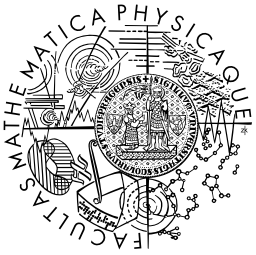


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Michal Popek

Animace grafových algoritmů v MonoGame

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D.
Studijní program: Informatika (B0613A140006)
Studijní obor: IPP6 (0613RA1400060010)

Praha 2024

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

Tímto bych rád poděkoval rodině za podporu během studia, Univerzitě Karlově za umožnění studia a především bych rád poděkoval panu RNDr. Martinovi Pergelovi, PH.D. za vedení této práce a za pomoc a konzultace při její psaní.

Název práce: Animace grafových algoritmů v MonoGame

Autor: Michal Popek

Katedra: Katedra softwaru a výuky informatiky

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

Abstrakt: Výstupem této bakalářské práce je výukový program zaměřený na grafové algoritmy, který je určen pro použití na technicky zaměřených středních a vysokých školách. Součástí programu je proprietární jazyk Cb, který je použit pro psaní již zmíněných grafových algoritmů. V textu této práce se budeme bavit o zpracovávání a prezentování algroritmů uživateli, tedy se podíváme na kompilátor a jeho konstrukci a na krokování algoritmů při běhu programu. Dále jsme pro vykreslování grafů využili pružinkového algoritmu pro automatické generování jeho rozložení do okna aplikace.

Klíčová slova: grafy, grafové algoritmy, kompilátor, MonoGame

Title: Animation of Graph Algorithms in MonoGame

Author: Michal Popek

Department: Department of Software and Computer Science Education

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

Abstract: The result of this bachelor thesis is an application for teaching graph algorithms intended to be used at technically oriented high schools and universities. The application comes with a proprietary language Cb, which is used to write before mentioned graph algorithms. In this text we shall discuss processing and presentation of algorithms, thus we will take a look at compiler and its construction and stepping of algorithms during run time. To draw the graphs we use force-directed graph drawing algorithms for generating the layout of graphs automatically on a computer screen.

Keywords: graphs, graph algorithms, compilers, MonoGame

Obsah

Úvod

Grafové algoritmy jsou jedny z nejdůležitějších algoritmů pro informatiky. Zahrnují například algoritmy pro prohledávání prostoru a hledání nejkratších cest, toky v sítích, nebo hledání minimální kostry. Algoritmy dále můžeme rozdělit podle grafů, na kterých běží. Tyto grafy můžeme rozdělit na ohodnocené a neohodnocené. Přestože se podle mě jedná o základní znalosti, studenti mohou mít s těmito algoritmy problémy.

Cílem této bakalářské práce bylo tedy sestrojit program, který by vizualizoval grafy a algoritmy na nich běžící v reálném čase. Program byl napsán v jazyce C# ve frameworku MonoGame.

Tento program by poté mohl sloužit při výuce algoritmů a programování, buď jako program pro přednášející nebo pro studenty na vyzkoušení napsání algoritmu a následný pohled na běh s možností pozastavení.

Program bere jako vstup soubory s algoritmy napsané v proprietátním jazyce, a tedy se budeme bavit i o kompilátoru, který tyto soubory převede do syntaktického stromu, který následně při běhu procházíme.

Taktéž si ukážeme, jak vypadá uživatelské rozhraní a jak může uživatel přidat další grafy a algoritmy do programu.

1. Požadavky programu

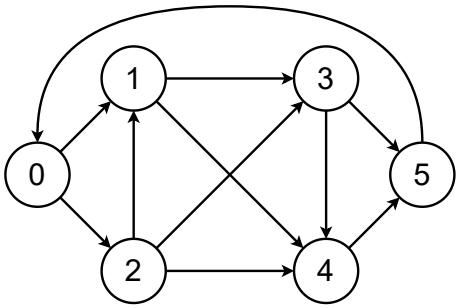
Než začneme s popisem implementace, podívejme se, co chceme implementovat. Chceme aplikaci napsanou v C#, která dostane jako vstup grafový algoritmus a graf, který bude vstupem algoritmu. Graf chceme vykreslit do okna a na tomto grafu poté budeme vizualizovat běh algoritmu.

1.1 Reprezentace grafů

Začněme s grafem, resp. s jeho reprezentací. Graf je množina vrcholů a množina hran, kde každá hrana je dvojice vrcholů.

První možná reprezentace je *matice sousednosti*, kde pro N vrcholů máme matici $N \times N$. Pro řádek i a sloupec j pak pole matice obsahuje 0, pokud neexistuje hrana mezi vrcholy i a j a 1 jinak. Matici můžeme upravit tak, aby hodnota pole (i, j) byla hodnota hrany mezi vrcholy i a j.

Jelikož chceme obecné grafové algoritmy, ohodnocení hran jednou hodnotou není dostatečné a tedy matice sousednosti odpadá.



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Obrázek 1.1: Orientovaný graf a jeho matice sousednosti

Další možná reprezentace je seznam vrcholů a seznam hran. Každá hrana má odkazy na vrcholy (počátek a konec pro orientované grafy) a vrchol má odkaz na hrany vycházející z něj. Vrcholy a hrany pak mají seznam proměnných (např. pro toky v sítích máme ve vrcholech *přebytek* a ve hranách *kapacitu* a *tok*).

Tato reprezentace je vyhovující. Uživatel určí počet vrcholů a dvojici indexů jako počátek a konec hrany. Pokud uživatel bude chtít ohodnocený graf (vrcholy i/nebo hrany), pak stačí, aby napsal proměnné, které bude chtít použít v algoritmu a předal jim ohodnocení. Musíme sice řešit orientaci hran (resp. zda je graf orientovaný nebo ne), ale tento problém je velice malý a jde snadno vyřešit.

Pro kreslení grafů je tato implementace vyhovující, jelikož vrcholům můžeme přiřadit pozici.

1.2 Kreslení grafů

Při kreslení grafů se musíme podívat, jak rozmístit vrcholy a jak popsat ohodnocení vrcholů a grafů. Říci uživateli, aby graf ručně uspořádal, je zvláštní a tedy chceme rozložení grafu najít algoritmicky. Jelikož budeme graf vykreslovat na 2D plochu okna, vyplatí se hledat 2D vykreslovací algoritmy.

Jednou skupinou takových algoritmů jsou *pružinkové algoritmy*, kde provádíme fyzikální simulaci natahování a smršťování kovových pružin.

Další metody kreslení grafů jsou hiearchické metody, kde pro orientovaný acklický graf přiřadíme vrcholu v nějakou vrstvu l. Pro hranu (u, v) přiřadíme vrcholu u vrstvu menší než l a pro hranu (v, w) přiřadíme vrcholu w vrstvu větší než l.

Jelikož se bavíme o obecných grafech (orientované/neorientované s možnými cykly), pouze pružinková metoda dává smysl.

1.3 Algoritmus

Posledním velkým dílem aplikace je vstup algoritmu.

Uživatel by mohl napsat algoritmus v nějakém jazyce a my bychom tento jazyk interpretovali. Existující jazyk by měl výhodu dokumentace a nebo by jej uživatel mohl znát předem. Dále by takto napsaný algoritmus mohl vyzkoušet mimo naší aplikaci. Problém je, že interpret nebo kompilátor je uzavřená schránka a my se do nich za standardních podmínek nedostaneme. Tedy zajistit krokování by bylo velmi obtížné.

Proto bylo rozhodnuto, že pro naše účely vytvoříme vlastní jazyk a výsledný algoritmus budeme krokovat my. Algoritmus můžeme zkompilovat tak, abychom ho snadno krokovali. Zároveň budeme mít přístup k paměti, což se bude hodit při předání grafu algoritmu a při následném čtení grafu.

2. Návrh programu

V této kapitole se podíváme na návrh celého programu a tedy se na něj podíváme velmi obecně. O jednotlivých dílech se budeme podrobněji bavit v dalších kapitolách tohoto textu.

