parametre funkcie s jej argumentami). Výnimku tvoria parametre lambdy, ktoré sú spojené s lambdou na stálo a počítajú sa spoločne ako jedna hodnota.

Po definovaní zloženého výrazu môžeme prejsť k popisu vyhodnocovacieho procesu.

Vyhodnotenie výrazu X

- Ak je X **hodnota** v typu jedného z:
  - jednoduchý tvar
  - zložený tvar
  - farba

– lambda

výsledkom je **hodnota** v.

- Ak je X zložený výraz (**lambda** l s **parametrami** $p_1...p_n$ a aspoň jedným argumentom a) tak:

  1. Získame **hodnoty** $v_c...v_d$ kde $1 <= c, d <= n$, ktoré sú **argumentami** $a_c...a_d$ (opäť vyhodnocovacím procesom)
  2. **Výsledná hodnota** v je aplikácia **lambdy** l na **hodnoty** $v_c...v_d$ z predošlého kroku.

Aby sme popísali celý vyhodnocovací proces, musíme ešte pochopiť, ako funguje aplikácia **lambdy** l

- Ak je l **lambda** s **parametrami** $p_1...p_n$ ktoré sú všetky spojené s argumentami $a_1...a_n$, výsledkom je hodnota, ktorú získame vykonaním **lambdy** l na argumenty $a_1...a_n$.
- Ak je l **lambda** s **parametrami** $p_1...p_n$ ktoré nie sú všetky spojené s argumentami, teda existuje parameter p_m kde $1 <= m <= n$, ktorý nie je spojený so žiadnou hodnotou, výsledkom je **lambda** k, v ktorej tele sú všetky spojené parametre nahradené ich argumentami a jej parametrami zostávajú len tie, ktoré pri jej aplikácii neboli spojené so žiadnymi hodnotami.

Ak použijeme abstrakciu, tak lambda si svoje argumenty *zapamätá* a stanú sa jej súčasťou.

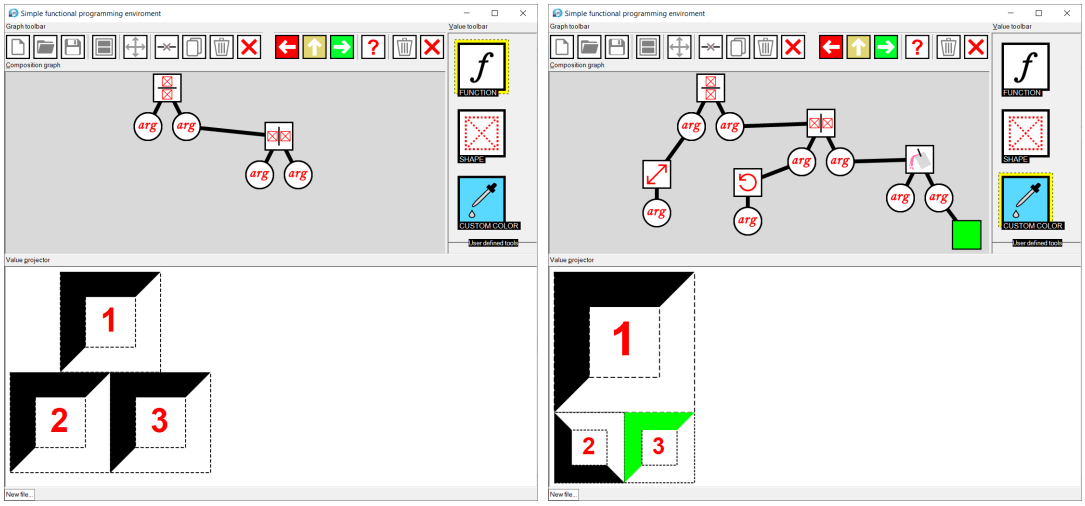
Ak sú všetky obsadené, tak s nimi rovno niečo spraví. Dôležité je pochopiť, že lambda si odkladá výpočet na neskôr. Ak spravíme niekde chybu, nemusíme sa o nej hneď dozvedieť.

Po definovaní vyhodnocovacieho procesu, môžeme konečne popísať premietnutie lambdy. Už vieme, že ak premietame lambdu, ktorá má všetky parametre spojené s argumentami, tak získame priamo hodnotu, ktorá sa skladá z argumentov lambdy. Rozdiel však nastáva, ak nejaký z jej parametrov je prázdny. V tomto prípade sa lambda vyhodnotí klasicky podľa vyhodnocovacieho procesu, ale v ďalšom kroku sa aplikuje na množinu špeciálnych jednoduchých útvarov, ktoré sú typu nešpecifikovaný tvar. Mohutnosť tejto množiny je pritom rovnaká, ako počet prázdnych parametrov lambdy.

Nešpecifikovanný tvar

Je špeciálny jednoduchý tvar, ktorý nie je užívateľovi prístupný. Užívateľ s ním prichádza do kontaktu iba pri projekcii lambdy.

Tento tvar sa zobrazuje so správnou farbou, veľkosťou aj uhlom ktorý by mal tvar, ak by bol argumentom príslušnej lambdy. Dodatočne v sebe nesnie informáciu o svojom poradí z poradia parametrov lambdy (obr. 19).



(a) Projekcia lambdy (b) Projekcia po úprave argumentov

Obr. 19: Znázornenie rozdielov pri projekcii lambdy pred a po úprave argumentov

4.3 Zhrnutie vlastností nášeho jazyka

Ak sa spätne pozrieme na vlastnosti funkcionálnych jazykov preberaných v sekcii funkcionálne programovanie, tak náš jazyk z nich spĺňa práve tieto:

- **funkcie prvej triedy** - s funkciami v našom jazyku pracujeme ako s každou inou hodnotou, môžeme ich vizualizovať aj pridávať do toolbaru,
- **čisté funkcie** - náš jazyk obsahuje čisté funkcie bez vedľajšieho efektu. Jediný vedľajší efekt ktorý nastáva, je pri vizualizácii hodnoty,
- **absencia mutátorov** - hodnoty nášeho jazyka sú nemenné,
- **anonymné funkcie** - náš jazyk je schopný vytvárať anonymné funkcie.
- **parciálna aplikácia** - náš jazyk podporuje parciálnu aplikáciu,
- **lenivé vyhodnocovanie** - náš jazyk vytvára lambdy lenivo, teda vyhodnocuje argumenty až keď je to skutočne nutné,
- **syntax jazyka podporujúca vyhodnocovanie výrazov** - prostredie je koncipované tak, aby bola práca s výrazmi jazyka čo najjednoduchšia.

5 Uživateľská príručka programu

Aplikácia nesie názov *Simple Functional Programming Enviroment*. Jej obsah je napísaný v anglickom jazyku, pretože sa v dnešnej dobe jedná o štandard a nakoľko predovšetkým slúži na prácu s grafickým jazykom, text je iba doplnková informácia pre jednotlivé časti prostredia.

Ovládanie prostredia je intuitívne a jednoduché. Postačuje nám k nemu iba počítačová myš (samozrejme ak nepočítame prácu so súbormi, kde klávesnicu používame k pomenovaniu súborov, no táto funkcionalita sa odporúča pre skúsenejších užívateľov). Pri ovládaní si vystačíme iba s ľavým tlačítkom myši *MOUSE1*. Výnimkou je *SCROLL* ktorý používame pri rolovaní obsahu jednotlivých častí a na priblíženie a oddialenie projektovaných hodnôt.

Klávesnicu je nutné využívať pri práci so súbormi a konkrétne pri ich otváraní a ukladaní. Ostatné ukladanie, ako napríklad priebežné ukladanie levelov je riešené prostredníctvom promptu respektíve dialógového okna a ich používanie zvládnu aj mladšie vekové kategórie detí.

Zdrojový kód aplikácie je napísaný v CAPI (Common Application Programming Interface). Jedná sa o knižnicu od spoločnosti LispWorks Ltd. pre tvorbu GUI. Táto knižnica poskytuje natívnu podporu pre bežne dostupné operačné systému ako Windows, MacOS a rôzne distribúcie Linuxu.

Aj keď bola aplikácia vytváraná v duchu prenositeľnosti na iné operačné systémy, jednotlivé časti ako napríklad súborový prehliadač, môžu na inom operačnom systéme pôsobiť odlišne, ale ich hlavný princíp zostáva zachovaný.

Táto dokumentácia popisuje verziu pre operačný systém Windows.

5.1 Layout

Po spustení aplikácie sa nám otvorí nové okno prostredia. Pozostáva z troch hlavných častí (obr. 21):

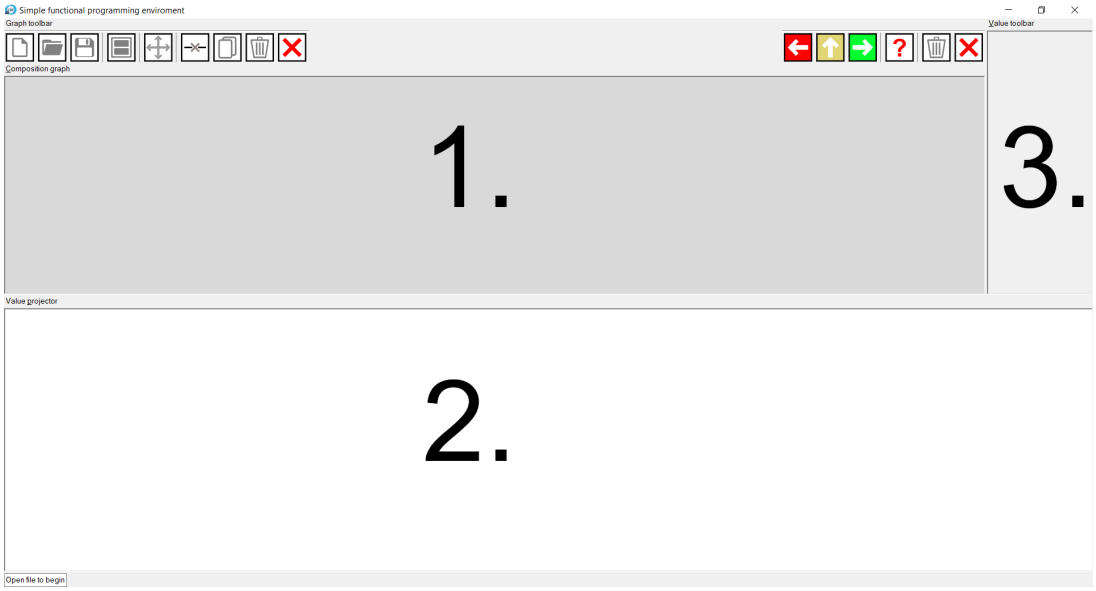
1. graf (composition graph)
2. projektor (value projector)
3. toolbar (value toolbar)

Veľkosť jednotlivých častí aplikácie je počítana dynamicky a nie je daná absolútnymi jednotkami, ale relatívnymi, vzhľadom k veľkosti počítačovej obrazovky.

