

Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Výuková aplikace programování
Josef Böhm

Bakalářská práce
2019

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: Josef Böhm
Osobní číslo: I15283
Studijní program: B2646 Informační technologie
Studijní obor: Informační technologie
Název tématu: Výuková aplikace programování
Zadávající katedra: Katedra informačních technologií

Zásady pro vypracování:

Cílem bakalářské práce je vytvořit program v Javě pro výuku základů programování pomocí vlastního jednoduchého programovacího jazyka. Teoretická část práce bude obsahovat popis problematiky návrhu a realizace kompilátoru (resp. interpretu) pro navržený výukový programovací jazyk. Dále bude formálně popsána gramatika navrženého programovacího jazyka. Navržený jazyk by měl poskytnout základní konstrukce (podmínky, cykly, funkce) a podporu pro práci s lokálními proměnnými ve funkcích. V praktické části bude realizována aplikace na platformě Java. Aplikace bude sloužit jako interpret navrženého jazyka s grafickým výstupem (na podobném principu jako u jazyků Logo/Baltik/apod.). Aplikace bude obsahovat základní příklady pro výuku programování.

Rozsah grafických prací:

Rozsah pracovní zprávy: 40 stran

Forma zpracování bakalářské práce: tištěná/elektronická

Seznam odborné literatury:

RICHARDSON, W. Clay. Professional Java, JDK 5th ed. Indianapolis, IN: Wrox, c2005, xxxi, 712 p. ISBN 07-645-7486-8 LI, Liwu. Java: Data Structures and Programming Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. ISBN 36-429-5851-6 DOS REIS, Anthony J. Compiler construction using Java, JavaCC, and Yacc Hoboken, N.J.: Wiley-IEEE Computer Society, c2012. ISBN 9780470949597

Vedoucí bakalářské práce:

Ing. Roman Diviš

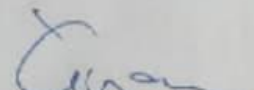
Katedra softwarových technologií

Datum zadání bakalářské práce: 31. října 2017

Termín odevzdání bakalářské práce: 12. května 2018



Ing. Zdeněk Němec, Ph.D.
děkan



Ing. Lenka Čepová, Ph.D.
pověřený vedením katedry

V Pardubicích dne 20. března 2018

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 3. 5. 2019

Josef Böhm

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu práce Ing. Romanu Divišovi za vstřícný přístup a cenné rady při zpracovávání bakalářské práce.

ANOTACE

Práce se zabývá vlastní tvorbou výukové aplikace programování včetně návrhu vlastního jazyka a implementací interpretu. Věnuje se teorii programovacích jazyků a zaměřuje se na problematiku jejich obecného návrhu. Práce dále obsahuje řešerši dalších výukových aplikací programování v různých formách jejich provedení.

Výsledkem této práce je aplikace pro výuku programování využívající vlastní navržený programovací jazyk. Aplikace obsahuje několik základních příkladů na ozkoušení algoritmických dovedností.

KLÍČOVÁ SLOVA

abstraktní syntaktický strom, interpret, kompilátor, výuková aplikace, programovací jazyk

TITLE

Educational application for programming.

ANNOTATION

The thesis is focused on development of application for programming education, including custom language and implementation of the interpreter. Furthermore, the thesis addresses the theory of programming languages and problematics of their general designs while including researches of other educational programs in different forms.

The result of the thesis is an application created to teach programming using its own programming language. The application contains a couple of basic examples in order to test users algorithmic skills.

KEYWORDS

abstract syntax tree, interpreter, compiler, tutorial application, programming language

OBSAH

Seznam obrázků	9
Seznam zkratk	10
Úvod.....	11
1 Programovací jazyky	12
1.1 Nízko-úrovňové programovací jazyky.....	12
1.2 Vysoko-úrovňové programovací jazyky	12
1.3 Programovací paradigmatata	13
1.3.1 Funkcionální paradigma	13
1.3.2 Logické paradigma	13
1.3.3 Procedurální paradigma	14
1.3.4 Objektově orientované paradigma	14
1.4 Rozdělení překladačů podle typu cílového programu.....	14
1.4.1 Kompilátor	14
1.4.2 Interpret.....	15
1.4.3 Hybridní překladače	15
1.4.4 Výhody a nevýhody jednotlivých druhů překladačů.....	15
2 Srovnání programovacích jazyků	17
2.1 Představení výukových aplikací včetně jazyků a vývojových prostředí.....	17
2.2 Nevýukové programovací jazyky	20
3 Syntaxe vlastního programovacího jazyka	21
3.1 Motivace	21
3.2 Použité nástroje a prostředí	21
3.3 Vlastnosti.....	21
3.4 Formální popis jazyka	22
4 Návrh interpretu jazyka.....	26
4.1 Obecný návrh interpretu.....	26
4.1.1 Interpret na bázi mezikódu	26
4.1.2 Interpret abstraktního syntaktického stromu.....	26

4.1.3	Interpret zřetězeného kódu.....	27
4.1.4	Kompilace typu – <i>Just-in-time</i>	27
4.1.5	Intepretace sebe samotného	27
4.2	Problematika návrhu řešení	27
4.2.1	Návrh řešení výrazů.....	28
4.2.2	Analýza rekurzivním sestupem.....	29
4.2.3	Postup při návrh jazyka	29
4.2.4	Inicializace vstupního kódu	30
4.2.5	Obecný návrh struktury interpretu abstraktního syntaktického stromu	30
4.2.6	Analýza vstupního kódu	31
5	Realizace cílové aplikace	32
5.1	Funkcionality	32
5.2	Představení řešení a výstupu aplikace	35
	Závěr.....	38
	Použitá literatura.....	39

SEZNAM OBRÁZKŮ

Obrázek 1: Ukázka vývojového prostředí programu <i>Karel</i>	17
Obrázek 2: Ukázka výukové webové aplikace <i>CodeCombat</i>	18
Obrázek 3: Ukázka výukové webové aplikace <i>CodinGame</i>	19
Obrázek 4: Ukázka výukové webové aplikace <i>CodeWars</i> . Zdroj: vlastní.	19
Obrázek 5: UML diagram popisující obecnou strukturu pro řešení výrazů	28
Obrázek 6: Ukázka hlavní části aplikace bezprostředně po jejím spuštění	32
Obrázek 7: Ukázka záložky <i>Edit function</i>	33
Obrázek 8: Ukázka záložky <i>Tutorial</i>	34
Obrázek 9: Ukázka první lekce. Zdroj: vlastní.....	34
Obrázek 10: Ukázka popisu příkazu <i>FUNCTION</i> . Zdroj: vlastní.	35
Obrázek 11: Demonstrace úspěšného řešení první lekce. Zdroj: vlastní.	35
Obrázek 12: Program vykreslující pomocí cyklu hlavní diagonálu. Zdroj: vlastní.....	36
Obrázek 13: Vyplnění matice s postupnou změnou barev v každé iteraci. Zdroj: vlastní...36	

SEZNAM ZKRATEK

EOF	End of file – Konec souboru
CPU	Central Processing Unit – Centrální procesorová jednotka
RAM	Random Access Memory – Operační paměť počítače
VRAM	Video Random Access Memory – Grafická operační paměť

ÚVOD

Cílem této bakalářské práce je vytvoření aplikace, která bude sloužit jako výukový nástroj pro studenty věnující se programování. Aplikace bude umožňovat uživateli psát vlastní programy v programovacím jazyce, který byl navržen pro tyto výukové účely.

Důvodem vzniku této práce je snaha rozvinout u studentů logické myšlení, které bude ve finále výborným nástrojem pro racionální uvažování a zlepšení dosavadních algoritmických dovedností. Každý student věnující se oboru informační technologie, by měl mít alespoň základní vědomosti o algoritmických procesech. Tyto zkušenosti se ovšem nepromítají jen do oblasti programování, ale také do běžných činnostech člověka. Z tohoto důvodu byl navržen takový výukový nástroj, aby si každý mohl vyzkoušet, zda je schopný abstraktní myšlenku uvést do konkrétní implementace a následně ověřit, zda jeho teze byla validní, či nikoliv.

K získání či rozšíření znalostí v oblasti programování existují výukové programy, které většinou obsahují vlastní vývojové prostředí včetně jazyka, který je na míru navržený pro konkrétní aplikaci. Tyto aplikace jsou většinou vyvíjeny s propracovanou grafikou, aby uživateli usnadnili celkovou orientaci a aby mohly konkurovat dalším aplikacím tohoto typu. Jsou k dispozici přímo ve webovém prohlížeči nebo jako spustitelné soubory.

Forma zpracování je většinou závislá na typu uživatelů, na které aplikace cílí. Pro ty nejmladší jsou k dispozici formou hry, kde uživatel pomocí jednoduchých příkazů ovládá průběh děje a pro starší uživatele jsou většinou ve formě konkrétního zadání, kde soutěží s ostatními uživateli o nejlepší implementaci a na základě těchto úkolů získává na svůj profil body, které o uživateli prozrazují jeho aktuální dovednosti.

Existují také výukové aplikace, které nejsou zajímavé z pohledu jejich užívání, ale naopak z implementační části. Tyto aplikace slouží jako demonstrační pomůcka při implementaci vlastních programovacích jazyků.

V oblasti programování řadíme tyto aplikace mezi kompilátory nebo interprety a jejich praktické využití bude v této práci probíráno včetně jejich obecného návrhu.

1 PROGRAMOVACÍ JAZYKY

Programovací jazyky jsou nástrojem určeným převedším pro přímou komunikaci s hardwarem. Poskytují možnost vyjadřovat abstraktní myšlenky člověka do konkrétní podoby, kterou je hardware počítače schopný zpracovat. Kapitola vychází z (Peterka 1995, Wirth 2005, Van Roy 2009, Dos Reis 2012, Hlavica 2004).

1.1 Nízko-úrovňové programovací jazyky

Toto označení platí pro jazyky, které jsou málo odlišné od strojových instrukcí CPU. Jejich konstrukce přináší možnost práce přímo s hardwarem počítače. Složitost těchto jazyků však náročná není, nicméně pro správné používání je třeba velmi detailní znalost používaného hardwaru.

Existují dvě generace těchto jazyků. První je prakticky strojový kód, který nepodléhá žádnému překladu a je přímo interpretován pomocí CPU. Druhou generací je jazyk symbolických instrukcí, známé také jako *assembler*. Přesto že se již nejedná o prostý strojový kód, je velmi nutná znalost architektury CPU včetně instrukční sady a registrů.

Překlad z tohoto jazyka je prováděn přímo do strojového kódu. Mezi tyto jazyky lze zařadit i jazyk *C*, a to hlavně z toho důvodu, že na rozdíl od jiných programovacích jazyků poskytuje přístup k paměti RAM a jiné nízko-úrovňové nástroje.