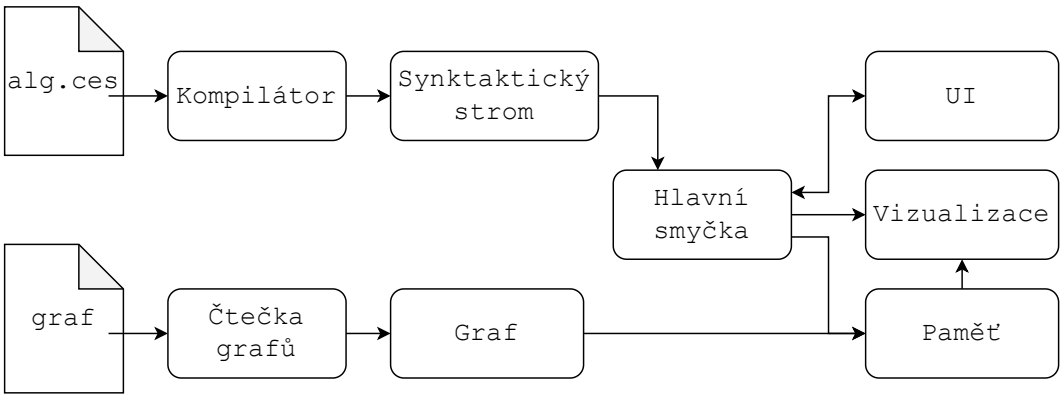
Program můžeme rozdělit do několika logických celků. Začněme kompilátorem, který dostane jako vstup algoritmus napsaný ve vlastním jazyce Cb a jeho výstupem je syntaktický strom, který dostane hlavní třída programu.

Dále máme čtečku grafů, která dostane soubor s popisem grafu. Vrcholům i hranám je možno přiřadit pole. Takové pole je proměnná s celočíselným typem *int*. Tento graf následně předáme reprezentaci paměti. S touto pamětí poté manipuluje zkompilovaný algoritmus přidáváním proměnných a jejich přepisováním a popřípadným mazáním.

Hlavní smyčku programu implementujeme pomocí frameworku MonoGame. Zde se staráme o krokování algoritmu a vykreslování grafu a uživatelského rozhraní.

S prvky uživatelského rozhraní může uživatel interagovat a pomocí nich nastavit, který algoritmus se má spustit na jakém grafu a následně např. pozastavit algoritmus nebo změnit rychlost jeho běhu.

Vizualizace grafu se zároveň stará o rozložení vrcholů do okna. Toto rozložení pak uživatel může změnit. Nakonec vrcholy a hrany grafů budou měněny podle toho, jak je k nim přistupováno.



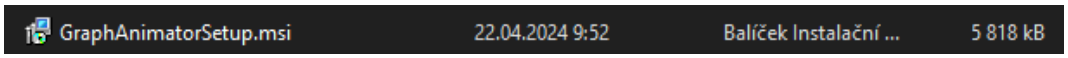
Obrázek 2.1: Grafové znázornění struktury programu

3. Uživatelská dokumentace

Než se podíváme na implementaci, bylo by vhodné se podívat, jak program funguje. V této kapitole se tedy podíváme na instalaci programu na Windows, jak vypadá uživatelské rozhraní, v jakém formátu očekáváme grafy a jak vypadá jazyk Cb.

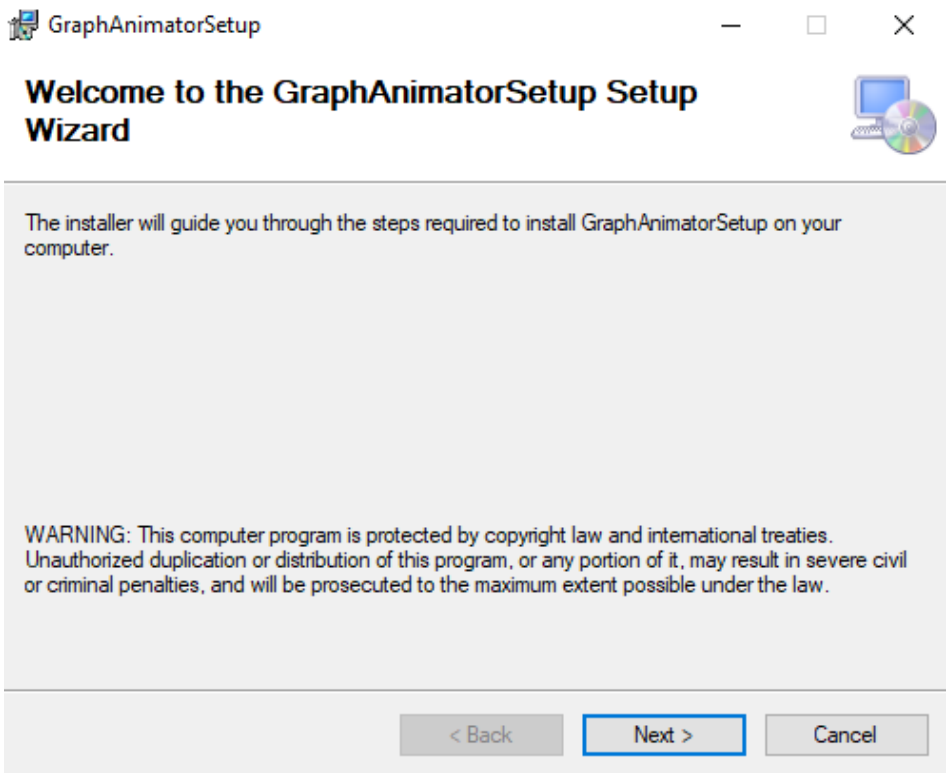
3.1 Instalace

V této kapitole si ukážeme instalaci programu. Instalace je krátká a jednoduchá. Prvním krokem je spuštění instalačního programu.



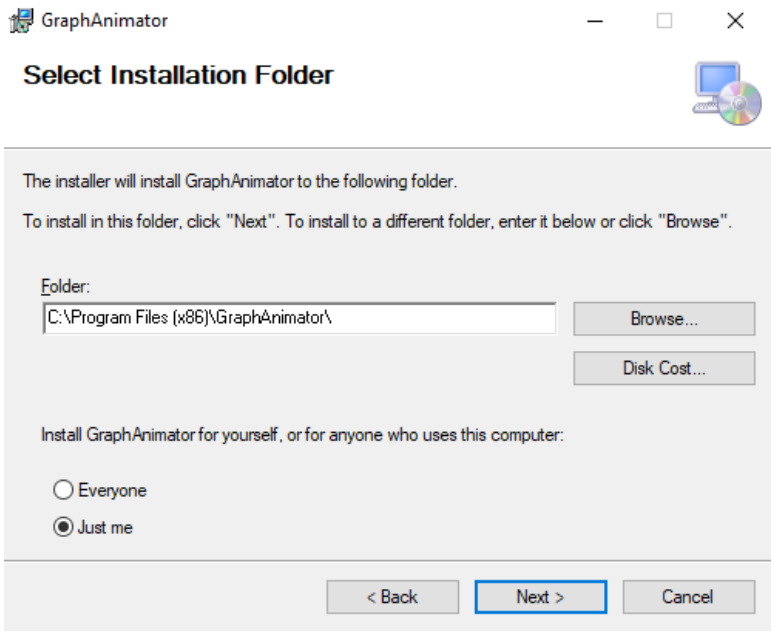
Obrázek 3.1: Instalační program

Při spuštění se objeví okno níže. Můžete stisknout *Next*.