5.1.1 Graf (composition graph)

Graf je plátno, na ktoré je možné pridávať hodnoty našeho jazyka. Hodnoty sa do grafu pridávajú *drag & drop* gestom a po ich pridaní sa v grafe zobrazí uzol, reprezentujúci pridávanú hodnotu.

Pri popise jazyka, sme používali termín krabička. Týmto termínom sme popisovali reprezentáciu hodnôt jazyka v prostredí, aby bolo pochopenie jazyka



Obr. 20: Screenshot prostredia okamih po jeho otvorení

čo najjednoduchšie. Vo zvyšku užívateľskej príručky tento termín nahradzujeme termínmi *uzol* pre krabičku grafu a *nástroj* pre krabičku v toolbare.

5.1.2 Projektor (value projector)

Projektor je nástroj na projekciu (vizualizáciu) hodnôt s ktorými práve pracujeme. Projektor je schopný premietnuť všetky hodnoty prostredia.

Na ovládanie projektoru sa používa *SCROLL*, ktorým zväčšujeme alebo zmenšujeme jeho obsah. Na pohyb vykresľovaných objektov sa používa *drag & drop* gesto.

5.1.3 Toolbar (value toolbar)

Nachádza sa na pravo od grafu a zoskupuje všetky dostupné hodnoty, s ktorými môže užívateľ v danom okamihu pracovať. Hodnoty sú tu zoradené z hora na dol. Môžeme ich nielen do grafu pridávať, ale aj s nimi pracovať opačným smerom, a teda môžeme hodnoty z grafu pridávať do toolbaru.

Pri popise jazyka, sme používali termín toolbar. Naše prostredie však obsahuje dva toolbary. Jeden hodnotový, ktorý zoskupuje dostupné hodnoty jazyka a jeden pre prácu s grafom. Vo zvyšku užívateľskej príručky tento termín nahradzujeme termínmi *hodnotový toolbar* a *toolbar pre ovládanie prostredia*.

5.2 Toolbar pre ovládanie prostredia

Prostredie obsahuje jeden hlavný toolbar (graph toolbar) ktorý môžeme rozdeliť na päť častí (obr. 22):

1. Manipulácia so súbormi
2. Zobrazenie/ukrytie projektoru
3. Manipulácia s grafom
4. Manipulácia s levelmi
5. Manipulácia s hodnotovým toolbarom

Obr. 21: Ukážka toolbar pre ovládanie prostredia a jeho piatich častí

5.2.1 Manipulácia so súbormi

Manipulácia so súbormi je priamočiara a jednoduchá. Je veľmi podobná ostatným vývojovým prostrediam a editorom. Súbory prostredia majú koncovku *.sfpe*. Na prácu so súbormi používame 3 tlačítka:

- Nový súbor
- Otvorenie existujúceho súboru
- Uloženie súboru

K ukladaniu súboru dochádza aj pri zatváraní prostredia. Pokiaľ je nejaký súbor otvorený, užívateľ je o tom upozornený dialógovým oknom a má možnosť otvorený súbor uložiť. Pokiaľ si zvolí možnosť áno, postupuje sa presne tak isto, ako by stlačil tlačidlo uloženia súboru.

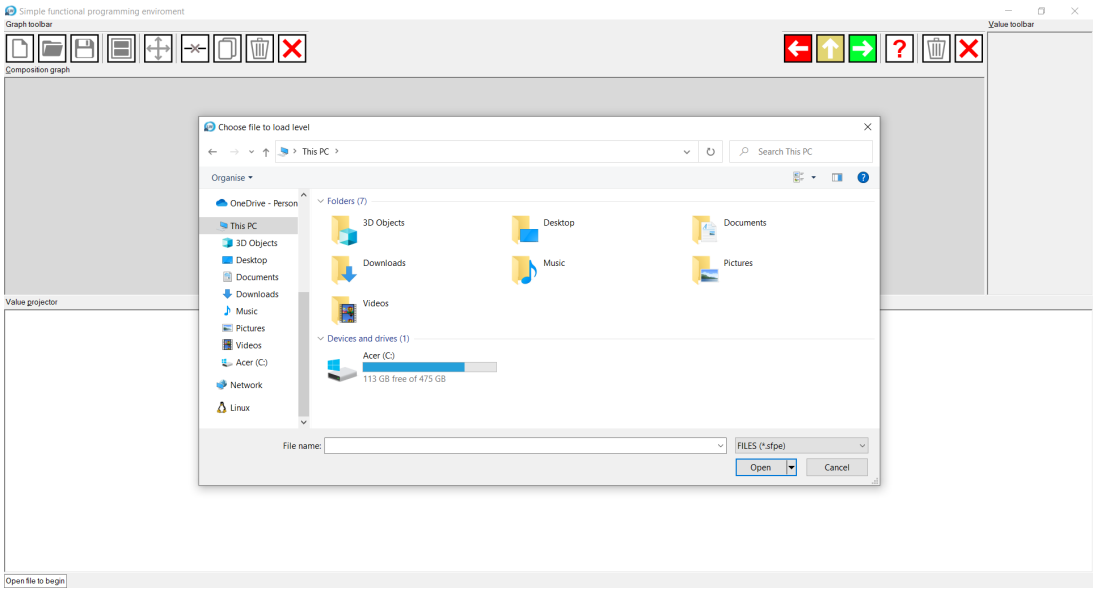
Nový súbor

Slúži na otvorenie nového súboru (obr. 23). Pokiaľ je nejaký súbor momentálne otvorený, prostredie to rozozná a opýta sa na uloženie otvoreného súboru od užívateľa. Prostredie sa vyčistí, z projektoru a grafu sa odstránia všetky uzly a hodnotový toolbar sa naplní východzími nástrojmi.

Obr. 22: Prostredie po otvorení nového súboru

Otvorenie existujúceho súboru

Prebieha klasicky pomocou dialógového okna (obr. 24). Pokiaľ je nejaký súbor momentálne otvorený, prostredie to rozozná a opýta sa užívateľa na uloženie otvoreného súboru.



Obr. 23: Súborový prehliadač pre otvorenie súboru

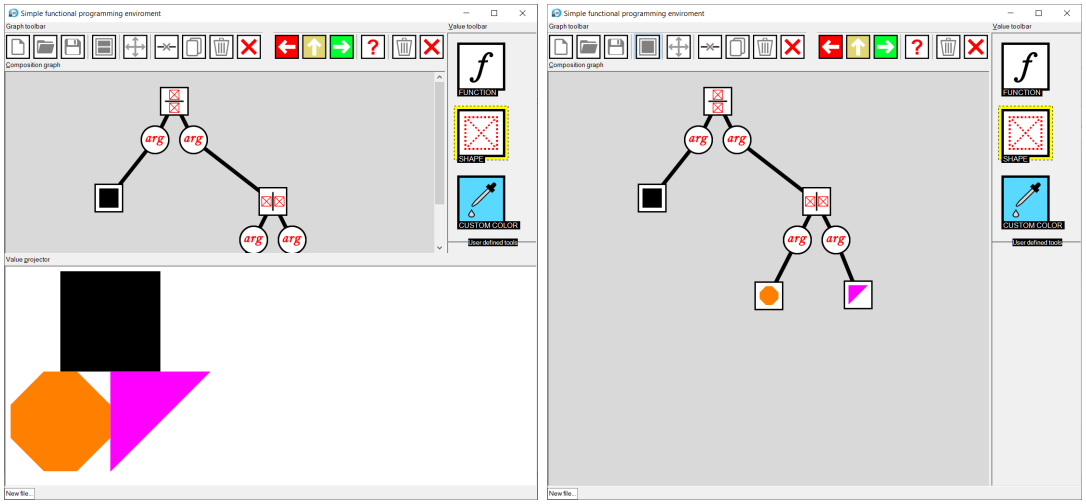
Pri zvolení nepodporovaného formátu súboru dôjde k chybe o ktorej je užívateľ upovedomený správou. Po otvorení súboru sa prostredie vyčistí, graf a hodnotový toolbar sa naplnia hodnotami uloženými v súbore.

Uloženie súboru

Prebieha tak, že pokiaľ otvorený súbor nie je nový, uloženie prebehne do súboru, ktorý bol zvolený pri jeho otváraní. Ak sa jedná o nový súbor, tak sa zobrazí okno so súborovým prehliadačom, slúžiace na uloženie súboru do počítača s uvedením jeho názvu.

Zobrazenie/ukrytie projektora

Ukryje projektor a prípadne ho opätovne zobrazí. Môže byť užitočné pri práci s veľkým množstvom hodnôt v grafe (obr. 25). Projektor si zobrazovanú hodnotu pamätá.



(a) Projektor je zobrazený (b) Projektor nie je zobrazený

Obr. 24: Prostredie s a bez zobrazeného projektoru

5.2.2 Manipulácia s grafom

Na manipuláciu s grafom sa využíva sada tlačidiel:

-  Presúvanie uzlu
-  Odpojenie uzlu
-  Kópia uzlu
-  Odstránenie uzlu
-  Vymazanie celého grafu

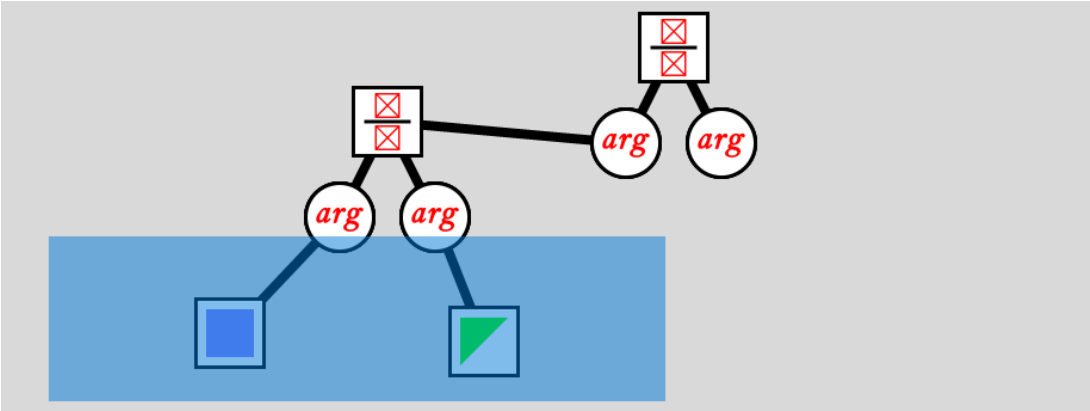
Premiestnenie uzlu

Na premiestnenie uzlu je najprv potrebné stlačiť príslušné tlačidlo a následne je možné presúvať uzol po nástenke *drag & drop* gestom. Vo východzom stave, bez zatlačeného tlačidla presúvania uzlu, nie je možné hýbať s uzlami, pretože prioritne je nastavené, aby po kliknutí na uzol, začala akcia jeho spájania.

