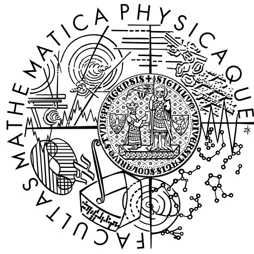


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Jiří Frantál

Didaktická visualizace grafových algoritmů

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce:	RNDr. Martin Pergel, Ph.D.
Studijní program:	Matematika
Studijní obor:	Matematika se zaměřením na vzdělávání - Informatika se zaměřením na vzdělávání

Praha 2020

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V .... dne ....

Podpis autora

V první řadě bych chtěl poděkovat své rodině, která mi po celou dobu práce poskytovala psychickou podporu a obecně mi zajišťovala to nejlepší zázemí, abych se mohl věnovat především své bakalářské práci a studiu.

Dík patří i mému kamarádovi Ráďovi, a to především za vytrvalou podporu a za to, že mě nutil do práce, i když se mi zrovna příliš nechtělo. Rovněž si dovolím za duševní podporu poděkovat jmenovitě i svým kamarádkám Majdě a Denče; a všem ostatním, kteří se starali o to, abych si během nouzového stavu a práce na bakalářské práci udržel zdravý rozum.

Na závěr bych chtěl také poděkovat RNDr. Martinu Pergelovi, Ph.D., vedoucímu mé bakalářské práce.

Název práce: Didaktická visualizace grafových algoritmů

Autor: Jiří Frantál

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: Tato práce se zabývá návrhem programovacího jazyka specializovaného na vizualizaci grafových algoritmů, vytvořením jeho interpretu a vývojového prostředí. Cílem bylo, aby byl výsledný program využitelný pro výuku grafových algoritmů a jejich implementace, a to především na středních školách. Na základě odborných didaktických poznatků se povedlo navrhnout jazyk podobný Pascalu, který byl upraven pro potřeby grafových algoritmů. Součástí práce je i stručné pojednání o interpretech a několik ukázkových grafů a algoritmů ilustrujících práci s naším programem.

Klíčová slova: didaktika, grafové algoritmy, interpret

Title: Didactic graph visualisation

Author: Jiří Frantál

Department: The Department of Software and Computer Science Education

Supervisor: RNDr. Martin Pergel, Ph.D., The Department of Software and Computer Science Education

Abstract: This thesis deals with the design of programming language specialized in visualisation of graph algorithms, creating its interpreter and integrated development environment. The aim was to make the created program usable for education of graph algorithms and their implementation, especially at high school. Based on didactic knowledge, we designed language similar to Pascal, but adapted for the needs of graph algorithms. Part of the thesis is also brief introduction to interpreters and several exemplary graphs and algorithms ilustrating work with our program.

Keywords: didactic, graph algorithms, interpreter

Obsah

Úvod **3**

1 Návrh jazyka **4**
1.1 Rozbor požadavků . . . 4
1.1.1 Pro koho je jazyk určen . . . 4
1.1.2 Obecné vlastnosti jazyka . . . 5
1.1.3 Konkrétní prvky jazyka . . . 6
1.2 Finální podoba návrhu . . . 7
1.2.1 Výběr výchozího jazyka . . . 7
1.3 Úpravy jazyka pro grafové algoritmy . . . 8
1.3.1 Specifikace grafu a implementace grafového algoritmu . . . 9
1.3.2 Vrcholy a hrany . . . 10
1.4 Výběr nástrojů . . . 11

2 Uživatelská příručka **13**
2.1 Spuštění programu . . . 13
2.2 Vývojové prostředí . . . 13
2.2.1 Textové pole pro zdrojový kód . . . 14
2.2.2 Tlačítka pro práci se zdrojovým kódem . . . 14
2.2.3 Výpis výstupu programu . . . 15
2.2.4 Panel pro vykreslení grafu . . . 15
2.2.5 Prvky ovládající běh programu a vykreslování . . . 17
2.3 Programovací jazyk a jeho základní vlastnosti . . . 18
2.4 Zdrojový kód algoritmu . . . 18
2.4.1 Struktura kódu . . . 19
2.4.2 Typy proměnných . . . 21
2.4.3 Operátory . . . 27
2.4.4 Příkazy . . . 27
2.4.5 Vestavěné funkce . . . 31
2.4.6 Komentáře . . . 32
2.5 Zdrojový kód grafu . . . 32
2.5.1 Struktura kódu . . . 33
2.5.2 Typy proměnných . . . 34
2.5.3 Příkazy . . . 35
2.5.4 Komentáře . . . 36

3 Programátorská dokumentace **37**
3.1 Analýza kódu . . . 37
3.1.1 Lexikální analýza . . . 37
3.1.2 Syntaktická analýza . . . 38
3.1.3 Vyhodnocování . . . 40
3.2 Struktura programu . . . 41
3.2.1 Balíček *drawing* . . . 41
3.2.2 Balíček *graphTools* . . . 41
3.2.3 Balíček *language.actions* . . . 41

3.2.4 Balíček *language.loading* 44
3.3 Kreslení grafu 45

4 Ukázkové algoritmy **47**
4.1 Kruskalův algoritmus 47
4.2 Algoritmus pro hledání silně souvislých komponent grafu 49
4.3 Simulace běhu konečného deterministického automatu 51

Závěr **53**

Seznam použité literatury **54**

Seznam obrázků **56**

Seznam tabulek **57**

A Přílohy **58**

Úvod

Grafové algoritmy jsou jednou z významných skupin algoritmů, s nimiž se žáci středních škol setkávají. V běžných programovacích jazycích (a jejich vývojových prostředích) však není možnost přehledně vizualizovat jejich běh bez předchozí implementace složitějšího grafického rozhraní.

Na internetu existuje mnoho animací prezentujících fungování konkrétních grafových algoritmů (za všechny vzpomeňme například vizualizace na *Wikipedii* či *YouTube*). Hlavní vevýhodou těchto animací je jejich neinteraktivita. Žák sleduje běh algoritmu na předem zvoleném grafu, který nemůže měnit a nemůže si tak ověřit, co se stane při jeho modifikaci — např. chování Dijkstrova algoritmu (Matoušek a Nešetřil, 2002) na grafu se zápornými hranami či Jarníkova algoritmu (Matoušek a Nešetřil, 2002) na nesouvislém grafu.

Naším cílem bylo vytvořit programovací jazyk určený pro implementaci grafových algoritmů, který by se žákům střední školy dobře ovládal, a vývojové prostředí, které bude schopné dostatečně dobře vizualizovat běh daného algoritmu a umožnit krokování algoritmu i změnu rychlosti jeho vykonávání.

V první kapitole této práce se podrobněji podíváme na zadání našeho úkolu. Na základě didaktických poznatků se podíváme na požadavky, které na náš jazyk budeme klást, a nakonec navrhneme náš nový programovací jazyk. Popíšeme si jeho vazbu na jiné programovací jazyky a prvky, které jazyk přizpůsobují práci s grafy. Na závěr se stručně zmíníme o výběru nástrojů, které nám pomohly náš jazyk vytvořit.

Druhá kapitola představuje uživatelskou příručku s podrobným popisem struktury jazyka. Řekneme si, jak náš program spustit a popíšeme si podrobně vývojové prostředí, v němž budeme s naším jazykem pracovat. Poté se podíváme na strukturu zdrojového kódu našeho programovacího jazyka a pomocí přehledných příkladů se seznámíme s jeho datovými typy a strukturami, operátory, příkazy a vestavěnými procedurami a funkcemi, jakožto i s deklarací procedur a funkcí vlastních.

Třetí kapitola je pak příručkou programátorskou, v níž si popíšeme podrobnosti implementace našeho programu, včetně stručného úvodu do interpretů (jelikož náš jazyk je interpretovaný). Seznámíme se tedy s lexikální a syntaktickou analýzou a nakonec uvedeme přehled jednotlivých balíčků spolu se třídami, které obsahují, včetně vzájemné souvislosti jednotlivých balíčků. Na závěr se zmíníme o způsobu, kterým budeme vykreslovat grafy v našem programu.

Ve čtvrté kapitole se podíváme na ukázkovou implementaci několika konkrétních algoritmů, abychom se prakticky seznámili s naším novým jazykem. Dostaneme tak možnost udělat si představu, jak vypadá v našem jazyce rozsáhlejší kód, a zároveň uvidíme ukázku různých typů grafových algoritmů, které je v našem jazyce možné implementovat.

1. Návrh jazyka

V této kapitole si rozebereme zadání naší práce a řekneme si, jak budeme tvořit náš programovací jazyk.

V závěru si rozebereme také motivace výběru použitých nástrojů.

1.1 Rozbor požadavků

Při návrhu programovacího jazyka byl brán ohled zejména na jeho zamýšlený účel, kterým je využití pro didaktické účely, konkrétně pro výuku grafových algoritmů (a jejich implementace) na střední škole.

Podívejme se podrobně na jednotlivé požadované aspekty jazyka a pokusme se zamyslet, jak jej navrhnout, aby danému účelu sloužil do nejlépe.

1.1.1 Pro koho je jazyk určen

Hubert a Stuart Dreyfusovi uvádí pět stádií vývoje programátora (Robins, Rountree a Rountree, 2003): **nováček** (novice), **pokročilý začátečník** (advanced beginner), **způsobilý** (competence), **zdatný** (proficiency) a **odborník** (expert).

Vytváříme jazyk určený s primárním využitím pro výuku na středních školách, uživatelé našeho jazyka (dále pro přehlednost označováni jako *žáci*) budou tedy nejspíše spadat do některé z počátečních kategorií. Chystáme-li se zaměřit náš program na výuku grafových algoritmů, můžeme předpokládat, že žáci již mají jisté základní programátorské znalosti a není nutné je učit základům programování. Můžeme si tedy bez problémů stanovit, že vetšina uživatelů bude spadat do druhé kategorie, tedy půjde o pokročilé začátečníky. To mimochodem také znamená, že můžeme očekávat, že žáci již ovládají nějaký programovací jazyk — byť třeba jazyk didaktický (např. Scratch či Pascal).

Jens Kaasbøll uvádí mezi didaktickými přístupy tzv. „sémiotický žebřík" (semiotic ladder), kde na nejnižší příčce stojí **syntaxe** (syntax), výše pak **sémantika** jazyka (semantics) a nejvýše **pragmatika** (pragmatics) (Kaasbøll, 1998). Aby se mohl žák plně soustředit na vyšší příčku, musí nejdříve zvládnout příslušné příčky nižší — tedy má-li se žák soustředit v našem případě na pochopení grafových algoritmů (což je cílem výuky v našem jazyce), musí dobře ovládat syntax a sémantiku našeho jazyka.

Podobně i Benedict Du Boulay uvádí jako dovednosti, které musí programátor zvládnout, **základní orientaci** (general orientation) v tom, k čemu programy slouží, **představu o stroji** (notional machine), tedy o fungování počítače jako takového, **notaci** (notation), tedy syntax a sémantiku, **struktury** (structures), čili základní schémata programování a **pragmatiku** (pragmatics), samotný vývoj programů (Robins a kol., 2003).

Chceme-li tedy zaměřit pozornost žáků na pochopení grafových algoritmů a jejich implementace, jinými slovy na pragmatiku jazyka, bylo by vhodné navrhnout syntaxi a sémantiku našeho nově vytvářeného jazyka co nejpodobnější něčemu, co už by žák mohl znát. Jinými slovy náš jazyk by měl být podobný nějakému

již existujícímu jazyku, u nějž je možnost, že jej žák, jenž se chystá používat náš jazyk, bude ovládat.

1.1.2 Obecné vlastnosti jazyka

Vizuální versus textový jazyk

Jedním z možných dělení programovacích jazyků (zvláště pak dětských) je na jazyky **vizuální** (či také **grafické**) a jazyky **textové** (Hornik, Musílek a Milková, 2019).

První skupina označuje jazyky, kde žák kód skládá z předpřipravených částí (typickými představiteli této skupiny jsou např. Scratch či Baltík). Výhodou těchto jazyků je, že odpadá nutnost učit se syntaxi jazyka — žák pouze skládá jednotlivé prvky k sobě jako dílky skládačky, a soustředí se tak pouze na pragmatickou část programování.

Do druhé skupiny patří jazyky, kde žák píše celý kód ručně (tedy většina pokročilých programovacích jazyků, např. Java či Python, ale i některé výukové, např. Pascal nebo Karel). Nevýhodou je větší chybovost programátorů, která může být způsobena nedostatečnou znalostí syntaxe, nepozorností či prostě překlepy, výhodou naopak je, že se těmito jazyky žák učí pečlivosti a připravuje se tak i na potenciální budoucí přechod na profesionální jazyky, které jsou běžně textové (Hornik a kol., 2019).

Pro tyto výhody jsme se rozhodli navrhnout náš jazyk jako textový. Vizuální jazyk by si mimo to vyžadoval větší znalosti ve tvorbě interpretů a bylo by náročnější navrhnout grafickou podobu vývojového prostředí tak, aby odpovídala některému z používaných didaktických jazyků.

Restriktivní versus volný jazyk

Dále můžeme dětské programovací jazyky rozdělit podle toho, co vše umožňují a dovolují svým uživatelům dělat, na jazyky **restriktivní**, které jsou vysoce specifické (a to i svými příkazy) a zaměřené na konkrétní programovací problémy, a **volné**, které umožnují svým uživatelům větší míru svobody se vývoji programů (Hornik a kol., 2019).

Ačkoli plánujeme vyvíjet jazyk pro výuku grafových algoritmů, budeme se klonit spíše k volnému typu jazyka. Příkazy v našem jazyce by měly být co možná nejobecnější, což si můžeme dovolit i díky tomu, že cílíme na žáky, kteří již mají jisté programátorské znalosti — a musíme tak učinit i proto, že se snažíme syntakticky a sémanticky navázat na již existující jazyk, v němž žáci umí programovat.

Objektový versus neobjektový jazyk

Françoise Détienneová poukazuje ve svých výzkumech na přirozenost objektově orientovaného programování, které je jednoduché na osvojení. Testované subjekty však měly předtím, než se začaly učit oběktově orientované programování, již jisté znalosti v neobjektovém programování (Robins a kol., 2003).

Susan Wiedenbecková zkoumala studenty druhého ročníku univerzity, kteří se učili buď objektový jazyk (C++), nebo neobjektový jazyk (Pascal). Ukázalo se,

že u krátkých programů nebyly zřetelnější rozdíly (pouze studenti učící se objektovému jazyku vykazovali lepší porozumění funkci programu), u delších programů si však vedli studenti učící se neobjektový jazyk lépe, což autorka přisuzuje složitější orientaci v řídících strukturách kvůli jejich komplikovanějšímu rozložení v kódu (Robins a kol., 2003).

Náš jazyk pojmeme jako neobjektový, jednak ve světle těchto výzkumů a jednak z prosté úvahy, že vzhledem k zaměření programu (grafové algoritmy) nebudeme potřebovat vytvářet vlastní objekty — jediné složitejší struktury v našem jazyce budou vrcholy a hrany, pro něž ale raději zavedeme speciální datové typy.

1.1.3 Konkrétní prvky jazyka

Řídící struktury

Řekli jsme si, že budeme směřovat vývoj našeho jazyka spíše k volnému přístupu. Budeme tedy požadovat veškeré běžné řídící struktury — podmínky (bez *else* větve i s ní), příkaz *switch*, *while-do* (případně *do-while*) cyklus a *for* cyklus. Vzhledem k tomu, že v mnoha grafových algoritmech chceme provést nějakou akci pro všechny vrcholy či hrany z nějaké množiny, budeme potřebovat kromě *for* cyklu přes čísla v nějakém intervalu i variantu *for-in* procházející všechny prvky v dané množině.

Samozřejmou součástí našeho jazyka musí být i přiřazovací příkaz.

Datové typy

Ze stejného důvodu budeme požadovat i všechny základní datové typy — logické hodnoty `true` a `false`, celá i reálná čísla (včetně nekonečných hodnot) a znaky. Vzpomeňme také, že žáci už umí programovat, bude nám tedy stačit pro reprezentaci daného druhu hodnot pouze jeden datový typ. Nebudeme tedy rozlišovat např. mezi jednobytovými a vícebytovými celými čísly a jejich znaménkovou a bezznaménkovou reprezentací či velikostí mantissy a exponentu u čísel reálných.

Zejména kvůli vykreslování budeme požadovat i datový typ označující barvu. Tu lze v běžné praxi implementovat pomocí jiných typů (typicky celočíselných hodnot), nám jde ale o grafickou vizualizaci, kde bude pohodlnější pracovat rovnou s barvou namísto pouhé její reprezentace.

Pro úplnost našeho návrhu budeme požadovat i datový typ reprezentující textový řetězec.

Datové struktury

Nejzákladnější datovou strukturou, která se nachází ve většině profesionálních i didaktických jazyků, je pole. Samozřejmě ho budeme chtít zahrnout i do návrhu našeho jazyka. Vzhledem k tomu, že se jedná o jazyk didaktický, bude nám stačit pole s pevnou délkou, neboť množství prvků, s nimiž budeme chtít pracovat, nebude zpravidla příliš velké, nebo bude dopředu známé (zadané učitelem).

Z pokročilejších struktur se pro implementaci grafových algoritmů hodí mít v jazyce zásobník a frontu (klasickou i prioritní). Součástí těchto struktur musí být samozřejmě i vestavěné metody pro přidání a odebrání prvku.

V řadě grafových algoritmů potřebujeme rozdělit prvky, zejména vrcholy či hrany, do množin (např. Jarníkův či Dijkstrův algoritmus). To je v praxi řešeno zpravidla proměnnou v daném vrcholu či hraně, která uchovává identifikátor množiny (typicky číselnou hodnotu). Z didaktických důvodů se bude hodit implementovat množinu, do níž bude možné vložit určité prvky.