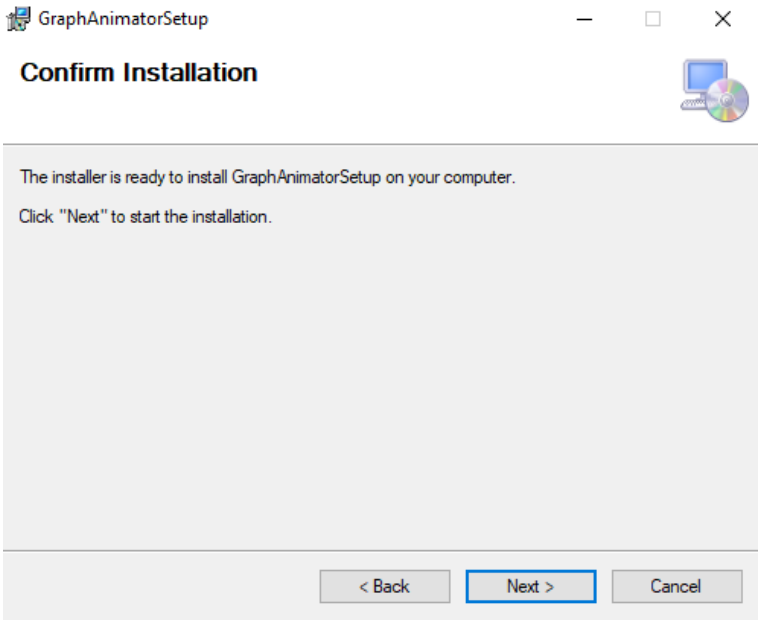
Obrázek 3.2: Uživatelské rozhraní pro instalaci

Poté budeme chtít zvolit složku, kam se má program nainstalovat. Po zvolení složky stiskněte *Next*.



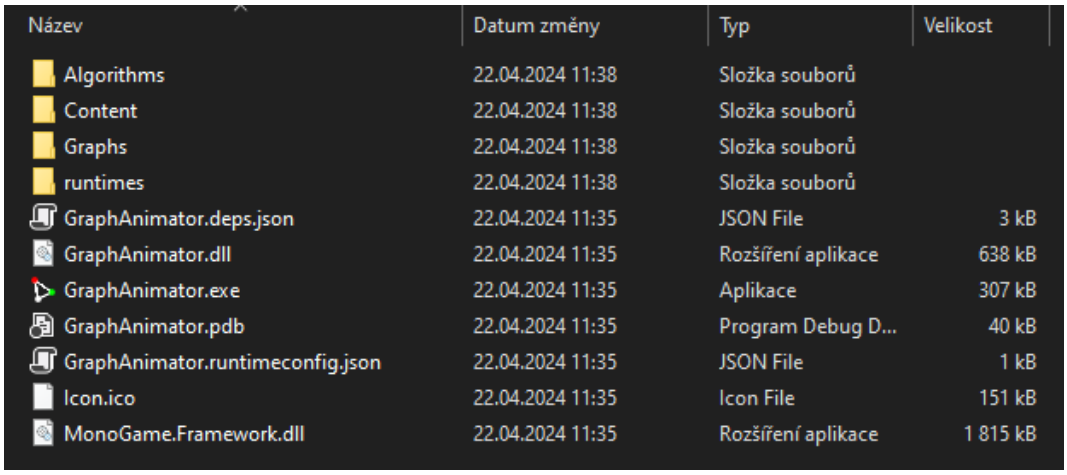
Obrázek 3.3: Výběr destinace instalace

Nakonec stačí potvrdit instalaci opětovným stisknutím tlačítak *Next*.



Obrázek 3.4: Potvrzení instalace

Složka, kde je program nainstalovaný, by měla vypadat takto:

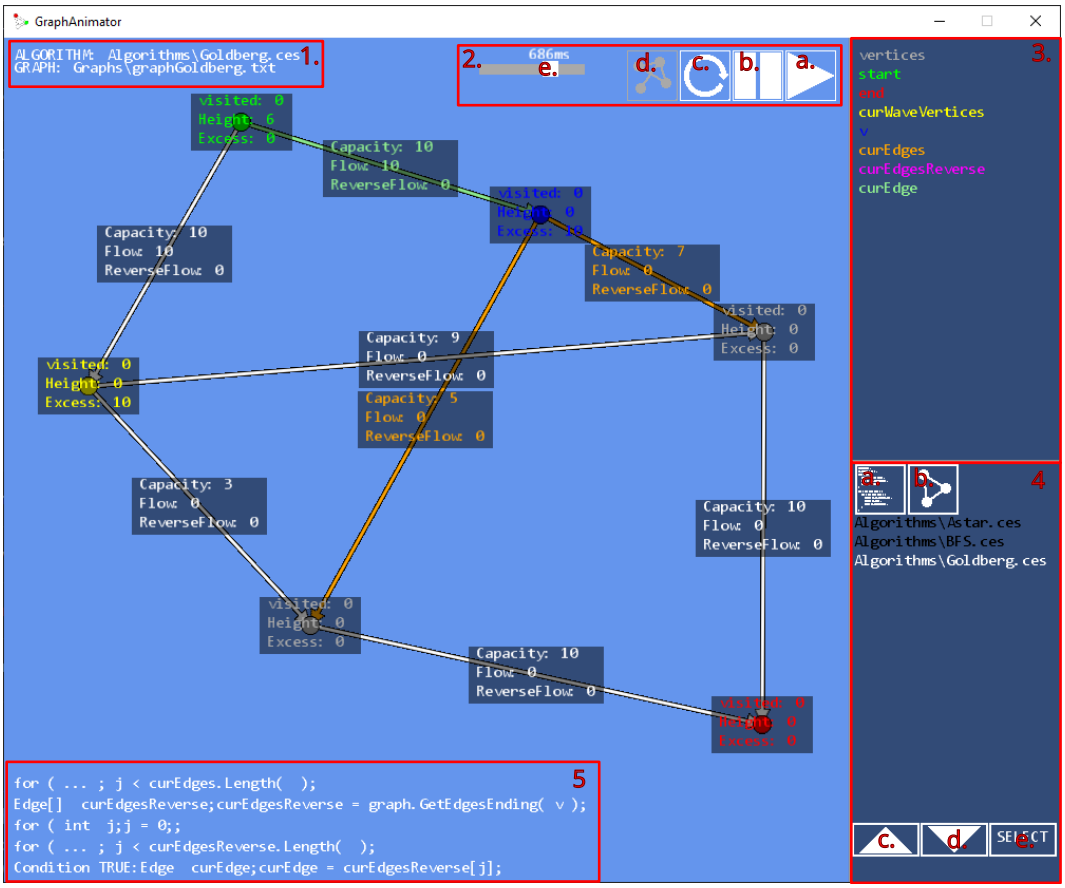


Obrázek 3.5: Složka s úspěšnou instalací

3.2 Uživatelské rozhraní

Rozeberme si uživatelské rozhraní 3.6 a jak se s ním pracuje. Podíváme se tedy na tlačítka a textové prvky uživatelského rozhraní

1. Aktivní algoritmus a graf. Graf i algoritmus jsou popsány názvem souboru.
2. Tlačítka pro ovládání běhu algoritmu.
   (a) Tlačítko **Spustit** spustí algoritmus.
   (b) Tlačítko **Pozastavit** pozastaví algoritmus.
   (c) Tlačítko **Restartovat** zastaví běh algoritmu, pokud algoritmus běží, a vrátí graf do počátečního stavu.
   (d) Tlačítko **Nové rozložení** vytvoří nové rozložení grafu.
   (e) Posuvník **Rychlost** mění rychlost běhu algoritmu.
3. Proměnné vrcholů a hran a barvy jim přiřazené. Proměnné jsou definované v algoritmu.
4. Ovládání pro čtení algoritmů a grafů, obsahuje seznam souborů grafů nebo algoritmů podle volby uživatele.
   (a) Tlačítko **Algoritmy** nastaví seznam souborů na seznam dostupných algoritmů.
   (b) Tlačítko **Grafy** nastaví seznam souborů na seznam dostupných grafů.
   (c) Tlačítko **Nahoru/Předchozí** označí předchozí soubor.
   (d) Tlačítko **Dolů/Další** označí následující soubor.
   (e) Tlačítko **Zvolit** zvolí označený soubor, přečte a načte graf/algoritmus.



Obrázek 3.6: Snímek uživatelského rozhraní

5. Právě provedená sekce kódu. Tento prostor je dále určen pro chybové hlášení při běhu algoritmu či čtení grafů a algoritmů.