1.2 Vysoko-úrovňové programovací jazyky

Jazyky v této kategorii jsou daleko lépe srozumitelné a poskytují tak možnost psaní vlastních komplexních programů. Oproti svému předchůdci jsou zdrojové kódy mnohem kratší a lépe čitelné, což snižuje pravděpodobnost výskytu logických a syntaktických chyb.

Mezi další nesporné výhody těchto jazyků patří absence znalosti hardwaru na kterém se vykonávají, což přináší znatelně lepší přenositelnost napříč různými platformami.

Charakteristiky těchto jazyků určují takzvaná programovací paradigmatata, které pomáhají klasifikovat oblast použitelnosti. Mezi tyto jazyky patří: *BASIC*, *C++*, *PHP*, *Perl*, *Python*, *Java*, *Delphi*,

1.3 Programovací paradigmatata

Různé druhy existujících programovacích jazyků byly navrženy s určitým záměrem pro jejich specifické využití. Na základě této skutečnosti existuje množina různých klasifikací nazývané programovacími paradigmaty, kam lze zařadit každý existující programovací jazyk, podle jeho charakteru a poskytovaných možností.

Obecně tyto klasifikace lze chápat jako fundamentální styl budování konkrétní struktury. Většina jazyků spadá do více druhů paradigmat. Jen velmi malá část existujících jazyků je totiž zařaditelná jen do jednoho. Vhodným příkladem je programovací jazyk *Scala*, ve kterém je možné použít funkcionální či objektově orientovaný styl programování.

1.3.1 Funkcionální paradigma

Tento způsob programování je založen především na volání funkcí. Proměnné jsou v těchto jazycích většinou neměnitelné, což znamená, že po přiřazení hodnoty již nelze tuto hodnotu modifikovat a je třeba založit novou. Také je zde kladen důraz na využívání rekurze. V zásadě se jedná o funkce, které obsahují jiné funkce.

Přednosti toho stylu jsou především v úspoře velikosti výsledného kódu. Hlavní výhodou bude nižší chybovost a rychlost. Za nevýhodu lze považovat horší čitelnost kódu.

Nutno také podotknout, že se jedná o jedno z nejstarších paradigmat (Jones 2009, s. 4-5). Typickými zástupci tohoto stylu jsou například: *Clean*, *Haskell*, *Erlang*, *Scala*.

1.3.2 Logické paradigma

Logické paradigma funguje na principu sestavení množiny pravidel, nazývané klauzule, které programátor nadefinuje. Při jejich spuštění systém při zpracování ověřuje, zda napsaná pravidla platí či nikoliv a podle toho se vykoná příslušná operace.

Tento styl má uplatnění především pro vývoj umělé inteligence, protože na rozdíl od klasických programovacích jazyků, které krok po kroku strojově provádí nadefinované instrukce, tento styl umožňuje napsat program takovým způsobem, že stačí pouze specifikovat co má program dělat, ale již nic neříká o tom, jakým způsobem by to měl dělat.

Interpret těchto jazyků má pak za úkol rozhodnout, jakým způsobem provede výpočet. Jedná se o formu deklarativního paradigma a jeho představitelé jsou: *Prolog*, *Absys*, *ASP*, *Datalog*, *Gödel*, *Mercury*, *MetaL*, *Twelf*.

1.3.3 Procedurální paradigma

V případě procedurální paradigma se příkazy vykonávají sekvenčně. V zásadě se jedná o definování funkcí či podprogramů, které obsahují sled příkazů, které se postupně vykonávají. Programátor tyto části kódu následně volá podle potřeby.

Tento způsob hojně využívá cykly. Jedná se o nejstarší typ paradigma. Do tohoto paradigma řadíme jazyky: *C/ C++, Java, ColdFusion, Pascal*.

1.3.4 Objektově orientované paradigma

Objektově orientované paradigma je založeno na komunikaci mezi třídami a objekty. Prvním jazykem napsaném v tomto stylu byl *Simula*. Tento styl zakládá na modelování určité problematiky z reálného prostředí do konkrétní podoby, ze které následně vzniknou objekty.

Objekt je ovšem pouze abstraktní pojem, který vyjadřuje nespécifikované vlastnosti a chování, a proto je ve třídě nutné specifikovat atributy a metody, které budou daný objekt konkretizovat (Jones 2009, s. 5-6).

Konstrukce tohoto paradigma je velmi blízká mentálnímu vnímání světa na fundamentální úrovni každého jedince a nabízí tak čistý pohled na problematiku budování určité abstraktní konstrukce a poměrně snadnou představu o budoucí realizaci konkrétní problematiky.

Mezi výhody patří ochrana dat, dědičnost a flexibilita. Jazyky, které jsem patří jsou: *Java, C++, Simula, Objective-C, Visual Basic .NET, Python, SmallTalk, Ruby*.

1.4 Rozdělení překladačů podle typu cílového programu

Existuje několik momentálně možných způsobů, jakým lze programový kód zpracovávat. Obecně se vždy jedná o nějaký program, zajišťující vykonání definovaných instrukcí. Tyto programy se rozdělují podle způsobu zpracování programového kódu, z čehož každý z nich poskytuje jedinečné vlastnosti, podle kterých se určují výhody či nevýhody. Následující podkapitoly tyto programy popisují.

1.4.1 Kompilátor

Kompilátor je program, který je schopný transformovat programový kód vyššího programovací jazyka do jazyka nižší úrovně, většinou do strojového kódu, který je procesor počítače schopen zpracovat (Peterka 1995).

Existence kompilátorů programátorům poskytuje mnohem větší komfort při vývoji aplikací, jelikož se nemusí zabývat složitými elementárními instrukcemi, které bývají při vývoji větších projektů velmi těžko srozumitelné a velice náročné na implementaci či údržbu. Tento problém spolehlivě řeší kompilované jazyky, které jsou mnohem intuitivnější a jednodušší. Výhodou kompilovaných jazyků je především rychlost.

1.4.2 Interpret

Interpret je program, který je umožňuje spustit zdrojový kód nějakého jazyka, aniž by se musel kompilovat do strojového kódu. Pokud bychom uvažovali o interpretu napsaném v nižším programovacím jazyce, v zásadě by se jednalo o podobnou strukturu strojového kódu, který produkují klasické kompilátory.

Zcela jiné vlastnosti lze očekávat, pokud bude interpret navržen ve vyšším programovacím jazyce. Navržená struktura bude obsahovat syntaktický strom, který bude složen z obecné struktury příkazů, či lze přímo vykonávat určité akce, bez žádné větší datové struktury. Poměrně mnoho operací při zpracování zdrojového kódu bude shodné s kompilátorem.

Hlavním rozdílem generovaných programů interpretem je, že kompilovaný kód je sestavitelný do samostatného souboru na úrovni strojového jazyka, kdežto interpretovaný kód je vlastně množina příkazů, které se na základě uživatelského vstupu podle vnitřních definic vykonají v jazyce, ve kterém byl interpret napsán (Peterka 1995).

1.4.3 Hybridní překladače

Tyto překladače pracují na principu převedení interpretovaného jazyka do mezikódu. Tím získáme určitý sled instrukcí, které nejsou vázány přímo na operační systém, ale na interpret, který je navržen tak, aby bylo možno spouštět tyto programy všude, kde je nainstalován.

Tento způsob je velice výhodný zejména v oblasti přenositelnosti kódu, protože v zásadě stačí, pokud program pro zpracování mezikódu existuje pro platformu, ve které se programátor pohybuje (Peterka 1995).

1.4.4 Výhody a nevýhody jednotlivých druhů překladačů

Hlavním kritériem pro porovnání výše zmíněných překladačů bude především kladen důraz na dobu zpracovávání, paměťové náročnosti a možnost přenositelnosti mezi různými platformami.

Nespornou výhodou kompilovaných programů bude především doba zpracování. Pokud bude programátor vyžadovat pro svoji aplikaci vysokou rychlost, pravděpodobně zvolí jazyk pracující na nižší úrovni. Tyto situace ovšem ovlivňují spousty faktorů, a nelze vždy jednoznačně říci, který typ jazyka bude nejvhodnější.

Interpretované jazyky především zpomaluje jejich samotná konstrukce, jelikož se jedná o program, který je spouštěný v jiném programu, což je náročné zejména na RAM počítače a výkon procesoru. Další vlastností interpretu je jeho rychlost překladu, která je oproti kompilátoru lepší.

Výkon hybridních překladačů stojí někde mezi kompilátorem a interpretem. Výhodou je podobně jako u interpretovaných jazyků multiplatformní podpora.

Díky celkovému výkonu dnešních počítačů včetně velikosti RAM a dalších faktorů, se již hranice mezi výhodou a nevýhodou stále více přibližuje vývoji v interpretovaných jazycích.

2 SROVNÁNÍ PROGRAMOVACÍCH JAZYKŮ

2.1 Představení výukových aplikací včetně jazyků a vývojových prostředí

Tyto jazyky jsou navrženy pro uživatele, kteří mají jako primární cíl učit se základním principům programování, nebo jen čistě pro zábavu. Konstrukce těchto jazyků je většinou nevhodná pro použití v komerční sféře či vývoje komplexnějších aplikací s racionálním významem. Existence výukových jazyků je tedy podmíněna existencí nevýukových.

Tyto jazyky a jejich vývojové prostředí, jsou k dispozici v nepřeberném množství. Zástupci těchto jazyků jsou například *Logo* či *Karel*, který je určen pro absolutní začátečníky a funguje na principu pohybu objektu na dvoudimenzionálním poli za pomoci jednoduchých instrukcí definující množinu pohybů, které jsou následně vykresleny (Lašťovička 2011).