Vzhledem k tomu, že hlavní funkcí našeho jazyka bude práce s grafy (tj. s vrcholy a hranami, pro které, jak jsme uvedli výše, budeme mít v našem programu zvláštní datové typy), není nutné, aby náš program obsahoval možnost vytvořit si vlastní datové struktury, tedy ani ukazatele do paměti. Práci s těmito prvky programování si žák může lépe vyzkoušet v jiném programovacím jazyce (případně je již zná z dřívějšího programování).

Uživatelské metody

Kromě existence vestavěných metod bychom chtěli, aby bylo žákům umožněno definovat si vlastní metody. Uživatelské metody by měly mít možnost mít libovolný počet argumentů (tedy mohou být i bez argumentů), které budou moci být předávány jak hodnotou, tak odkazem.

Vstup a výstup

Hlavní funkcí našeho jazyka má být práce s grafy, můžeme tedy bez větších problémů potlačit práci s klasickým textovým vstupem a výstupem, neboť většinu informací bude program zjišťovat z grafu a stejně tak může skrze graf prezentovat i svůj výstup. Práce se soubory může být také potlačena.

1.2 Finální podoba návrhu

1.2.1 Výběr výchozího jazyka

Jak jsme si řekli dříve, budeme vycházet z již existujícího jazyka. Zkusme se zamyslet, který jazyk zvolit jako základ k vývoji našeho nového jazyka.

Hned na začátek uveďme, že nelze (a ani není nutné) uvažovat všechny jazyky používané pro výuku programování. Probereme pouze ty nejčastější z nich.

Vizuální programy, jakými jsou například Scratch nebo Baltík, jsme z výběru vyřadili už v úvaze nad požadavky, kdy jsme se rozhodli, že náš jazyk bude textový. Tyto jazyky příliš staví právě na své vizuální podobě a přetvořit je do podoby textové je nemožné.

Podobně i jazyky specializované na speciální úkoly — jako například Karel (pohyb robota po obrazovce) nebo Logo (vykreslování geometrických útvarů) — jsou jako vzor pro náš nový programovací jazyk nevhodné, protože jsou příliš vázány na svůj specializovaný účel.

Java (a další jí podobné jazyky) není i přes svou popularitu mezi profesionálními programátory doporučována jako didaktický jazyk, zejména pro její složitou syntaxi (Grandell, Peltomäki, Back a Salakoski, 2006; Böszörményi, 1998) a řadu pro začínajícího programátora obtížně pochopitelných konstruktů — např. složité vztahy mezi třídami, nemožnost předávání parametrů odkazem apod. (Böszörményi, 1998). Ty plynou zejména z jejího v základu objektového návrhu, který jsme už dříve vyhodnotili jako nevhodný pro začínající programátory.

László Böszörményi uvádí jako alternativu pro Javu jazyk Modula-3 (Böszörményi, 1998), který je svou syntaxí podobný jazyku Pascal. Samotný Pascal byl od začátku vyvíjen jako didaktický jazyk (Jensen a Wirth, 1974), dal by se tedy považovat za ideální vzor pro náš jazyk. Nejvetší nevýhodou Pascalu je jeho stáří a celková zastaralost vůči moderním jazykům (Grandell a kol., 2006). Brian Kernighan uvádí ve svém článku (Kernighan, 1981) řadu negativ Pascalu jako programovacího jazyka, z nichž však řada byla vyřešena v pozdějších verzích jazyka, např. ve Free Pascalu (van Canneyt, 2017) (např. pořadí vyhodnocování logických výrazů, bitové operace, příkaz *break*, *else* větev v *case* příkazu), nebo nás z didaktického hlediska nezajímají (neoddělený překlad jednotlivých částí programu).

Další alternativou je jazyk Python, který se primárně používá jako profesionální programovací jazyk — což je jeho nesporná výhoda při výuce budoucích programátorů. Mezi jeho hlavní výhody patří zejména jednoduchá syntaxe — kód v Pythonu je podobný pseudokódu (Grandell a kol., 2006) — a možnost přejít plynule od nejjednodušších neobjektových programů k objektovému programování bez nutnosti změny jazyka. Jednoduché syntaxe je však mimo jiné dosaženo i pomocí neinicializace proměnných a dynamickým typováním. To z didaktického hlediska nemusí být vhodné pro začínající programátory a může to způsobovat více chyb v jejich programech (Grandell a kol., 2006).

Jak vidíme, je téměř nemožné vybrat vzorový jazyk, který by byl ve všem ideální. Z požadavků, které jsme si pro náš jazyk stanovili, a z krátkého rozboru některých potenciálních kandidátů vyšel nakonec jako finální volba jazyk Pascal. Přirozeně musíme původní podobu Pascalu upravit tak, aby byla implementace grafových algoritmů v našem novém jazyce co možná nejpohodlnější — kromě datových typů pro vrcholy a hrany Pascal nenabízí například mnoho datových struktur, které bychom v našem novém jazyce požadovali, a některé jeho funkce jsou naopak pro náš účel nadbytečné (např. deklarace vlastních typů) či zcela nevhodné (např. návěští používaná pro *goto* příkaz).

1.3 Úpravy jazyka pro grafové algoritmy

Jak už jsme zmínili, je třeba do našeho nového jazyka přidat některé prvky umožňující či usnadňující implementaci grafových algoritmů, které v původním Pascalu nejsou. Jedná se zejména o nové datové typy **color** (barva), **vertex** (vrchol) a **edge** (hrana) a datové struktury **stack** (zásobník), **queue** (fronta) a **set** (množina). Upozorněme zde, že množina v našem jazyce má být zcela jiná struktura než množina v Pascalu — bude se jednat o strukturu, do níž lze přidávat prvky, zjišťovat incidenci prvku v množině a provádět množinové operace (sjednocení, průnik, rozdíl). Samozřejmě požadujeme, aby každý prvek byl v konkrétní množině právě jednou.

Všechny datové struktury — tedy pole (**array**), zásobník, fronta a množina — budou pro jednoduchost obsahovat pouze jednoduché datové typy. Toto omezení nijak výrazně neomezí možnosti implementace grafových algoritmů.

Nakonec do našeho programu nebyla implementována prioritní fronta, která je potřeba pro řadu grafových algoritmů. To je zejména důsledkem faktu, že návrh prioritní fronty by vyžadoval implementaci lambda výrazů (nebo podobného konceptu, který by nějakým způsobem porovnával dva obecné prvky), což neza-

padá příliš dobře do struktury Pascalu, tedy i v našem jazyce by lambda výrazy působily poněkud nepatřičně.

V případech, kdy je prioritní fronta potřeba, ji lze implementovat například takovouto jednoduchou funkcí (která vybírá z množiny s hranu, která má nejmenší hodnotu proměnné w). Přesné vysvětlení jednotlivých částí kódu uvedeme v kapitole 2.4.

```
function ExtractMin(var s : set of edge) : edge;
var
 min : integer;
 e : edge;
 t : set of edge;

begin
 min := maxInt;
 for e in s do
  if e.w <= min then
   begin
    min := e.w;
    ExtractMin := e;
   end;
 Add(ExtractMin, t);
 s := s - t;
end;
```

Je zřejmé, že tato implementace má časovou složitost $\mathcal{O}(n)$ vzhledem k velikosti množiny s. Pokud bychom požadovali lepší časovou složitost, lze toho dosáhnout například implementací haldy pomocí pole, nebo ještě lépe implementací pomocí grafu — což je vhodný grafový algoritmus, který lze využít jako úlohu pro žáky.

1.3.1 Specifikace grafu a implementace grafového algoritmu

Základním problémem, který je třeba vyřešit, je otázka vytvoření grafu, nad kterým budeme provádět příslušný grafový algoritmus. To obnáší jednak výčet samotných vrcholů a hran, jednak specifikaci proměnných, které budou vrcholy a hrany obsahovat.

Nakonec jsme se rozhodli pro řešení, které před prováděním samotného kódu popisujícího daný grafový algoritmus, načte z jiného zdrojového kódu podobu příslušeného grafu. Tyto kódy jsou v oddělených souborech, aby bylo možné jednoduše měnit grafy, nad kterými chceme provádět grafový algortimus (žák může například dostat — případně si sám napsat — několik souborů obsahujících různé grafy, nad kterými může zkoušet svůj algoritmus), případně obráceně.

Rozeberme si nyní základní myšlenku těchto dvou zdrojových kódů trochu podrobněji (přesně ji popíšeme v kapitole 2).

Zdrojový kód grafu

Zdrojový kód popisující podobu grafu se skládá ze dvou částí.

- Deklarace proměnných vrcholů a hran (pomocí uvozovacích slov **vertexVar** a **edgeVar**) a následného výčtu jednotlivých proměnných (obdobně jako je v Pascalu realizována deklarace proměnných pod klíčovým slovem **var**).

- Vytvoření jednotlivých vrcholů a hran (orientované a neorientované varianty) pomocí pomocí vestavěných procedur *AddVertex*, *AddEdge* (případně *AddEdgeOr*) uzavřených do bloku **begin-end** ukončeného tečkou.

V průběhu vývoje byla uvažována i možnost deklarace proměnných grafu (pomocí klíčového slova **graphVar**). Nakonec bylo od tohoto nápadu upuštěno, a to z několika důvodů. Zejména se nepovedlo odkaz na tyto proměnné organicky včlenit do struktury jazyka pro akce nad grafem (jako je tomu u vrcholů a hran — to popíšeme později podrobněji). Dále se tento koncept ukázal jako vcelku zbytečný (původní myšlenka ukládání hodnot proměnných mezi dvěma běhy programu není pro demonstraci principu grafových algoritmů nezbytná).

Vrcholy a hrany mohou v sobě obsahovat všechny typy proměnných, se kterými lze pracovat v kódu pro provádění akcí. Samozřejmě je také možné, aby vrcholy nebo hrany neobsahovaly žádné proměnné. V takovém případě je potřeba specifikovat pouze číslo vrcholu, resp. čísla vrcholů, které hrana propojuje.

Zdrojový kód algoritmu

Většina prvků tohoto kódu byla převzata z původního Pascalu. Přesné specifikaci jazyka se budeme opět věnovat v kapitole 2.4, popišme ale alespoň strukturu zdrojového kódu.

Základní odlišností oproti Pascalu je výrazné zjednodušení strukutury kódu. Program v Pascalu se skládá z hlavičky, definice navěští (kvůli příkazu *goto*), definice konstant, definice vlastních typů, deklarace proměnných, deklarace procedur a funkcí a vlastního programu (Jensen a Wirth, 1974). V našem jazyce jsme tuto strukturu poněkud zjednodušili. Ponechali jsme pouze poslední tři části, tedy

1. deklaraci proměnných,

2. deklaraci procedur a funkcí,

3. vlastní program.

Jak je vidět, ponechali jsme nejdůležitější části programu. Hlavička slouží ke specifikaci parametrů při externím volání programu, my však budeme spouštět náš program pouze z vývojového prostředí, proto tuto část můžeme vynechat. Příkaz *goto* byl vynechán jako obecně nevhodný (Dijkstra, 1968) — tím spíše pro žáky, kteří s programováním začínají. Konstanty lze bez větších problémů nahradit pomocí proměnných. Definice vlastních typů byla vynechána zejména z důvodu náročné implementace a nepotřebnosti tohoto konceptu (o tom jsme se zmiňovali výše).

1.3.2 Vrcholy a hrany

Navrhujeme program, který je primárně určený pro práci s grafovými algoritmy, bylo tedy nutné přidat do jazya dva další typy: vrchol (**vertex**) a hranu (**edge**).

Každý vrchol má jednoznačný identifikátor, kterým je přirozené číslo v rozsahu od 1 do 999 (tedy například vrchol číslo 5 napíšeme jako `#5`). Omezení na rozsah identifikátorů je zavedeno čistě z důvodu přehlednějšího vykreslování při vizualizaci grafu (čtyř- a vícemístná čísla by byla příliš malá, pokud by se měla vejít do prostrou pro ně vyhrazeného).

Z toho vyplývá, že v grafu může být maximálně 999 vrcholů, ovšem s přihlédnutím k tomu, že je jazyk určen pro didaktické účely, a tedy budou algoritmy povětšinou prezentovány na malých grafech, není toto omezení nijak omezující.

Při návrhu hran jsme měli v zásadě dvě možnosti. Vzhledem k tomu, že vrcholy mají jedinečný identifikátor, nabízelo by se, aby jej měly také hrany. Hranu mezi vrcholy `#1` a `#2` bychom tak mohli značit například jako `#(1,2)`. To by bylo přehledné, ovšem znamenalo by to, že mezi dvěma vrcholy může vést pouze jedna hrana (aby byl identifikátor jednoznačný), nebo, v případě orientovaných hran, aby mohla existovat jedna hrana `#(1,2)` a jedna hrana `#(2,1)`. To by bylo ale značné omezení pro implementovatelné algoritmy (např. pro simulace automatů).

Doplněním této myšlenky byla možnost, že by se v případě více hran vybírala konkrétní hrana náhodně. Tuto myšlenku jsme však velmi rychle zavrhli, neboť by v programech mohla způsobit množství chyb (de facto by pak symbol `#(1,2)` nebyl identifikátorem, neboť by vybíral náhodnou hranu, byť ze značně omezené množiny).

Konečnou volbou bylo, aby identifikátor `#(1,2)` označoval množinu hran (tj. ekvivalenci typu **set of edge**, který bychom použili při deklaraci proměnné) vedoucí mezi vrcholy `#1` a `#2`.

Hrany samotné nemají žádný speciální identifikátor, a to právě z již výše zmíněného důvodu, že mezi dvěma vrcholy může vést více hran (jak jsme zmínili výše). Pod pojmem *hrana mezi vrcholy* `#1` *a* `#2` tedy chápeme spíše množinu hran mezi těmito dvěma vrcholy (poznamenejme pouze, že při vypisování datových struktur, např. množiny, je každá hrana mezi vrcholy `#1` a `#2` vypsána jako `#(1,2)`, příp. `#(2,1)`). Z této množiny lze pak při běhu programu procházet všechny hrany pomocí *for-in* cyklu. Takováto množina hran může mít libovolný počet prvků, dokonce může být i prázdná (pokud mezi příslušnými vrcholy neexistuje žádná hrana).

Jak už bylo naznačeno výše, hrany existují dvojího typu — orientované a neorientované. Tato vlastnost hrany se udává už při jejím vytvoření (použití procedury či funkce — podle toho, zda hranu specifikujeme už v kódu popisujícím graf, nebo ji přidáme až za běhu algoritmu — *AddEdge* pro neorientovanou hranu nebo *AddEdgeOr* pro orientovanou hranu) a nelze ji později změnit.

Všechny proměnné vrcholů a hran, tak, jak byly deklarovány v kódu pro tvorbu grafu, jsou dostupné z kódu pro akce. Lze z nich číst, i do nich zapisovat.

1.4 Výběr nástrojů

Celý program (vývojové prostředí i samotný interpret) je psaný v programovacím jazyce Java, konkrétně byla použita verze *Java 13*. Jako vývojové prostředí pro vývoj našeho programu jsme použili *NetBeans 11.3*. Java jako vývojový jazyk byla zvolena kvůli dostatku možností, které jako jazyk nabízí, ale zejména proto, že existuje množství nástrojů pro lexikální a syntaktickou analýzu, které jsou s Javou kompatibilní.

Nakonec byly zvoleny nástroje *JFlex* pro lexikální analýzu a *Java CUP* pro analýzu syntaktickou. Tyto nástroje z gramatiky automaticky generují kód v Javě, tutíž bylo snadné jejich výstupy integrovat do programu bez nutnosti dalších nastavení. Práce s oběma zvolenými nástroji je navíc velmi snadná na naučení a práce s nimi je do pro programátora pohodlná. To byl také faktor, který z původního výběru možných nástrojů vyřadil zpočátku uvažované analyzátory *ANTLR* či *GNU Bison*.

2. Uživatelská příručka

V této kapitole se podrobněji podíváme na podobu našeho programovacího jazyka a vývojového prostředí. Vzhledem k tomu, že je náš program zaměřen na vizualizaci, jsou tyto dvě části (tedy jazyk a vývojové prostředí) pevně svázány.

2.1 Spuštění programu

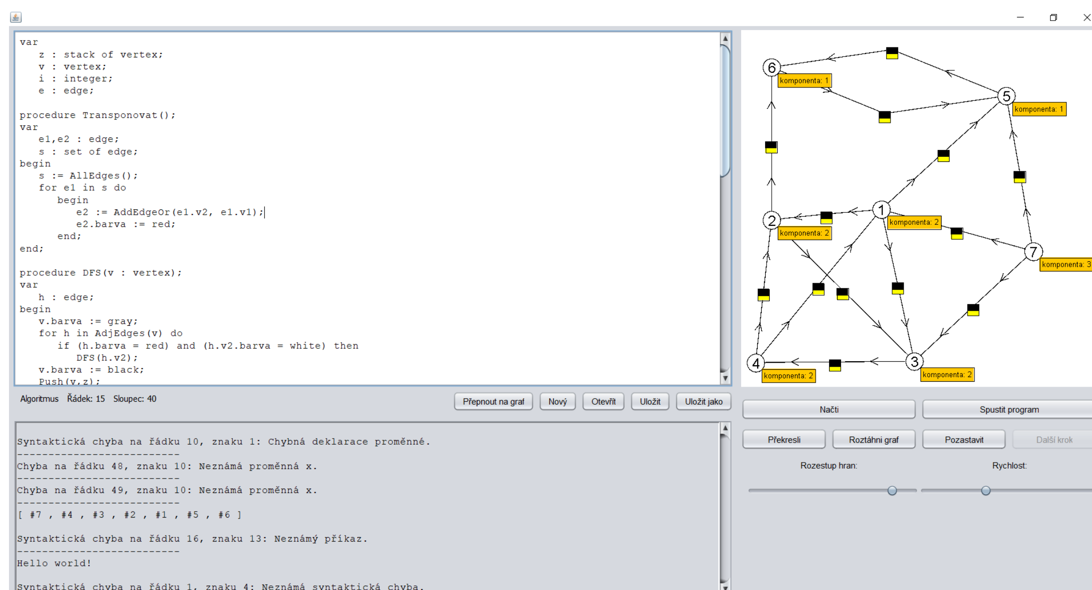
Pro spuštění programu je nutné mít nainstalovanou Javu — ideálně verzi 13 nebo novější (pro nižší verze není zaručena funkčnost programu). Pokud Javu nemáme nainstalovanou, můžeme tak učinit jednoduše pomocí instalačního programu ze stránek `https://java.com/en/download/`.