Tlačítka se mohou stát neaktivní, tedy na ně nebude moci kliknout. Tato tlačítka jsou označena zešednutím. Tlačítka mohou být blokována např. běžícím algoritmem.

3.3 Grafy

Při čtení grafu očekáváme nějaký formát. Tento formát si rozebereme v této kapitole.

V našem souboru 3.7 očekáváme označení, zda je graf orientovaný nebo neorientovaný (*DIRECTED* nebo *UNDIRECTED*). Tento parametr je volitelný, jeho nenapsání však označí graf jako neorientovaný.

Dále máme definici vrcholů. Očekáváme tedy *Vertices* a *počet_vrcholů*.

Poté očekáváme názvy parametrů/polí vrcholů a jejich hodnoty. Parametry jsou na jedné řádce a v následujících řádcích se nachází *číslo_vrcholu* a poté hodnoty parametrů v pořadím definovaném výše. Pokud si nežádáme žádné parametry, můžeme tyto parametry přeskočit. Vrcholy vždy mají parametr *visited*.

Nakonec očekáváme hrany *Edges*. Na stejném řádku se mohou nacházet parametry hrany. Dále očekáváme na každém řádku *číslo_vrcholu1* (počáteční vrchol), *číslo_vrcholu2* (koncový vrchol) a nakonec očekáváme hodnoty parametrů

```
DIRECTED/UNDIRECTED

Vertices #
param1 param2 ...
0 value1 value2 ...
1 value1 value2 ...
2 value1 value2 ...
...

Edges param1 param2 ...
vertex1 vertex2 value1 value2 ...
vertex1 vertex2 value1 value2 ...
vertex1 vertex2 value1 value2 ...
...
```

Obrázek 3.7: Formát souboru s grafem

ve stejném pořadí jako byly definované dříve.

Pro vrcholy a hrany platí, že názvy parametrů jsou slova (text), kde každý parametr je oddělen mezerou a hodnoty jsou čísla také oddělená od sebe mezerou.

Čtení končí koncem souboru. Pokud není dodržen formát souboru, zahlásíme chybu, ukončíme čtení a vrátíme prázdný graf. Pokud uživatel měl nějaký graf načtený a pokusí se načíst chybný graf, ponecháme v okně graf již načtený.

Po načtení vznikne na obrazovce nákres grafu. Uživatel má možnost přesouvat vrcholy kliknutím a táhnutím. Pokud ani přetažení vrcholů nezachrání nákres grafu, je zde vždy možnost generovat nový nákres pomocí tlačítka *Nové rozložení* 2d.

3.4 Jazyk Cb

Jazyk Cb (čteme *ces*) je proprietární jazyk vytvořený pro tento program. Jazyk je lehce inspirován jazykem C#. Zde si projdeme základy a konstrukty tohoto jazyka, včetně seznamu typů a funkcí. Hlavní účel jazyka Cb je konstrukce grafových algoritmů. V této kapitole se budeme dívat na stavební bloky jazyka Cb. Pro lepší porozumění je doporučeno projít si soubory s algoritmy, se kterými program přichází.

3.5 Typy proměnných

V jazyce Cb máme 5 typů proměnných:

- **int**: celočíselný typ.
- **bool**: booleovský typ, který může nabývat hodnot *true/false*.
- **Vertex**: třída pro vrcholy grafu.
- **Edge**: třída pro hrany grafu.

- **Graph**: třída pro reprezentaci grafu.

Inicializace pak vypadá takto:

```
int number = 26;
bool condition = false;
Vertex vertex = ... ;
Edge edge = ... ;
```

Je nutné podotknout, že proměnná typu **Graph** s názvem *graph* je vytvořena při čtení souboru s grafem. Uživatel nemá možnost novou proměnnou typu **Graph** vytvořit.

Pro typy **Vertex** a **Edge** nelze vytvořit nové instance. Lze se ale odkázat na již existující instance, které se nacházejí v grafu (viz. podkapitola *Seznam metod*).

Dále zde máme speciální typ **List**. Prázdný **List** celočíselných proměnných se definuje takto:

```
int[] numbers = [];
```

List lze vytvořit pro libovolný typ kromě typu **Graph**. Pro přístup do listu lze použít index.

```
int number = number[3]; #index musí být validní,
                        #tedy větší než -1 a menší než délka
```

Dále každé proměnné je určen blok působnosti (definován {}). Na proměnnou nelze odkazovat před jejím vytvořením a nebo po skončení bloku, ve kterém byla vytvořena. Pokud proměnná nebyla vytvořena v nějakém bloku, má globální působnost a lze se na ni odkázat kdekoliv v souboru.

3.5.1 Vertex a Edge

Pro typy **Vertex** a **Edge** existuje operátor '.', tento operátor umožňuje přístup k polím, která byla definována v souboru s definicí grafu. Typ **Vertex** navíc obsahuje proměnou *visited*. Všechny tyto hodnoty jsou typu **int**.

3.6 Konstrukty

Jazyk definuje několik větvících a cyklických konstruktů:

- **if/else**: větvící konstrukt.
- **while**: cyklus, dokud platí podmínka, provádí blok kódu.
- **do-while**: cyklus, provádí blok kódu, dokud platí podmínka.
- **for**: cyklus, kde vytvoříme číselnou proměnnou, kterou můžeme po skončení bloku kódu iterovat. Před počátkem průběhu blokem kódu se provede kontrola podmínky.

- # značí komentář, který platí do konce řádku

Podívejme se tedy na příklady, jak tyto konstrukty vypadají v kódu.
If/else:

```
if(number < 3) #libovolná podmínka
{
    ... # je li podmínka pravdivá
}
else
{
    ... #není li podmínka pravdiví
}
```

If:

```
if(number < 3) #libovolná podmínka
{
    ... # je li podmínka pravdivá
}
```

While:

```
while(number < 3) #libovolná podmínka
{
    ... #obsah cyklu
}
```

Do-while:

```
do
{
    ... #obsah cyklu
} while (number < 3);
```

For:

```
for (int i = 0; i < 3; i = i + 1)
{
    ... #obsah cyklu
}
```

For cyklus je maličko složitější, pojďme si rozebrat obsah kulatých závorek. První je inicializace proměnné, která se provede jako první. Dále je podmínka, která se zkontroluje po inicializaci a po iteraci. Poslední je iterace, ta se provede po provedení následujícího bloku kódu. **For** cyklus typicky skončí nesplněním podmínky.

Pro cykly ale existují další konstrukty:

- **continue;**: přeskočí celý blok kódu a skočí na podmínku (nebo iteraci v případě **for** cyklu.
- **break;**: přeskočí celý blok kódu a ukončí cyklus.
- **return;**: ukončí celý algoritmus

3.7 Seznam operátorů

$=$	přiřazení
$\|\|$	nebo/disjunkce
$\&\&$	a/konjuknkce
$== !=$	rovnost
$< <= >= >$	porovnání
$+ -$	sčítání/odčítání
$* / \%$	krát/děleno/modulo
$! - unary$	negace/unární mínus
$[]$	přístup do pole

Obrázek 3.8: Priorita operátorů

- = očekává proměnnou vlevo a výraz vracející hodnotu stejného typu, jako je vlevo.
- ||, && očekávájí na obou stranách **bool** a vrací **bool**
- <, <=, >=, > očekávájí na obou stranách **int** a vrací **bool**
- +, -, *, /, % očekávají na obou stranách **int** a vrací **int**
- **!** očekává vpravo **bool** a vrací **bool**
- **-** (unární) očekává vpravo **int** a vrací **int**
- [ ] očekává vlevo název proměnné typu T, mezi závorkami očekává číselnou hodnotu a vrací hodnotu T

3.8 Seznam metod

Pro ukázku metod zde bude vyzobrazená proměnná daného typu.