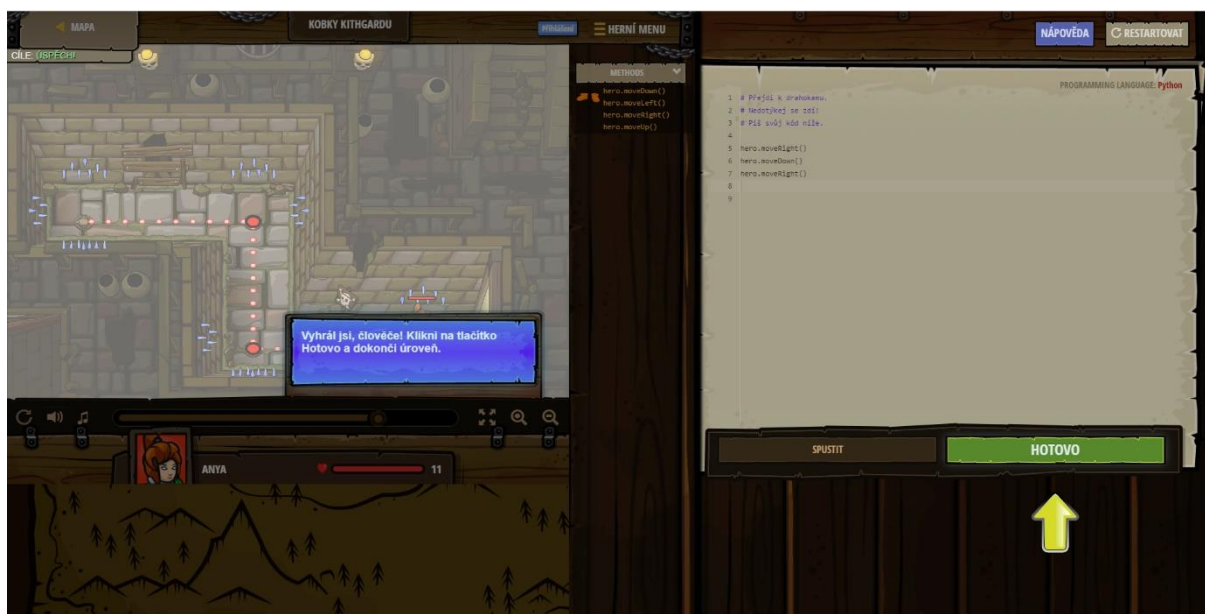


Obrázek 1: Ukázka vývojového prostředí programu *Karel*.

Zdroj: vlastní.

Moderní varianty těchto aplikací v zásadě přebírají stejné vlastnosti jako těch původních a jsou poskytovány v mnohem lepší grafice i celkového zpracování.

Jedním ze zástupců webové verze je například *CodeCombat*, což je výuková aplikace zaměřující se na deriváty klasických programovacích jazyků, mezi které patří *JavaScript*, *Python* či *CoffeScript*. Aplikace funguje na principu stejném jako *Karel*.



Obrázek 2: Ukázka výukové webové aplikace *CodeCombat*

Zdroj: vlastní.

Další volně dostupnou variantou těchto výukových aplikací je bezesporu *CodinGame*, což je graficky i funkcionálně velmi propracovaná aplikace. Její vlastnosti jsou podobné předchozí variantě. Uživatel má k dispozici grafický výstup, ve kterém se nachází připravená hra v závislosti na dosažené úrovni.

V této hře se programuje pomocí klasických programovacích jazyků, které nejsou nijak upravované. Tato varianta je z pohledu přínosu velmi výhodná v oblasti výuky, jelikož se uživatel může profilovat v jazyce, který je mu blízký a má možnosti si vyzkoušet široké spektrum jazyků a programovat stejné implementace v každém z nich. Aplikace uživatelům nabízí celkem 27 programovacích jazyků, ve kterých je možné plnit jednotlivé úkoly. Mezi tyto jazyky patří například: *Java*, *C/C++*, *C#*, *Python3*, *Javascript*, *Bash*, *ObjectiveC*, *Swift*, *Pascal* a další.

Další formou výukových aplikací jsou takové, jež neposkytují žádné grafické prostředí, ale soustředí se především na konstrukci algoritmů. Uživatel má k dispozici zadání, které specifikuje, jaký výstup program očekává a výsledná implementace je plně v jeho rukou.

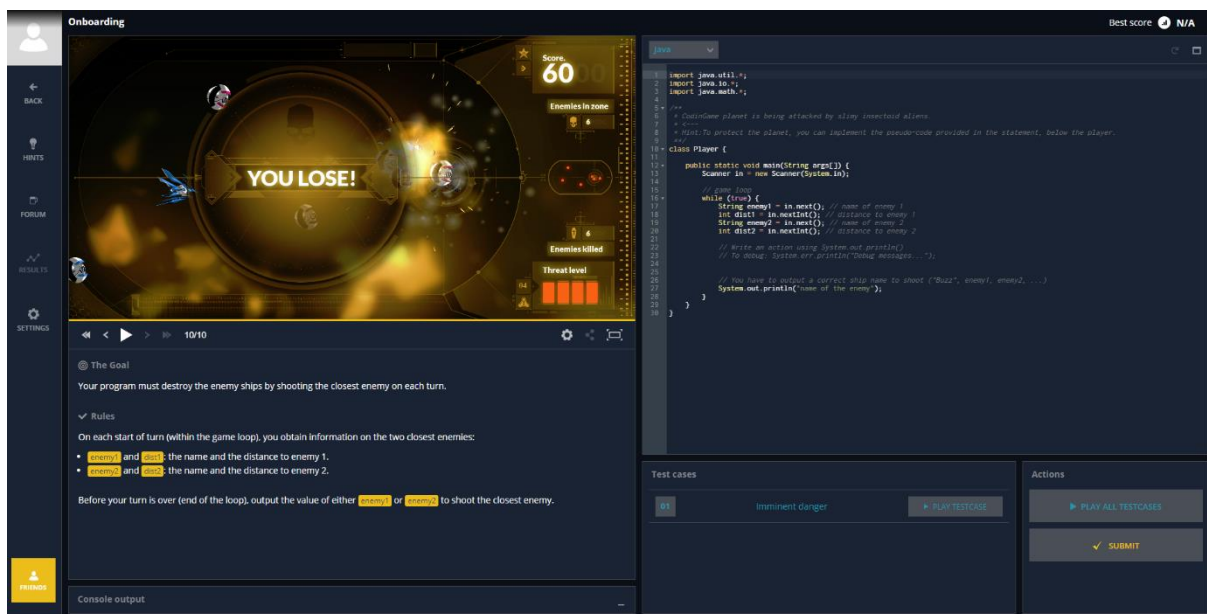
Vyhodnocuje se především kvalita a správnost výsledného kódu a porovná se s dalšími uživateli, na základě čehož obdrží bodové ohodnocení v závislosti uvedených faktorů.

Příkladem této aplikace se například *CodeWars*. Ostatně jako předchozí typy podporuje velké množství jazyků, ve kterém je možné danou implementaci realizovat.

Dokonce zde nechybí ani podpora verzovacího systému *Git*, zprostředkovaný na cloudové službě *GitHub*, do kterého lze ukládat hotové projekty, respektive implementace jednotlivých úloh, ke kterým budou moci přistupovat ostatní uživatelé, aby se mohli eventuálně inspirovat

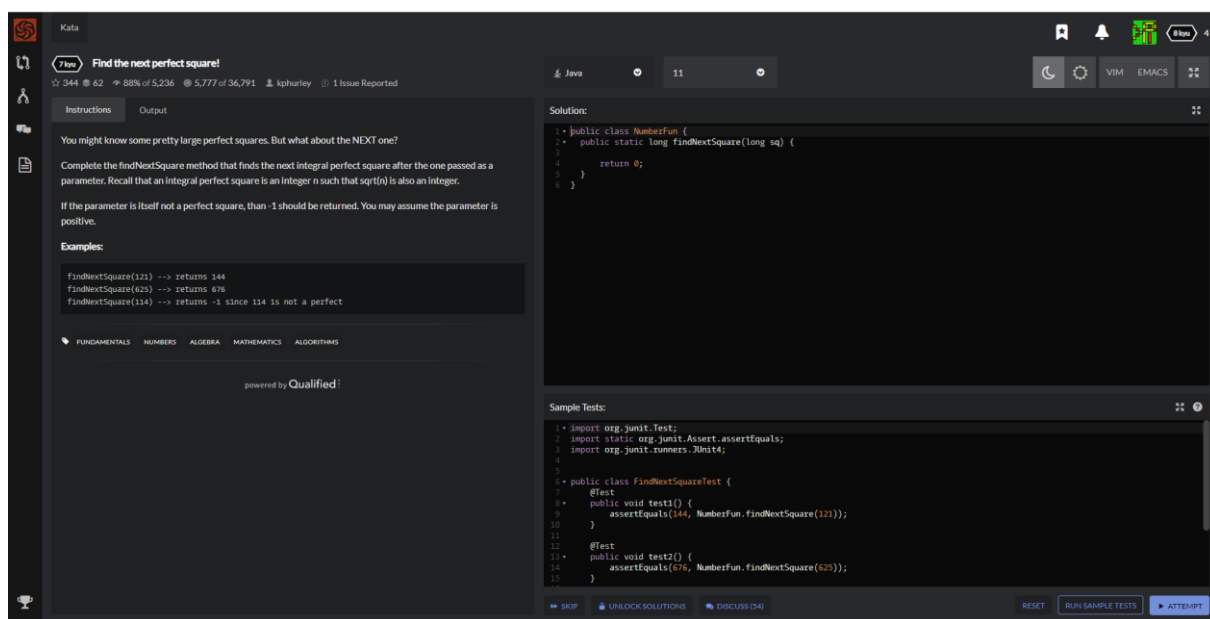
pro napsání vlastní verze. Do této kategorie lze zařadit i takové typy výukových aplikací, které jsou zajímavé především z implementační části. Tyto aplikace poskytují programátorovy koncept, kterým by se mohl inspirovat při návrhu vlastního interpretu. Mezi představitele tohoto typu řadíme například jazyk *PL/0*, který byl vyvinut právě pro tyto účely.

Absolvováním těchto programů je výsledkem rozvinutí algoritmického myšlení, které je nepostradatelným prvkem pro programování kvalitních aplikací nebo alespoň získání určitého základu a povědomí, co to vlastně znamená programovat a celkem rychle tak zjistí, zda mu tato disciplína vyhovuje.



Obrázek 3: Ukázka výukové webové aplikace CodinGame

Zdroj: vlastní.



Obrázek 4: Ukázka výukové webové aplikace CodeWars. Zdroj: vlastní.

2.2 Nevýukové programovací jazyky

Jazyky tohoto charakteru jsou primárně určeny pro vývoj jakýchkoliv programových struktur. Oproti výukovým jazykům nabízí mnohonásobně více možností a modifikací. Mezi nevýukové jazyky můžeme zařadit i jazyky, které vznikly čistě z akademických důvodů, jejichž záměrem bylo vytvořit kompilátor či interpret, bez ohledu na to, zda je možné ve výsledném jazyce vůbec možné tvořit nějaké smysluplné programy.

Tyto jazyky by byly extrémně nevhodné především pro začátečníky, protože propracovanost těchto jazyků nemusí splňovat uživatelské očekávání, a může nastat situace, kde se student bude muset potýkat s častými pády aplikace, či nepředvídatelnému chování. Ve finále by toto mohlo být rozhodujícím faktorem proto, zda se student bude danému oboru chtít vůbec věnovat. Mezi typické zástupce těchto jazyků patří například: *Ook!*, *WhiteSpace*, *LOLCODE*, *l33t* (Raduta 2010).