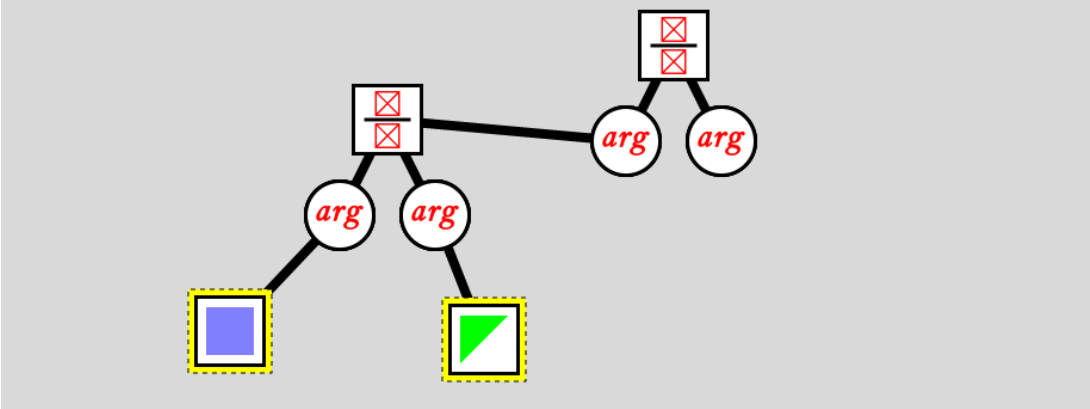
Pred popísaním funkcionality nasledujúcich tlačidiel, je nutné popísať vyznačovanie uzlov na nástenke.

Vyznačovanie uzlov

Nastáva, keď užívateľ klikne ľavým tlačidlom myši *MOUSE1* na nástenku v mieste, kde sa nevyskytuje žiadny uzol. Za stáleho držania zatlačeného tlačidla môže začať hýbať myšou po nástenke. S užívateľovým pohybom sa na nástenke vykresľuje priehľadný svetlo modrý obĺžnik (obr. 26 a).



(a) Proces zvýrazňovania uzlov



(b) Výsledok zvýraznenia uzlov

Obr. 25: Uzly pred a po ich zvýraznení

Po pustení zatlačeného tlačidla myši, dôjde k zvýrazneniu všetkých uzlov nachádzajúcich sa vo zvýrazňovacom obdĺžniku (obr. 26 b).

Odpojenie uzlu

Odpojí všetky vyznačené uzly na nástenke.

Kópia uzlu

Skopíruje všetky vyznačené uzly na nástenke. Skopírované uzly sú posunuté od kopírovaných uzlov. Kopírovanie uzlov je pohodlné v tom, že pri kopírovaní dôjde aj k automatickému spojeniu kopírovaných uzlov, čiže pokiaľ sú nejaké dva uzly spojené a my ich skopírujeme, ich kópie budú spojené.

Odstránenie uzlu

Odstráni všetky vyznačené uzly na nástenke.

Vymazanie celého grafu

Vymaže všetky uzly v grafe.

5.2.3 Manipulácia s levelmi

Interné súbory zabudované do prostredia nazývame levely. Na prácu s nimi slúži sada štyroch tlačidiel:

-  Predošlý level
-  Posledný level
-  Ďalší level
-  Cieľ levelu

Predošlý level

Vráti aktuálny level o jeden späť. Pokiaľ je nejaký level otvorený, užívateľ bude o tom upozornený a má možnosť uložiť otvorený level.

Posledný level

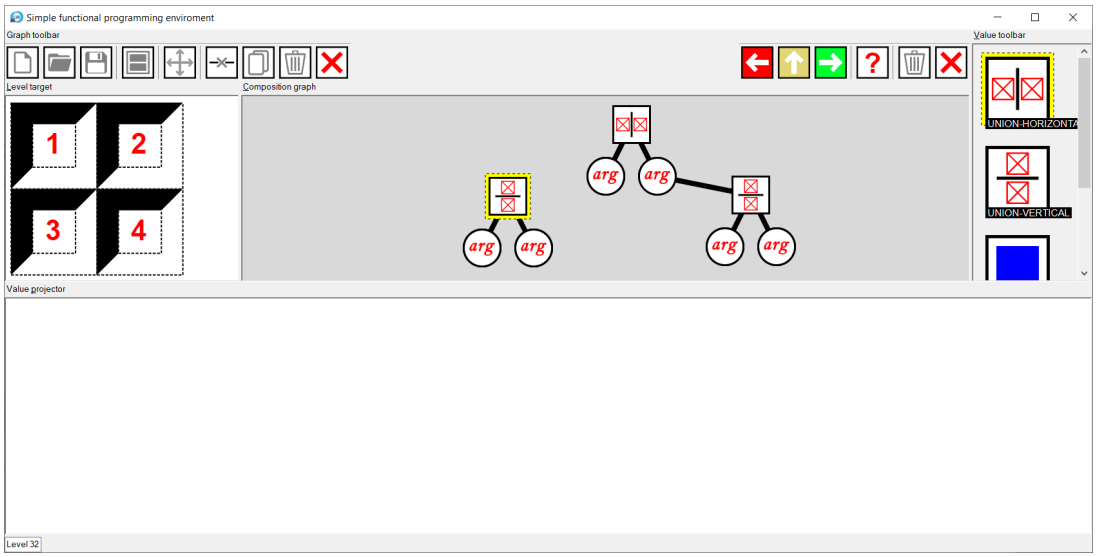
Nastavý aktuálny level na posledný zo všetkých levelov prostredia. Pokiaľ je nejaký level otvorený, užívateľ bude o tom upozornený a má možnosť uložiť otvorený level.

Ďalší level

Posunie aktuálny level o jeden dopredu. Pokiaľ je nejaký level otvorený, užívateľ bude o tom upozornený a má možnosť uložiť otvorený level.

Cieľ levelu

Je tlačidlo, ktoré zobrazí nové okno projektoru v ľavej časti prostredia s tak zvaným cieľom levelu (obr. 27). Je to hodnota, ktorú sa užívateľ má snažiť replikovať. Toto okno slúži ako pomôcka pre riešenie levelu. Táto funkcionalita bola pridaná po odporúčaní pri testovaní detí. Po opätovnom kliknutí na toto tlačidlo sa okno zavrie.



Obr. 26: Prostredie s levelom čislo 32 a otvoreným cieľom levelu

5.2.4 Manipulácia s hodnotovým toolbarom

Na manipuláciu s hodnotovým toolbarom sa využívajú dve tlačidlá:

- Odstránenie vyznačej hodnoty
- Vymazanie všetkých hodnôt

Tieto tlačidlá slúžia na odstránenie užívateľsky definovaných hodnôt. S hodnotami definovanými prostredím alebo jednotlivými levelmi nie je možné manipulovať.
Vyznačenie položiek hodnotového toolbaru je jednoduché a na vyznačovanú hodnotu stačí kliknúť. Hodnota sa vyznačí aj automaticky, pri jej pridaní na plátno. V danom okamihu môže byť vyznačená iba jedna hodnota.

Odstránenie vyznačenej hodnoty

Odstráni vyznačenú hodnotu. Pokiaľ taká hodnota neexistuje, s toolbarom sa nič nestane. Súčasťou akcie je prompt, ktorý slúži na potvrdenie vykonávaných zmien.

Vymazanie všetkých hodnôt

Odstráni všetky hodnoty v toolbare ktoré sú užívateľsky definované. Súčasťou akcie je prompt, ktorý slúži na potvrdenie vykonávaných zmien od užívateľa.

5.3 Ukážokové riešenie levelu

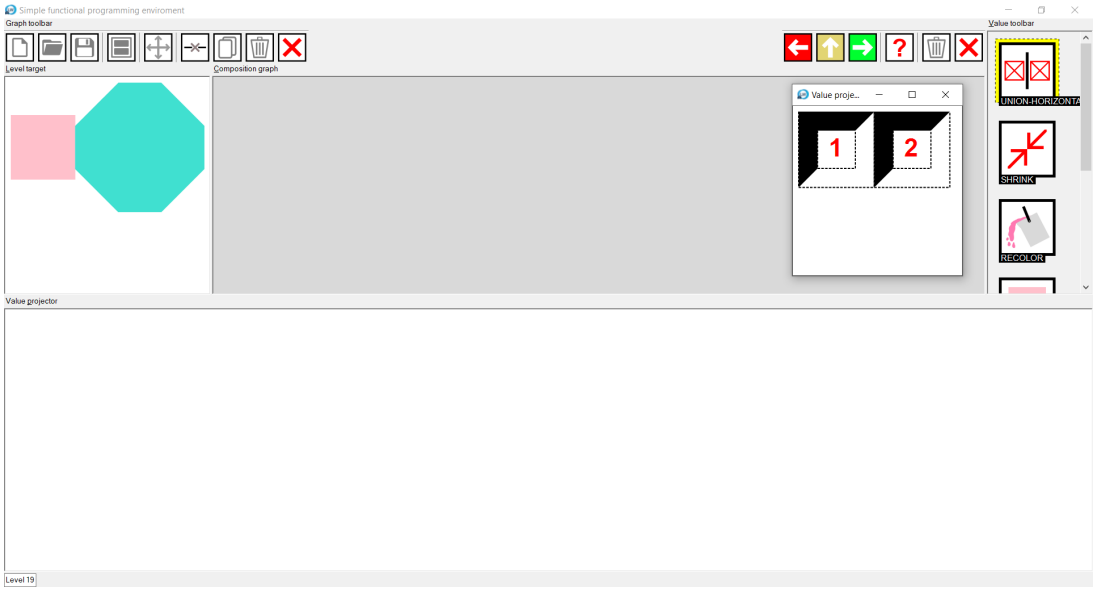
V nasledujúcej časti si ukážeme prácu s prostredím vyriešením levelu číslo 19.

Po otvorení levelu sa nám vyplní obsah prostredia (obr. 28), tak ako pri bežnom otvorení súboru (levely sú interne tiež súbory).

Obr. 27: Level číslo 19 pred jeho riešením

Klikneme na tlačidlo cieľu levelu, aby sme videli, akú hodnotu máme replikovať.

Pre pohodlnú prácu v prostredí, možeme hodnoty v hodnotovom toolbary premietať, bez nutnosti ich pridávať na nástenku. Na hodnotu, ktorú chceme premietnuť, klikneme dva krát po sebe a otvorí sa nám nová inštancia hodnotového projektoru so zobrazenou hodnotou (obr. 29).