3.8.1 Graph

```
graph.GetVertices(); vrací Vertex[] seznam vrcholů v grafu
graph.GetStart(); vrací Vertex[] seznam počátečních vrcholů v grafu
graph.GetEnd(); vrací Vertex[] seznam koncových vrcholů v grafu
graph.GetEdges(); vracé Edge[] sezanm hran
graph.GetNeighbours(Vertex v); vrátí Vertex[] seznam vrcholů sousedící
                               s vrcholem v
graph.IsStart(Vertex v); vrátí bool zda je vrchol počáteční nebo ne
graph.IsEnd(Vertex v); vrátí bool zda je vrchol koncový nebo ne
graph.GetEdgesEnding(Vertex v); vrátí Edge[] seznam hran končící
                                vrcholem v
```

3.8.2 Edge

```
edge.GetStart(); vrací Vertex počátek hrany
edge.GetEnd(); vrací Vertex konec hrany
edge.GetOther(Vertex v); vrací Vertex druhý vrchol hrany
                         (tedy jiný než v)
```

3.8.3 Vertex

```
vertex.SetPrev(Vertex v); nevrátí nic, nastaví předchozí vrchol
                           (vhodný pro cesty)
vertex.GetPrev(); vrací Vertex předchozí vrchol
vertex.HasPrev(); vrací bool zda má vrchol předchozí vrchol
```

3.8.4 List

List funguje na všech typech proměnných, tedy máme implementovanou nějakou generiku. Typem T označíme všechny typy najednou. Platí, že do listu typu T můžeme vkládat pouze hodnoty typu T a získat z něho pouze hodnoty typu T (např. int[] seznam čísel obsahuje pouze číselné hodnoty int).

```
list.Append(T t); nevrací nic, přidá hodnotu t na konec listu
list.Push(T t); nevrací nic, přidá hodnotu t na konec listu
list.Enqueue(T t); nevrací nic, přidá hodnotu t na konec listu
list.Sort(string s); nevrací nic, setřídí hodnoty podle názvu pole
                     (určeno pro Vertex[] a Edge[])
list.Sort(); nevrací nic, setřídí hodnoty (určeno pro int[])
list.Pop(); vrací T hodnotu na konci listu
list.Deque(); vrací T hodnotu na počátku listu
list.IsEmpty(); vrací bool zda je list prázdný
list.Length(); vrací int délku listu
```

3.9 Ukázky

Nakonec si v této sekci ukážeme pár příkladů kódu pro sestrojení částí algoritmů.

Začněme iterování listem 3.9. V tomto případě vezmeme seznam vrcholů v grafu a všechny nastavíme na navštívené, tedy nastavíme pole *visited* na 1.

```
Vertex[] vertices = graph.GetVertices();
for (int i = 0; i < vertices.Length(); i = i + 1)
{
    Vertex curV = vertices[i];
    curV.visited = 1;
}
```

Obrázek 3.9: Ukázka iterování listem

Zde 3.10 vidíme ukázku použití *Listu* jako fronty (například pro *Prohledávání do šířky*). V tomto případě máme frontu vrcholů, kde začneme počátkem grafu. Poté vezmeme vrchol z fronty, najdeme jeho sousedy a ty dáme do fronty.

Je vhodné podotknout, že běh programu neskončí, jelikož nekontrolujeme navštívené vrcholy. Zajištění, abychom nenavštěvovali navštívené vrcholy, je čtenáři ponecháno jako cvičení (nápověda: vrcholy mají pole *visited*).

```
Vertex[] queue = graph.GetStart();
while(!queue.IsEmpty())
{
    Vertex current = queue.Dequeue();
    Vertex[] neighbours = graph.GetNeighbours(current);
    int size = neighbours.Length();
# Add neighbours to queue
    for (int i = 0; i < size; i = i + 1)
    {
        Vertex next = neighbours[i];
        queue.Enqueue(neighbours[i]);
    }
}
```

Obrázek 3.10: Ukázka fronty

Dále si ukážeme použití *Listu* jako zásobníku (například pro *Prohledávání do hloubky*) 3.11. Kód je velmi podobný kódu fronty 3.10, akorát jsme vyměnili funkce *Enqueue()* a *Dequeue()* za *Push()* a *Pop()*.

Opět platí, že tento kód zásobníku neskončí ze stejného důvodu, jako kód fronty.

```
Vertex[] stack = graph.GetStart();
while(!stack.IsEmpty())
{
    Vertex current = stack.Pop();
    Vertex[] neighbours = graph.GetNeighbours(current);
    int size = neighbours.Length();
# Add neighbours to stack
    for (int i = 0; i < size; i = i + 1)
    {
        Vertex next = neighbours[i];
        stack.Push(neighbours[i]);
    }
}
```

Obrázek 3.11: Ukázka zásobníku

Pokud budeme chtít sestrojit prioritní frontu (například pro Dijkstrův algoritmus), tedy frontu vrcholů setříděnou podle nějakého pole, pak můžeme použít frontu a tu poté setřídit podle daného parametru. Toto vidíme v kódu níže 3.12.

Je vhodné podotknout, že parametr, podle kterého třídíme, musí být v definici grafu, jinak přistupujeme k neznámé proměnné, a tedy nastane výjimka.

```
Vertex[] queue = graph.GetStart();
while(!queue.IsEmpty())
{
    Vertex current = queue.Dequeue();
    Vertex[] neighbours = graph.GetNeighbours(current);
    int size = neighbours.Length();
# Add neighbours to queue
    for (int i = 0; i < size; i = i + 1)
    {
        Vertex next = neighbours[i];
        queue.Enqueue(neighbours[i]);
        # Vertex must contain fields 'distance' and 'cost'.
        next.distance = current.distance + next.cost;
    }
    queue.Sort("distance");
}
```

Obrázek 3.12: Ukázka prioritní fronty

Pro všechny ukázky kódu platí, že jejich chování je na grafu vidět, je-li algoritmus spuštěn, jelikož máme *List* vrcholů či samostatné proměnné vrcholů *Vertex*. Tyto proměnné jsou pak barevně označeny (jak bylo popsáno v podkapitole o uživatelském rozhraní 3).

4. MonoGame

4.1 Úvod

Jak je psáno v Capellman a Salin (2020), MonoGame je *open source* framework pro C# primárně dělaný pro vývoj her. Tento framework je multiplatformní a funguje na operačních systémech pro počítače a mobilní telefony, a MonoGame funguje dokonce i na konzolích.

MonoGame poskytuje hlavní herní smyčku a v ní metody *Update()* a *Draw()*. *Update()* slouží ke čtení vstupu hráče a následné aktualizaci hry a herní logiky a *Draw()* slouží ke kreslení do herního okna. Zároveň MonoGame poskytuje způsob nahrání herního obsahu, např. obrázků, 3D modelů, zvuků a videí. Součástí monogame je i aplikace *MonoGame pipeline app*, která pomáhá při vkládání herního obsahu do hry.

4.2 Použití

Hlavní třídou frameworku monogame je třída *Game*, ve které máme přístup k již zmíněným metodám *Update()* a *Draw()*. Dále zde máme metody *Initialize()* a *LoadContent()* (viz. MonoGame Team). Tyto metody tvoří základ herní logiky a poslouží nám jako kostra pro vývoj aplikace.

Další důležité třídy, které budeme používat, jsou *GraphicsDeviceManager* a *SpriteBatch*. *GraphicsDeviceManager* slouží k inicializaci a prezentování herního okna. *SpriteBatch* pak slouží k samotnému kreslení obrázků a textu.

Implementace nekonečné smyčky za standardních podmínek není obtížná. Nejtěžší na dělání her je dle mého názoru kreslení, obzvláště pokud člověk pracuje např. s OpenGL. MonoGame toto kreslení značně zjednodušuje. Pokud chce uživatel něco nakreslit, stačí, aby nahrál obrázek na začátku spuštění aplikace a pak určil, kde se má tento obrázek nakreslit. Narozdíl od OpenGL, kde obraz na okno se mapuje do ([-1, 1], [-1, 1]) (výška, šířka), v MonoGame říkáme pixely, kam chceme obrázek nakreslit. Pro mě je přístup MonoGame příjemnější, minimálně pro jednoduché hry či aplikace, kde nehýbeme s kamerou.

```
void Main()
{
    Initialize();
    LoadContent();
    while(...) //nekonečná smyčka, dokud nevyžádáme konec
    {
        ReadInput();
        Update();
        Draw();
    }
}
```

Obrázek 4.1: Zjednodušení herní smyčky

Přidávání herního obsahu je také jednoduché, pokud mluvíme např. o přidávání obrázků (jiný herní obsah, jako jsou např. zvuky, nejsou součástí programu). Pro tuto aplikaci jsem vytvořil několik malých obrázků (*sprites*), které poté vykreslujeme. Pro kreslení vrcholů je nutno pouze určit místo vykreslení. Kreslení hran je maličko obtížnější, jelikož musíme určit, kde by byl střed hrany, škálování délky hrany a rotaci. Když jsme toto schopni vypočítat, můžeme parametry dát do metody *spriteBatch.Draw()* a o více se starat nemusíme.