Nyní už můžeme spustit náš program, a to buď prostým dvojklikem na soubor *graph_algorithms_visualisation.jar*, nebo z příkazového řádku (kde se musíme přepnout do složky, v níž se nachází soubor *graph_algorithms_visualisation.jar*) pomocí příkazu

```
java -jar graph_algorithms_visualisation.jar
```

2.2 Vývojové prostředí

Pro začátek si popíšeme vývojové prostředí našeho programovacího jazyka. Vývojové prostředí je pojato jako okenní aplikace. Jeho podobu můžeme vidět níže (Obrázek 2.1).

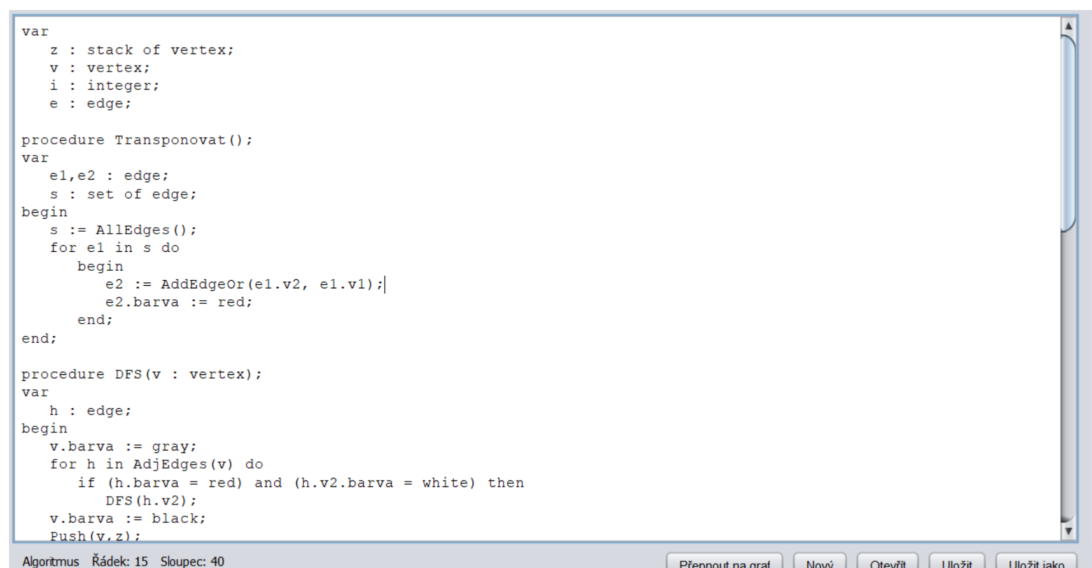


Obrázek 2.1: Ukázka podoby celého programu.

Skládá se celkem z pěti hlavních částí obsahujících dále jednotlivé prvky, které si nyní podrobně popíšeme.

2.2.1 Textové pole pro zdrojový kód

V levém horním rohu se nachází největší prvek naší aplikace — textové pole, kam užitatel píše zdrojový kód programu (Obrázek 2.2).



Obrázek 2.2: Ukázka textového pole pro zápis zdrojového kódu.

Pomocí tlačítek pod nimi (která důkladněji představíme později) může uživatel přepínat mezi zdrojovým kódem grafu a zdrojovým kódem algoritmu. Oba tyto zdrojové kódy se tedy píší do tohoto textového pole.

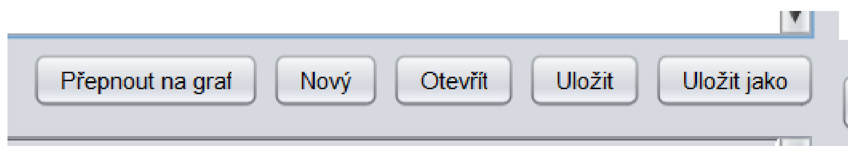
Při vykonávání zdrojového kódu algoritmu se postupně, jeden po druhém, označují zelenou barvou řádky, které se právě vykonávají (vždy pouze ten aktuální řádek). Zdrojový kód grafu nelze krokovat, zejména kvůli jeho značné jednoduchosti.

Při nalezení běhové chyby v kódu se kód zastaví a řádek, na němž se chyba nachází, se označí červeně. Při kliknutí na textové pole se příslušný řádek odznačí.

Vlevo pod textovým polem nalezneme popisek, zda je právě otevřený zdrojový kód grafu (*Graf*) či algoritmu (*Algoritmus*). V případě, že uživatel právě upravuje kód, nalezneme zde i pozici, na níž se nachází kurzor (počítáno od jedničky).

2.2.2 Tlačítka pro práci se zdrojovým kódem

Pod textovým polem se nachází pět tlačítek pro práci se zdrojovým kódem (Obrázek 2.3).



Obrázek 2.3: Ukázka tlačítek pro práci se zdrojovým kódem grafu.

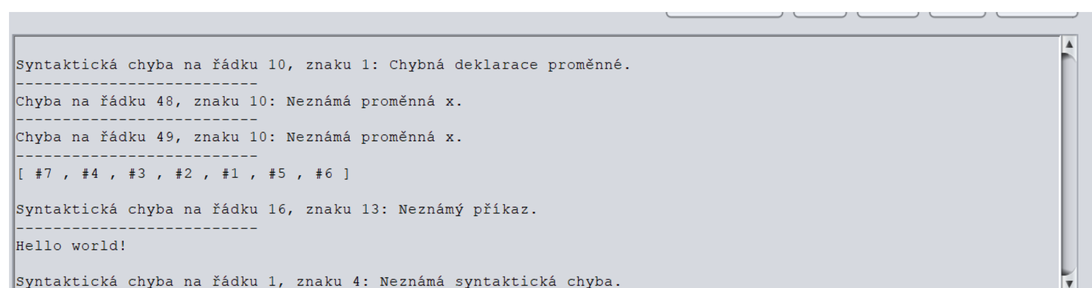
Jejich funkce je následující.

- **Přepnout na algoritmus/Přepnout na graf** — přepíná mezi zdrojovým kódem grafu a algoritmu, přesněji řečeno mezi jejich zobrazením v textovém poli. Při přepnutí zdrojového kódu se změní text na tlačítku. Neukládá přepínaný kód, ovšem tento lze samozřejmě znovu otevřít po opětovném kliknutí na tlačítko.
- **Nový** — vymaže aktuální kód z textového pole a vytvoří nový prázdný neuložený kód. Neukládá odstraňovaný kód.
- **Otevřít** — poskytne uživateli možnost otevřít dříve vytvořený a uložený kód. Ten pak napíše místo aktuálně otevřeného kódu, který vymaže. Neukládá odstraňovaný kód.
- **Uložit** — poskytne uživateli možnost uložit dříve uložený kód. Pokud aktuální kód dosud nebyl uložen, funguje stejně jako tlačítko **Uložit jako**.
- **Uložit jako** — poskytne uživateli možnost uložit aktuální kód do nového souboru.

Soubory, do nichž se ukládá kód grafu, jsou běžné textové soubory (přípona *.txt*). Kód lze tedy upravovat i v jakémkoli jiném textovém editoru (ovšem spouštět ho lze pouze po načtení do našeho vývojového prostředí).

2.2.3 Výpis výstupu programu

V levém dolním rohu se nachází prostor, do něhož je vypisován výstup programu i chybové hlášky (Obrázek 2.4).



Obrázek 2.4: Ukázka pole pro výpis výstupu programu a chybových hlášek.

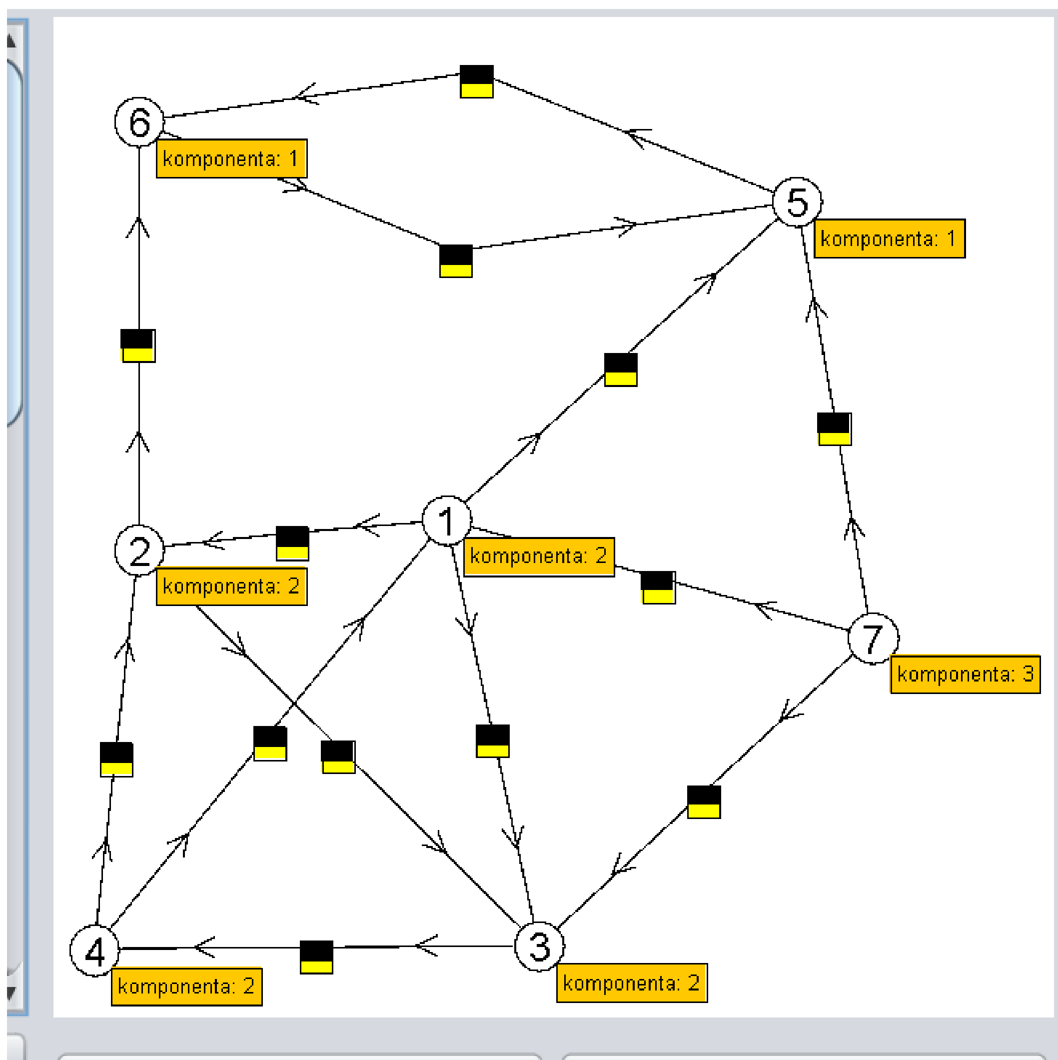
Chronologicky nejmladší zprávy jsou řazeny naspod již vypsaného textu. Výpisem lze listovat pomocí posuvníku.

Výpis se neodstraňuje při opětovném spuštění programu. Vymaže se až po opětovném spuštění vývojového prostředí.

2.2.4 Panel pro vykreslení grafu

V pravém horním rohu se nachází hlavní složka našeho vývojového prostředí — panel pro vykreslení grafu (Obrázek 2.5).

Tento panel se překresluje při načtení grafu, manuálním překreslení pomocí tlačítka **Překresli** (to podrobněji popíšeme později) a také po každém provedení



Obrázek 2.5: Ukázka panelu pro vykreslení grafu.

příkazu (nepřepočítávají se pozice vrcholů, toho lze dosáhnout pouze příkazem *redraw*).

Vrcholy mají podobu kruhů (ohraničených černou kružnicí), v nichž je uvedeno číslo vrcholu. Vrchol může být vyplněn barvou či barvami, které jsou uchovávány v proměnných vrcholu typu **color** (pouze v atomických proměnných, nikoli v datových strukturách, např. barvy v poli typu **array**[**a..b**] **of color** se do vrcholu nevykreslí). Pokud vrchol neobsahuje žádnou proměnnou tohoto typu, je barva vrcholu bílá.

Vpravo dole u vrcholu se nachází tabulka udávájící seznam všech proměnných a jejich hodnot (vyjma proměnných typu **color**, které jsou vykreslovány jako výplň vrcholu samotného) pro příslušný vrchol.

Hrany jsou vykreslovány jako křivky spojující příslušné vrcholy. V případě jedné hrany mezi dvěma různými vrcholy jde o úsečku, v případě multihrany mezi dvěma různými vrcholy jde o lomenou čáru, v případě, že se jedná o smyčku (tj. hranu z vrcholu do stejného vrcholu), vykreslí se hrana jako kružnice.

Pokud hrana obsahuje alespoň jednu proměnnou typu **color**, bude hrana vykreslena barvou rovnou hodnotě první proměnné tohoto typu, kterou hrana ob-

sahuje. Pokud hrana neobsahuje žádnou proměnnou typu **color**, je barva hrany černá (implicitní hodnota proměnné typu **color** pro hrany — tomu se více věnujeme v kapitole 2.4.2).

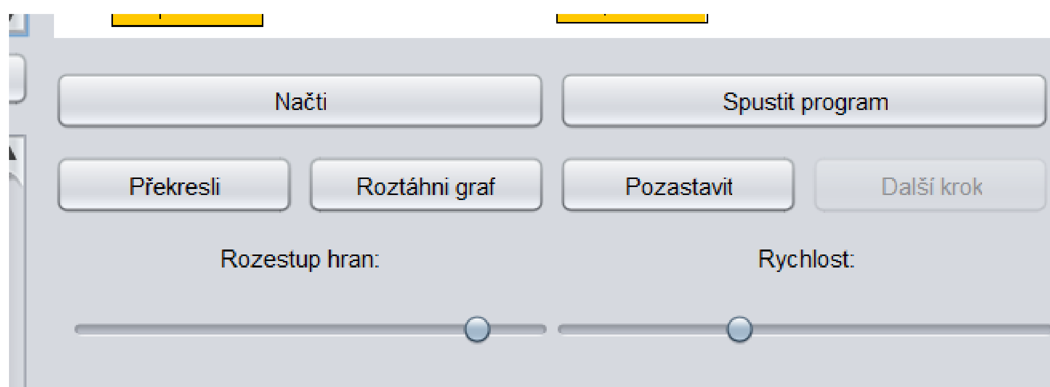
Uprostřed hrany je tabulka udávající seznam proměnných a jejich hodnot pro příslušnou hranu (vyjma proměnných typu **color**). Hodnoty proměnných typu **color** (pokud hrana nějaké obsahuje) jsou vykreslovány v horní části tabulky jako jedno- či vícebarevný pruh.

Je-li hrana orientovaná, nachází se v jedné čtvrtině a třech čtvrtinách délky hrany šipka udávající směr hrany.

Podobu grafu lze také ručně upravovat pomocí přetažení vrcholů myší.

2.2.5 Prvky ovládající běh programu a vykreslování

V pravém dolním rohu se nachází sada tlačítek a posuvníků ovládajících běh programu a vykreslování grafu (Obrázek 2.6).



Obrázek 2.6: Ukázka prvků pro ovládání běhu programu a vykreslování.

V levé části jsou to následující prvky.

- **Načti** — Načte graf popsaný ve zdrojovém kódu grafu do paměti a vykreslí jej do panelu pro vykreslování grafu.
- **Překresli** — přepočítá („náhodně“) pozice vrcholů aktuálního grafu a vykreslí jeho novou podobu.
- **Roztáhni graf** — přeškáluje graf tak, aby vyplňoval celý panel pro vykreslování grafu (hodí se zejména po ruční úpravě podoby grafu).
- **Rozestup hran** — posuvník udávající rozestup multihran a velikost kružnic představujících smyčky.

V pravé části jsou to pak tyto prvky.

- **Spustit program/Zastavit program** — započne nebo ukončí běh programu popsaného ve zdrojovém kódu algoritmu.
- **Pozastavit/Spustit** — přejde do režimu krokování kódu nebo tento režim opustí.

- **Další krok** — provede další příkaz v programu (lze použít pouze v režimu krokování).
- **Rychlost** — posuvník udávající rychlost provádění jednotlivých příkazů (má smysl pouze pokud je vypnuto krokování).