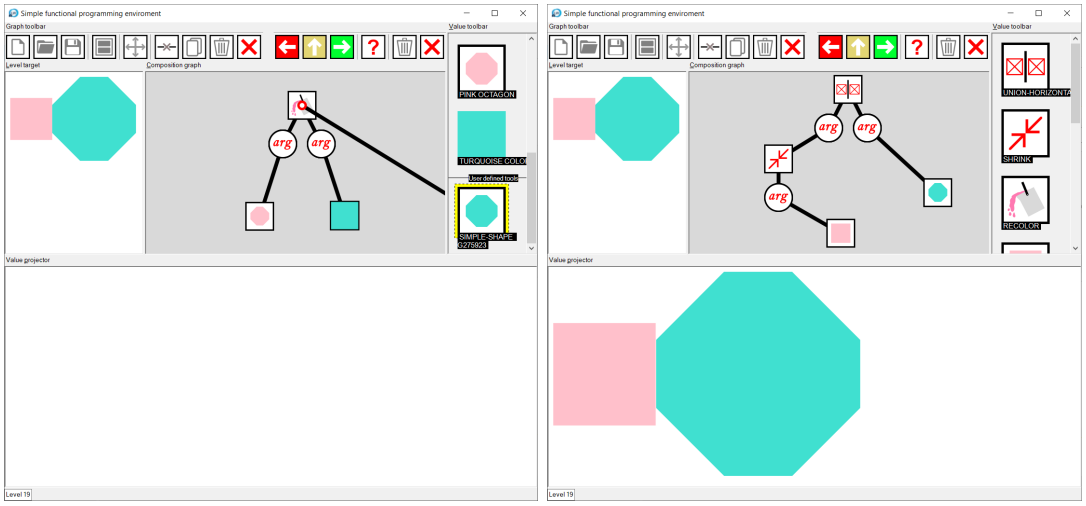
Obr. 28: Prostredie otvorení po cieľu a projekcii prvého itemu priamo z toolbaru

Gestom *drag & drop* pridáme hodnoty do grafu a spojíme argumenty s parametrami.

Riešenie úlohy si rozdelíme na jednoduchšie časti. Najprv si vytvoríme šesťuholník ktorý prefarbíme na tyrkysovo. Tento objekt si odložíme do toolbaru.

Pre sprehladnenie grafu si ho môžeme vyčistiť.

Následne už len vyriešime zvyšok levelu a môžeme prejsť na ďalší (obr. 30). Level samozrejme máme možnosť uložiť.



(a) Medzi krok pri riešení levelu (b) Vyriešenie levelu 19

Obr. 29: Celkové riešenie levelu 19

6 Popis levelov prostredia

Ako sme naznačili v sekcii užívateľská príručka, prostredie obsahuje sadu levelov. Jedná sa o súbory zabudované do prostredia, obsahujúce vopred definované hodnoty, s ktorými môže užívateľ pracovať. Cieľom každého levelu, je replikovať hodnotu, ktorú cieľ levelu obsahuje. Obtiažnosť levelov sa postupne zvyšuje, tak ako aj množstvo vedomostí, potrebných na ich vyriešenie. Užívatelia takto majú možnosť sa postupne zoznámiť s prostredím a prácou v ňom. Cieľom týchto levelov, je postupne naučiť deti princípom funkcionálneho programovania jednoduchou a hravou formou. Prostredie obsahuje 34 levelov.

Výhodnou levelov je to, že interne sú to súbory. Platia pre ne všetky vlastnosti týkajúce sa práce so súbormi. Vieme ich otvárať ale hlavne ukladať. Užívateľ si teda môže odložiť vyriešenie levelu na neskôr. Postup do ďalšieho levelu nie je nijak podmienený, aby nevznikla frustrácia z ich riešenia.

Levely vieme rozdeliť na dva druhy:

- Levely zoznamujúce užívateľa s prácou v prostredí (level 1-11),
- Levely obsahujúce cieľ, ktorý má užívateľ replikovať (level 12-34).

6.1 Levely zoznamujúce užívateľa s prácou v prostredí

Tieto levely slúžia ako ukážka pre prácu v prostredí. Pri výuke detí, sa v tejto časti očakáva pomoc asistenta respektíve vyučujúceho, ktorý je s prostredím plnohodnote oboznámený a bude deťom nápocný.

- **level 0** - Úplne prvý level prostredia bez žiadnych hodnôt. Jedná o prostredie hneď po jeho otvorení. Tu sa deti oboznámia s prostredím, zistia čo je hodnotový toolbar, projektor a graf. Prostredie sa celé vysvetlí bez zbytočných zabiehaní do detailov. Taktiež sa oboznámia s ovládaním levelov. Ako sa presunúť o level späť a o level dopredu. Toolbar prostredia sa v tejto časti ďalej nevysvetľuje.
- **level 1** - Je predstavený základný typ hodnoty jednoduchý tvar. Vysvetlí sa, ako presne sa pridávajú hodnoty z toolbaru a ako ich môžeme vizualizovať v projektore.
- **level 2** - Základná hodnota typu jednoduchý tvar môže mať viacero druhov. Tie sú všetky predstavené v tomto leveli. Predstavený nie je však jednoduchý tvar typu neviditeľný tvar. Deti v tomto levely pochopia, že rôzne hodnoty sa dajú zoskupiť podľa určitej vlastnosti a nepriamo tak pochopia pojem dátový typ.
- **level 3** - Ukážka novej hodnoty typu farba.
- **level 4** - Podobne ako s hodnotou typu jednoduchý tvar, existujú ďalšie druhy farby. Rôznych hodnôt typu farba môže byť však neporovnateľne viac.

- **level 5** - Deti sa v tomto levely naučia novému typu hodnoty lambda, ako aj jemne nazrú do vyhodnocovacieho procesu jazyka.

  Následne krok po kroku sa im predstavia primitíva zmenšenie a zväčšenie.

- **level 6** - Lambdy otočenie vľavo a otočenie vpravo. Deti sa tu naučia ako fungujú rotácie.

- **level 7-9** - Lambdy vertikálne spojene, horizontálne spojenie, prekrytie postupne, v tomto poradí. Vysvetlíme, že tieto lambdy rôzne spájajú jednoduché tvary a vytvárajú z nich zložitejšie (teda hodnoty typu zložený tvar).

  V týchto leveloch deti pochopia, že lambda môže mať aj iný počet parametrov. Taktiež si skúsia rôzne poradia zapojenia argumentov a zistia, že na poradí argumentov lambdy záleží.

- **level 10** - Lambda prefarbenie. Tu sa deťom zíde hodnota typu farba, pretože budú prefarbovať jednoduché tvary.

  Deti sa naučia, že na dátových typoch parametrov lámbd záleží a nemôžeme ich len tak spojiť.

- **level 11** - Predstavenie cielu levela. Tu vysvetlíme, ako môžeme cieľ levelu zobraziť a že v ďalších leveloch sa budú deti snažiť tento cieľ napodobniť.

  Týmto ukončujeme prvú časť levelov o zoznamovaní s prostredím.

6.2 Levely obsahujúce cieľ, ktorý má užívateľ replikovať

Tieto levely sú priamočiarejšie a mnohé nevyžadujú toľko vysvetľovania od vyučujúceho. Tu deti pochopia pridávanie hodnôt do toolbaru, vytváranie vlastnej lambdy alebo jednoduchý tvar druhu neviditeľný tvar.

- **level 12-16** - Obyčajné levely na precvičenie si doposiaľ získaných vedomostí.

- **level 17** - Predstavenie nového druhu jednoduchého tvaru, neviditeľný tvar. V tomto leveli sa objasní (ak si to deti ešte nevšimli), že vertikálne spojenie a horizontálne spojenie centrujú menší z argumentov na stred a prečo sa nám kvôli tomu hodí neviditeľný tvar.

- **level 18-20** - Ďalšie levely na precvičenie si doposiaľ získaných vedomostí.

- **level 21** - Cieľom levelu je vytvoriť list, zreťazením štyroch rovnakých zložených tvarov typu horizontálne spojenie. Je to skvelý príklad vysvetliť deťom, ako sa dá zložitejšia úloha rozdeliť na menšie, jednoduchšie časti a vytvoriť tak kód ktorý je znovu použiteľný. V tomto levely sa vysvetlí pridávanie hodnôt do toolbaru tak ako aj možnosť si ich z toolbaru prezrieť. Zároveň sa uvedie nový typ hodnoty zložený tvar.

- **level 22** - Vytvorenie jednoduchého tvaru druhu hviezda, za pomoci iných tvarov.
- **level 23** - Vytvorenie jednoduchého tvaru druhu šesťuholník, za pomoci iných tvarov.
- **level 24** - Vytvorenie jednoduchého tvaru druhu hviezda s využitím lambdy prekrytie. Pre zvýšenie kontrastu bola použitá šedá farba pre trojuholník.
- **level 25** - Vytvorenie jednoduchého tvaru druhu šesťuholník s využitím lambdy prekrytie. Pre zvýšenie kontrastu bola použitá šedá farba pre trojuholník.
- **level 26** - Vytvorenie pyramídy. V neskoších leveloch sa náchadza úloha vytvoriť lambdu, ktorá vytvára pyramídu.
- **level 27** - Prvý level, v ktorom deti vytvoria vlastnú lambdu. Ukážeme, že lambdy si pamätajú svoje argumenty. Ak máme lambdu dvoch parametrov a len jeden je naviazaný na argument (iba jedna gulička z dvoch je spojená) tak nám vyhodnotením vznikne nová lambda, ktorá si v sebe pamätá tú hodnotu, s ktorou bola spojená a zostanú jej už len tie parametre, ktoré neboli so žiadnou hodnotou spojené. Cieľom levelu je vytvoriť lambdu, ktorá akceptuje jeden argument typu tvar a prefarbí ho na zeleno. Level je doplnený o útvary, na ktorých môžu otestovať deti správnosť vytvorenej lambdy.
- **level 28** - Cieľom levelu je vytvoriť lambdu, ktorá akceptuje jeden argument typu *farba* a tou prefarbí útvar vo vnútri lambdy. Level je doplnený o farby, na ktorých môžu otestovať deti správnosť vytvorenej lambdy.
- **level 29** - Cieľom levelu je vytvoriť lambdu, ktorá akceptuje jeden argument typu *tvar* ktorý prefarbí a zväčší. Samozrejme, táto lambda sa dá vytvoriť aj tak že ho najrpv zväčšíme a potom prefarbíme. Deti takto opäť zistia, že neexistuje iba jeden spôsob na vyriešenie určitého problému.
- **level 30** - Cieľom levelu je vytvoriť lambdu, ktorá akceptuje jeden argument typu *tvar* a otočí ho do prava. Háčik je v tom, že k dispozícii máme iba lambdu *rotácia vľavo*. Deti sa tu naučia, že niektoré lambdy slúžia iba ako programová abstrakcia a že ich vieme vytvoriť aj z iných lámbd.
- **level 31** - Cieľom levelu je vytvoriť lambdu vertikálne spojenie pomocou lambdy horizontálne spojenie. Opäť tu dokazujeme, že niektoré lambdy vieme vytvoriť z už existujúcich.
- **level 32** - Doposiaľ sme pracovali s lambdami, ktoré mali maximálne 2 argumenty. Lambdy však môžu mať ľubovoľné množstvo argumentov (nie celkom, v našom prostredí je maximálny počet argumentov nastavený na 64).

- **level 33** - Cieľom je replikovať lambdu. Jedná sa pravdepodobne o najťažší level z celého prostredia.
- **level 34** - Posledný level prostredia. Jeho cieľom je vytvoriť lambdu, ktorá z jej argumentov vytvorí pyramídu.

7 Programátorská príručka programu

Prostredie bolo vytvorené pomocou softvéru LispWorks®[11]. LispWorks® je cross-platformová implementácia ANSI Common Lisp. Zdrojový kód prostredia je napísaný v programovacom jazyku Common Lisp.