3 SYNTAXE VLASTNÍHO PROGRAMOVACÍHO JAZYKA

3.1 Motivace

Existuje mnoho variant programovacích jazyků. Můžeme je rozdělit na ty, které jsou pro začátečníky z hlediska syntaxe jednodušší a intuitivnější nebo naopak méně vhodné. Vlastnosti jazyka poté determinují míru použitelnosti a složitosti.

Cílem této práce bylo vytvořit takovou podobu jazyka, aby uživatel nemusel studovat složité konstrukce pro jeho použití a na druhou stranu, aby obsahoval nepostradatelné součásti, ve kterých je možné tvořit zajímavé programy.

3.2 Použité nástroje a prostředí

Celá aplikace byla programována v Javě ve vývojovém prostředí NetBeans IDE 8.2 (Build 201609300101). Podstatná část studia se tomuto jazyku věnovala nejvíce, což bylo hlavní kritérium při volbě. Stejně to bylo i případě vývojového prostředí.

Pro návrh grafické části byl využíván nástroj SceneBuilder 10.0.0. a návrh programové části byl realizován v JavaFX. Verze používaného JVM Version 8 Update 201 (build 1.8.0_201-b09).

Vše bylo vytvářeno v operačním systému Windows 10 Education 64bitový (10.0, build 17134). Hardware počítače obsahoval grafickou kartu NVIDIA GeForce GTX 660 s VRAM 1998 MB. Procesor byl AMD FX-6300 Six-Core taktován na ~4.0 Ghz. Počítač obsahoval RAM o velikosti 8192 MB.

3.3 Vlastnosti

Navržený jazyk poskytuje základní příkazy pro práci s proměnnými včetně rozhodovacích podmínek a cyklů. Základním stavebním kamenem toho jazyka je funkce. Z pohledu klasifikace tento jazyk spadá do kategorie *procedurálního paradigma*. Existuje zde speciální bezparametrická funkce s názvem *main*, která je spouštěna jako první. Pokud tato funkce chybí, program nemá co spouštět a je ukončen s chybovým hlášením o nepřítomnosti této funkce.

Funkce si mohou uživatelé definovat sami, nebo mohou použít sadu funkcí, které poskytují různé matematické operace či základní příkazy pro práci s textovými řetězci. V rámci výukové aplikace zde byla implementována možnost práce s jednoduchým grafickým výstupem, nýbrž samotný jazyk nemá specifikovaná žádná klíčová slova, která by umožňovala pracovat s obecnými grafickými prvky ani zvukovým vstupem či výstupem. Funkce podporují vstupní parametry.

Tento jazyk také podporuje *rekurzi*, což uživateli umožní v těle funkce volat tutéž funkci znovu. Proměnné jsou dynamického typu, které podporují číselnou hodnotu či textový řetězec. Do proměnné je možné uložit výsledek libovolného matematického či logického výrazu, včetně textového řetězce. V případě logického výrazu se vyhodnotí číselnou hodnotou, kde nula znamená nepravda a jedna pravda. Logický výraz se očekává především v příkazech, které definují podmínku či cyklus.

Kromě výrazu klasicky sestaveného z binárních operátorů je možné za přiřazovacím operátorem volat funkci, která vrací hodnotu. Pokud je zavolána funkce, jenž nic nevrací, je do proměnné automaticky přiřazena číselná hodnota nula. Jazyk nerozlišuje malá a velká písmena. Není zde implementována žádná podpora přístupu k síťovým službám ani přístup k souborovému systému. Interpret aplikace byl realizován na principu analýzy rekurzivním sestupem, z čehož je patrné, že se bude jednat bezkontextový typ gramatiky.

Přehled základních vlastností jazyka:

- procedurální jazyk,
- bezkontextová LL (2) gramatika,
- dynamická typová kontrola,
- slabě typovaný,
- podpora rekurze,
- interpretovaný.

3.4 Formální popis jazyka

Spustitelný program je sestaven z deklarovaných funkcí včetně přítomnosti speciální funkce, která pokud chybí, není program možné spustit.

```
program = { funkce } EOF;
```

Deklarace obecné funkce se velice podobá té hlavní, změna bude především v možnosti přidávat parametry a unikátní identifikátor.

```
funkce = 'FUNCTION', mezera, identifikátor, '(', (identifikátor, {'',',',  
identifikátor}), ')', '{', {příkaz}, '}';
```

Příkaz obsahuje vždy jednu konkrétní typ operace

```
příkaz = (deklarace-proměnné | modifikace-proměnné | inkrementace-proměnné |  
dekrementace-proměnné | cyklus | podmínka-příkaz | volání-funkce | konec-  
cyklu | návrat-hodnoty);
```

Pro deklaraci proměnné jsou stanovena následující pravidla.

```
deklarace-proměnné = 'VAR', mezera, identifikátor, '=', vstupní-hodnota,  
';;';
```

Příkaz pro modifikaci proměnné bude téměř totožný jako předchozí případ.

```
modifikace-proměnné = 'SET', mezera, identifikátor, '=', vstupní-hodnota,  
';;';
```

Použití příkazu pro inkrementaci proměnné bude vypadat následovně.

```
inkrementace-proměnné = 'INCREASE', mezera, identifikátor, ';;';
```

Pro dekrementaci budou pravidla téměř shodná s přechozí variantou.

```
dekrementace-proměnné = 'DECREASE', mezera, identifikátor, ';;';
```

Volání funkce je možné díky následujícímu postupu.

```
volání-funkce = 'CALL', mezera, specifikovaná-funkce, ';;';
```

Po deklaraci, je možné obecně tuto funkci použít v tomto tvaru.

```
specifikovaná-funkce = identifikátor, '(', (vstupní-hodnota, {'',',', vstupní-  
hodnota}), ')';
```

Podporované vstupní hodnoty.

```
vstupní-hodnota = (výraz | podmínka | řetězec);
```

Pravidla pro cyklus jsou obdobná pro deklaraci funkce.

```
cyklus = 'REPEAT', '(', podmínka, ')', '{', {příkaz}, '');
```

Stanovení podmínky má podle pravidel následující tvar.

```
podmínka-příkaz = 'IF', '(', podmínka, ')', '{', {příkaz}, '}', ('ELSE',  
'{', {příkaz}, '}');
```

Podmínka je definována všemi porovnávacími operátory, přičemž jsou seřazeny podle priority.

```
podmínka = výraz, { ('|'|'|' | '&&' | '==' | '!=' | '<' | '<=' | '>' | '>=' ) },  
výraz};
```

Výraz má následující stavbu.

```
výraz = term, { ('+' | '-'), term};
```

Term je částí výrazu, která vykonává pouze část binární operace, má následující složení.

```
term = faktor, { ('*' | '/'), faktor};
```

Faktor představuje již konkrétní hodnotu přijímanou výrazem.

```
faktor = (identifikátor | číslo | ('(', výraz, ')') | řetězec |  
specifikovaná-funkce);
```

Pro identifikátory platí tato struktura.

```
identifikátor = znak-abecedy, {znak-abecedy | číslice};
```


Číslo může být v následujícím tvaru.

```
číslo = číslice, {čísllice};
```

Číslice je terminální sada těchto povolených znaků.

```
čísllice = ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9');
```

Řetězec je definován následující formou.

```
řetězec = '"', {všechny-znaky-bez-uvozovek}, '"';
```

Všechny znaky jsou zapsány tímto způsobem.

```
všechny-znaky-bez-uvozovek = ? všechny tisknutelné znaky bez uvozovek ? ;
```

Znaky abecedy obsahují malá velká písmena.

```
znak-abecedy = ('a' ... 'z') | ('A' ... 'Z');
```

Mezera obsahuje veškerá bílá místa v řetězci.

```
mezera = ? všechny bílé znaky ? ;
```

4 NÁVRH INTEPRETU JAZYKA

4.1 Obecný návrh interpretu

Interpret může fungovat na odlišných principech. Volba těchto principů závisí čistě na výsledných potřebách programátora. Některé druhy návrhů mohou být navzájem mezi sebou kombinovány a v praxi se tak většinou uplatňuje, což přináší velmi mnoho benefitů, kterým je například rychlost. Veškeré formy těchto návrhů obsahují následující podkapitoly, kde je princip fungování patřičně vysvětlen. Kapitola vychází z (Peterka 1995, Wirth 2005, Jones 2009, Dos Reis 2012).

4.1.1 Interpret na bázi mezikódu

Realizace této varianty je zprostředkována pomocí překladu programového kódu do takzvaného mezikódu. Tento výstup však nemá nic společného se strojovým kódem, který produkují klasické kompilátory.

Jedná se o sadu instrukcí, která musí být následně opět zpracovaná, a to buď do již spustitelné formy, nebo klasického strojového kódu. Mezikód lze chápat jako částečné zpracování, jelikož byly provedeny jen základní operace, které ještě neznamenaají spustitelný program. K tomu potřebujeme mít k dispozici virtuální stroj, což je opět nějaký program, který s tímto typem výstupu již umí pracovat.

Pokud je pro určitou platformu existující verze tohoto virtuálního stroje, není problém spustit jakýkoliv program napsaný v tomto druhu interpretu, jelikož zpracovaný kód není vázán na konkrétní hardware počítače. Díky tomu způsobu získáme snadno přenositelné programy mezi různými platformami.

4.1.2 Interpret abstraktního syntaktického stromu

Tyto interprety fungují zcela odlišným způsobem. V zásadě nedochází k žádnému překladu do nějakého konkrétního kódu, který by byl dalším nástrojem spouštěn. Dochází však k sestavování struktury nazývané abstraktní syntaktický strom. Tyto stromy jsou reprezentací jednotlivých příkazů a výrazů, které se pak postupně procházejí a provádějí podle vnitřních definic. Výhoda této metody pramení v jednotné struktuře programu a snazší analýze procesů za běhu programu. Oproti předchozí metodě nabízí také mnohem větší rychlost.

4.1.3 Interpret zřetězeného kódu

Tento interpret též produkuje mezikód, jako jeho první varianta zde uvedená, nicméně realizace je již patrně odlišná. Jeho realizace je založena na ukazatelích na jednotlivé funkce či nějaké posloupnosti instrukcí, jež mají být vykonány. Interprety tohoto typu jsou používány především pro realizace virtuálních strojů. Výhoda používání je především ve snížení režii na dotazy o další příkazy.