4.3 Proč MonoGame?

MonoGame jsem zvolil z následujících důvodů: jelikož se jedná o interaktivní aplikaci, kterou chceme vypnout, až když uživatel řekne, pak implementace nekonečné smyčky je první zjednodušení. Dále jednoduchost nahrávání a vykreslování obsahu bylo pro mě také velké plus při výběru, v čem budu tuto aplikaci dělat.

Zároveň by bylo vhodné podotknout, že logiku v *Update()* a *Draw()* si do určité míry implementujeme sami. Tedy víme, v jakém pořadí se prvky vykreslují a aktualizují. Pokud toto porovnám s herním enginem jako je Unity, tak Unity nám dá možnost pro každý objekt napsat metodu *Update()*, ale v jakém pořadí jsou tyto metody volány my nevíme. Což může způsobit mírný nedeterminismus chování.

Nakonec bych rád zmínil, že ve frameworku MonoGame jsem již pár projektů dělal a mám tedy pozitivní zkušenost. Zároveň idea fameworku mi vyhovuje, jelikož mi stačí implementovat logiku aplikace a MonoGame obstará ty obtížnější věci, jako je vykreslování na obrazovku.

5. Kompilátor

5.1 Úvod

Jak je psáno v Aho a kol. (2006), programovací jazyky jsou pro člověka způsob, jak napsat program spustitelný na počítači či jiném stroji. Takový jazyk je pro člověka typicky čitelný, ale tento jazyk musí být převeden do formy, které bude rozumět stroj spouštějící tento program.

Při převodu z jednoho jazyka do druhého provede kompilátor několik kroků. Prvním krokem v našem překladu je lexikální anlýza. V tomto kroku kompilátor čte zdrojový kód a uspořádává znaky do lexémů.

Tyto lexémy pak v druhém kroku, syntaktické analýze, převede kompilátor na syntaktický strom. Tento strom pak představuje strukturu programu, kde každý vrchol je nějaká operace.

Dalším krokem kompilátoru je sémantická analýza, kde provádíme kontrolu konzistence s jazykem a také kontrolu typů. Např. přiřazení *bool* hodnoty do *int* proměnné by vyvolalo výjimku a ukončilo kompilaci.

V tomto kroku my končíme. Vytvoříme syntaktický strom a s tím následně pracujeme, když spouštíme uživatelem napsané algoritmy. Za standardních podmínek při kompilaci poté následuje generování mezikódu, jeho optimalizace a nakonec generujeme výsledný kód.

Při tvorbě tohoto kompilátoru vycházím stále z knihy od Aho a kol. (2006). V dalších sekcích této kapitoly se budeme bavit o jazyku Cb a o tvorbě syntaktického stromu.

5.2 Jazyk Cb

Aby uživatel mohl psát algoritmy, které program bude dále zpracovávat, musí mít uživatel nějaký způsob popisu toho algoritmu. Proto jsme vytvořili jazyk Cb. Jedná se o bezkontextový jazyk inspirovaný jazyky jako je C#, ale je o něco primitivnější.

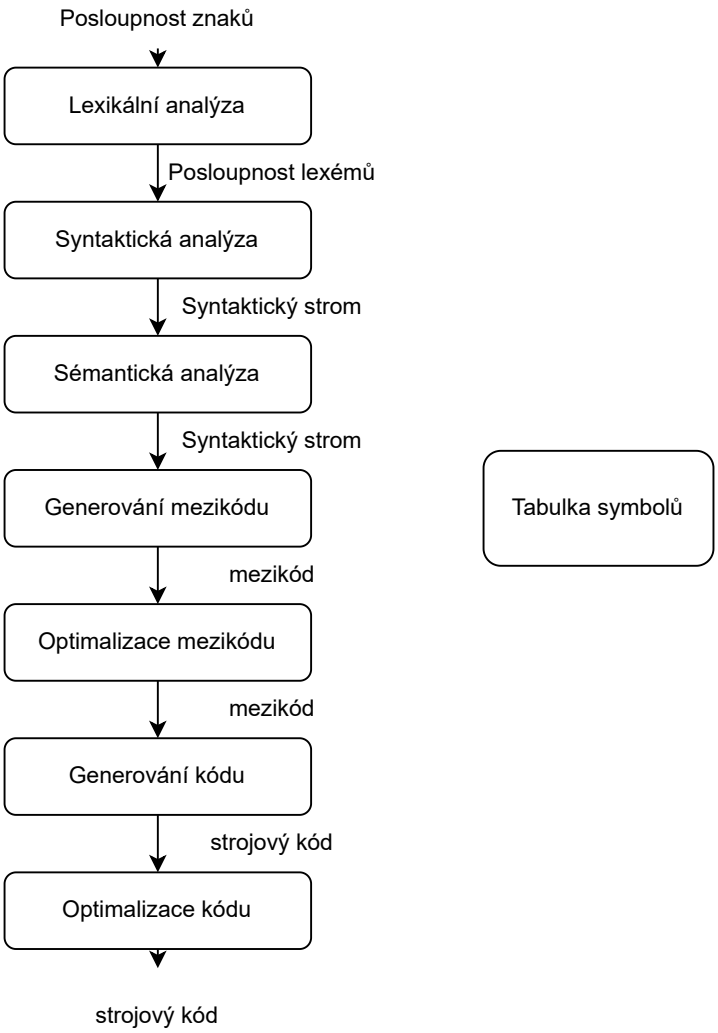
Při tvorbě jazyka bylo pro nás důležité zanechat základní konstrukty větvení (*if-else*) nebo cyklů (*for*, *while*, *do-while*). Dále jsou důležité proměnné, do kterých uživatel může zapisovat, a samozřejmě operátory a jejich priority, aby uživatel mohl vytvářet matematické výrazy.

Proměnných je pět typů: **int**, **bool**, **Vertex**, **Edge** a **Graph**. Graph je typ, který uživatel nemůže vytvořit v algoritmu samotném, ale jedna proměnná tohoto typu je vždy dána programem, když se přečte popis grafu ze souboru.

Dále máme definované listy konstruktem **type**[], kde **type** je jeden z dostupných typů (opět platí, že Graph není povolený typ). Listy může uživatel definovat prázdné a nebo je zkopírovat, např. funkce Vertex.GetNeighbours() vrací list vrcholů sousedících s vrcholem volající funkci (funkce vrací Vertex[]).

Tedy máme nějaké typy a některé typy mají definované funkce a nebo pole. Jazyku bych ovšem neříkal objektově orientovaný. Uživatel nemá možnost vytvářet nové typy a ani nemůže definovat nové funkce.

Vraťme se ale k proměnným. Každá proměnná má totiž nějaký rozsah. Pokud je proměnná v nějakém bloku, kde blok je definován složenými závorkami {}, které



Obrázek 5.1: Diagram kompilátoru a jeho kroků

jsou součástí konstruktů jako jsou *if* nebo *while*, pak má proměnná lokální rozsah a tedy životnost pouze do konce bloku. Proměnné definované mimo libovolný blok jsou pak globální proměnné a ty žijí do konce programu.