Prečo práve Common Lisp?

Common Lisp[12] je dialekt programovacieho jazyka Lisp. Je známy tým, že je mimoriadne flexibilný, má vynikajúcu podporu pre objektovo orientované programovanie a rýchle prototypovanie. Má tiež mimoriadne výkonný systém makier, ktorý umožňuje prispôsobiť jazyk vašej aplikácii. Nakoľko našim cieľom cielom bolo vytvoriť vlastný jazyk, Common Lisp sa nám pre túto prácu hodil ideálne. S Common Lispom sme pred začatím práce mali aj osobné skúsenosti, kedy sme s ním pracovali naprieč tromi semestrami predmetu *Paradigmata programování* vyučovanom na Univerzite Palackého v Olomnouci.

Prečo práve LispWorks®?

Ako sme už spomínali, LispWorks® je cross-platformová implementácia ANSI Common Lisp. To je obrovská výhoda, pretože Common Lispový kód vyvíjaný v LispWorks® je funkčný (niekedy však s malými zmenami) na platforme *Windows*, *macOS*, rôznych distribúciach *Linuxu* a iných operačných systémoch. Súčasťou LispWorks® je CAPI (The Common Application Programming Interface). Jedná sa prenosný súbor nástrojov pre tvorbu grafických užívateľských rozhraní. Samotné LispWorks® IDE bolo napísané prostredníctvom tohto nástroja. Ponúka kvalitnú dokumentáciu[13] a množstvo vzorových príkladov. S LispWorks® máme taktiež osobnú skúsenosť, z hodín predmetu *Paradigmata programování*. CAPI je postavené na CLOS (Common Lisp Object System)[14]. To znamená, že môže interagovať s nami definovanými objektami, môžeme rozširovať triedy CAPI pomocou vlastných mixinou[15], redefinovať prípadne definovať vlastné metódy a podobne. CAPI nám v tomto zmysle ponúka obrovskú flexibilitu.

LispWorks® je však komerčný software, preto sme pri vývoji použili jeho bezplatnú verziu *LispWorks® Personal Edition*. Prostredie je v tejto verzii v určitých častiach obmedzujúce. Je to najmä v limite veľkosti heapu ale predovšetkým daná verzia neumožňuje export projektov do binárnch súborov. Napriek tomu, LispWorks® Personal Edition nám ponúka všetko, čo pre vývoj tejto práce potrebujeme a preto sme sa rozhodli pre prácu v ňom.

Pri vývoji sme tiež využili knižnicu CL-STORE[16]. Je to balík ponúkajúci

[11] http://www.lispworks.com/products/lispworks.html
[12] https://common-lisp.net/
[13] http://www.lispworks.com/documentation/pdf/lw80/capi-w-8-0.pdf
[14] https://en.wikipedia.org/wiki/Common_Lisp_Object_System
[15] https://en.wikipedia.org/wiki/Mixin
[16] https://cl-store.common-lisp.dev/

serializáciu a deserializáciu objektov Common Lispu z prúdov. Táto knižnica sa nám vynikajúco hodila pri ukladaní levelov do súborov.

V nasledujúcej časti si popíšeme to najzaujímavejšie z implementácie aplikácie.

7.1 Vyhodnocovací proces

Vstupnou bránou vyhodnocovacieho procesu je metóda *eval-graph* (zdrojový kód 3).

```
1 (defmethod eval-graph ((graph composition-graph) node)
2   (eval-subgraph node))
```

Zdrojový kód 3: Definícia metódy *eval-graph*

V grafe reprezentujeme hodnoty našeho jazyka ako uzly. Pri vyhodnocovaní tak môžeme k celému výrazu pristupovať ako ku datovej štruktúre *strom*, kde jeho koreňom je uzol, ktorý sa práve snažíme vyhodnotiť. Tento strom môžeme rekurzívne prechádzať (zdrojový kód 4).

```
1 (defmethod eval-subgraph ((node composition-graph-node))
2   (get-subgraph-expression node))
3
4 (defmethod eval-subgraph ((node composition-graph-node-argument))
5   nil)
```

Zdrojový kód 4: Definície metódy *eval-subgraph*

Definícia metódy *eval-subgraph* pre triedu *composition-graph-node-argument* je pre očistenie prípadu pri pokuse o jeho spojenie s projektorom.

Vyhodnotením podgrafu znamená získať výraz našeho jazyka. To majú nastarosti jednotlivé uzly (zdrojový kód 5).

Získanie výrazu je pre všetky uzly rovnaké, až na uzly reprezentujúce lambdu a parametre. Pokiaľ parameter nie je spojený s iným uzlom, vráti nám unikátny symbol. Získanie výrazu z uzlu, ktorý reprezentuje lambdu, znamená aplikovať danú lambdu na jej argumenty (zdrojový kód 6).

Pri aplikácii lambdy dochádza k vybudovaniu novej lambdy, kde každý výskyt parametru, ktorý je spojený s argumentom je nahradený týmto argumentom. Pokiaľ výsledná lambda, neobsahuje žiadne argumenty, dochádza k vyhodnoteniu jej tela. Vyhodnotenie jej tela prebieha prostredníctvom vyhodnocovacieho procesu v jazyku Common Lisp. Takto sa vyhodnotenie tela lambdy v našom jazyku odkladá až na najneskoršiu možnú dobu.

```
(defmethod get-subgraph-expression ((node composition-graph-node))
  (node-value node))

(defmethod get-subgraph-expression ((node
    composition-graph-node-sfpe-lambda))
  (apply-sfpe-lambda (node-value node)
                (mapcar #'get-subgraph-expression
                      (child-nodes node))))

(defmethod get-subgraph-expression ((node
    composition-graph-node-argument))
  (if (child-nodes node)
     (get-subgraph-expression (first (child-nodes node)))
    (gensym "unspecified")))
```

Zdrojový kód 5: Definície metódy *get-subgraph-expression*

```
(defun do-apply-sfpe-lambda (sfpe-lambda args new-lambda-list
    old-lambda-list)
  (dolist (arg args)
    (process-sfpe-lambda-argument arg new-lambda-list))
  (eval-sfpe-lambda
   `(sfpe-lambda ,(reverse new-lambda-list)
              ,(create-sfpe-lambda-expression
                (sfpe-lambda-lambda-expression sfpe-lambda)
                old-lambda-list
                args))))
```

Zdrojový kód 6: Definícia funkcie *do-apply-sfpe-lambda*

7.2 Ukladanie do súboru

Hodnoty nášeho jazyka reprezentujeme pomocou datovej štruktúry list. CL-STORE ponúka robustnú podporu pre jeho serializáciu. Hodnoty nášeho jazyka sme tak mohli pohodlne ukladať do súboru bez žiadnej výraznej zmeny. Problémom však bolo, že hodnoty v grafe reprezentujeme pomocou objektov. Konkrétne sa jedná o inštancie triedy *composition-graph-node* ktorá je potomkom triedy *capi:drawn-pinboard-object* z knižnice CAPI. CL-STORE síce ponúka serializáciu aj pre objekty, no túto možnosť sme nepovažovali za ideálnu. Ďalšou komplikáciou je, že náš graf obsahuje aj objekty, ktoré priamo nie sú súčasťou nášeho jazyka. Sú to inštancie triedy *composition-graph-edge*, ktoré reprezentujú väzby argumentov na parametre. Trieda *composition-graph-edge* je potomkom triedy *capi:line-pinboard-object* tiež z knižnice CAPI.

Pri serializácii sme postupovali veľmi podobne, ako pri vyhodnocovaní výrazov, kde sme použili abstrakciu stromu. Každý uzol reprezentujúci hodnotu má definovanú metódu *store-graph-node* ktorá slúži na prípavu uzlov pre ich seriali-

záciu (zdrojový kód 7).

```
1  (defmethod store-graph-node ((node composition-graph-node))
2    (capi:with-geometry node
3      (list (node-value-type node)
4            (node-value node)
5           capi:%x% capi:%y%
6           (icon-name node))))
7
8  (defmethod store-graph-node ((node composition-graph-node-argument))
9    (when-let (child-nodes (child-nodes node))
10     (store-graph-node (first child-nodes))))
11
12 (defmethod store-graph-node ((node
       composition-graph-node-sfpe-lambda))
13   (nconc (call-next-method)
14          (list (mapcar #'store-graph-node
15                        (child-nodes node)))))
```

Zdrojový kód 7: Definície metódy *store-graph-node*

Každý uzol ukladáme vo formáte (*node-value-type node-value x y icon-name child-nodes*) kde:

- *node-value-type* - dátový typ hodnoty, ktorú uzol reprezentuje,
- *node-value* - hodnota, ktorú uzol reprezentuje,
- *x y* - súradnice uzlu v grafe,
- *icon-name* - názov ikonu ktorú uzol zobrazuje,
- *child-nodes* - list potomkov uzlu v ronakom formáte. Môže byť aj *NIL* ak uzol nemá potomkov.

Uzly v grafe rekurzívne prechádzame a získame list, kde každá položka reprezentuje jeden zložený výraz, ktorý je opäť vo formáte listu. Týmto spôsobom sa zbavíme nutnosti ukladania hrán do súboru. Pred ukladaním je ešte nutné zistiť, ktoré uzly sa nachádzajú na vrchole zloženého výrazu. To je veľmi jednoduché, pretože to sú všetky uzly, ktoré nemajú rodiča (zdrojový kód 8).

```
1  (defmethod prepare-to-store ((graph composition-graph))
2    (do-prepare-to-store graph (capi:layout-description graph)))
3
4  (defmethod do-prepare-to-store ((graph composition-graph) nodes)
5    (when-let (node (car nodes))
6      (if (and (typep node 'composition-graph-node)
7               (not (parent-node node)))
8          (cons (store-graph-node node)
9                (do-prepare-to-store graph (cdr nodes)))
10       (do-prepare-to-store graph (cdr nodes)))))
```

Zdrojový kód 8: Definícia metódy *prepare-to-store*

Pri opätovnom otváraní súboru tak môžeme len prechádzať jednotlivé výrazy, ktoré spracúvame rekurzívne a postupne spájame s potomkami, tak, ako sú uložené. Pri pridávaní využívame metódy *make-and-add-node*, ktorá vytvorí uzol a pridá ho do grafu z reprezentácie v súbore a následne *connect-graph-nodes* pre ich spojenie.

Ukladanie hodnôt v hodnotovom toolbare je o niečo jednoduchšie, pretože sme nemuseli riešiť žiadne zapojenie hrán no v princípe je veľmi podobné a taktiež využívame pri ukladaní reprezentáciu pomocou listu.

8 Testovanie detí

Neoddeliteľnou súčasťou tvorby programu je jeho testovanie. Vychádzajúc z charakteru našej práce, ako subjekty pre testovanie prostredia boli vybrané dieťa mladšieho a staršieho školského veku. Obe deti boli v príbuzenskom vzťahu (súrodenci) a v príbuzenskom vzťahu s osobou, ktorá realizovala následné testovanie (autor práce). Testovanie prebiehalo oddelene a v domácom prostredí detí (v ich izbách) v popoludňajších hodinách, približne tri hodiny po návrate zo školy. V miestnosti, kde prebiehalo testovanie, sa okrem dieťaťa a osoby, ktorá riadila testovanie, nikto nenachádzal. Rodičia boli v čase testovania na rovnakom mieste (doma), avšak v inej miestnosti.