4.1.4 Kompilace typu – *Just-in-time*

Tento způsob umožňuje některé části programového kódu kompilovat přímo do strojového za účelem získání vyšší rychlosti. Rychlost překladu do mezikódu je podmíněna velmi složitým zpracováním, nicméně překlad z mezikódu do strojového je již velmi rychlý, jelikož veškerou práci odvedl interpret, který mezikód generoval. Navíc může provést určité formy optimalizace pro daný procesor, pokud určité typy instrukcí ovšem podporuje. Tyto překlady jsou prováděny pouze za běhu aplikace. Většina moderních jazyků jako je například *Java* nebo *Python* tento způsob kompilace již má zabudovaný.

4.1.5 Intepretace sebe samotného

Jedná se o takový typ interpretu, který je napsán ve stejném jazyce, který interpretuje. Tudíž lze říci, že programový kód napsaný v nějakém jazyce je v tom samém jazyce interpretován. Prvotní fáze vývoje ovšem vyžaduje naprogramovat interpret v jiném jazyce, pro který existuje kompilátor. Teprve poté, lze tuto realizaci provést.

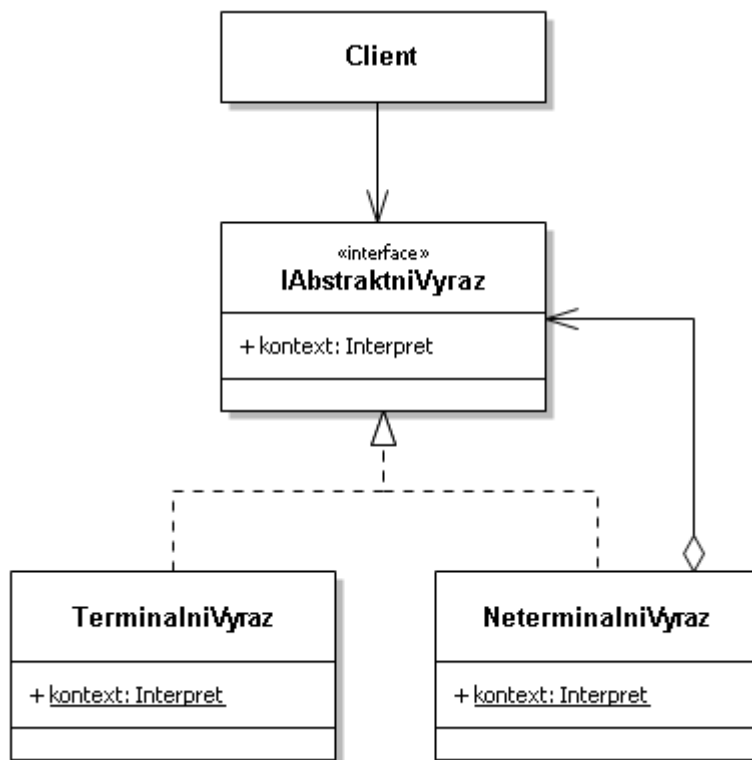
4.2 Problematika návrhu řešení

Vlastní implementace či návrh konstrukce vlastního jazyka a jeho interpretu vyžaduje kromě důkladné přípravy také solidní informace věnující se této problematice.

Následující podkapitoly obecně poskytují informace o jednotlivých částech návrhu, které ve výsledku pomohou či rovnou zodpoví na otázky, které by si eventuálně programátor sám sobě mohl pokládat nebo naopak mohou rozšířit širší povědomí v oblastech, o kterých ani neuvažoval.

4.2.1 Návrh řešení výrazů

Pro řešení toto typu problému je zapotřebí struktura tříd takového způsobu, aby bylo možné sestavit abstraktní syntaktický strom.



Obrázek 5: UML diagram popisující obecnou strukturu pro řešení výrazů

Zdroj: vlastní.

Obrázek definuje strukturu pro abstraktní syntaktický strom. Je třeba zmínit, že obecný návrh nedefinuje přesné metodiky pro budování tohoto stromu, takže výsledná implementace je již čistě na programátorovi. Tyto stromové struktury sestavují takzvané syntaktické analyzátoři. Kontext uchovává veškerá data, které se následně zpracovávají. Abstraktní výraz je zde základním stavebním kamenem pro ostatní typy výrazů, obecně v třídní klasifikaci jako rozhraní. Finálním výrazem se zde myslí takový typ výrazu, který je dále nedělitelný a obsahuje již konkrétní hodnotu. Neukončený výraz zde bude představovat množinu tříd implementujících rozhraní typu abstraktní výraz, které budou představovat zástupce jednotlivých binárních operátorů (Jones 2009, s. 1-3).

Tyto třídy by měli obsahovat metody, které budou vracet konkrétní hodnoty. Každý binární operátor bude ve své třídě obsahovat dvě proměnné typu *výraz*, a metodu která bude schopná z obecného výrazu zajistit příslušnou binární operaci.

Pokud je výraz terminální, patřičná třída vrátí konečnou hodnotu. V případě neterminálního výrazu provede opět binární operaci z hodnot, jež má k dispozici. Tímto postupem se pokračuje až do té fáze, kdy je strom kompletně z analyzován. Výstupem bude nějaká konkrétní hodnota, se kterou se podle potřeb dále pracuje.

4.2.2 Analýza rekurzivním sestupem

Pokud má programátor rozmyšlenou syntaktickou strukturu jazyka, který bude chtít zpracovávat, může využít této metody, která rekurzivně shora dolů zkoumá pravidla gramatiky. Představitelem této analýzy je prediktivní syntaktický analyzátor.

V zásadě se jedná o takový styl zpracování kódu, který reflektuje napsaná pravidla. Na vstupním kódu neprovádí žádné dodatečné úpravy. Tento analyzátor dopředu ví, co má očekávat. Pakliže je vstupu něco, co by tam být nemělo, analýza končí s chybovým hlášením. Předpokladem tohoto stylu je především takový typ navržené gramatiky, která neobsahuje žádná nejednoznačná pravidla. Tyto jazyky řadíme mezi bezkontextové. Tím je myšleno, že každý zapsaný kus kódu, musí do detailu splňovat přesné specifikace. Analyzátor toho typu pracuje v lineárním čase (Wirth 1996, s. 18-20).

4.2.3 Postup při návrhu jazyka

Hlavním cílem při budování interpretu je třeba mít celistvou představu o několika důležitých aspektech. V prvotní fázi návrhu je třeba uvědomit si, k čemu bude výsledný interpret, respektive jazyk, sloužit. Proto je nutná kompletní představa o návrhu jazyka, nebo jakéhokoliv z řetězeného textu logické struktury. Tím je myšlena především syntaxe jednotlivých příkazů.

Dále je třeba blíže specifikovat, jakého charakteru bude výsledný jazyk. V případě návrhu jazyka, který bude pracovat s proměnnými, je třeba zohlednit, jaké typy vstupních hodnot bude jazyk moci umět zpracovat. Proměnné mohou být dynamického typu, kdy se program v průběhu zpracování sám rozhoduje, jakého je vstupní hodnota charakteru nebo statické typy, kde si programátor bude moci sám definovat, který datový typ chce použít.

Potom je nutné zvážit, jaké typy matematických výrazů bude výsledný jazyk umět zpracovat. Tím je myšleno, zda budou podporovány závorky, volání funkce přímo ve výrazu nebo například podpora unárních operátorů.

Pokud bude jazyk obsahovat rozhodovací podmínky či nějaké typy cyklu, je nutné zakomponovat podporu pro práci s logickými typy výrazů, s čímž přichází další bod uvažování, a to konkrétně o booleovském typu. Variant řešení je opět několik. Jednou z variant řešení

tohoto problému je vyhodnocení logického výrazu do číselné hodnoty, která bude reprezentovat kladné nebo záporné vyhodnocení.

Také je možné přímo zavést logický typ, nicméně se tím výsledná implementace stává obtížnější z důvodu ošetřování, kde všude se tento typ výrazu může vyskytovat a je nutné implementovat obrané mechanismy, které v případě chyby programátora včas zasáhnou a nezpůsobí pád programu nebo jiné nežádoucí či nepředvídatelné chování aplikace.

4.2.4 Inicializace vstupního kódu

V prvotní fázi je nutné upravit vstupní kód do takové podoby, aby se mohl použít pro další zpracování. V této surové formě může obsahovat velmi mnoho bílých mezer, které bude třeba vhodným algoritmem odchytit a odstranit.

Dále bude nutné zjistit, co se na vstupu nachází. Může se jednat o klíčové slovo, identifikátor, separátor, číslo nebo třeba řetězec. Podle potřeb lze registrovat i pozici kódu v editoru, kterou je možné použít například pro lepší orientaci, v kterých místech nastala chyba. Po tomto kroku bude výstupem kontejner typu třídy, uchováající následující data (Wirth 1996, s. 9-12).

4.2.5 Obecný návrh struktury interpretu abstraktního syntaktického stromu

Základní myšlenkou je každému klíčovému slovu, jež bude představovat příkaz, vytvořit třídu či strukturu, která bude obsahovat vhodné atributy v závislosti charakteru daného příkazu.

Tím pádem je nutné definovat nějaké rozhraní, ze kterého budou vycházet všechny definované příkazy. Pro implementaci třídy, která bude představovat cyklus, bude ve své třídě potřebovat deklarovat proměnnou typu výraz, do které bude uložena podmínka.

Poté bude třeba použít vhodný datový kontejner, do kterého bude možné ukládat hodnoty typu příkaz. Příkaz bude stejně jako výraz rozhraním, kde jednotlivé implementace budou představovat konkrétní příkazy.

Tyto atributy jsou naprosto dostačující proto, aby mohl být cyklus zrealizován. Obdobným postupem může být řešena třída reprezentující podmínku, akorát zde bude třeba přítomnosti dalšího datového kontejneru. V případě kladného vyhodnocení výrazu, budou spuštěny příkazy z prvního kontejneru, a v případě záporného vyhodnocení, pokud existují, budou vykonány z druhého.

Třída funkce bude obsahovat kontejner, který uchovává názvy vstupních atributů. Zde není třeba žádné speciální třídy, a stačí tak s názvy pracovat jako s obyčejným textem. Dalším atributem, který mimo jiné definuje jedinečnost funkce, bude název.

Posledním atributem bude kontejner s příkazy. Existence funkcí kromě deklarace potřebuje také třídu, která bude představovat takový typ příkazu, jež bude dané funkce umět spouštět. Třída bude obsahovat kontejner uchovávající vstupní parametry ve formě výrazů a proměnou definující volanou funkci.

Třída bude implementovat rozhraní pro příkaz a pro použití funkce v závislosti na jejím výstupu také rozhraní pro výraz. Tím pádem je nutné počítat také s příkazem, který bude moci kdykoliv funkci opustit a vrátit nějaký výraz. Vytvoříme tedy třídu, která bude obsahovat proměnou typu výraz a implementaci nějakého mechanismu, který zaručí, že bude funkce ukončena.