2.3 Programovací jazyk a jeho základní vlastnosti

Nyní se zaměřme na samotný jazyk. Jak už jsme uvedli v kapitole 1.2.1, náš jazyk vychází z Pascalu, pro člověka znalého tohoto jazyka tedy nebude mít jistá část této kapitoly větší přínos. Přesto se ale přikláníme k tomu, popsat podrobně i ty části, které jsou shodné s Pascalem, abychom zamezili případným nejasnostem a nedorozuměním a aby naše příručka byla kompletní.

Na úvod zmiňme, že náš jazyk je stejně jako Pascal tzv. „case-insensitive“, tedy nezáleží na velikosti písmen u jednotlivých příkazů, proměnných a klíčových slov (stále ale rozlišujeme velikost písmen u obsahu znakových a textových proměnných — např. textové řetězce `'Abc'` a `'aBc'` se navzájem nerovnají). Uvádíme-li proto v klíčových slovech nebo názvech procedur či funkcí velká písmena, činíme tak pouze pro přehlednost.

V kapitole 1.3.1 jsme uvedli, že náš jazyk se skládá ze dvou částí — zdrojového kódu grafu (určeného pro popis podoby grafu) a zdrojového kódu algoritmu (určeného pro implementaci příslušného algoritmu). Zaměřme se nejdříve na druhou část, tedy zdrojový kód algoritmu. To proto, že zdrojový kód grafu do jisté míry syntakticky a sémanticky vychází právě ze zdrojového kódu algoritmu (přebírá např. způsob deklarace proměnných či jejich typy).

2.4 Zdrojový kód algoritmu

Než se pustíme do podrobného rozebírání struktury kódu, dovolme si na ukázku malý příklad. Následující zdrojový kód ukazuje, jak může například vypadat náš program.

```
var
   e : edge;

function BarvaHrany(e : edge) : color;
begin
   if (e.vaha < 0) then
      BarvaHrany := red
                 else
      BarvaHrany := black;
end;

begin
   for e in AllEdges() do
      e.barva := BarvaHrany(e);
end.
```

Myšlenkově i graficky je náš kód rozdělen do tří částí.

V první deklarujeme globální proměnné, v tomto případě proměnnou *e* typu **edge** (tedy typ reprezentující hranu grafu). Tu v hlavním kódu použijeme jako řídící proměnnou *for-in* cyklu, musíme ji tedy deklarovat předem (podobně jako je to nutné v běžném Pascalu u běžného *for* cyklu).

Druhá část je deklarace vlastní funkce *BarvaHrany(edge) : color*, která přebírá jeden argument typu **edge** a vrací hodnotu typu **color** (tedy typ reprezentující barvu). Lze snadno odhadnout, že tato funkce vrací červenou barvu pro hrany se zápornou váhou a černou pro hrany s nezápornou váhou.

Třetí částí je samotné hlavní vlákno kódu. Zde obsahuje jeden *for-in* cyklus, který postupně probírá množinu hran grafu a nastavuje barvu každé z nich podle výsledku funkce *BarvaHrany*.

Nyní se podívejme na jednotlivé části kódu podrobně a systematicky.

2.4.1 Struktura kódu

Jak už jsme předeslali v kapitole 1.3.1 a jak jsme viděli v úvodním příkladu, zdrojový kód se skládá se tří částí, jejichž pořadí je pevně dáno. Konkrétně jde o

1. deklaraci proměnných,
2. deklaraci procedur a funkcí,
3. hlavní vlákno kódu.

Podívejme se nyní na tyto jednotlivé části podrobněji.

Deklarace proměnných

Derklarace globálních proměnných je uvozena klíčovým slovem **var**. Následuje seznam deklarovaných proměnných v následujícím tvaru.

```
<seznam-názvů-proměnných> : <typ-proměnných>;
```

Na jednom řádku může být deklarována jedna proměnná, nebo i více proměnných. V prvním případě je na místě seznamu názvů proměnných pouze jediný název, ve druhém seznam názvů oddělených čárkami. Proměnných může být deklarován libovolný počet (jak v jednotlivých řádcích, tak celkově v celém programu).

Název proměnné má podobu libovolně dlouhého řetězce skládajícího se z písmen anglické abecedy a číslic, přičemž název musí začínat písmenem. Názvy proměnných musí být unikátní.

Deklarace proměnných tedy může vypadat například takto.

```
var
   n1, n2 : integer;
   slovo : string;
```

Snadno poznáme, že se jedná o deklaraci celočíselných proměnných *n1* a *n2* a proměnné *slovo* reprezentující textový řetězec.

Deklarace procedur a funkcí

Vlastní procedury a funkce mohou být v našem programu deklarovány v libovolném pořadí. Na rozdíl od Pascalu, kde bylo nutné, aby každá procedura/-funkce užívala jen již deklarované procedury a funkce, je v našem jazyce možné volat z dříve deklarované procedury/funkce i proceduru či funkci deklarovanou později.

Každá procedura i funkce je určena svým názvem a počtem a typem parametrů. V tomto ohledu musí být každá procedura i funkce unikátní.

Procedura je uvozena hlavičkou. Ta sestává z klíčového slova **procedure**, následuje jméno funkce a v závorkách uzavřený seznam parametrů (v našem jazyce je na rozdíl od Pascalu nutné psát za proceduru/funkci závorky i když nepřijímá žádné parametry) a středník. Následuje deklarace lokálních proměnných (ta je syntakticky stejná jako deklarace globálních proměnných) a poslední částí je tělo procedury uzavřené do bloku **begin-end** (syntaxe stejná jako u blokového příkazu, který bude podrobněji rozebrán později) a středník ukončující celou deklaraci funkce.

Deklarace funkce se liší drobnými změnami v hlavičce. Ta je uvozena klíčovým slovem **function** a za závorkou obsahující seznam parametů je dvojtečka a za ní návratový typ funkce.

Deklarace parametrů funkce má podobnou syntaxi, jako deklarace proměnných.

```
<seznam-názvů-parametrů> : <typ-parametrů>
```

V případě deklarace jednoho parametru daného typu obsahuje neznam názvů parametrů pouze jeden název, v případě deklarace více parametrů jsou jednotlivé názvy odděleny čárkami. Jednotlivé deklarace jsou odděleny středníkem.

Například hlavička funkce *f*, která přebírá dva parametry typu **integer** a jeden parametr typu **boolean** a vrací typ **real**, bude vypadat přibližně následovně.

```
function f(a,b : integer; c : boolean) : real;
```

Parametry procedury/funkce mohou být předávány hotnotou i odkazem. Implicitně jsou parametry předávány hodnotou, a to pro všechny typy. Pro předávání odkazem je nutné před deklaraci příslušného parametru (příp. parametrů) přidat klíčové slovo **var**.

Lokální proměnné mají vyšší prioritu než globální, to znamená, pokud existuje lokální a globální proměnná stejného jména, bude se odkaz na tuto proměnnou v kódu procedury/funkce chápat jako ona lokální proměnná. V kódu procedury-/funkce může být použit odkaz na globální proměnnou. Pokud však byla funkce/procedura zavolána z jiné funkce/procedury, nelze již odkázat na proměnnou této dříve volané funkce/procedury).

Ve funkci určíme její návratovou hodnotu tak, že v jejím těle dosadíme do proměnné, která má stejné jméno jako funkce samotná (tuto proměnnou není třeba deklarovat mezi lokálními proměnnými). V případě, že ve funkci nebude nikde určena návratová hodnota, použije se implicitní hodnota pro daný typ (o těchto implicitních hodnotách podrobněji pohovoříme později).

Pro lepší pochopení uvedeme příklady deklarací jedné procedury a jedné funkce.

Za procedury ukážeme proceduru *Vymena*, která prohodí obsah dvou proměnných typu **integer**. Můžeme zde vidět deklaraci lokálních proměnných a zejména klíčové slovo **var** v hlavičce funkce udávající, že argumenty a a b jsou předávány odkazem.

```
procedure Vymena(var a,b : integer);
var
   x : integer;
begin
   x := a;
   a := b;
   b := x
end;
```

Za funkce ukážeme funkci *Fib* počítající n-tý člen Fibonacciho posloupnosti. Pro záporná čísla vrací chybovou hodnotu -1, pro $n = 0$ a $n = 1$ vrací číslo 1 a pro zbylá kladná čísla počítá rekurzivně příslušnou hodnotu.

```
function Fib(n : integer) : integer;
begin
   if (n < 0) then
      Fib := -1
              else
   if (n = 0 or n = 1) then
      Fib := 1
                    else
      Fib := Fib(n-1) + Fib(n-2)
end;
```

Hlavní vlákno kódu

Hlavní vlákno kódu tvoří blok **begin-end** ukončený tečkou, ve kterém se nachází seznam příkazů. Jednotlivé příkazy jsou odděleny jedním středníkem, na konci bloku příkazů může, ale nemusí být středník (oddělující prázdný příkaz).

Jednotlivé příkazy probereme podrobně v příslušné sekci později.

2.4.2 Typy proměnných

Typy proměnných bychom mohli rozdělit na dvě skupiny — datové typy a datové struktury.

Při deklaraci jsou všechny proměnné nastaveny na svou implicitní hodnotu. To je jedna z odlišností oproti originálnímu Pascalu, kde proměnné nebývají automaticky inicializovány.

Datové typy

V našem jazyce existují následující datové typy.

- **integer** — celé číslo v rozsahu od -2^{31} do $2^{31}-1$. Může obsahovat i hodnoty ∞ a $-\infty$ (zapisované jako `infinity`, resp. `-infinity`). Maximální a minimální hodnotu označují hodnoty `maxInt` a `minInt`. Implicitní hodnotou je číslo `0`.

- **real** — desetinné číslo v rozsahu od $(2-2^{-52})\cdot 2^{1023}$ do 2^{-1074}. Může obsahovat i hodnoty ∞ a $-\infty$ (zapisované jako `infinity`, resp. `-infinity`). Při zápisu musí být uvedena celá část (byť by byla nulová), desetinná může být vynechána. Číslo může být doplněno znakem `e`, za kterým následuje celé číslo označující exponent v exponenciálním tvaru čísla. Implicitní hodnotou je číslo `0.0`.

- **boolean** — logická hodnota `true` (pravda) nebo `false` (nepravda). Implicitní hodnotou je hodnota `false`.

- **char** — znak Unicode. Zapisuje se uzavřený do apostrofů. Implicitní hodnotou je mezera (`' '`).

- **string** — libovolně dlouhý textový řetězec. Zapisuje se uzavřený do apostrofů. Implicitní hodnotou je prázdný řetězec (`''`).

  Pomocí syntaxe `str[i]` lze přistoupit k i-tému znaku v textovém řetězci uloženém v proměnné *str* typu **string**. S proměnnoou typu **string** lze tedy do jisté míry pracovat jako s jednorozměrným polem znaků. Textové řetězce v našem jazyce jsou však nemodifikovatelné (tedy nelze dosadit konkrétní znak na konkrétní místo v řetězci).

- **color** — barva určená pomocí RGB kódu. Lze ji zadat pomocí funkce *colorRGB(integer,integer,integer)*, kde jednotlivá čísla musí být z rozsahu 0–255. Existuje také několik předdefinovaných barev zastoupených klíčovými slovy `black`, `white`, `gray`, `red`, `green` a `blue`. Implicitní hodnotou je barva `white`, tedy bílá (vyjma situace, kdy je proměnná typu **color** proměnnou hrany — pak je implicitní hodnotou barva `black`, tedy černá).

Vzhledem k tomu, že je program implementován v Javě, odpovídají datové typy (spolu se svými parametry) odpovídajícím typům v jazyce Java (Oracle, 2019).

K těmto základním typům přiřadíme pro naše potřeby, tedy implementaci grafových algoritmů, ještě další dva typy.

- **vertex** — vrchol grafu. Implicitní hodnota je nulová hodnota `NIL`.

- **edge** — hrana grafu. Implicitní hodnota je nulová hodnota `NIL`.

Vrchol a hrana jsou inspirovány typem *record*, který se vyskytuje v původním Pascalu. Vrcholy a hrany lze ukládat do proměnných příslušných typů (**vertex** a **edge**) a zároveň lze přistupovat k proměnným deklarovaným ve vrcholech a hranách pomocí syntaxe podobné právě té, s jakou se pracuje v Pascalu s typem record.

```
<hotnota-typu-vertex/edge>.<proměnná-vrcholu/hrany>
```

Každá hrana má také automaticky proměnné s názvy *v*1 a *v*2, které označují první a druhý vrchol hrany (v případě neorientované hrany jsou vrcholy vnímány v pořadí, v jakém byly deklarovány ve zdrojovém kódu grafu).

Bude-li v proměnné typu **vertex**/**edge** hodnota `NIL`, způsobí čtení proměnné vrcholu/hrany běhovou chybu.

Odkazy na proměnnou vrcholu/hrany se mohou samozřejmě řetězit — pokud například vrchol `#1` obsahuje proměnnou *vrchol* typu **vertex** a proměnnou *cislo* typu **integer**, můžeme pomocí symbolu `#1.vrchol.cislo` přistoupit na hodnotu proměnné *cislo* vrcholu, který je uložený v proměnné *vrchol* vrcholu `#1`.

```
begin
   #2.cislo := 14;
   #1.vrchol := #2;
   writeLn(#1.vrchol.cislo)
end.
```

První řádek přiřadí do proměnné *cislo* vrcholu `#2` hodnotu `14`. Druhý řádek přiřadí do proměnné *vrchol* vrcholu `#1` vrchol `#2`. Třetí řádek pak vypíše hodnotu proměnné *cislo* vrcholu, který je uložen v proměnné *vrchol* vrcholu `#1` — tedy vypíše číslo `14`.

Datové struktury

Jak už jsme zmínili v kapitole 1.1.3, v našem jazyce existují čtyři datové struktury: **array** (pole), **set** (množina), **stack** (zásobník) a **queue** (fronta).

Řekněme si nejdříve něco k první z nich, tedy k poli (**array**).

- **array** — pole. Implicitní hodnotou je pole daných rozměrů, kde jeho hodnoty jsou nastaveny na implicitní hodnotu.

  Pole může být jednorozměrné i vícerozměrné. Rozsah pole je zadáván jako rozmezí mezi dvěma celými čísly. Při deklaraci je (stejně jako v Pascalu) nutné uvést pevné hranice všech rozměrů pole. Na rozdíl od Pascalu však mohou být identifikátory pole pouze celá čísla (tj. hodnoty typu **integer**).

  Deklarace jednorozměrného a vícerozměrného pole tedy může vypadat například následovně.

  ```
  var
     pole1 : array[1..4] of integer;
     pole2 : array[-2..2,0..12] of boolean;
     pole3 : array[-2..2] of array[0..12] of boolean;
  ```

  Deklarace proměnných *pole2* a *pole3* ukazuje dva možné způsoby deklarace vícerozměrného pole. Proměnné *pole2* a *pole3* jsou stejného typu.

  Např. k prvku na třetí pozici v poli *pole1* můžeme přistoupit pomocí syntaxe `pole1[3]`. K prvku na pozici (-1,9) v poli *pole2* můžeme přistoupit buď pomocí syntaxe `pole2[-1,9]` nebo `pole2[-1][9]`. Tyto dvě podoby odkazu na prvek mají úplně stejný význam.

Další tři datové struktury mají podobný styl deklarace. Při deklaraci uvedeme jméno datové struktury, klíčové slovo **of** a jméno typu, který datová struktura uchovává. Množinu, zásobník a frontu uchovávající celočíselné hodnoty (tj. typu **integer**) tedy deklarujeme následovně.

```
var
   mnozina   : set of integer;
   zasobnik : stack of integer;
   fronta    : queue of integer;
```

Nyní se podívejme na další specifika jednotlivých datových struktur.

- **set** — množina. Implicitní hodnotou je prázdná množina.

  Každá hodnota může být v množině pouze jednou. Přidáním prvku, který v dané množině již existuje, nedojde k modifikaci množiny.

  Při deklarci množiny je nutné uvést datový typ, který má uchovávat. Množina může uchovávat pouze jeden datový typ. Množinu lze upravovat pomocí následující procedury.

  – *procedure Add(<hodnota>,set)* — přidání prvku do množiny.

- **stack** — zásobník. Implicitní hodnotou je prázdný zásobník.

  Zásobník lze upravovat pomocí následujících procedur a funkcí.

  – *procerude Push(<hodnota>,stack)* — přidání prvku na vrchol zásobníku.
  – *function Pop(stack) : <hodnota>* — odebrání prvku z vrcholu zásobníku.
  – *function Peek(stack) : <hodnota>* — zjištění hodnoty na vrcholu zásobníku bez jejího odebrání.

- **queue** — fronta. Implicitní hodnotou je prázdná fronta.

  Frontu lze upravovat pomocí následujících procedur a funkcí.

  – *procerude Enqueue(<hodnota>,queue)* — přidání prvku na do fronty.
  – *function Dequeue(queue) : <hodnota>* — odebrání prvku z fronty.
  – *function Peek(queue) : <hodnota>* — zjištění hodnoty na začátku fronty bez jejího odebrání.

Všechny datové struktury mohou uchovávat pouze jednoduché datové typy, tj. **integer**, **real**, **boolean**, **char**, **string**, **vertex** a **edge**. Při vkládání prvku musí být přidávaný prvek správného typu, jinak dojde k běhové chybě.