Profil testovaných detí:

- **Dieťa 1** - dievča vo veku 11 rokov, navštevuje 5. ročník základnej školy (2. stupeň základnej školy). Dieťa absolvuje 1 hodinu informatiky týždenne (v súčasnosti tretí rok).
- **Dieťa 2** - chlapec vo veku 8 rokov, navštevuje 2. ročník základnej školy (1. stupeň základnej školy). Dieťa zatiaľ predmet informatika nenavštevuje.

Obe deti študujú na rovnkej základnej škole (ZŠ Nábrežie Mládeže, Nitra, Slovenská republika).

8.1 Priebeh testovania dieťaťa 1

Level 0 - Dieťa bolo oboznámené s prostredím. Pri opise prostredia sme postupovali nasledovne: *"Úplne na pravo máme okienko (toolbar) ktoré bude mať v sebe krabičky (hodnoty). Tie budeme vedieť hocikam na nástenku (graf) prilepiť, akoby sme lepili štítky na chladničku. Tie si vieme pozrieť ako v skutočnosti vyzerajú. Na to máme dole biele okienko (projektor). V pravo hore máme tri tlačítka (tlačitka na manipuláciu s levelmi). Tie nám slúžia na zmenu levelu (mnohé deti poznajú aj termín* level *z počítačových hier). Zelené tlačítko je pre level dopredu, červené pre level dozadu. teraz môžeš prejsť na ďalší level."*

Level 1, 2 - Dieťaťu bol predstavený typ hodnoty jednoduchý tvar. Pri vysvetľovaní bola použitá abstrakcia krabičiek: *"Každá krabička má s sebe niečo (nejakú hodnotu). Krabičky môžeme pridávať a prezerať si, čo obsahujú."*

Dieťatu sme predviedli názornú ukážku, ako pridávame hodnoty do grafu a ako ich môžeme vizualizovať. Následne sme im nechali čas, nech si to vie vyskúšať.

Tu sa nám ukázala prvá nedokonalosť prostredia. Pôvodne bolo gesto pridávania hodnôt tak, že bolo nutné hodnotu v toolbare označiť a potom tlačidlom myši *MOUSE1* pridať na nami zvolené miesto v grafe. Napriek tomu, že sa gesto *drag & drop*, môže zdať užívateľsky menej prívetivé, dieťa sa ako prvé pokúsilo pridať hodnotu touto cestou. Po uvážení sme *drag & drop* implementovali aj v našom prostredí.

Level 3-4 - Dieťa bolo oboznámené s novým typom hodnoty farba. Pri vysvetľovaní bola použitá abstrakcia: *"Existujú krabičky, ktoré v sebe nemajú iba štvorček alebo kruh ale v krabičke môže byť aj farba."*

Level 5 - Dieťaťu bol predstavený nový typ hodnoty lambda. Pri popise toho, čo je lambda, sme postupovali takto: *"Je to špeciálna krabička, ktorá je vždy spojená s guličkami. Tieto guličky sa dajú spojiť s inými krabičkami. Lambda zoberie krabičky, ktoré sú s jej guličkami spojené a niečo s nimi urobí. To čo s nimi urobí, závisí od toho o akú lambdu sa jedná."* Ďalej sme vysvetlili, že lambda sa vyhodnocuje rovnako ako ostatné krabičky a to spojením s projektorom.

Level 6-9 - Dieťaťu boli postupne ukázané všetky primitíva prostredia.

Level 10 - Dieťa bolo oboznámené s lambdou prefarbenia. Tu sa nám potvrdila, správnosť návrhu nášeho jazyka. Dieťa bolo schopné prefarbiť príslušný tvar v leveli bez toho, že by sme mu to nejak, museli vysvetliť. Po zístení príčiny, ako je môžné, že to vedelo spraviť samo, povedalo, že túto akciu pozná z grafického editora.

Level 11 - Dieťatu sme vysvetlili, ako sa pracuje s cielom daného levelu.

Level 12-20 - V týchto leveloch už nebola toľko potrebná náša asistencia, s výnimkou levelu 17. Dieťa pracovalo v prostredí samostatne. Po vyriešení jednotlivých levelov počkalo na kontrolu správnosti riešenia a následne pokračovalo ďalej.

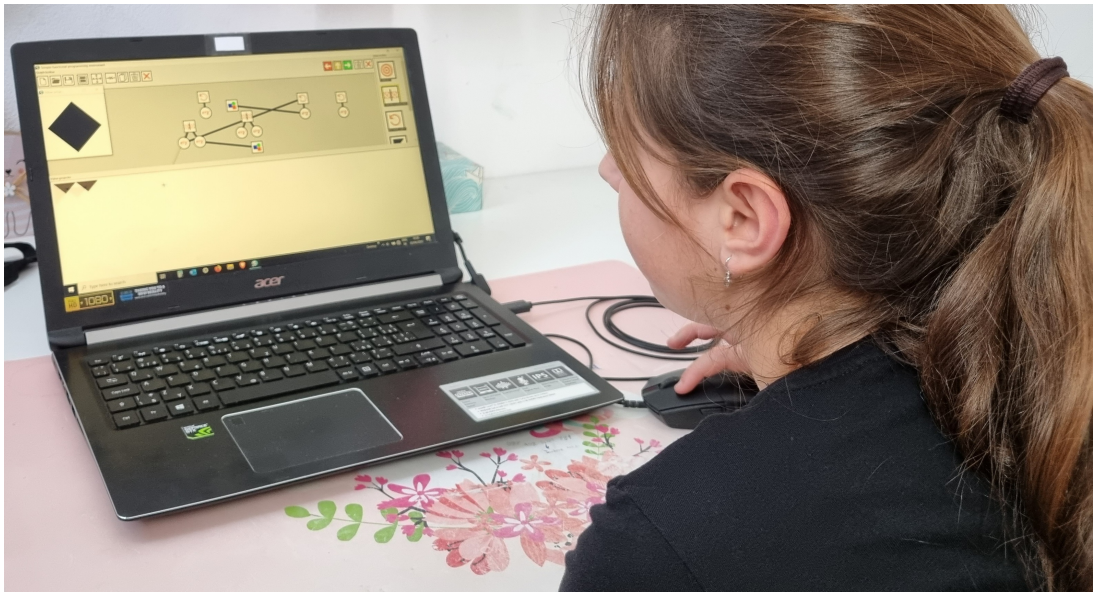
level 21 - Dieťa sa tu naučilo ako funguje pridávanie hodnôt do toolbaru, tak ako aj možnosť si ich z toolbaru prezrieť. Naučí sa aj nový typ hodnoty zložený tvar.

level 22-26 - Dieťa postupne riešilo všetky úlohy. Pri niektorých sa viacej natrápilo, ale nakoniec ich zvládlo všetky (obr. 31).

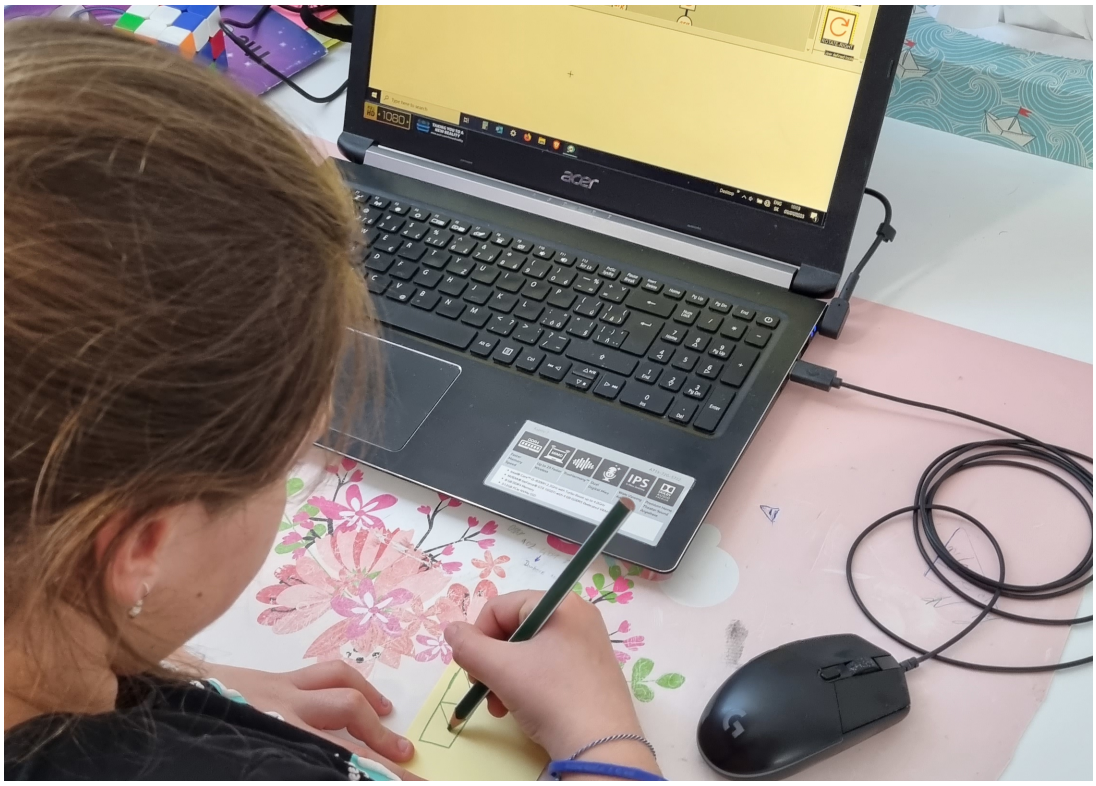
level 27 - Prvý level, kde dieťa vytvorilo vlastnú lambdu. Pri vysvetľovaní sme postupovali tak, že najprv sme dieťaťu objasnili, ako si môže premietnuť lambdu v projektore. Najprv bez žiadnych argumentov a potom s niektorými argumentami. Objasnili sme, že tak ako sme si mohli pridávať vlastné tvary do toolbaru, tak môžeme aj lambdy. A nakoniec sme mu ukázali, že lambdy si pamätajú svoje argumenty. Z tejto funkcionality dieťa neskrývalo nadšenie.

level 28-34 Dieťa postupne vyriešilo všetky levely až na level 33. Jedná sa však o skutočne ťažký level a nevyžadovali sme nutne od neho riešenie.

Pri riešení úloh s lambdami nás prekvapilo, že dieťa samo od seba zobralo papier a snažilo sa nakresliť si riešenie (obr. 32). Bolo plne zainteresované a práca v prostredí ho nesmierne bavila.