Nicméně i pro práci s proměnnými budou potřeba nějaké vhodné třídy. Tyto třídy již budou ovšem velice triviální. Deklarace proměnné lze realizovat třídou, která uchovává název proměnné spolu s výrazem. Dále je třeba třídy, která bude schopna modifikace již definované hodnoty proměnné. Třída bude naprosto shodná s třídou pro deklaraci.

Při implementaci tak bude naprosto jasné, co programátor vyžaduje, a navrhování logiky bude tak díky tomuto návrhu jednodušší.

4.2.6 Analýza vstupního kódu

Zde bude potřeba třídy, která bude výstup inicializační části zpracovávat. Předchozí část kapitoly popsala struktury tříd, do kterých bude možné ukládat informace o jednotlivých příkazech. Níže vytvořená třída bude tyto informace vytvářet a řetězit je do struktury abstraktního syntaktického stromu.

Třída bude obsahovat metody starající se o jednotlivé příkazy, včetně metod zpracovávající výrazy. Každá metoda bude mít v závislosti na příkazu přesně definovanou strukturu. Například metoda zpracovávající příkaz pro deklaraci proměnné, bude očekávat klíčové slovo, které tento příkaz definuje.

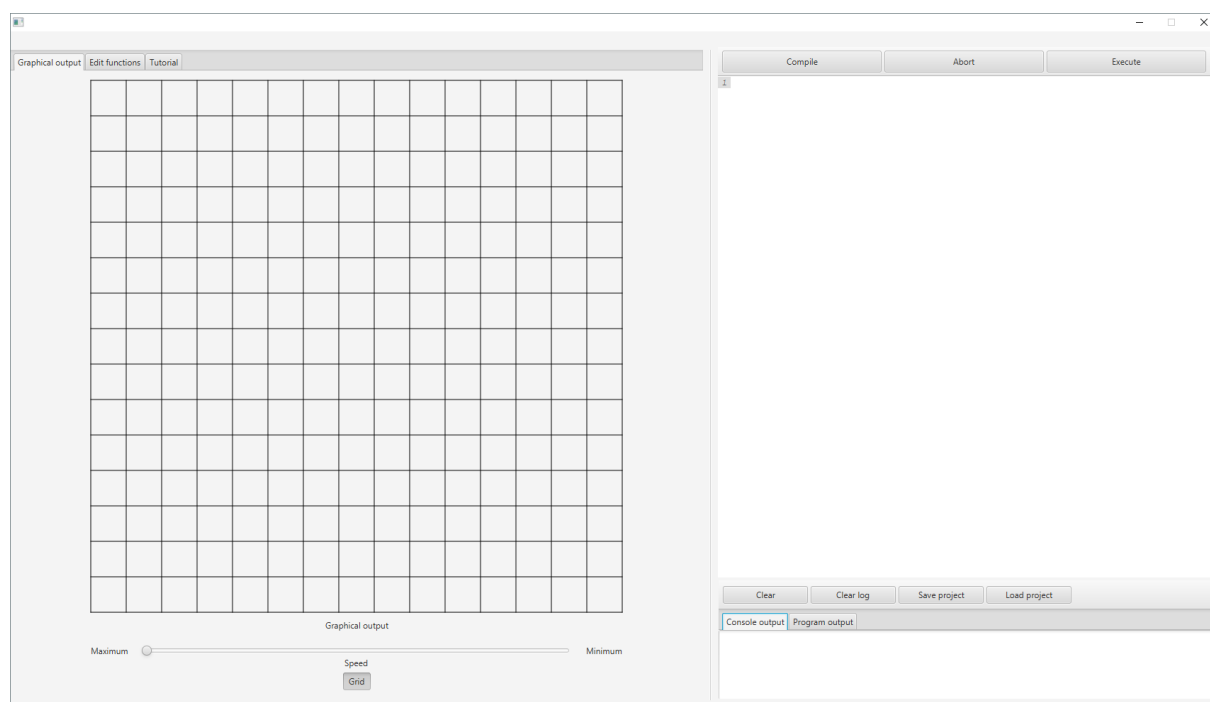
V případě že je tento příkaz v kontejneru přítomen, pokračuje se s kontrolou dalších pravidel. V případě neočekávaného vstupu analýza končí chybovým hlášením o konkrétní chybě a v nejlepším případě vypíše, který kus kódu byl očekáván a případně který byl místo něho nalezen.

5 REALIZACE CÍLOVÉ APLIKACE

4.1 Funkcionality

Aplikace nabízí studentovi poměrně bohaté vývojové prostředí s širokou škálou dostupných funkcí. Zóna pro vstup programového kódu zbarvuje klíčová slova a znaky, aby se dalo v napsaném kódu lépe orientovat.

K dispozici je také číslování řádků poskytující možnost velmi rychlé opravy nevalidní části. Prostor obsahuje textový výstup pro výpis informací, které v kódu definuje programátor a odděleně pro výstup chybových hlášení. Součástí je i grafický výstup, který je zde ve formě matice 15 x 15, kam je možné vykreslovat jednotlivé bloky.



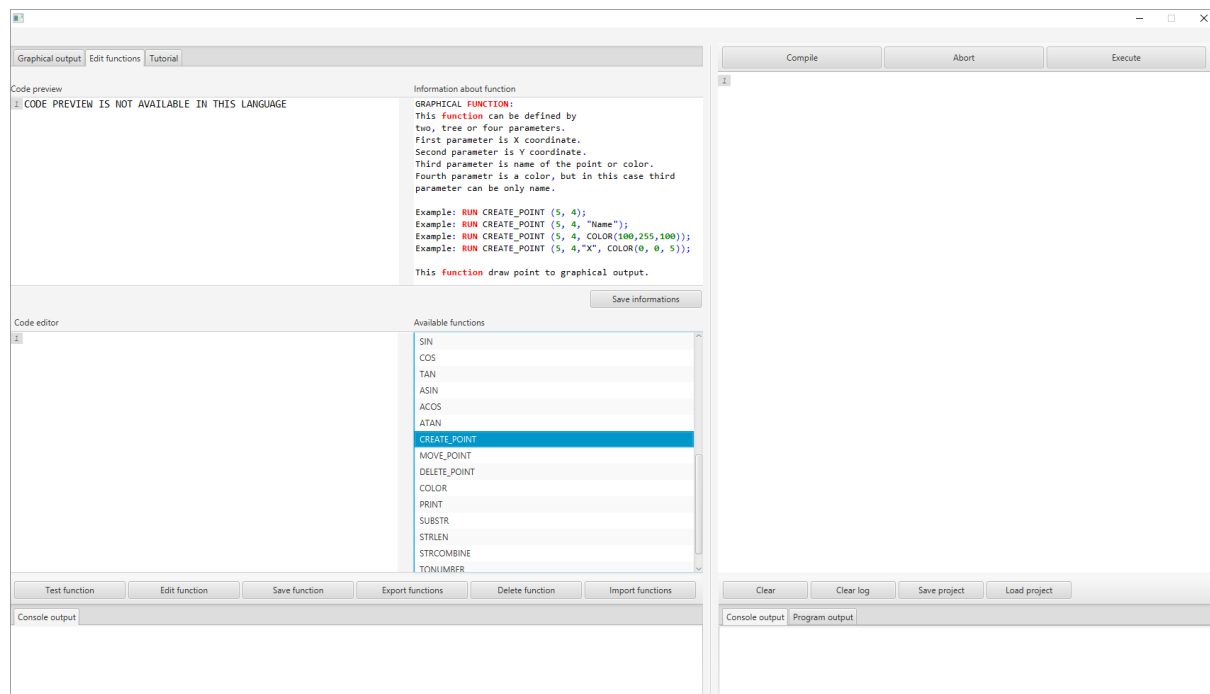
Obrázek 6: Ukázka hlavní části aplikace bezprostředně po jejím spuštění

Zdroj: vlastní.

Celá aplikace je kompletně v anglickém jazyce, a to z toho důvodu, že znalost tohoto jazyka je z pohledu programátora prakticky nezbytná. Existují ovšem i kvalitní materiály, které jsou k dispozici v českém jazyce, nicméně jejich množství je velmi limitováno což může programátora celkově omezit.

Na obrázku 6, v levé části je pro přehlednost vykreslena mřížka představující celkový rozsah grafického výstupu, kterou je možné potlačit. Tlačítko *Compile* slouží pouze k sestavení kódu a kontroluje správnost přítomných příkazů z pohledu gramatiky. Tlačítko *Execute* již zpracovaný kód vykoná.

Tlačítko *Abort* je zde hlavně z důvodu, aby programátor mohl přerušit grafický výstup, nebo odblokovat program v případě nechtěného zacyklení. Student má možnost si napsaný kód uložit do souboru a opět jej načíst. Rychlost animace grafického výstupu lze řídit posuvníkem pod mřížkou, kterou ovšem nelze řídit za běhu.



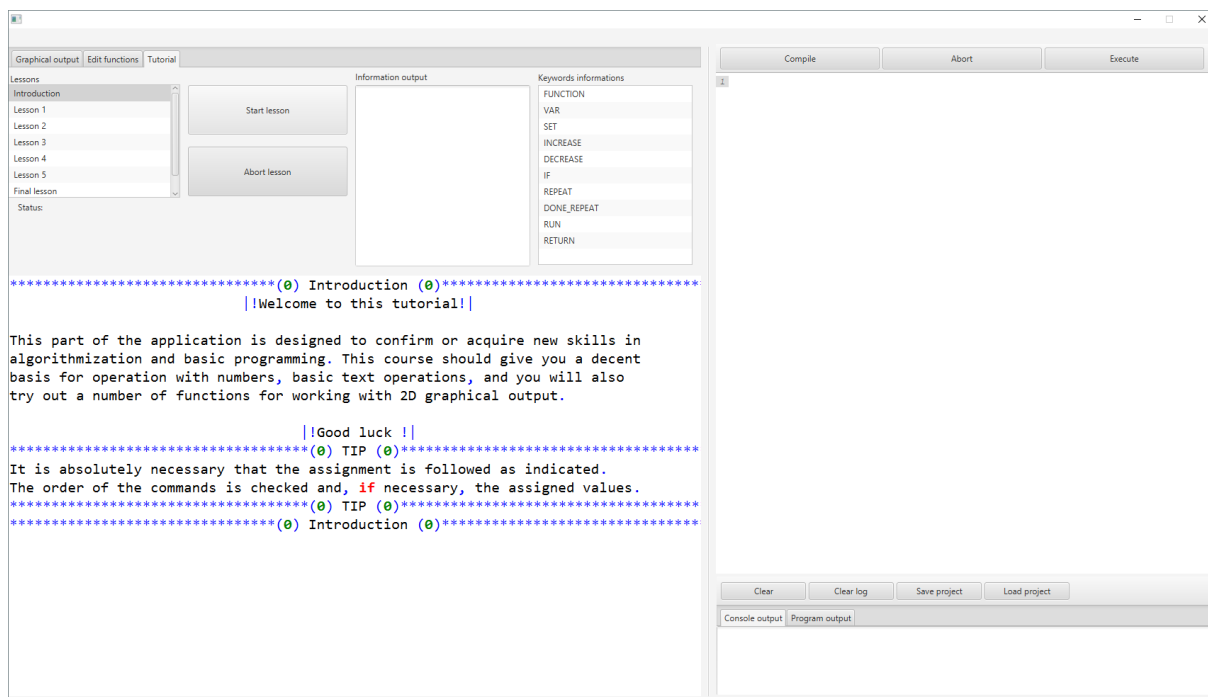
Obrázek 7: Ukázka záložky *Edit function*

Zdroj: vlastní.