Operátor	Název operace	Typy operandů	Typ výsledku
=	rovná se	datové typy	**boolean**
		vertex, **vertex**	**boolean**
		edge, **edge**	**boolean**
		set, **set**	**boolean**
<>	nerovná se	datové typy	**boolean**
		vertex, **vertex**	**boolean**
		edge, **edge**	**boolean**
		set, **set**	**boolean**
>	větší	číselné typy	**boolean**
		boolean, **boolean**	**boolean**
>=	větší nebo rovno	číselné typy	**boolean**
		boolean, **boolean**	**boolean**
<	menší	číselné typy	**boolean**
		boolean, **boolean**	**boolean**
<=	menší nebo rovno	číselné typy	**boolean**
		boolean, **boolean**	**boolean**
	je podmnožinou	**set**, **set**	**boolean**
`in`	je prvkem	datový typ, **set**	**boolean**
><	symetrický rozdíl	**set**, **set**	**set**

Tabulka 2.1: Operátory 1. úrovně (expressions).

Operátor	Název operace	Typy operandů	Typ výsledku
+	součet	číselné typy	**integer**
			real
	sjednocení	**set**, **set**	**set**
−	rozdíl	číselné typy	**integer**
			real
	množinový rozdíl	**set**, **set**	**set**
`or`	disjunkce	**boolean**, **boolean**	**boolean**
	bitový součet	**integer**, **integer**	**integer**
`xor`	ostrá disjunkce	**boolean**, **boolean**	**boolean**
	bitová nonekvivalence	**integer**, **integer**	**integer**

Tabulka 2.2: Operátory 2. úrovně (simple expressions).

Operátor	**Název operace**	**Typy operandů**	**Typ výsledku**
*	násobení průnik	číselné typy **set**, **set**	**integer** **real** **set**
/	dělení	číselné typy	**integer** **real**
div	celočíselné dělení	**integer**, **integer**	**integer**
mod	modulo	**integer**, **integer**	**integer**
and	konjunkce bitový součin	**boolean**, **boolean** **integer**, **integer**	**boolean** **integer**
shl	bitový posuv vlevo	**integer**, **integer**	**integer**
shr	logický bitový posuv vpravo	**integer**, **integer**	**integer**

Tabulka 2.3: Operátory 3. úrovně (terms).

Operátor	**Název operace**	**Typy operandů**	**Typ výsledku**
not	negace bitová negace	**boolean** **integer**	**boolean** **integer**
−	unární minus	**integer** **real**	**integer** **real**
+	unární plus	**integer** **real**	**integer** **real**
**	mocnina	**integer**, **integer** **real**, **integer**	**integer** **real**

Tabulka 2.4: Operátory 4. úrovně (factors).

2.4.3 Operátory

Operátory jsou zpracovány podle příručky Free Pascalu (van Canneyt, 2017). V tabulkách níže uvádíme přehled všech operátorů v našem jazyce. Operátory jsou rozděleny do čtyř skupin sestupně podle priority vyhodnocování — **expressions** (Tabulka 2.1), **simple expressions** (Tabulka 2.2), **terms** (Tabulka 2.3) a **factors** (Tabulka 2.4). Pokud jsou ve výrazu použity dva operátory ze stejné skupiny, vyhodnocují se zleva.

Kterákoli část výrazu také může být uzavřena v kulatých závorkách — pak dojde k přednostnímu vyhodnocení výrazu v závorce.

Operace máme binární a unární. To je odlišeno i ve sloupci *Typy operandů*, kde je v případě unární operace jeden typ, v případě binární operace dva typy oddělené čárkou, a to v pořadí, v němž je možné je použít — to je důležité u operace mocniny (Tabulka 2.4), kde je možné použít jako druhý operand pouze typ **integer** (tedy např. pokus o vyhodnocení výrazu `4**0.5` skončí chybou).

Ve sloupci *Typy operandů* můžeme narazit na zkrácený zápis *datové typy* či *číselné typy*.

Datové typy jsou jednoduché datové typy popsané v kapitole 2.4.2 — tedy **integer**, **real**, **boolean**, **char**, **string** a **color**.

Pro operátory `=` a `<>` (Tabulka 2.1) je nutné, aby byly operandy vzájemně porovnatelné — to znamená stejné nebo typů **integer** a *real*. Porovnávat lze i typy **vertex** a **edge** vůči hodnotě `NIL` či dvě hodnoty `NIL` navzájem. V případě podovnávání množin musí být obě množiny stejného typu (tedy musí uchovávat stejné typy hodnot). V jiném případě dojde k chybě.

Pro operátor `in` (Tabulka 2.1) je nutné, aby byl příslušný datový typ stejný, jako typ, který uchovává daná množina. V opačném případě dojde k chybě.

Číselné typy jsou typy **integer** a **real**. Ve sloupci *Typy operandů* (Tabulka 2.1, Tabulka 2.2 a Tabulka 2.3) zastupuje všechny možné kombinace těchto typů. Pokud je alespoň jeden z operandů typu **real**, je výsledek typu **real**. V opačném případě je výsledek typu **integer**.

Při porovnávání operandů typu **boolean** (Tabulka 2.1) používáme pravidlo, že hodnota `true` je větší než hodnota `false`.

Operace modulo (Tabulka 2.3) je definována tak, jak ji definuje Niklaus Wirth ve své přiručce (Jensen a Wirth, 1974), tj. $a \mod b = a - (\lfloor a/b \rfloor \cdot b)$.

2.4.4 Příkazy

Kód programu sestává z jednotlivých příkazů. Jednotlivé příkazy jsou odděleny nejvýše jedním středníkem a za posledním příkazem středník být nemusí (neodděluje už totiž žádný další příkaz), ale také může (oddělení prázdného příkazu).

Blokový příkaz

Seznam několika příkazů lze uzavřít do bloku. Tím je umožněno dívat se na seznam příkazů jako na jeden příkaz, což je výhodné především u řídících struktur, kde je syntakticky povolen pouze jeden příkaz.

```
begin
   prikaz;
```

```
   ...
   prikaz(;)
end
```

Řídící struktury

V našem jazyce je k dispozici několik řídících struktur — podmínky a cykly. Jejich princip fungování je stejný jako v jiných jazycích, proto si popíšeme zejména jejich syntaxi.

- **If-podmínka**

  Příkaz sloužící k rozhodnutí o provedení určitého příkazu (či oprovedení alternativního příkazu — *else* větev) podle platnosti podmínky.

  ```
  if podminka then prikaz
  ```

  ```
  if podminka then prikaz1
              else prikaz2
  ```

  Problém s vnořenými podmínkami je vyřešen podle příručky Niclause Wirtha (Jensen a Wirth, 1974). To znamená, že kód

  ```
  if podminka1 then if podminka2 then prikaz1 else prikaz2
  ```

  je chápán následovně.

  ```
  if podminka1 then
     begin
        if podminka2 then prikaz1
                     else prikaz2
     end
  ```

- **Příkaz case**

  V některých dalších jazycích bývá tento příkaz označován též jako *switch*. Slouží k rozeskoku do jednotlivých větví podle jedné hodnoty. Jedná se prakticky o kompaktnější zápis soustavy *if-else* podmínek.

  ```
  case promenna of
     hodnota : prikaz;
     ...
     hodnota : prikaz
  end
  ```

  ```
  case promenna of
     hodnota : prikaz;
     ...
     hodnota : prikaz;
     else prikaz
  end
  ```

- **While-cyklus**

  Slouží k opakovanému provádění příslušného příkazu. Příkaz se provádí, dokud je pravdivostní hodnota podmínky `true`.

```
while podminka do prikaz
```

- **Repeat-until-cyklus**

  Slouží k opakovanému provádění příslušné řady příkazů. Příkazy se provádí provedou jednou a poté se provádí, dokud je pravdivostní hodnota podmínky `false`.

```
repeat
   prikaz;
   ...
   prikaz;
until podminka
```

- **For-cyklus**

  Slouží k opakovanému provádění příkazu, pokud předem víme, kolikrát se má příkaz provést. Příkaz postupně dosazuje do řídící proměnné (kterou je třeba deklarovat mezi dalšími proměnnými na začátku kódu) typu **integer** celá čísla od počáteční do cílové hodnoty (postupně zvyšuje nebo snižuje — v závislosti na klíčovém slově **to** nebo **downto** — řídící proměnnou o jedna). Cyklus probíhá všechny hodnoty včetně obou krajních.

```
for promenna := hodnota1 to hodnota2 do
   prikaz
```

```
for promenna := hodnota1 downto hodnota2 do
   prikaz
```

- **For-in-cyklus**

  Slouží k opakovanému provádění příkazu pro všechny prvky určité množiny (**set**). Příkaz postupně dosazuje do řídící proměnné prvky z uvedené množiny a vykonává příkaz. Řídící proměnná musí být stejného typu, jako prvky uvedené množiny.

```
for promenna in mnozina do
   prikaz
```

Přiřazovací příkaz

Jako přiřazovací operátor slouží symbol `:=`. V příkazu musí být na jeho levé straně uvedena proměnná a na pravé straně libovolná proměnná, konkrétní hodnota, výraz či funkce (vestavěná nebo definovaná uživatelem).

Například dosazení hodnoty `14` do proměnné *cislo* vypadá následovně.

```
cislo := 14
```

Při pokusu o dosazení nesprávného typu vyvolá příkaz běhovou chybu. Výjimkou je typ **integer**, který lze dosadit do typu **real** (ne však naopak, byť by v proměnné typu **real** bylo uloženo celé číslo).

```
var
   a : integer;
   b : real;

begin
   a := 1;
   b := a;
   a := b;
end.
```

V tomto kódu tedy druhý řádek proběhne bez problémů, třetí řádek však již vyvolá běhovou chybu.

Vestavěné procedury

Kromě procedur modifikujících datové struktury (které jsou uvedeny v příslušném oddílu této práce) existují v jazyce procedury pro výpis.

- *Write(<seznam-parametrů>)* — vypíše uvedené parametry za sebe jako textový řetězec.
- *WriteLn(<seznam-parametrů>)* — vypíše uvedené parametry za sebe jako textový řetězec a odřádkuje.

Další skupinou vestavěných procedur jsou procedury pro mazání vrcholů a hran (pro jejich vkládání slouží funkce, o kterých pojednáme později).

- *DeleteVertex(vertex)* — vymaže vrchol uvedený v parametru a všechny hrany vedoucí z vrcholu a do něj.
- *DeteteEdge(edge)* — vymaže hranu uvedenou v parametru.

Příkazy ovládající vizualizaci

Náš jazyk a přidružené vývojové prostředí slouží k vizualizaci grafových algoritmů. Budou se nám tedy hodit příkazy, které budou nějakým způsobem ovládat vizualizaci.

Při vykonávání grafu se kód krokuje po jednotlivých příkazech, přesněji řečeno po každém příkazu se na okamžik zastaví, aby bylo možné sledovat běh programu. Tuto vlastnost lze ovládat pomocí následujících příkazů.

- *StepOff* — vypne krokování kódu.
- *StepOn* — zapne krokování kódu.

Po provedení každého příkazu dojde k opětovnému vykreslení grafu (včetně například aktualizace barvy vrcholu/hrany či hodnoty proměnné v tabulkách u vrcholů a hran). Nedojde však k přepočítání pozic vrcholů (což by se hodilo např. při přidání nové hrany či nového vrcholu). Manuálně lze toto přepočítání vyvolat ve zdrojovém kódu následujícím příkazem.

- *Redraw* — přepočítá pozice vrcholů a znovu vykreslí graf.

2.4.5 Vestavěné funkce

V našem jazyce existuje několik vestavěných funkcí. Uveďme specifikaci těch, které již nebyly zmíněny jinde.

- Aritmetické funkce.
  - *Abs(integer/real) : integer/real* — vrací absolutní hodnotu čísla (je definována i pro nekonečné hodnoty).
  - *Sqrt(integer/real) : integer/real* — vrací odmocninu z čísla (pro nekonečno vrací nekonečno).
  - *Even(integer) : boolean* — zjišťuje, zda je číslo sudé (v případě nekonečné hodnoty vrátí `false`).
  - *Odd(integer) : boolean* — zjišťuje, zda je číslo liché (v případě nekonečné hodnoty vrátí `false`.
- Funkce nad datovými strukturami.
  - *Length(stack/queue/set) : integer* — vrátí počet prvků v zásobníku-/frontě/množině.
  - *Length(string) : integer* — vrátí délku textového řetězce.
- Funkce k přidávání vrcholů a hran.
  - *AddVertex(integer) : vertex* — přidá do grafu vrchol s daným číslem a vrátí ho jako výsledek funkce.
  - *AddEdge(vertex, vertex) : edge* — přidá do grafu neorientovanou hranu mezi danými vrcholy a vrátí ji jako výsledek funkce.
  - *AddEdgeOr(vertex, vertex) : edge* — přidá do grafu orientovanou hranu mezi danými vrcholy a vrátí ji jako výsledek funkce.
- Funkce vracející množiny vrcholů a hran.
  - *AdjVertices(vertex) : set of vertex* — vrátí množinu vrcholů, do nichž vede hrana z vrcholu v argumentu.
  - *AdjEdges(vertex) : set of edge* — vrátí množinu hran, které vedou z vrcholu v argumentu.
  - *AllVertices() : set of vertex* — vrátí množinu všech vrcholů v grafu.
  - *AllEdges() : set of edge* — vrátí množinu všech hran v grafu.

2.4.6 Komentáře

V našem jazyce lze psát komentáře dvěma způsoby:

- **Řádkový komentář** — je uvozen dvěma lomítky a ukončen koncem řádku.

```
// priklad radkoveho komentare
```

- **Blokový komentář** — je uzavřen mezi složené závorky. Jako koncová závorka se bere ta, která je nejbližší uvozující závorce.

```
{
   priklad
   blokoveho
   komentare
}
```

2.5 Zdrojový kód grafu

Podobně jako v kapitole 2.4, kde jsme popisovali zdrojový kód algoritmu, si i zde uvedeme na ukázku malý příklad, který ukazuje, jak může například vypadat náš program.

```
vertexVar
   cislo : real;

edgeVar
   vaha : integer;
   barva : color;

begin
   AddVertex(1, 3.14);
   AddVertex(2, default);
   AddVertex(3, -8);

   AddEdge((1,3), 4,green);
   AddEdge((2,3), -1,blue);
   AddEdge((3,3), 0,blue);
   AddEdgeOr((2,1), -5,default);
   AddEdgeOr((1,3), 9,red);
end.
```

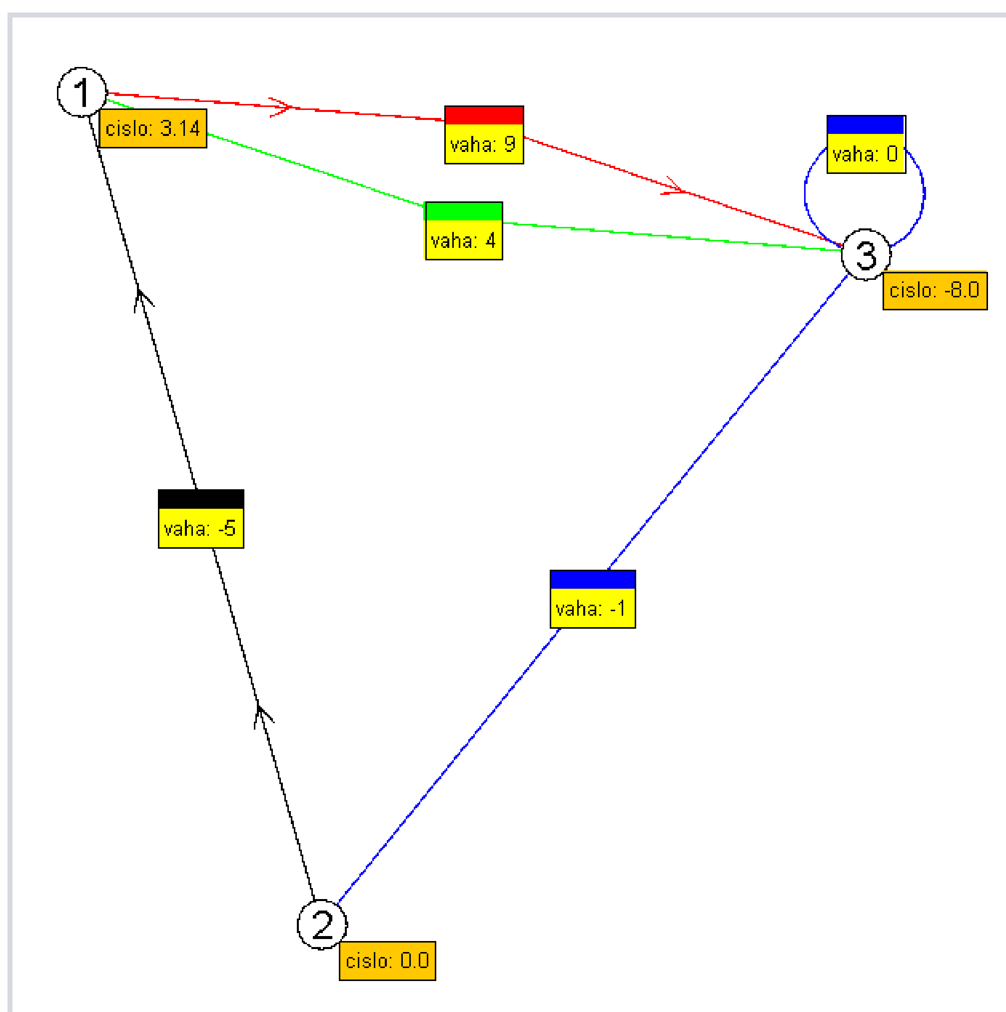
Myšlenkově je kód rozdělen na dvě části.

První z nich je deklarace proměnných. Ta je ještě přirozeně rozdělena na dvě další podčásti — deklaraci proměnných pro vrcholy (uvozenou klíčovým slovem **vertexVar**) a deklaraci proměnných pro hrany (uvozenou klíčovým slovem **edgeVar**). Vrcholům deklarujeme proměnnou *cislo* typu **real**, hranám pak proměnnou *vaha* typu **integer** a proměnnou *barva* typu **color**.