Bloky nám tedy tvoří úrovně. Nejvyšší úroveň je globální a bloky tedy zvyšují lokalitu proměnných. Pokud chceme vytvořit proměnnou, stejnojmenná proměnná se nesmí nacházet ve vyšší úrovni v daný moment. Toto je podobné (pokud ne stejné) proměnným v jazyce C#. Ale jak je psáno v Aho a kol. (2006), v C++ má proměná při její definici přiřazen blok, ve kterém byla definována. Pokud pak přistupujeme k proměnné nějakého jména, a takových proměnných je definováno více, pak vezmeme proměnnou definovanou v nejhlubším bloku. Toto mi přijde maličko matoucí a proto jsem se rozhodl o rozsah proměných jako je v C#. Tedy rozsah proměnných je statický.

$$
\begin{aligned}
stmts &\rightarrow stmts\ stmt \\
&\mid \epsilon \\
\\
stmt &\rightarrow \text{if } (or) \text{ \{ } stmts \text{ \} } else \text{ \{ } stmts \text{ \} } \\
&\mid \text{if } (or) \text{ \{ } stmts \text{ \} } \\
&\mid \text{while } (or) \text{ \{ } stmts \text{ \} } \\
&\mid \text{do \{ } stmts \text{ \} while (} or \text{);} \\
&\mid \text{for (} init \text{ ; } or \text{ ; } assign \text{) \{ } stmts \text{ \} } \\
&\mid \text{continue;} \\
&\mid \text{break;} \\
&\mid \text{return;} \\
&\mid createVar; \\
&\mid assign; \\
&\mid var.chars(arguments); \\
&\mid var.chars(); \\
\\
createVar &\rightarrow \text{int } assign \\
&\mid \text{bool } assign \\
&\mid \text{Vertex } assign \\
&\mid \text{Edge } assign \\
&\mid \text{int[] } assignList \\
&\mid \text{bool[] } assignList \\
&\mid \text{Vertex[] } assignList \\
&\mid \text{Edge[] } assignList \\
\\
assignList &\rightarrow var = [] \\
&\mid var = var \\
\\
assign &\rightarrow var\ =\ or \\
&\mid var[factor]\ =\ or
\end{aligned}
$$

Obrázek 5.2: Gramatika jazyka Cb, část 1.

$$
\begin{aligned}
or &\rightarrow and \;||\; or \\
&\mid and \\
\\
and &\rightarrow equality \;\&\&\; and \\
&\mid equality \\
\\
equality &\rightarrow compare \;==\; equality \\
&\mid compare \;!=\; equality \\
&\mid compare \\
\\
compare &\rightarrow add \;<\; compare \\
&\mid add \;<=\; compare \\
&\mid add \;>\; compare \\
&\mid add \;>=\; compare \\
&\mid add \\
\\
add &\rightarrow term \;+\; add \\
&\mid term \;-\; add \\
&\mid term \\
\\
term &\rightarrow factor \;*\; term \\
&\mid factor \;/\; term \\
&\mid factor \;\%\; term \\
&\mid factor \\
\\
factor &\rightarrow (\; or \;) \\
&\mid \;-\, number \\
&\mid number \\
&\mid \;-\, var \\
&\mid !var \\
&\mid var \\
&\mid string \\
&\mid \text{true} \\
&\mid \text{false}
\end{aligned}
$$

Obrázek 5.3: Gramatika jazyka Cb, část 2.

$$
\begin{aligned}
var &\rightarrow chars \\
&\mid chars.chars \\
&\mid chars.chars(arguments) \\
&\mid chars.chars() \\
&\mid chars[factor]
\end{aligned}
$$

$$
\begin{aligned}
arguments &\rightarrow arguments, or \\
&\mid or
\end{aligned}
$$

$$
string \rightarrow "chars"
$$

$$
\begin{aligned}
chars &\rightarrow chars\ char \\
&\mid chars\ digit \\
&\mid char
\end{aligned}
$$

$$
\begin{aligned}
char &\rightarrow \text{a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z} \\
&\mid \text{A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z}
\end{aligned}
$$

$$
\begin{aligned}
number &\rightarrow number\ digit \\
&\mid digit
\end{aligned}
$$

$$
digit \rightarrow \text{0|1|2|3|4|5|6|7|8|9}
$$

Obrázek 5.4: Gramatika jazyka Cb, část 3.

5.3 Konstrukce kompilátoru

Jak bylo psáno v úvodu 5.1 a návrhu programu, kompilátor je implementován v jazyce C# a jak bylo psáno v úvodu této kapitoly, výstupem je syntaktický strom.

Syntaktický strom stavíme v průběhu lexikální analýzy, jelikož gramatika nám určuje, co můžeme očekávat a pokud se nám naše očekávání neobjeví, vyhodíme výjimku, jelikož napsaný algoritmus nedodržuje gramatiku určenou jazykem Cb. Sémantická analýza se pak provádí během konstrukce syntaktického stromu, neboť operátory očekávají typ, se kterým pracují.

Pojďme se ale podívat na implementaci samotné gramatiky (viz. obrázky 5.25.35.4). Začněme od nejprimitivnějších prvků, čísel (*number*) a slov (*chars*). v obou případech je to čtení znaků ve while cyklu dokud nenarazíme na nečíselný nebo nealfanumerický znak.

Var označuje proměnnou a volání funkce, přístup indexem nebo přístup k poli.

Factor pak vybírá mezi proměnnou, hodnotou (včetně bool hodnot), unární negací a závorkou, kde uvnitř je další výraz.

Dále máme v gramatice binární operátory seřazené podle jejich priority. Podívejme se na jeden z nich, např. **term**, ostatní fungují obdobně. Způsob napsání

$$term \rightarrow factor * term$$

by implikovalo použití rekurze. Zprvu jsme to tak implementovali, avšak pro jednoduchost a zamezení problému s přetečením zásobníku byla rekurze změněna na **while** cyklus ze kterého vystoupíme, když nemáme vhodný symbol operátoru (například pro náš případ to jsou operátory * (krát), / (děleno), a % (modulo)).

Poté máme **assign** (přiřazení), které očekává proměnnou a hodnotu, kterou do ní zapsat. To může být nějaký matematický výraz, proměnná, ze které hodnotu bereme a nebo hodnota samotná. Pro **assignList** (přiřazení do listu) očekáváme proměnnou stejného typu nebo prázdný list, který je označen hranatými závorkami [].

Vytvoření proměnné (**createVar**) je pak přiřazení, akorát očekáváme před přiřazením typ.

Nakonec máme konstrukty **if**, **while**, **do-while**, **for**. Který konstrukt nebo inicializace proměnné se vybere je určeno přečtením prvního slova a následně *switch* provede výběr. S neznámým slovem se zachází jako s proměnnou a tedy očekáváme přiřazení do proměnné nebo volání metody či funkce. Pokud se jedná o funkci, návratová hodnota je ignorována.

Klíčová slova jsou uložená v tabulce. Krom identifikace slouží tabulka k tomu, aby klíčové slovo nebylo použito jako proměnná. Proměnné pak mají vlastní tabulku. Tabulka je ve skutečnosti *List slovníků*. Slovník nám pro název proměnné určuje její typ. List slovníků jsme ale zvolili kvůli rozsahu a životnosti proměnných. Jelikož každá proměnná má nějaký rozsah určený blokem, ve kterém byla vytvořena, potřebovali jsme zajistit, aby se nepřistupovalo k proměnným, které ještě nebo již neexistují. Tedy pokud je vytvořen nový blok (a tedy nová hloubka), vytvoříme nový slovník v listu slovníků. Pokud blok skončí, pak náležící (resp. poslední) slovník smažeme. Tedy replikujeme životnost proměnných během kompilace stejně jako by životnost proměnných vypadala během běhu programu.

Nakonec máme ještě tabulky pro funkce a metody pro typy *Vertex*, *Edge* a *Graph*, kde tabulky jsou slovníky, kde klíč slovníku je název funkce a hodnota je typ návratové hodnoty (včetně *void*).

5.4 Konstrukce syntaktického stromu

Pro každý neterminál máme funkci, která gramatiku implementuje. Každá funkce pak vrací uzel (*Node*), který popisuje daný operátor či konstrukt. Tyto uzly jsou pak sestavené do syntaktického stromu. Příkazy (*stmt*) jsou pak spojeny pomocí speciálního uzlu *Sequence*.