Tato záložka je zde určená čistě pro funkce. Aplikace nabízí možnost testování vlastních funkcí a následně jejich ukládání. Veškeré dodatečně vytvořené funkce lze importovat do binárního souboru včetně možnosti jejich opětovného načtení.

Ke každé funkci je možné přidat vlastní komentář, který popisuje, co daná funkce vykonává. Aplikace již má v sobě zabudované interní funkce včetně jejich podrobného popisu, jak fungují a jak se zapisují na konkrétních příkladech.

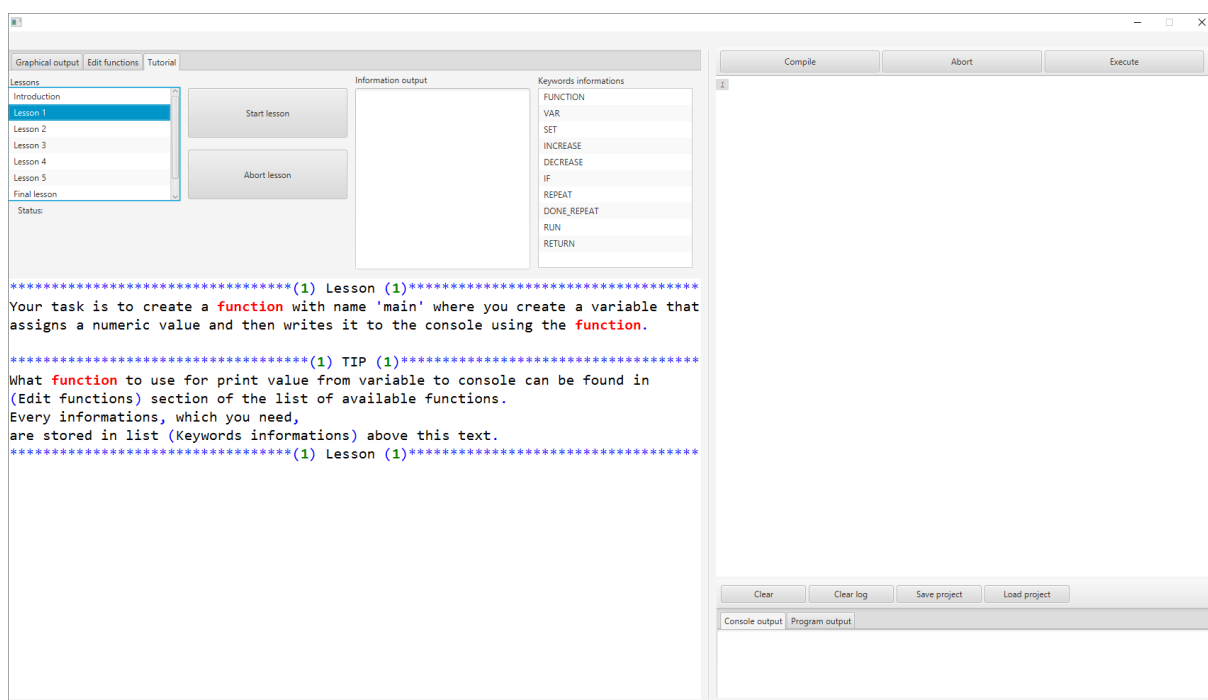
Podpora je zde pro matematické výrazy, řetězce a jednoduché funkce poskytující možnost grafického výstupu. Jejich vymazání je nemožné, stejně tak modifikace, protože tyto funkce nejsou napsány v navrženém jazyce.



Obrázek 8: Ukázka záložky *Tutorial*

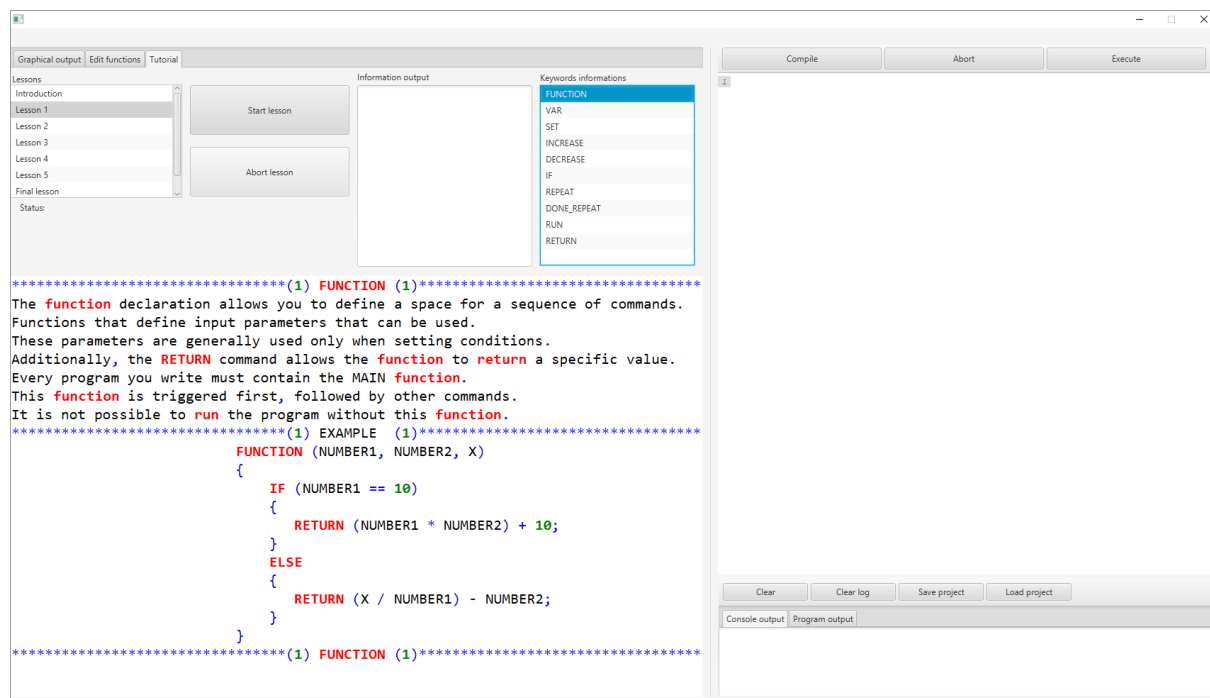
Zdroj: vlastní.

Poslední záložka je stěžejní část výukové aplikace. Pro studenta je zde připraveno pět lekcí. Každá z těchto lekcí obsahuje přesný popis, jak by měl student konkrétní program napsat. Finální lekce je testem, který ověřuje, že student chápe používání veškerých příkazů včetně pravidel jazyka. Kontrola správnosti jednotlivých lekcí je plně automatizována.



Obrázek 9: Ukázka první lekce. Zdroj: vlastní.

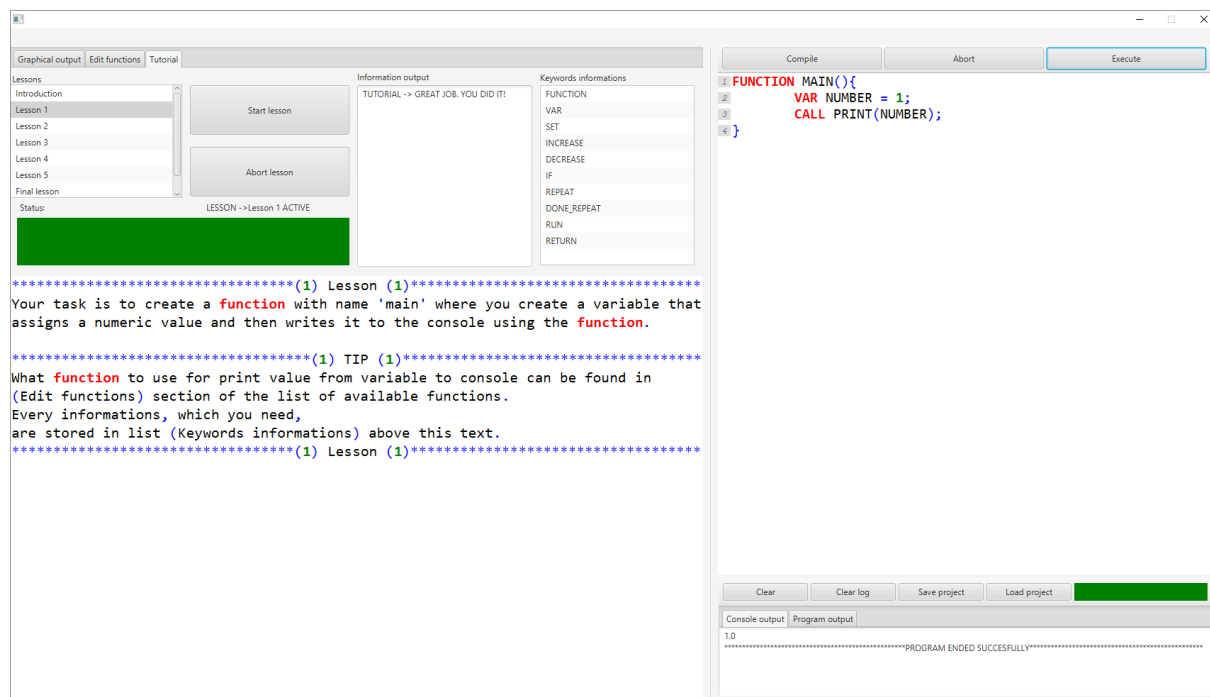
Všechny dostupné příkazy jsou podobně jako interní funkce formou komentáře a ukázky plně k dispozici v levé části záložky.