Ve druhé části specifikujeme podobu grafu. První tři řádky přidávají do grafu vrcholy `#1`, `#2` a `#3`, přičemž inicializuje hodnoty jejich proměnné. Další tři řádky přidávají do grafu neorientované hrany mezi vrcholy `#1` a `#3`, `#2` a `#3` a smyčku ve vrcholu `#3`. Poslední dva řádky přidávají do grafu orientované hrany z vrcholu `#2` do vrcholu `#1` a z vrcholu `#1` do vrcholu `#3`.

Klíčové slovo `default` představuje implicitní hodnotu dané proměnné, tedy hodnota proměnné *cislo* vrcholu `#2` je `0.0` a hodnota proměnné *barva* orientované hrany z vrcholu `#2` do vrcholu `#1` je `black` (tedy černá barva).

Pokud bychom tedy dle uvedeného kódu nechali vývojovým prostředím vykreslit grafickou reprezentaci grafu, dostali bychom přibližně takový výsledek, jako vidíme na obrázku níže (Obrázek 2.7).



Obrázek 2.7: Graf vytvořený podle zdrojového kódu z úvodu kapitoly 2.5

Nyní se opět podíváme na jednotlivé části kódu podrobně a systematicky.

2.5.1 Struktura kódu

Zdrojový kód grafu je v mnohém podobný zdrojovému kódu algoritmu, který jsme popisovali v předchozí kapitole. Skládá ze dvou částí, jejichž pořadí je pevně dáno.

1. deklarace proměnných
2. vlastní program

Nyní se na tyto dvě části podíváme podrobněji.

Deklarace proměnných

Deklarace proměnných se skládá ze dvou dalších podčástí — deklarace proměnných hran a deklarace proměnných vrcholů. Tyto části mohou být v kódu zapsány v libovolném pořadí a dokonce může jedna z nich (případně obě) chybět.

Deklarace proměnných vrcholu je uvozena klíčovým slovem **vertexVar**, deklarace proměnných hrany pak klíčovým slovem **edgeVar**. Syntaxe deklarace jedné proměnné je pak stejná jako u deklarace globální proměnné ve zdrojovém kódu algoritmu.

`<seznam-názvů-proměnných> : <typ-proměnných>;`.

Syntakticky tedy klíčové slovo **vertexVar**, resp. **edgeVar**, zastupuje klíčové slovo **var** v deklaraci proměnných ve zdrojovém kódu algoritmu.

Názvy proměnných musí být jedinečné pro každou z obou skupin (tj. vrcholy a hrany), ne však napříč mezi vrcholy a hranami (pro vrchol tedy může existovat proměnná se stejným názvem jako pro hranu).

Deklarace proměnných vrcholu tedy může vypadat například takto.

```
vertexVar
   n1, n2 : real;
   b : color;
```

Snadno nahlédneme, že jde o deklaraci proměnných *n1* a *n2* uchovávajících reálná čísla, a proměnnou *b* reprezentující barvu daného vrcholu.

Vlastní program

Podobně jako je tomu u hlavního vlákna kódu ve zdrojovém kódu pro algoritmy, tvoří i zde vlastní program blok **begin-end** ukončený tečkou, v němž se nachází seznam příkazů. Jednotlivé příkazy jsou pak rovněž odděleny jedním středníkem a na konci bloku příkazů může, ale nemusí být středník (oddělující prázdný příkaz).

Jednotlivé příkazy probereme podrobně v příslušné sekci později.

Hlavní tělo kódu tvoří blok **begin-end** ukončený tečkou, ve kterém se nachází seznam příkazů.

2.5.2 Typy proměnných

Datové typy a datové struktury jsou pojaty stejně jako ve zdrojovém kódu algoritmu, pouze s následujícími odchylkami.

- Pro typy **integer** a **real** nejsou podporovány nekonečné hodnoty.
- Pry typ **color** jsou podporovány pouze předdefinované barvy.
- Nelze dosazovat prvky do datových struktur, je možné používat pouze jejich implicitní hodnotou.

2.5.3 Příkazy

Ve zdrojovém kódu grafu existují pouze tři příkazy, a to procedury pro vytváření vrcholů a hran.

- *addVertex(integer,<seznam-hodnot>)* — přidá do grafu vrchol s daným identifikačním číslem a uvedenými hodnotami proměnných.
- *addEdge((integer,integer),<seznam-hodnot>)* — přidá do grafu neorientovanou hranu mezi vrcholy, jejichž čísla jsou uvedena v závorce, a uvedenými hodnotami proměnných.
- *addEdgeOr((integer,integer),<seznam-hodnot>)* — přidá do grafu orientovanou hranu mezi vrcholy, jejichž čísla jsou uvedena v závorce, a uvedenými hodnotami proměnných.

Je důležité zmínit, že každá hrana (orientovaná neorientovaná) musí být přidávána až tehdy, když jsou již přidány oba její krajní vrcholy (tj. příkaz k jejímu vytvoření musí být v kódu až pod příkazy k vytvoření jejích krajních vrcholů).

Zastavme se na chvíli u seznamu hodnot, který předáváme proceduře. Syntakticky se může jednat o seznam libovolných hodnot oddělených čárkami. Sémanticky tyto hodnoty představují iniciální hotnoty proměnných příslušného vrcholu nebo hrany, a to v tom pořadí, v jakém byly proměnné deklarovány v příslušné sekci na začátku zdrojového kódu. Je tedy zřejmé, že každná hodnota musí být stejného typu jako příslušná proměnná.

Uveďme jednoduchý konkrétní příklad. Předpokládejme, že proměnné vrcholu jsme deklarovali následovně.

```
vertexVar
   x : char;
   y : integer;
```

Ve vlastním programu pak použijeme tento příkaz.

```
AddVertex(4, 'a',8)
```

Tím dojde k vytvoření vrcholu `#4`, v němž bude proměnná x obsahovat znak `'a'` a proměnná y číslo `8`.

V některých případech se nám hodí určit jako hodnotu proměnné její implicitní hodnotu. Tuto hodnotu zastupuje pro všechny typy hodnota `default`. Tuto hodnotu je také třeba použít vždy při inicializaci proměnných typu **array**, **set**, **stack** a **queue**.

Za speciální případ můžeme považovat situaci, kdy pro vrcholy či hrany nedeklarujeme žádné proměnné. V tom případě v příkazu k vytvoření vrcholu či hrany pouze vynecháme seznam hodnot. Například pro vytvoření vrcholu `#4` tedy použijeme příkaz

```
AddVertex(4)
```

a pro vytvoření neorientované hrany mezi vrcholy `#1` a `#2` následující příkaz.

```
AddVertex((1,2))
```

Obdobně pro orientované hrany.

Vidíme, že i v případě vynechaného seznamu hodnot musíme ponechávat závorky obsahující vrcholy hrany.

2.5.4 Komentáře

Stejně jako v kódu pro akce nad grafy jsou i zde povoleny jak řádkové, tak blokové komentáře, a to ve stejné podobě jako v kódu pro vykonávání akcí.

3. Programátorská dokumentace

V této kapitole si projdeme vytvořený program a jeho jednotlivé složky z programátorského hlediska a vyložíme si celkové fungování programu.

Náš program je implementován v jazyce Java, čemuž jsme uzpůsobili i výběr nástrojů (tomu jsme se více věnovali v kapitole 1.4).

3.1 Analýza kódu

Naším úkolem je vytvořit interpret programovacího jazyka a jeho vývojové prostředí. Než přistoupíme k samotnému popisu našeho jazyka, řekněme si nejdříve něco k interpretům obecně.

Hned na začátek je vhodné ujasnit si rozdíl mezi dvěma důležitými pojmy v oblasti implementace programovacích jazyků — **překladač** (compiler) a **interpret** (interpreter).

Překladač je program, který transformuje (překládá) zdrojový kód vyššího programovacího jazyka do nízkoúrovňového jazyka, tzv. strojového kódu, který je následně prováděn procesorem (Janssen a Janssen, 2020). Překladač tedy nejdříve přeloží celý kód, a tento výsledek překladu je následně vykonáván již bez závislosti na původním zdrojovém kódu

Interpret naproti tomu je program, který přímo vykonává instrukce zdrojového kódu, a to buď přeložením do mezikódu, který následně vykonává, nebo pouze pomocí syntaktické analýzy a následného přímého vykonávání jednotlivých příkazů (Janssen a Janssen, 2020).

My jsme si pro naši práci vybrali typ interpretu, který se označuje jako „tree-walk interpreter“. Fungování tohoto interpretu sestává ze tří částí — **lexikální analýza**, **syntaktická analýza** a **vyhodnocování** (Nystrom, 2020). Proberme si postupně tyto tři části.

3.1.1 Lexikální analýza

Na zdrojový kód nahlížíme jako na proud (stream) znaků. Nástroj lexikální analýzy, tzv. **lexer**, tento proud znaků přetváří na proud tzv. **tokenů**. Tokeny jsou „slova“, z nichž se skládá náš kód — např. klíčová slova, názvy proměnných, funkcí a procedur, číselné, textové či jiné konstanty, operátory nebo jiné symboly (např. středníky, dvojtečky, čárky, závorky apod.) (Nystrom, 2020).

Uveďme si jako jednoduchý konkrétní příklad zpracování aritmetického výrazu. Mějme například následující výraz.

```
2*(14+9)
```

Na začátku má náš ukázkový kód podobu proudu znaků. Naším cílem v rámci syntaktické analýzy je převést tento proud znaků na proud tokenů, v našem případě tedy snadno odhadneme, že jde o čísla, operátory a závorky (skutečný lexer zpracovává zdrojový kód podle předepsané gramatiky). Po zpracování lexerem tudíž dostaneme následující proud tokenů.

2	*	(	14	+	9	)

K vytvoření lexeru jsme použili nástroj JFlex (Klein, Rowe a Décamps, 2020). Vzhledem k tomu, že máme dva zdrojové kódy (kód grafu a kód algoritmu), máme i dva lexery — *language.loading.lexer.java* (gramatika popsána v souboru *lexer-loading.lex*) pro zpracování zdrojového kódu grafu a *language.actions.lexer.java* (gramatika popsána v souboru *lexer-actions.lex*) pro zpracování zdrojového kódu algoritmu.

Oba soubory obsahující gramatiky lexerů jsou po strukturální stránce stejné. Lze je rozdělit do tří částí.

Uživatelský kód

Uživatelský kód obsahuje určení balíčku, v němž se má nacházet výsledný lexer, a import balíčků potřebných pro fungování lexeru.

Předvolby a deklarace

Na začátku této části jsou uvedeny předvolby (options) pro fungování lexeru. V našem případě jsme v obou lexerech použili tyto předvolby (Klein a kol., 2020).

- **%class lexer** — explicitní název třídy, ve které se bude nacházet vygenerovaný kód lexeru (v našem případě se tedy třída bude jmenovat *lexer.java*).
- **%cup** — příkaz, který zajišťuje kompatibilitu s nástrojem Java CUP, který jsme použili pro syntaktickou analýzu (o tom podrobněji pojednáme v kapitole 3.1.2).
- **%unicode** — příkaz udávající znakovou sadu.
- **%line** — zapnutí počítání řádků. Počet řádků (počítáno od nuly) je následně uložen v proměnné *yyline*.
- **%column** — zapnutí počítání sloupců. Počet sloupců (počítáno od nuly) je následně uložen v proměnné *yycolumn*.
- **%caseless** — příznak ignorace velikosti písmen.

Následuje uživatelský kód (**%{ ... %}**). Ten v našem případě obsahuje pouze dvě metody sloužící ke snadnějšímu zpracování jednotlivých tokenů.

Na konci této části je definice identifikátorů zastupujících regulární výrazy.

Lexikální pravidla

Ve třetí části kódu se nachází seznam lexikálních pravidel — klíčových slov, názvů funkcí, symbolů, hodnot a komentářů.

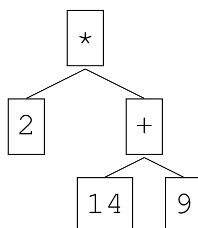
3.1.2 Syntaktická analýza

V této fázi je již kód zpracován na proud tokenů. Nástroj syntaktické analýzy, tzv. **parser**, tento proud znaků převádí podle zadané gramatiky na **abstraktní syntaktický strom** (abstract syntax tree) (Nystrom, 2020).

Uveďme si opět jednoduchý příklad na ukázku. Lexikální analýzou jsme dostali následující proud tokenů.

`2` `*` `(` `14` `+` `9` `)`

Naším cílem je převést tento proud znaků na abstraktní syntaktický strom, který by náš výraz reprezentoval. V našem případě jsou gramatikou všeobecně známá pravidla pro vyhodnocování aritmetických výrazů (skutečný parser zpracovává proud tokenů na základě předepsané gramatiky). Po zpracování parserem tedy dostaneme následující abstraktní syntaktický strom.



Vidíme, že z původního proudu tokenů byly vynechány závorky. To proto, že pro následné vyhodnocování již nejsou třeba — stromová struktura udržuje hierarchii operací sama o sobě.

Samotný strom je tvořený jednotlivými třídami (o tom podrobněji pojednáme v kapitole 3.1.3) — máme tedy třídy pro čísla a operace, tj. sčítání a násobení.

K vytvoření parseru jsme použili nástroj Java CUP (Hudson (1999)). Stejně jako u lexikální analýzy, i zde máme zvlášť parser pro každou část kódu — *language.loading.parser.java* (gramatika popsána v souboru *parser-loading.cup*) pro zpracování zdrojového kódu grafu a *language.actions.parser.java* (gramatika popsána v souboru *parser-actions.cup*) pro zpracování zdrojového kódu algoritmu.

Struktura souboru obsahujícího gramatiku parseru je rozdělena do následujících částí.

Uživatelský kód

Na začátku kódu se nachází název balíčku výsledného parseru a import balíčků pro jeho správné fungování.

Následuje uživatelský kód (**parser code {: . . . :}**) obsahující tři metody ovládající komunikaci mezi parserem a zbytkem programu.

- *getParser(String)* — dostane jako parametr zdrojový kód programu v našem jazyce a vygeneruje pro tento kód derivační strom za pomoci příslušného lexeru, který je obsažen ve stejné třídě jako parser.

- *addProgram(LoadProgram/Program)* — slouží pro předání hotového abstraktního syntaktického stromu do proměnné *program*, odkud pak je pak možné na něj odkázat.

- *addError(int, int, String)* — buduje seznam syntaktických chyb, který se pak kontroluje před vykonáváním programu a případně se vypíše na výstup.

Gramatika

Oddíl popisující gramatiku jazyka je rozdělen ještě do několika dalších oddílů.

- *Terminály* — deklarace terminálů, případně jejich návratových typů.
- *Neterminály* — deklarace neterminálů, případně jejich návratových typů.
- *Precedence* — seznam levých a pravých precedencí pro případ konfliktních situací.
- *Gramatická pravidla* — deklarace gramatických pravidel pro parser za pomoci terminálů a neterminálů.

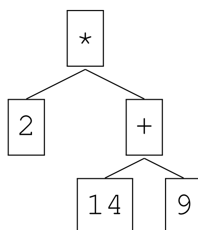
3.1.3 Vyhodnocování

V poslední fázi práce interpretu vyhodnocujeme abstraktní syntaktický strom. Jeho uzly jsou tvořeny třídami reprezentujícími jednotlivé prvky programu — např. konstanty (čísla, logické hodnoty, textové řetězce, barvy), operátory (sčítání, porovnávání, přiřazení) nebo jména konstant, ale i celé příkazy, řídící struktury, deklarace proměnných či deklarace procedur a funkcí (Nystrom, 2020).

Všechny tyto třídy obsahují metodu, po jejímž zavolání se interpretují a případně vrátí návratovou hodnotu (konkrétně v případě výrazů, konstant či funkcí). Tato metoda se pro přehlednost jmenuje u různých typů tříd různě (např. pro konstanty, výrazy a funkce je to metoda *eval()*, pro příkazy a procedury metoda *exec()*, pro deklaraci proměnných metoda *decl()* apod.), pokaždé však slouží pro vyhodnocení daného uzlu.

V případě, že je uzel koncový, provede pouze sám sebe (vrátí hodnotu, deklaruje proměnnou atd.), pokud není koncový, dá příkaz k vyhodnocení svého podstromu (či případně více jednotlivých podstromů) a teprve poté, co se interpretují všechny uzly v daném podstromě, pokračuje ve svém běhu dál (např. sečte výsledky vyhodnocených podstromů a výsledek předá vyššímu uzlu jako svou návratovou hodnotu).

Uveďme opět konkrétní příklad. Syntaktickou analýzou jsme dostali následující abstraktní syntaktický strom.



Na začátku dáme příkaz k vyhodnocení kořene — tedy násobení. Třída pro násobení má dva podstromy. Oběma tedy předá příkaz k interpretování. Levý podstrom vrátí hodnotu `2`, pravý podstrom mající v kořeni třídu pro sčítání nejdříve zadá příkaz k vyhodnocení oběma svým podstromům a poté, co dostane jejich výsledky (hodnoty `14` a `9`), tyto výsledky sečte a výsledek (hodnotu `23`) předá vyššímu uzlu jako svůj výsledek. Nyní teprve může třída pro násobení oba výsledky svých podstromů vynásobit a dostat tak výsledek — hodnotu `46`.

Na našem příkladu si také můžeme všimnout, že jednotlivé uzly nemusí mít žádné povědomí o tom, jak vypadá celý strom — pouze vrací hodnotu, případně vyžadují od svých podstromů jejich hodnoty.