Obr. 30: Dieťa 1 pri riešení levelu 22



Obr. 31: Dieťa 1 pri náčrte riešenia

Pri testovaní sme progresívne obmieňali terminológiu za zložitejšiu. Výrazy ako *krabičky* postupne nahradili *hodnoty*, *guličky parametre* a *krabičky spojené s*

guličkami argumenty. Dieťa terminológiu akceptovalo a spätne sme sa k predošlím termínom už nevracali.

Konverzácia s dieťaťom po ukončení testovania: Autor práce: *"Takže, ty si celý čas pracovala s nemennými hodnotami, argumenty si viazala na parametre, vyhodnocovala si zložené výrazy, vytvárala si anonymné funkcie a ani si o tom nevedela."* a dieťa odpovedalo: *"Ale nie, ja som predsa robila iba s myškou!"*.

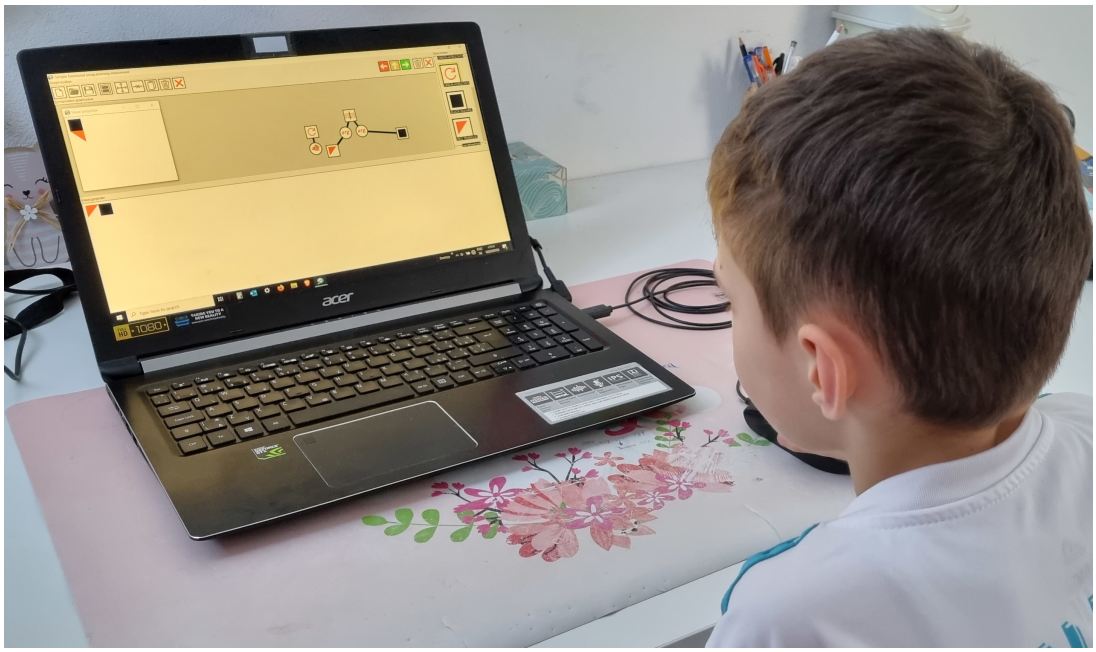
Celkový priebeh testovania bol 1 hodina a 15 minút. Po ukončení testovania dieťaťa 1 nasledovalo testovanie dieťaťa 2.

Dieťa 1 bolo úlohami natoľko zaujaté, že odišlo do inej miestnosti, kde na papier vyriešilo jednu z úloh a neskôr sa vrátilo s lepším riešením.

8.2 Priebeh testovania dieťaťa 2

Testovanie prebiehalo viac menej rovnako (obr. 33) a poukazujeme iba na rozdieli alebo zaujímavosti, na ktoré sme narazili.

Level 1 - Tak ako dieťa 1 aj dieťa 2 sa pokúsilo najprv hodnotu z toolbaru pridať pomocou gesta *drag & drop*.

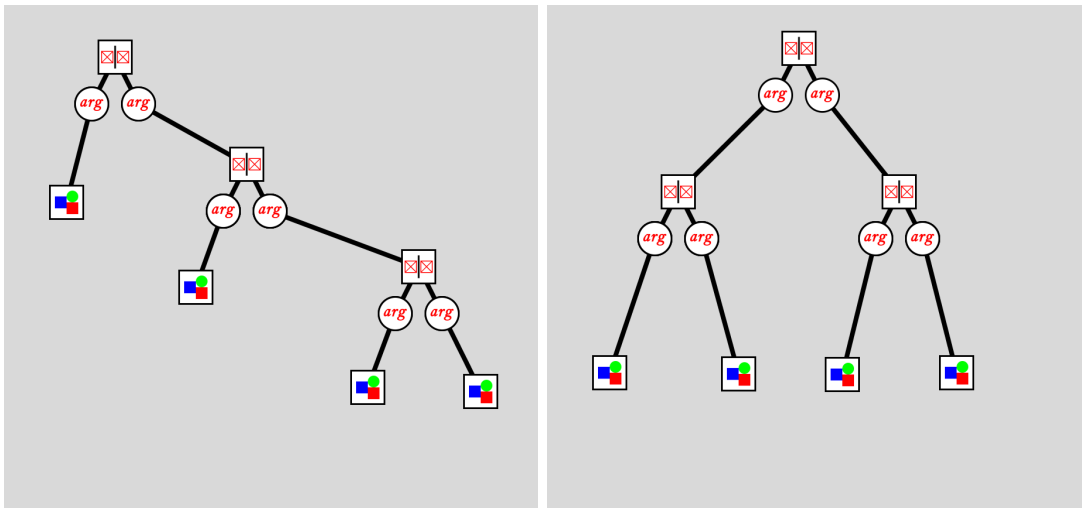


Obr. 32: Dieťa 2 počas testovania

Level 18 - Pri riešení tohto levelu sme dieťaťu poradili aby skúsilo vymeniť poradie argumentov. Pojmom rozumel a následne zmenil poradie argumentov.

Level 21 - Cieľom levelu je vytvoriť sériu štyroch po sebe opakujúcich sa zložených útvarov. Za zmienku určite stojí, že zatiaľ čo dieťa 1, si úlohu rozdelilo na

4 rovnaké časti a tie nakoniec postupne pospájalo, dieťa 2 si rozdelilo úlohu na dve polovice, ktoré spojilo dokopy (obr. 34).



(a) Riešenie levelu 22 dieťaťom 1 (b) Riešenie levelu 22 dieťaťom 2

Obr. 33: Porovnanie riešení levelu 21 testovaných detí

Level 22 - Po vyriešení levelu 21 nám však dieťa oznámilo, že už ho to nebaví, na čo sme zareagovali jeho pochvalou a ukončili sme testovanie.

Celkový priebeh testovania bol 45 minút.

Záver

Počítačové programy vytvorené pomocou funkcionálnej programovacej paradigmy, sa skladajú z vyhodnocovania výrazov. Naproti tomu iné paradigmy, odrážajú princípy, na ktorých je digitálny počítač postavený.

Výuka programovania má svoje opodstatnenie nie len u dospelých ale predovšetkým aj u detí. Táto výuka môže byť pre ne veľkým benefitom do budúcna a zo znalostí, ktoré nadobudnú programovaním, budú čerpať celý život, bez ohľadu na profesiu, ktorú si v budúcnosti vyberú. Prostredí pre výuku funkcionálneho programovania však nie je mnoho.

Pri tvorbe prostredia je nutné vymyslieť vhodnú abstrakciu, pomocou ktorej by sa deti hravou formou mohli naučiť základom funkcionálneho programovania a postupne tak odkrývali jeho možnosti. Pri tvorbe sme sa inšipirovali hrou skladania kociek, ktorú sme previedli do 2D podoby. Aby sme znížili vstupnú hranicu veku pre prácu v prostredí, naše prostredie spĺňa niekoľko vlastností:

- je na používanie čo najjednoduchšie,
- jeho jazyk je vizuálny,
- obmedzená znalosť čítania a písania je na prácu v ňom dostatočná.

Po vytvorení prostredia, ktorá reflektuje vyššie spomenuté kritériá, sme prešli na testovanie detí. Pri testovaní sa nám potvrdila korektnosť našeho návrhu, kedy sme boli schopní vhodne vysvetliť základné princípy funkcionálneho programovania a objasniť tak pojmy ako: *hodnoty*, *argumenty*, *funkcie*, *aplikácia funkcie*, *anonymné funkcie* alebo *parciálna aplikácia*.

Prostredie má značný potenciál do budúcnosti a môže tak byť rozšírené o ďalšiu funkcionalitu, ako sú funkcie vyššieho rádu alebo ďalšie primitíva. Jeho zmysel vidíme aj pri výuke programovania na školách, kedy môže byť uvedené do procesu výuky programovania už hneď zo začiatku. Prostredie nevyžaduje prílišnú znalosť čítania a písania. V testovaní sme boli schopní naučiť princípom (až na anonymné funkcie) aj dieťa, ktoré doposiaľ nenavštevuje predmet informatika na základnej škole.

Z vyššie uvedeného môžeme konštatovať, že prostredie sa nám podarilo vytvoriť a úspešne sme tak predstavili mnohé princípy funkcionálneho programovania deťom.

Conclusions

Computer programs created using the functional programming paradigm, consist of evaluating expressions. Other paradigms, in contrast, reflect the principles on which a digital computer is built.

Teaching programming has its justification not only to adults but also and especially to children. They can benefit greatly in the future and will draw on the knowledge they gain from programming throughout their lives, regardless of the profession they choose in the future. However, there are not many environments for learning functional programming.

When designing an environment, it is necessary to devise a suitable abstraction through which children can learn the basics of functional programming in a playful way and gradually discover its possibilities. We were inspired by the game of stacking cubes, which we converted into a 2D form. In order to lower the entry age threshold for working in the environment, our environment satisfies several characteristics:

- is as simple as possible to use,
- it's language is visual,
- limited reading and writing skills are sufficient to work in it.

After creating an environment that reflects the above criteria, we moved on to testing the children. During testing, we confirmed the correctness of our design, where we were able to adequately explain the basic principles of functional programming, clarifying concepts such as *values*, *arguments*, *functions*, *function application*, *anonymous functions*, or *partial application*.

The environment has considerable potential for the future and can thus be extended with additional functionality, such as higher-order functions or other primitives. We can also see its sense in teaching programming in schools, where it can be introduced into the process of teaching programming right from the start. The environment does not require too much reading and writing. In testing, we were able to teach the principles (except for the anonymous functions) even a child who has not yet attended a computer science course in primary school.

From the above, we can conclude that we were able to create environment that successfully introduced many of the principles of functional programming to children.

A Obsah přiloženého dátového média

bin/
: Tento priečinok je prázdny. Inštrukcie, ako spustiť vývojové prostredie, nájdete v súbore README.txt, ktorý sa nachádza v koreňovom priečinku.

doc/
: Teoretickú časť bakalárskej práce nájdete v súbore kidiplom.pdf. Súbor doc.zip obsahuje zdrojový text teoretickej časti a vložené obrázky.

src/
: Tento priečinok obsahuje zdrojové texty nami vytvoreného prostredia spoločne s vytvorenými levelmi.