Obrázek 10: Ukázka popisu příkazu *FUNCTION*. Zdroj: vlastní.

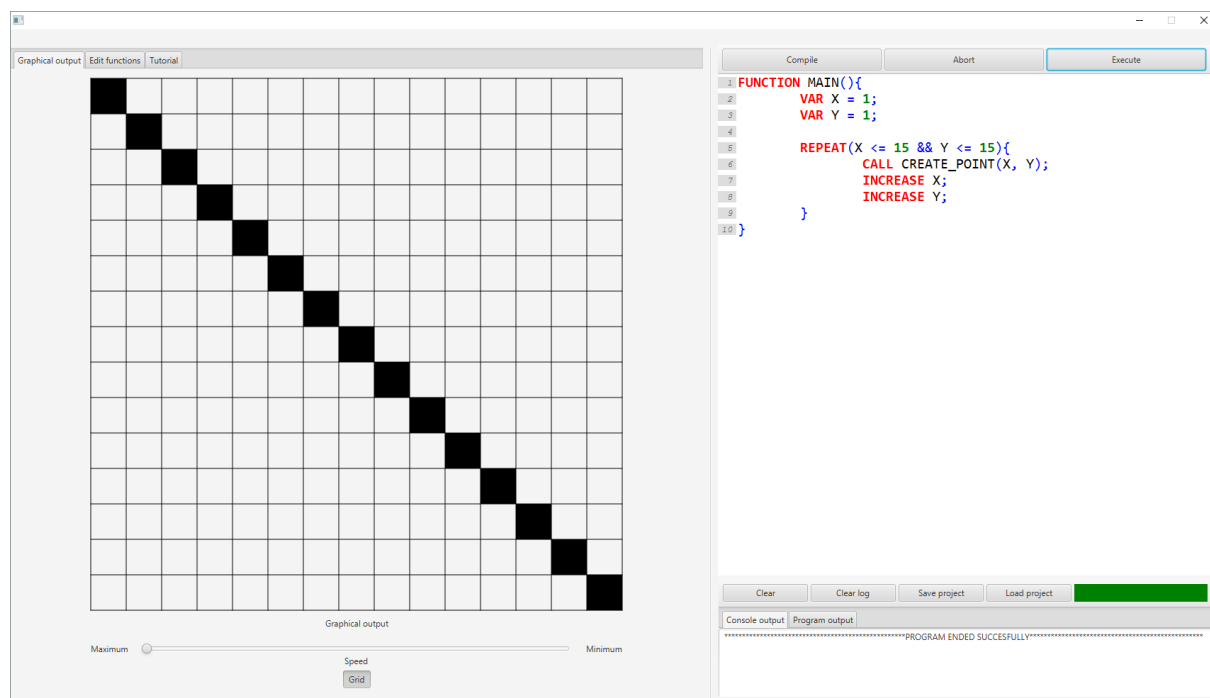
4.2 Představení řešení a výstupu aplikace

Následující program bude demonstrace postupu řešení první lekce.



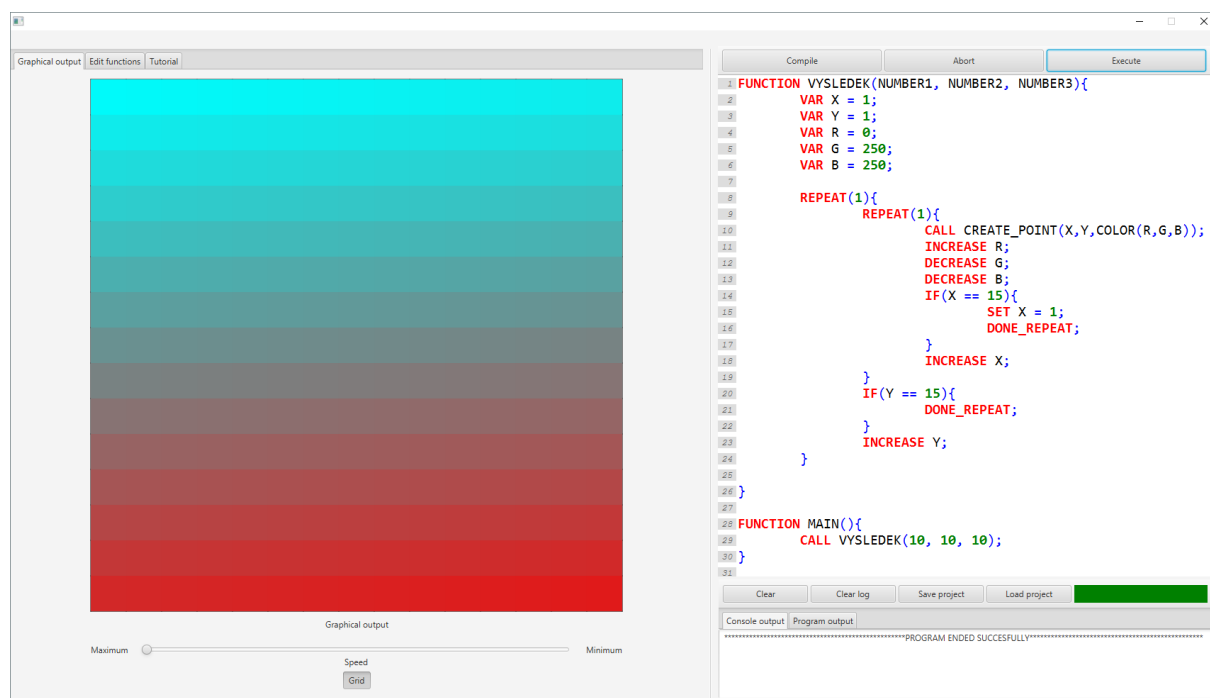
Obrázek 11: Demonstrace úspěšného řešení první lekce. Zdroj: vlastní.

Na obrázku 11 je požadováno deklarovat hlavní funkci, ve které deklaruje proměnou libovolného výrazu a následně jej vhodnou funkcí vypíše do konzole.



Obrázek 12: Program vykreslující pomocí cyklu hlavní diagonálu. Zdroj: vlastní.

Následující ukázka vykresluje pomocí jednoduchého cyklu diagonálně černé bloky. Pro grafické prostředí je k dispozici funkce, která má umožňuje posun již existujících bodů.



Obrázek 13: Vyplnění matice s postupnou změnou barev v každé iteraci. Zdroj: vlastní.

Komplexnější program nabízí dva vnořené cykly umožňující postupně vykreslit všechna pole v matici. Postupně klesající modrá a zelená složka barev se stoupající červenou vytvoří efekt zřetelně viditelný na obrázku 13.

V případě překročení definovaného rozsahu se hodnota upraví na hodnotu opačnou, takže se matice jeví jako nekonečné pole. V případě pokusu o vytvoření bloku, který by se nacházel mimo rozsah matice je příkaz ignorován.



Obrázek 14: Vyplnění matice s náhodnou změnou barev v každé iteraci

Zdroj: vlastní.

Tento program oproti předchozímu pomocí interních funkcí COLOR a RND generuje v každé iteraci náhodnou barvu z definovaného rozsahu pro jednotlivé bloky.

ZÁVĚR

Po několika měsících vývoje je výsledkem této práce plně funkční výuková aplikace, která obsahuje mnoho funkcionalit nezbytných pro výuku. Přítomnost jednoduchého grafického prostředí uživateli umožňuje pracovat s grafickými elementy jen za použití interních funkcí bez hlubší znalosti obecného fungování grafiky, což může být velmi silným podnětem pro vytváření vlastních programů právě v této aplikaci.

Díky možnosti exportu vlastních funkcí může být výhodou, zejména při vytváření vlastních složitějších programových struktur, které je možné sdílet spolu s ostatními uživateli a tím tak rozšiřovat možnosti při vývoji.

Práce na tomto projektu byla výbornou zkušeností zejména při návrhu vlastní verze programovacího jazyka či algoritmu pro jeho zpracování. Aplikace byla testována dobrovolníky, kteří se podíleli zejména na odhalování chyb. Předmětem testů byla zahrnuta i přívětivost uživatelského prostředí, které dopadly výborně. Většina dobrovolníků prošla všemi lekcemi bez jediného problému a následně byly schopni psát vlastní verze programů, což lze považovat za celkový úspěch.

Navržený jazyk je vhodný i pro naprosté začátečníky, což je velmi výhodné zejména v oblasti výuky naprostých základů. Výhodou také je, že se uživatel nemusí soustředit na složitou syntaxi jazyka a může tak věnovat čas návrhu programového řešení jemu zvolenému algoritmu. Intuitivní grafické prostředí poskytuje solidní základ pro práci s programovým kódem, kde se jeho dílčí části kódu specificky zbarvují, čímž přináší velmi přehlednou strukturu.

POUŽITÁ LITERATURA

DOS REIS, Anthony J. *Compiler construction using Java, JavaCC, and Yacc*. Hoboken, N.J.: Wiley-IEEE Computer Society, c2012. [cit. 2019-04-21]. ISBN 9780470949597.

HLAVICA, Tomáš. *Od strojového kódu k programovacím jazykům*. Masarykova Univerzita, [online]. 2004. [cit. 2019-04-21]. Dostupné z: <https://www.fi.muni.cz/usr/jkucera/pv109/sl5.htm>

LAŠTOVIČKA, Petr. *Robot KAREL 4.2* [online]. 6.9.2011 [cit. 2019-04-22]. Dostupné z: <http://petr.lastovicka.sweb.cz/ostatni.html#karel>

JONES, JoelRoy. *Abstract Syntax Tree Implementation Idioms*. Department of Computer Science. University of Alabama, 2009 [cit. 2019-04-21]. Dostupné také z: <https://www.hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf>

PETERKA, Jiří. Compiler vs. interpreter. *Computerworld* [online]. 1995, **95**(6), 1 [cit. 2019-04-21]. Dostupné z: <http://www.earchiv.cz/a95/a506c120.php3>

RADUTA, Bogdan. Top 13 Most Absurd Programming Languages. *Top Design Magazine – Web Design and Digital Content* [online]. Copyright © 2010 [cit. 22.04.2019]. Dostupné z: <http://www.topdesignmag.com/top-13-most-absurd-programming-languages/>

VAN ROY, Peter. *Programming Paradigms for Dummies: What Every Programmer Should Know* [online]. 2012 [cit. 2019-04-21]. Dostupné z: https://www.researchgate.net/publication/241111987_Programming_Paradigms_for_Dummies_What_Every_Programmer_Should_Know

WIRTH, Niklaus. *Compiler construction*. Reading, Mass.: Addison-Wesley Pub. Co., c1996. [cit. 2019-04-21]. ISBN 02-014-0353-6.