3.2 Struktura programu

Nyní si popíšeme strukturu programu podle balíčků. U každého si uvedeme k čemu slouží třídy v něm obsažené, jejich návaznost na jiné třídy v programu a rovněž nejdůležitější metody.

3.2.1 Balíček *drawing*

V balíčku *drawing* se nachází dvě třídy. Jednou z nich je třída *GraphDrawingGUI*, která představuje vývojové prostředí pro náš jazyk. Právě tato třída také obsahuje metodu *main(String args[])*, tedy hlavní metodu celého kódu.

Druhou z nich je třída *Global*. Tato třída sem byla zařazena původně pro svou úzkou souvislost s vykreslováním grafu, postupně do ní ale byly přidány i další atributy a metody, např. nástroje pro práci s globálními i lokálními proměnnými, metody pro přístup k vrcholům a hranám, odkaz na okenní aplikaci vývojového prostředí nebo nástroje pro práci s uživatelskými procedurami a funkcemi.

3.2.2 Balíček *graphTools*

Balíček *graphTools* obsahuje třídy reprezentující graf a jeho části včetně tříd určených pro vykreslování grafu.

Třída *Graph* představuje instanci grafu. Obsahuje seznamy vrcholů a hran a metody pro práci s nimi — jejich ukládání, přístup k nim, jejich úpravu či odstranění — a pro vykreslování grafu (tomu věnujeme samostatný oddíl na konci této kapitoly).

Při vykreslování se používají také instance třídy *Vector*, které reprezentují dvoudimenzionální vektor.

Třídy *Vertex* a *Edge* představují instance vrcholů a hran. Obsahují atributy pro jejich identifikaci, proměnné, které jsou v každém vrcholu a každé hraně uloženy, metody pro jejich vykreslování (určení polohy vrcholů a finální vykreslení podoby pomocí třídy *Graphics2D*).

Třída *Table* a její konkretizace *TabVertex* a *TabEdge* slouží k reprezentaci a vykreslení informačních tabulek (obsahujících seznam proměnných) vrcholů a hran.

3.2.3 Balíček *language.actions*

Balíček *language.actions* obsahuje třídy určené k vykonávání zdrojového kódu algoritmu. Jedná se o hlavní a nejrozsáhlejší balíček v celém našem programu. S některými jeho částmi pracují i třídy v balíčku *language.loading*, které jsou určeny pro načítání grafu podle zdrojového kódu grafu (podrobněji se tomuto balíčku věnujeme v kapitole 3.2.4).

V tomto balíčku nalezneme třídy *Nod*, tedy reprezentaci obecného uzlu derivačního stromu, od níž přímo či nepřímo dědí většina tříd v tomto balíčku,

a třídu *Program*, která reprezentuje uživatelský kód, který se po spuštění vlákna (tato třída dědí od třídy *Thread*) začne vykonávat.

Hlavními třídami tohoto balíčku jsou ale třídy *lexer*, lexikální analyzátor kódu, *sym*, tabulka tokenů, a *parser*, syntaktický analyzátor kódu — ten buduje abstraktní syntaktický strom složený z instancí tříd, které jsou v dalších podbalíčcích balíčku *language.actions*, tj. deklarací proměnných, procedur a funkcí příkazů, výrazů atd. Podrobněji jsme o tom pojednali v kapitolách 3.1.2 a 3.1.3.

Balíček *language.actions.commands*

Hlavní třídou tohoto balíčku je třída *Cmd*, od níž dědí všechny ostatní v tomto balíčku. Jedná se o reprezentaci obecného příkazu (řídící struktury, procedury,. . .). Obsahuje dvě metody — *waitCmd()*, která zajišťuje čekání mezi dvěma příkazy (při vykonávání je kód uměle zdržován, a to z didaktických důvodů, aby bylo možné sledovat jeho běh) a *exec()*, která provede příkaz reprezentovaný konkrétní třídou.

Třídy, jejichž název začíná na *Cmd* reprezentují řídící struktury (vyjma tříd *CmdAssignment*, která reprezentuje přiřazovací příkaz, a *CmdError*, která reprezentuje neznámý příkaz a při jejímž vytváření se ve třídě *language.actions.parser* přidá do seznamu syntaktická chyba).

Třídy, jejichž název začíná na *Proc* reprezentují vestavěné procedury. O těch jsme podrobněji pojednali v příslušné sekci v kapitole 2.

Zvláštní funkci mají třídy *MyProc*, *Redraw* a *Stepping*. Třída *MyProc* reprezentuje příkaz k vykonání uživatelem definované procedury (ta je rozpoznána podle jména a počtu a typu argumentů). Třída *Redraw* reprezentuje stejnojmenný příkaz určený k přepočítání pozic vrcholů grafu. Třída *Stepping* reprezentuje příkazy *stepOff* a *stepOn*, které vypínají a zapínají krokování kódu.

Balíček *language.actions.declaration*

V tomto balíčku se nachází třídy reprezentující deklaraci proměnných a uživatelských procedur a funkcí.

Třída *Declaration* označuje obecnou deklaraci proměnné. Její hlavní metodou je metoda *decl()*, která provede deklaraci dané proměnné a vrátí seznam proměnných. Od této třídy dědí ostatní třídy začínající *Decl*, které reprezentují deklaraci proměnných jednoduchých datových typů a datových struktur. Tyto třídy nepředstavují instance těchto proměnných (k tomu slouží třída *Variable* a třídy začínající *Str* v balíčku *language.actions.structures*), pouze zabezpečují jejich deklaraci.

Třídy začínající *Def* reprezentují deklaraci uživatelských procedur a funkcí. Tyto třídy nepředstavují instance těchto procedur/funkcí (k tomu slouží třídy *RawProcedure* a *RawFunction* v balíčku *language.actions.structures*), pouze zabezpečují jejich deklaraci.

Třídy *VariableList* a *RoutineList* pak reprezentují seznam proměnných, resp. rutin (procedur a funkcí), které lze pomocí těchto tříd jednoduše deklarovat najednou.

Balíček *language.actions.exceptions*

Navzdory svému názvu naznačujícímu množné číslo obsahuje tento balíček pouze jednu třídu, a to *MyException*. Tato třída představuje speciální výjimku pro náš jazyk. Zastupuje všechny běhové chyby, které se následně liší pouze vypisovanou zprávou.

Balíček *language.actions.structures*

Hlavními součástmi tohoto balíčku jsou třídy popisující proměnné — třída *Variable*, která popisuje proměnnou jednoduchého datového typu, a třídy začínající na *Str* (*StrArray*, *StrQueue*, *StrSet* a *StrStack*), které popisují datové struktury. Součástí balíčku je i výčtový typ *VariableType* určující typ proměnné (a také hodnoty *NIL* a chybový typ proměnné *error*).

Množina (*StrSet*), zásobník (*StrStack*) a fronta (*StrQueue*) jsou implementovány pomocí běžných prvků dostupných v Javě (*HashSet*, *Stack* a *LinkedList*). O něco složitější je implementace pole (*StrArray*). Jednorozměrné pole by bylo možné implementovat jednoduše pomocí jednorozměrného pole v Javě (a taky je tak implementováno), složitější je situace u vícerozměrných polí, neboť v Javě nelze deklarovat obecné n-rozměrné pole. Přistoupili jsme tedy k implementaci pomocí jednorozměrného pole, které může obsahovat libovolnou proměnnou (tj. *Variable*) — a tedy i *StrArray*. Dvourozměrné pole je tedy implementováno jako pole polí, trojrozměrné pole jako pole polí polí atd.

Další dvě důležité třídy jsou *RawProcedure* a *RawFunction*, které reprezentují uživatelem deklarované procedury a funkce. Tyto třídy se vytvoří při deklaraci dané procedury/funkce a skladují se ve třídě *Global*, odkud se kopírují vždy, když je potřeba je použít v kódu programu.

Dále tento balíček obsahuje struktury potřebné pro fungování jiných komponent programu — třídu *Branch* reprezentující větev příkazu case a třídu *Interval*, která se užívá pro specifikaci rozměrů pole (věrni myšlence Pascalu jsme ponechali deklaraci rozměru pole intervalem).

Také je sem zařazena třída *Nil* reprezentující nulovou hodnotu. Tato hodnota se používá pouze pro inicializaci proměnných typu **vertex** a **edge** (nelze říci, že by jeden vrchol byl významěјší než jiný, a tedy si zasloužil být implicitní hodnotou; podobně u hrany), případně pro opětovné vynulování její hodnoty.

Balíček *language.actions.values*

Nejrozsáhlejším balíčkem našeho programu je právě balíček obsahující třídu *Value*, která označuje obecnou hodnotu, a třídy, které od ní dědí. Nejdůležitější metodou je metoda *eval()*, která vrátí hodnotu, kterou třída reprezentuje (tedy například číselnou konstantu v případě třídy *ValueInteger* nebo součet v případě třídy *ExprPlus*). Třídy v tomto balíčku bychom mohli rozdělit do tří hlavních skupin.

Třídy začínající na *Value* reprezentují konkrétní elementární hodnoty, případně hodnoty reprezentované proměnnými nebo odkazující na složitější struktury (toto se týká tříd *ValueVertex* a *ValueEdges*, které obsahují pouze hodnoty, které tvoří identifikátory vrcholu či množiny hran, tříd reprezentujících proměnné

náležící vrcholům či hranám a tříd reprezentujících proměnné odkazující na konkrétní prvek v poli).

Další skupinou jsou třídy začínající na *Expr*. Tyto třídy reprezentují elementární binární (někdy však i unární — *ExprUnMinus*, *ExprUnPlus* a *ExprNot*) operace. Tyto třídy lze samozřejmě do sebe libovolně zanořovat. Operandy mohou být jakékoli třídy odvozené od třídy *Value*.

Poslední skupinou jsou třídy začínající na *Func*, které reprezentují vestavěné funkce programu. O těch jsme podrobněji pojednali v kapitole 2.4.5.

Speciální postavení má třída *MyFunc*, kterou jsme nezařadili do žádné ze zmíněných skupin. Tato třída reprezentuje použití uživatelem definované funkce. Jedná se o obdobu třídy *MyProc* z balíčku *language.actions.commands* a stejně jako ona je rozpoznávána podle jména a počtu a typu svých argumentů.

Balíček *language.actions.variables*

Tento balíček obsahuje třídy reprezentující samotné proměnné, zejména pak třídu *VariableID*, od které dědí další třídy tohoto balíčku. Třídy v tomto balíčku reprezentují odkazy na proměnné v programu — konkrétně třída *VarSimple* na obyčejnou proměnnou (ať už jednoduchého datového typu nebo datovou strukturu), třída *VarVEVariable* na proměnnou vrcholu či hrany a třída *VarArray* na konkrétní prvek v poli.

Instance těchto tříd bývají použity na místech, kde je potřeba do proměnné přiřadit nějakou hodnotu, tj. například na levé straně přiřazovacího příkazu.

3.2.4 Balíček *language.loading*

Balíček *language.loading* obsahuje třídy určené k vykonávání zdrojového kódu grafu.

Podobně jako v balíčku *language.actions* (ten je podrobně popisován v kapitole 3.2.3) i zde nalezneme třídu reprezentující uzel derivačního stromu; zde je pojmenována *LoadNod*.

Třída *GraphMaking* slouží k přidávání vrcholů do grafu. Hlavním významem tohoto mezistupně mezi třídami reprezentujícími přidání hrany (v balíčku *language.loading.commands*) a metodami v třídě *Graph* (v balíčku *graphTools*) je notnost vyplnit proměnné každého vrcholu zadanými hodnotami.

Stejně jako v balíčku *language.actions* se i zde nachází třídy *lexer*, *sym* a *parser* tvořící jednotlivé složky interpretu.

Balíček *language.loading.commands*

V tomto balíčku se nachází třídy reprezentující příkazy pro vytvoření grafu. Hlavní třídou je *LoadCmd*, reprezentace obecného příkazu — obdobně jako ve třídě *language.actions.commands* (podrobně popsána v kapitole 3.2.3).

Třída *LoadCmdError* reprezentuje neznámý příkaz a při jejím vytvoření se do seznamu chyb ve třídě *language.loading.parser* přidá syntaktická chyba.

Zbývající procedury jsou reprezentací příkazů (tedy tří vestavěných procedur, o nichž jsme podrobněji pojednali v příslušné sekci v kapitole 2.4.4) pro vytvoření vrcholů a hran.

Balíček *language.loading.program*

Tento balíček obsahuje dvě třídy — *LoadVariableDeclaration*, která slouží k hromadné deklaraci proměnných vrcholů a hran, a *LoadProgram*, která představuje program pro vytvoření grafu (deklarace proměnných vrcholů a hran a seznam příkazů).

Balíček *language.loading.values*

Tento balíček obsahuje třídy reprezentující hodnoty, které se ukládají do proměnných vrcholů a hran — obdobně jako ve třídě *language.actions.values* (o té jsme se podrobně rozepsali v kapitole 3.2.3).

Třída *LoadValue* označuje obecnou hodnotu typu jiného než **vertex** nebo **set of edge** — k vytoření hodnot těchto typů slouží třídy *LoadValueVertex* a *LoadValueEdges*. Tyto typy jsou unikátní, neboť odkazované vrcholy či hrany v době vytvoření daného vrcholu/hrany nemusí existovat. To je vyřešeno pomocí tříd *LoadVertexLink* a *LoadEdgesLink*, které slouží jako odkazy na vrcholy či množiny hran a později v programu se vyhodnotí přímo na své skutečné hodnoty.

3.3 Kreslení grafu

Ačkoli je naše práce z velké části zaměřena na vizualizaci, nekladli jsme na preciznost kreslení grafu (myšleno zjištění vhodné vzájemné pozice vrcholů) příliš velký důraz, a to z několika důvodů.

Předně je kreslení grafů příliš složitým problémem, než abychom se mu mohli v naší práci věnovat v celé jeho šíři. Náš program slouží didaktickým účelům, proto můžeme předpokládat, že nebude třeba zobrazovat příliš velké grafy (na velkých grafech by nebyl příliš dobře vidět princip fungování algoritmu — naším cílem je vytvořit program pro přehlednou vizualizaci grafových algoritmů, nikoli pro zpracování velkých dat). Také jsme přidali možnost znovu vynutit překreslení grafu (tlačítko **Překresli** ve vývojovém prostředí) a rovněž možnost ruční úpravy podoby grafu.

Ke kreslení jsme vybrali základní fyzikální algoritmus, který se snaží o minimalizaci délky hran a maximalizaci vzdáleností mezi vrcholy (Kratochvíl, 2008). Naše představa je tedy taková, že vrcholy jsou odpuzující se nabité částice a hrany jsou pružiny přitahující k sobě své krajní vrcholy.

Přitažlivou sílu jednotlivých hran počítáme pomocí Hookova zákona (Kratochvíl, 2008), tedy

$$F_{i,j} = k \cdot |i{,}j|\,,$$

kde i a j jsou krajní vrcholy hrany, k je váha pružiny (pro všechny hrany stejná) a $|i{,}j|$ je vzdálenost vrcholů i a j.

Odpudivou sílu jednotlivých vrcholů počítáme pomocí Coulombova zákona (Kratochvíl, 2008), tedy

$$F_{i,j} = \frac{Q^2}{|i{,}j|^2},$$

kde i a j jsou vrcholy, Q je náboj vrcholu (pro všechny vrcholy stejný) a $|i,j|$ je vzdálenost vrcholů i a j.

Diskrétní simulací vždy vypočítáme celkovou sílu, která na každý vrchol působí (součet dílčích sil), a z té pak rychlost podle vzorce

$$v_i = d \cdot (F_i \cdot \Delta t + v_i)\,,$$

kde i je konkrétní vrchol, F_i součet sil, které na něj působí, $\Delta\ t$ časový úsek simulace a d odpor prostředí (Kratochvíl, 2008). Novou pozici vrcholu vypočítáme pomocí vzorce

$$s_i = s_i + v_i \cdot \Delta t,$$

kde i je konkrétní vrchol, v_i rychlost vrcholu a $\Delta\ t$ časový úsek simulace (Kratochvíl, 2008).

Simulace běží buď do chvíle, kdy je celková energie — počítáno pomocí vzorce $E = \sum_{i \in V} {v_i}^2$ (Kratochvíl, 2008) — menší než 1, nebo maximálně 10000 iterací.

Za váhu pružiny k jsme zvolili číslo 0,1, za náboj vrcholu Q číslo 500, za odpor prostředí d číslo 0,75 a za časový úsek $\Delta\ t$ číslo 1. Čísla byla vybrána experimentálním zkoušením, kdy jsme subjektivně posuzovali vzhled výsledných grafů.

4. Ukázkové algoritmy

V této kapitole si představíme některé grafové algoritmy, které lze implementovat s pomocí našeho jazyka, abychom viděli, jak v našem jazyce vypadá rozsáhlejší kód.

Vybrali jsme tři algoritmy — Kruskalův algoritmus (hledání minimální kostry grafu), algoritmus pro hledání silně souvislých komponent grafu a simulaci běhu konečného deterministického automatu. Jednotlivé algoritmy byly vybrány tak, aby ilustrovaly co možná nejrůznorodější problémy řešitelné pomocí grafů.

U jednotlivých algoritmů se zaměříme pouze na nejdůležitější či nejzajímavější části. Ve zdrojovém kódu grafu nás tedy bude zajímat jen deklarace proměnných, ve zdrojovém kódu algoritmu pak zejména hlavní vlákno, případně důležité funkce či procedury. Kompletní zdrojové kódy následně nalezneme v příloze.