README.txt
: Inštrukcie pro inštálaciu a spustenie vývojového prostredia.

install/
: Inštalátor LispWorks® Personal Edition pre operačný systém Microsoft Windows.

Literatúra

[1] HUNT, Andrew; THOMAS, David. Pragmatic Programmer, The: From Journeyman to Master. [online]. Publisher: Addison Wesley. First Edition October 13, 1999 [cit. 2023-03-20]. ISBN 0-201-61622-X, 352 strán. Dostupné z: https://www.cin.ufpe.br/~cavmj/104The%20Pragmatic%20Programmer,%20From%20Journeyman%20To%20Master%20-%20Andrew%20Hunt,%20David%20Thomas%20-%20Addison%20Wesley%20-%201999.pdf

[2] PETŘÍČEK, Tomáš; SKEET, Jon. Real-World Functional Programming: With Examples in F# and C#. [online]. Publisher: Manning Publications Co. 2009-11-30. [cit. 2023-03-20]. ISBN 978-1933988924. 500 strán. Dostupné z: https://doc.lagout.org/programmation/Functional%20Programming/Functional%20Programming%20For%20The%20Real%20World.pdf

[3] NØRMARK, Kurt. Functional programming in Scheme With Web Programming Examples. Department of Computer Science, Aalborg University, Denmark [online]. 2003-9. [cit. 2023-03-21]. Dostupné z: https://homes.cs.aau.dk/~normark/prog3-03/pdf/all.pdf

[4] HARRISON, John. Introduction to Functional Programming. Department of Computer Science and Technology [online]. 1997-12-03. [cit. 2023-04-02]. Dostupné z: https://www.cl.cam.ac.uk/teaching/Lectures/funprog-jrh-1996/all.pdf

[5] KRUPKA, Michal. Paradigmata programování 1 ◇ poznámky k přednášce 3. Rekurze 1. 2019-10-16. [cit. 2023-04-03].

[6] KRUPKA, Michal. Paradigmata programování 1 ◇ poznámky k přednášce 9. Funkce vyššího řádu. 2019-12-03. [cit. 2023-04-03].

[7] KRUPKA, Michal. Paradigmata programování 1 ◇ poznámky k přednášce 8. Vedlejší efekt. 2019-12-03. [cit. 2023-04-03].

[8] HASKELL.ORG. Anonymous function. HaskellWiki [online] 2021-04-12. [cit. 2023-04-10]. Dostupné z: https://wiki.haskell.org/Anonymous_function

[9] HASKELL.ORG. Currying. HaskellWiki [online] 2020-03-22. [cit. 2023-04-10]. Dostupné z: https://wiki.haskell.org/Currying

[10] HASKELL.ORG. Partial application. HaskellWiki [online] 2020-05-13. [cit. 2023-04-10]. Dostupné z: https://wiki.haskell.org/Partial_application

[11] KRUPKA, Michal. Paradigmata programování 2 ◇ poznámky k přednášce 6. Líné vyhodnocování, přísliby a proudy. 2020-03-26. [cit. 2023-04-15].

[12] LONSDORF Brian. Professor Fisby's Mostly Adequate Guide to Functional Programming ResearchGate | GitBook - Where technical teams document. [online]. 2015. [cit. 2023-04-15]. Dostupné z: https://mostly-adequate.gitbook.io/mostly-adequate-guide/ch03

[13] WIKIPEDIA. Pure function | Wikipedia, the free encyclopedia [online]. 2023-04-09. [cit. 2023-04-26]. Dostupné z: https://en.wikipedia.org/wiki/Pure_function

[14] PARLANTE Nick. Decomposition & Style | Stanford Computer Science. [online]. 1996. [cit. 2023-04-28]. Dostupné z: https://cs.stanford.edu/people/nick/compdocs/Decomposition_and_Style.pdf

[15] WILLIAMS Cassidy. Functional Programming 101 | GitHub: Let's build from here. [online]. 2022-07-12. [cit. 2023-04-28]. Dostupné z: https://github.com/readme/guides/functional-programming-basics

[16] OUTRATA Jan. Paralelní programování | Phoenix. [online]. 2007-02. [cit. 2023-04-26]. Dostupné z: https://phoenix.inf.upol.cz/~outrata/courses/pp4_pp/texts/parallel.pdf

[17] ODERSKY Martin. "Working Hard to Keep It Simple OSCON Java 2011 | YouTube. [online]. 2011-07-26. [cit. 2023-04-27]. Dostupné z: https://www.youtube.com/watch?v=3jg1AheF4n0

[18] ERLANG.ORG. Processes | Index - Erlang/OTP. [online]. 2023-04. [cit. 2023-04-27]. Dostupné z: https://www.erlang.org/doc/reference_manual/processes.html

[19] MIŠKERÍK, Martin. Digitálny svet stále trápi nedostatok pracovníkov, čo vyvoláva tlak. Veľké problémy má aj Slovensko. Týždenník o ekonomike a podnikaní - TREND.sk [online]. 2022-09-25. [cit. 2023-04-22]. Dostupné z: https://www.trend.sk/trend-archiv/it-firmy-hladaju-ihlu-kope-sena-sikovni-ludia-odchadzaju-skoly-neprodukuju-novych

[20] MITRO, Matúš. Slovensko akútne potrebuje IT-čkárov. Aktuálne je najviac pracovných ponúk v histórii. FonTech.sk - Spravodajstvo o technológiach, elektromobilite, smartfónoch či vesmíre a vede. FonTech.sk je najlepšie miesto pre všetkých nadšenov inovácií. [online]. 2021-04-16. [cit. 2023-04-22]. Dostupné z: https://fontech.startitup.sk/slovensko-akutne-potrebuje-it-ckarov-aktualne-je-najviac-pracovnych-ponuk-v-historii/

[21] W. SEBESTA, Robert. CONCEPTS of Programming Languages. [online]. Publisher: Pearson. 10th edition. 2012. [cit. 2023-04-22]. ISBN 978-0-13-139531-2. Dostupné z: https://www.ime.usp.br/~alvaroma/ucsp/proglang/book.pdf

[22] STACKOVERFLOW. 2020 Developer Survey. Stack Overflow Insights - Developer Hiring, Marketing, and User Research [online]. 2022. [cit. 2023-04-22]. Dostupné z: https://insights.stackoverflow.com/survey/2020

[23] STEPHANE Chaudron; DI GIOIA Rosanna; GEMO Monica. Young Children (0-8) and Digital Technology A qualitative study across Europe. JRC Publications Repository [online]. Publisher: Publications Office of the European Union. 2018. [cit. 2023-04-22]. ISBN 978-92-79-77766-0. Dostupné z: https://publications.jrc.ec.europa.eu/repository/handle/JRC110359

[24] EURÓPSKA ÚNIA. Digital Education Action Plan (2021-2027). European Education Area [online]. 2021. [cit. 2023-04-22]. Dostupné z: https://education.ec.europa.eu/focus-topics/digital-education/action-plan

[25] VEE Annette. Understanding Computer Programming as a Literacy. University of Pittsburgh [online]. 2013. [cit. 2023-04-22]. Dostupné z: https://d-scholarship.pitt.edu/21695/1/24-33-1-PB.pdf

[26] GHASEMI Babak; HASHEMI Masoud. Foreign language learning during childhood. ScienceDirect.com | Science, health and medical journals, full text articles and books. [online] Procedia - Social and Behavioral Sciences Volume 28, 2011, Strany 872-876. 2011. [cit. 2023-04-23]. Dostupné z: https://www.sciencedirect.com/science/article/pii/S1877042811025997?ref=pdf_download&fr=RR-2&rr=7bc10a23c8e1b33b

[27] PRUCHA Jan.Psychologie učení. Teoretické a výzkumné poznatky pro edukační praxi. 2020. [cit. 2023-04-29]. Praha : Grada, 2020. ISBN 978-80-271-2853-2.

[28] CIFTCI Serdar; BILDIREN Ahmet. The effect of coding courses on the cognitive abilities and problem-solving skills of preschool children. Computer Science Education. 1-19. 10.1080/08993408.2019.1696169. ResearchGate | Find and share research. [online]. 2020-01. [cit. 2023-04-23]. Dostupné z: https://www.researchgate.net/publication/337642905_The_effect_of_coding_courses_on_the_cognitive_abilities_and_problem-solving_skills_of_preschool_children#:~:text=The%20research%20conducted%20by%20%C3%87iftci,solving%20skills%20were%20statistically%20insignificant.

[29] FESAKIS Georgios; MAVROUDI Elisavet. Problem solving by 5–6 years old kindergarten children in a computer programming environment: A case study. Computers & Education. 63. 87–97. 10.1016/j.compedu.2012.11.016. ResearchGate | Find and share research. [online]. 2013-04. [cit. 2023-04-23]. Dostupné z: https://www.researchgate.net/publication/257171388_Problem_solving_by_5-6_years_old_kindergarten_children_in_a_computer_programming_environment_A_case_study

[30] LOVÁSZOVÁ Gabriela; GALBAVÁ Ľudmila; PALMÁROVÁ Viera; TOMCSÁNYIOVÁ Monika. Malé programovacie jazyky. Štátny pedagogický ústav - ŠPÚ. [online]. 2010. [cit. 2023-04-29]. ISBN 978-80-8118-066-8. Dostupné z: https://www.statpedu.sk/files/sk/o-organizacii/projekty/projekt-dvui/publikacie/male_programovacie_jazyky.pdf

[31] COMPUTATIONAL EDUCATION LAB. Chapter 1: Introducing Karel the Robot. CompEdu Lab. [online]. 2020-04-01. [cit. 2023-04-29]. Dostupné z: https://compedu.stanford.edu/karel-reader/docs/python/en/chapter1.html

[32] INFOVEK. Baltík 3. SGP Systems - Baltie C# 3D Game visual programming teaching tools for kids, children, youth and adults. [online]. 2003-02-28. [cit. 2023-04-29]. Dostupné z: https://www.sgpsys.com/infovek/zakl_inf3.htm

[33] Kolektív autorov. Scratch: Programming for All. News + Updates – MIT Media Lab. [online]. 2009-11. [cit. 2023-04-30]. doi:10.1145/1592761.1592779. Dostupné z: https://web.media.mit.edu/~mres/papers/Scratch-CACM-final.pdf

[34] BRAUN Bryan. Scratch is a big deal. Bryan Braun - Frontend Developer. [online]. 2022-07-16. [cit. 2023-04-30]. Dostupné z: https://www.bryanbraun.com/2022/07/16/scratch-is-a-big-deal/

[35] WARSKI Adam. Functional programming for kids?. SoftwareMill - proactively transforming your business with technology. [online]. 2022-06-21. [cit. 2023-04-30]. Dostupné z: https://softwaremill.com/functional-programming-for-kids/

[36] SMITH Chris. CodeWorld. GitHub - google/codeworld: Educational computer programming environment using Haskell. [online]. 2023-02-12. [cit. 2023-04-30]. Dostupné z: https://github.com/google/codeworld