4.1 Kruskalův algoritmus

Prvním algoritmem, který si ukážeme, je Kruskalův algoritmus pro hledání minimální kostry v grafu.

Kruskalův algoritmus funguje tak, že seřadí všechny hrany do posloupnosti podle velikosti jejich ohodnocení (od nejmenšího po největší) a následně se pokouší postupně přidat ho množiny hran podgrafu (pozdější kostry) původního grafu všechny hrany této posloupnosti s tím, že uspěje v případě, že přidání dané hrany nevytvoří cyklus. Nejpozději po projití všech hran dostaneme kostru grafu (Matoušek a Nešetřil, 2002).

Nejdříve si připravíme graf, jehož kostru budeme hledat. Jak už jsme řekli, konkrétní vrcholy a hrany grafu nejsou důležité. Zajímat nás bude pouze deklarace proměnných vrcholů a hran.

U hran budeme potřebovat celočíselnou proměnnou, která nám udává ohodnocení hrany, na které budeme brát zřetel při výběru hran, které budeme postupně zkoušet přidávat do budované kostry. Pro jednoduchost budeme pracovat pouze s celočíselnými ohodnoceními hran. Dále budeme potřebovat zjišťovat, zda přidáním hrany nevytvoříme cyklus, což provedeme rozdělením vrcholů do množin podle aktuálních komponent podgrafu (budoucí kostry). To z praktických důvodů neimplementujeme pomocí datového typu **set of vertex**, ale pomocí identifikátoru, který bude udávat, do jaké množiny vrchol patří. Budeme tedy potřebovat celočíselnou proměnnou pro vrcholy. Na závěr, pro lepší přehlednost, obarvíme hrany, které patří do kostry, na což si pro hrany zavedeme proměnnou označující barvu.

Deklarace proměnných ve zdrojovém kódu grafu tedy bude vypadat následovně.

```
vertexVar
   mnozina : integer;

edgeVar
   vaha : integer;
   barva : color;
```

Nyní už můžeme přistoupit k samotné implementaci Kruskalova algoritmu.

Posloupnost hran reprezentujeme polem (**array**), které má tu vlastnost, že uchovává své prvky v pevném pořadí (na rozdíl od množiny), lze přistupovat k jednotlivým prvkům a rovněž lze s jeho prvky manipulovat (na rozdíl od fronty či zásobníku). Jak jsme zmiňovali v kapitole 1.1, je náš jazyk určený pro didaktické účely, a tedy graf, s nímž budeme pracovat, bude spíše menšího rozsahu. Bude nám proto stačit pole o stu prvcích (navíc je náš kód jen ukázkový a jeho podobu je možné pro případnou prezentaci ve výuce poupravit — je jen nutné, aby mělo pole alespoň takovou velikost, jaký je počet hran v grafu).

Hrany, které přidáme do kostry, budeme přidávat do množiny (**set of edge**), kdybychom chtěli kód nějak rozšířit a s výslednou kostrou nějak dále pracovat. Hrany kostry budeme také hned při přidání do množiny obarvovat na červeno.

V příkladovém kódu níže byly vynechány implementace pomocných procedur, aby bylo možné se soustředit na samotné jádro algoritmu.

- *Inicializace()* — přidá všechny hrany grafu do pole a zajistí, aby číslo množiny bylo u každého vrcholu jiné.
- *Setrizeni()* — setřídí hrany v poli vzestupně podle jejich ohodnocení. Vzhledem k malé velikosti grafu není potřeba příliš řešit efektivnost algoritmu (žáci se tedy nemusí soustředit na volbu a implementaci třídícího algoritmu a mohou zaměřit svou pozornost na samotný Kruskalův algoritmus).
- *Sjednoceni(integer, integer)* — sjednotí množiny vrcholů určené svými čísly. Tuto proceduru budeme volat pokaždé, když přidáme hranu do kostry.

```
var
   i : integer;
   arr : array[1..100] of edge;
   s : set of edge;

procedure Inicializace();
   \\ implementace procedury

procedure Setrizeni();
   \\ implementace procedury

procedure Sjednoceni(i : integer; j : integer);
   \\ implementace procedury

begin
   Inicializace();

   Setrizeni();

   for i := 1 to Length(AllEdges()) do
      begin
         if arr[i].v1.mnozina <> arr[i].v2.mnozina then
            begin
               Add(arr[i], s);
               arr[i].barva := red;
```

```
                Sjednoceni(arr[i].v1.mnozina, arr[i].v2.mnozina);
            end;
      end;
end.
```

Hlavní vlákno kódu je, jak vidíme, v jádru jednoduché. Na začátku inicializujeme pole a vrcholy a setřídíme hrany v poli podle jejich ohodnocení. Poté pomocí *for* cyklu postupně probíráme všechny hrany uložené v poli *arr* a pokoušíme se je přidat do budované kostry — to se podaří, pokud jsou identifikátory množin ve vrcholech hrany různé. Zároveň se také přidávaná hrana obarví a sjednotí se množiny, které tato hrana propojuje.

Oba zdrojové kódy s implementací všech procedur nalezneme v příloze v souborech `1-graf-kruskal.txt` (zdrojový kód grafu) a `1-algoritmus-kruskal.txt` (zdrojový kód algoritmu).

4.2 Algoritmus pro hledání silně souvislých komponent grafu

Druhým algoritmem, který si ukážeme, je algoritmus pro hledání silně souvislých komponent grafu (Mareš a Valla, 2017).

Pro běh algoritmu budeme potřebovat prázdný zásobník. Algoritmus nejdříve vytvoří transponovaný graf. Na něm následně spouštíme algoritmus prohledávání do hloubky (DFS), dokud neprozkoumáme všechny vrcholy. Vždy, když opouštíme vrchol, vložíme ho do zásobníku, který jsme si vytvořili na začátku. Poté postupně odebíráme vrcholy ze zásobníku a z každého z nich spustíme DFS, přičemž vstupujeme pouze do vrcholů, které ještě nejsou součástí žádné komponenty. Pro každý takto vytvořený DFS-stromu přiřadíme všem jeho vrcholům stejný identifikátor komponenty.

Opět si nejdříve připravíme graf, v tomto případě orientovaný. Budeme potřebovat jednu proměnnou pro vrcholy, do níž v poslední fázi algoritmu zapíšeme identifikátor komponenty. Pro jednoduchost budeme komponenty označovat celými čísly. Dále budeme vytvářet transponovaný graf, pro který použijeme vrcholy původního grafu, musíme tedy nějak odlišit hrany původního a transponovaného grafu — a to nejlépe barvou, budeme tedy potřebovat proměnnou typu **color** pro hrany. Pro správný běh DFS budeme potřebovat ve vrcholech uchovávat jejich aktuální barvu (bílou, šedou a černou pro nenavštívené, navštívené a uzavřené vrcholy).

```
vertexVar
   komponenta : integer;
   barva : color;

edgeVar
   barva : color;
```

Nyní už můžeme jednoduše implementovat samotný algoritmus.

Jak už jsme uvedli v popisu algoritmu, budeme potřebovat zásobník, do něhož budeme ukládat vrcholy. V poslední fázi algoritmu také budeme potřebovat celočíselnou proměnnou pro postupné počítání označení komponenty grafu.

V kódu jsme opět neimplementovali žádné procedury a soustředili jsme se pouze na hlavní vlákno programu. Řekněme si ale, co která procedura dělá.

- *Transponovat()* — pro všechny hrany v grafu přidá do grafu novou hranu vedoucí v opačném směru a mající červenou barvu.
- *DFS(vertex)* — spustí z daného vrcholu rekurzivně prohledávání do hloubky pro všechny jeho sousední vrcholy, s nimiž ho spojuje červená hrana (tedy běží na transponovaném grafu).
- *DFS(vertex, integer)* — spustí z daného vrcholu rekurzivně prohledávání do hloubky pro všechny jeho sousední vrcholy, s nimiž ho spojuje černá hrana (tedy běží na transponovaném grafu) a které ještě nemají číslo komponenty (přesněji řečeno číslo jejich komponenty je 0).

```
var
   z : stack of vertex;
   v : vertex;
   i : integer;
   e : edge;

procedure Transponovat();
   \\ implementace procedury

procedure DFS(v : vertex);
   \\ implementace procedury

procedure DFS(v : vertex; i : integer);
   \\ implementace procedury

begin
   Transponovat();

   for v in AllVertices() do
      if v.barva = white then
         DFS(v);
   for v in AllVertices() do
      v.barva := white;

   i := 1;
   while length(z) > 0 do
      begin
         v := Pop(z);
         if (v.komponenta = 0) then
            begin
               DFS(v,i);
               i := i + 1;
            end;
```

```
      end;
   for v in AllVertices() do
   v.barva := white;

   for e in AllEdges() do
      if e.barva = red then
         DeleteEdge(e);
end.
```

Hlavní vlákno kódu je rozděleno do čtyř částí. Nejdříve vytvoříme transponovaný graf. Poté, ve druhé části, ze všech vrcholů, které v době, kdy přijdou na řadu, budou stále bílé, spustíme proceduru *DFS(vertex)*. Tím dostaneme zásobník s topologicky seřazenými vrcholy, které budeme v tomto pořadí následně probírat — k tomu slouží třetí část kódu, v níž ze všech vrcholů, které ještě nemají určenou komponentu, spustíme proceduru *DFS(vertex, integer)* s postupně se navyšujícím identifikátorem komponent. Ve čtvrté části kódu pouze vymažeme z grafu červené hrany (tj. hrany transponovaného grafu).

Kompletní zdrojové kódy jsou opět obsaženy v příloze, konkrétně v souborech `2-graf-komponenty.txt` (zdrojový kód grafu) a `2-algoritmus-komponenty.txt` (zdrojový kód algoritmu).

4.3 Simulace běhu konečného deterministického automatu

Třetím a posledním algoritmem, který si ukážeme, je simulace běhu konečného deterministického automatu.

Konečný deterministický automat je pětice obsahující stavy automatu, vstupní abecedu, přechodovou funkci (δ), počáteční stav a množinu koncových stavů. Přechodová funkce δ dostane jako argumenty stav, v němž se automat nachází, a aktuálně čtený znak ze vstupu a vrátí stav, do něhož se má automat přesunout.

V naší reprezentaci budou stavy reprezentovány vrcholy a funkce δ hranami. Ve hranách tedy budeme potřebovat uchovat znak, kterým se přejde do dalšího stavu. Ve vrcholech pak budeme držet informaci, zda daný vrchol je nebo není koncový. Pro přehlednější vizualizaci běhu algoritmu také přidáme vrcholům a hranám proměnnou typu **color**, jejíž úpravou budeme vizualizovat běh algoritmu.

```
vertexVar
   fin : boolean;
   c : color;

edgeVar
   ch : char;
   c : color;
```

V proměnných si budeme potřebovat pamatovat vrchol, který reprezentuje aktuální stav automatu, a textový řetězec reprezentující čtené slovo.

Hlavní součástí je, jak už jsme si řekli, funkce δ, která z aktuálního vrcholu a čteného znaku vypočítá nový stav. Tuto funkci jsme proto v implementaci nevynechali. Naše funkce *Delta(vertex, char)* projde všechny okolní hrany daného vrcholu a vybere z nich tu, ve které je správný znak, a tuto hranu zvýrazní. V této implementaci také můžeme vidět příkazy *StepOff* a *StepOn*, které nám umožňují nezobrazovat průběh kódu, který vybírá správnou hranu.

```
var
   actVert : vertex;
   s : string;
   i : integer;

function Delta(v : vertex; c : char) : vertex;
var
   e, deltaEdge : edge;
begin
   StepOff;
   for e in AdjEdges(v) do
      if (e.ch = c) then
         begin
            deltaEdge := e;
            Delta := e.v2;
         end;
   StepOn;
   deltaEdge.c := red;
   deltaEdge.c := black;
end;

begin
   s := {vstupni slovo};

   actVert := {pocatecni vrchol};
   actVert.c := red;
   for i := 1 to length(s) do
      begin
         actVert.c := white;
         actVert := Delta(actVert,s[i]);
         actVert.c := red;
      end;

   WriteLn(actVert.final)
end.
```

Hlavní vlákno kódu je pak velice jednoduché. Pomineme-li příkazy ovládající zvýrazňování aktuálního vrcholu, skládá se pouze z inicializace vstupního slova a počátečního vrcholu a z *for* cyklu, který postupně volá funkci δ a aktualizuje vrchol reprezentující aktuální stav. Na konci vypíšeme, zda jsme či nejsme v koncovém stavu, tedy zda slovo bylo či nebylo automatem přijato.

Zdrojové kódy opět nalezneme v příloze, a to v souborech `3-graf-automat.txt` (zdrojový kód grafu) a `3-algoritmus-automat.txt` (zdrojový kód algoritmu).

Závěr

Podařilo se nám vytvořit jazyk a jeho vývojové prostředí vhodné pro interaktivní vizualizaci grafových algoritmů. Program je využitelný pro výuku na střední škole — jak bylo požadováno na začátku — a díky svému provedení může sloužit jako pomůcka, jak žákům ukázat programování jako zajímavý obor lidské činnosti.

Nebyly implementovány všechny prvky jazyka, které bychom pro úplnost očekávali — omezená je zejména paleta vestavěných procedur a funkcí (tu lze však podle potřeby jednoduše rozšířit), také cykly, procedury a funkce nelze ukončit předčasně (chybí implementace klíčových slov *break* a *continue*). Pro implementaci většiny grafových algoritmů to však není nijak omezující, navíc je možné náš jazyk o tyto prvky s nevelkou námahou rozšířit.

Seznam použité literatury

BÖSZÖRMÉNYI, L. (1998). Why Java is not my favorite first-course language. *Software-Concepts & Tools*, **19**(3), 141–145.

DIJKSTRA, E. W. (1968). Letters to the editor: go to statement considered harmful. *Communications of the ACM*, **11**(3), 147–148.

GRANDELL, L., PELTOMÄKI, M., BACK, R.-J. a SALAKOSKI, T. (2006). Why complicate things? Introducing programming in high school using Python. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 71–80.

HORNIK, T., MUSÍLEK, M. a MILKOVÁ, E. (2019). Didaktika programování. URL `https://imysleni.cz/images/vyukove_materialy/UHK_Didaktika_programovani.pdf`.

HUDSON, S. E. (1999). Cup user's manual. URL `https://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html`.

JANSSEN, D. a JANSSEN, C. (2020). Technopedia. URL `https://www.techopedia.com/`.

JENSEN, K. a WIRTH, N. (1974). *Lecture Notes in Computer Science*. III. Series. Springer-Verlag, Berlin. ISBN 3-540-06950-X.

KAASBØLL, J. J. (1998). Exploring didactic models for programming. In *NIK 98–Norwegian Computer Science Conference*, pages 195–203. Citeseer.

KERNIGHAN, B. W. (1981). Why Pascal is not my favorite programming language.

KLEIN, G., ROWE, S. a DÉCAMPS, R. (2020). Jflex user's manual, version 1.8.2. URL `https://jflex.de/manual.html`.

KRATOCHVÍL, J. (2008). Kreslení grafů. Bakalářská práce, Univerzita Karlova v Praze, Matematicko-fyzikální fakulta.

MAREŠ, M. a VALLA, T. (2017). *Průvodce labyrintem algoritmů*. CZ.NIC, z. s. p. o., Praha. ISBN 978-80-88168-22-5.

MATOUŠEK, J. a NEŠETŘIL, J. (2002). *Kapitoly z diskrétní matematiky*. Nakladatelství Karolinum, Praha. ISBN 80-246-0084-6.

NYSTROM, R. (2020). Crafting interpreters. URL `https://www.craftinginterpreters.com/contents.html`.

ORACLE (2019). JDK 13 Documentation. URL `https://docs.oracle.com/en/java/javase/13/`.

ROBINS, A., ROUNTREE, J. a ROUNTREE, N. (2003). Learning and teaching programming: A review and discussion. *Computer science education*, **13**(2), 137–172.

van Canneyt, M. (2017). Reference guide for Free Pascal, version 3.0.4. URL https://www.freepascal.org/docs-html/ref/ref.html.

Seznam obrázků

2.1 Ukázka podoby celého programu. 13
2.2 Ukázka textového pole pro zápis zdrojového kódu. 14
2.3 Ukázka tlačítek pro práci se zdrojovým kódem grafu. 14
2.4 Ukázka pole pro výpis výstupu programu a chybových hlášek. . . 15
2.5 Ukázka panelu pro vykreslení grafu. 16
2.6 Ukázka prvků pro ovládání běhu programu a vykreslování. 17
2.7 Graf vytvořený podle zdrojového kódu z úvodu kapitoly 2.5 . . . 33

Seznam tabulek

2.1 Operátory 1. úrovně (expressions). 25
2.2 Operátory 2. úrovně (simple expressions). 25
2.3 Operátory 3. úrovně (terms). 26
2.4 Operátory 4. úrovně (factors). 26

A. Přílohy

Nedílnou součástí práce je i elektronická příloha. V příslušném adresáři (komprimovaném do formátu ZIP) se nachází

- adresář `program` obsahující soubor `graph_algorithms_visualisation.jar`, tedy samotný program, který je jádrem celé naší práce, a adresář `lib` obsahující knihovnu pro správnou funkci parseru,
- adresář `src`, který obsahuje zdrojové kódy vytvořeného programu,
- adresář `javadoc` obsahující automaticky generovanou dokumentaci k našemu programu ve formátu *Java Doc*,
- adresář `ukazkove_zdrojove_kody` obsahující šest zdrojových kódů, které jsou kompletními verzemi kódů uvedených v kapitole 4.