Každý uzel má definovaný interface *INode<T>*, kde *T* je návratový typ funkce *ReturnResult()*. *ReturnResult()* je implementace funkčnosti operátorů a konstruktů. Pro binární operátory je návratová hodnota triviální, násobení a sčítání vrací *int*, porovnání a logické *AND* a *OR* vrací *bool*, přístup do listu pomocí [] vrátí jeden prvek daného typu a podobně.

Pro běh programu musel být vytvořen nový typ: *BreakType*, který může dosáhnout čtyř hodnot: *Standard*, *Continue*, *Break* a *Return*. Tento typ nám pomáhá implementovat logiku konstruktů *continue*, *break* a *return*. Pokud se nám někde ocitne *break*, pak zastavíme, co děláme, *break* propagujeme, dokud nenarazíme na nějaký cyklus a ten cyklus ukončíme. *Continue* a *return* fungují podobně, akorát *continue* přeskočí na novou iteraci cyklu a return ukončí celý algoritmus.

V diagramu 5.5 můžeme pak vidět uzly a děti a návratovou hodnotu funkce *ReturnResult()*. Děti mají očekávaný typ, který vrací jejich *ReturnResult()*. Tento typ pak pomáhá v sémantické části tvorby syntaktického stromu.

Je nutné podotknout, že diagram 5.5 je reprezentace syntatického stromu v programu, tedy je psaná v C#.

Rád bych ještě podotknul, že *for* cyklus lze napsat pomocí *while* cyklu. To lze udělat tak, že vytvoření proměné dáme před *while*, kam dáme podmínku je triviální a iteraci dáme na konec příkazů v následujícím bloku. Problém byl ale *continue*. Jelikož *continue* přeskočí na kontrolu podmínky ve *while* cyklu, nedostaneme se na iteraci a mohli bychom být nekonečně zaseknuti v cyklu. A tak dostal *for* vlastní uzel.

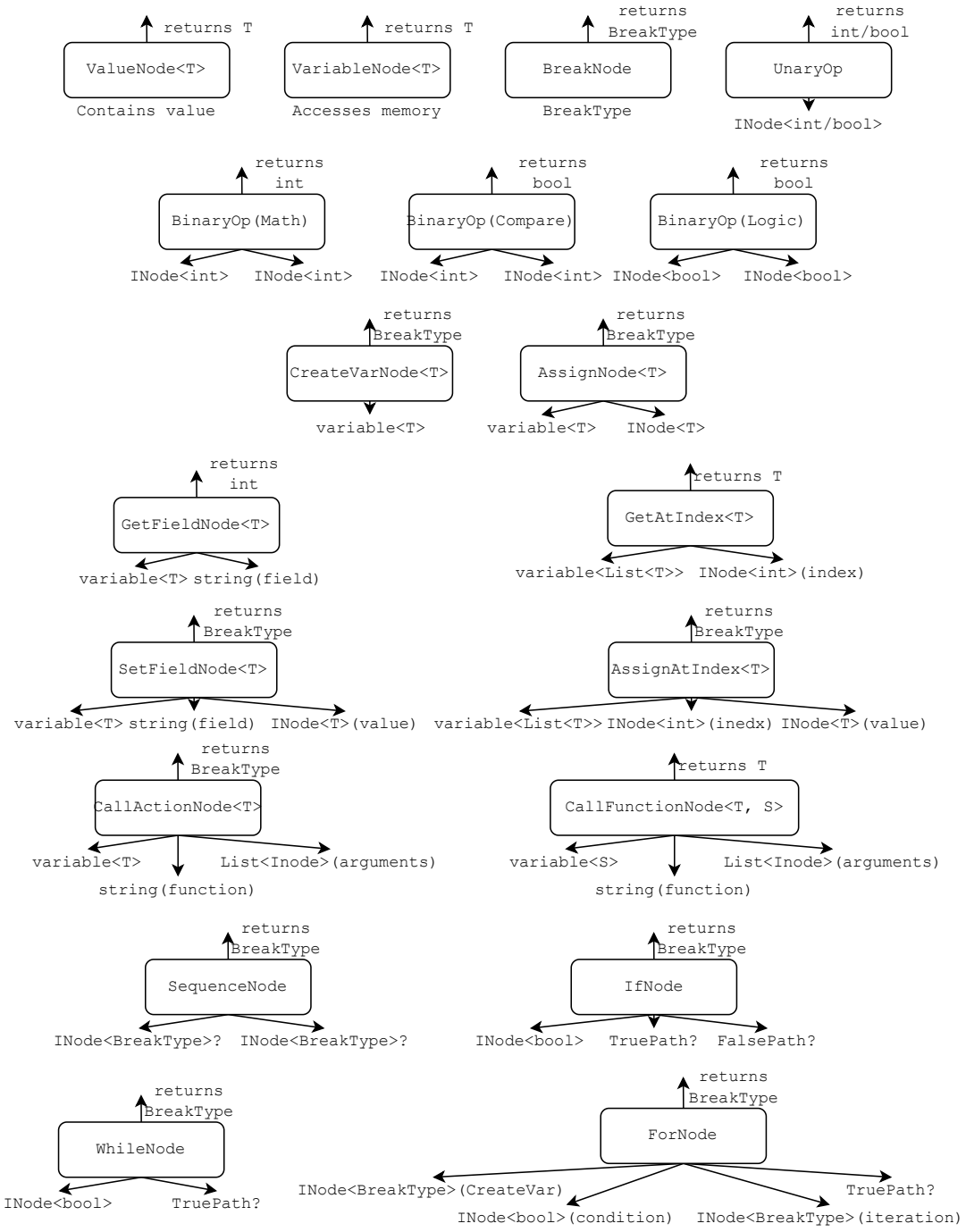
5.5 Průchod syntaktickým stromem

V této sekci se budeme bavit o průchodu syntaktickým stromem bez krokování. Je však vhodné podotkount, že chování uzlů je podobné pokud ne stejné, pokud porovnáme krokovaný a nekrokovaný běh.

Syntaktický strom popisuje strukturu programu jak byl zkompilován. Jelikož se jedná o strom, můžeme použít jednoduchý algoritmus *Depth First Search* pro průchod stromem. Jak přesně budeme procházet stromem určují uzly samotné a někdy i návratová hodnota z dítěte uzlu určuje průchod. Vše zmíněné v této kapitole bylo implementováno ve funkci *ReturnResult()*.

5.5.1 Pamět

Implementace paměti je v souboru Memory.cs.



Obrázek 5.5: Diagram uzlů syntaktického stromu

Před popisem jak fungují jednotlivé uzly si popíšeme jak funguje vnitřní paměť pro syntaktický strom.

Stejně jako program se píše v blocích, je paměť rozdělena do vrstev. Pokaždé, když začne nový blok, zažádáme paměť o novou vrstvu a když blok skončí, požádáme o odstranění poslední vrstvy. Samotné proměnné jsou pak uloženy ve slovníku, kde název proměnné je klíč a hodnota slovníku je wrapper pro hodnotu uloženou v paměti. Paměť je tedy vnitřně typu *List<Dictionaty<string, IMemoryBlock>>*, kde *IMemoryBlock* je již zmíněný wrapper pro hodnotu v paměti.

V paměti vždy může existovat jméno proměnné jednou. Některé programovací jazyky povolí pro každý blok proměnnou se stejným názvem, ale jak jsem již zmiňoval při popisu jazyka, toto může být matoucí. Pokud tedy existuje v paměti proměnná s nějakým názvem a uživatel chce přidat proměnnou se stejným názvem, vyhodíme výjimku.

5.5.2 Rozbor jednoduchých uzlů

Pojďme tedy rozebrat uzly 5.5 jednotlivě, počínaje od nejjednoduších.

ValueNode<T> má v sobě uloženou hodnotu. Typ hodnoty je určen typem T. BreakNode se chová podobně, akorát má uloženou hodnoty typu *BreakType*.

VariableNode<T> vezme z paměti hodnotu typu T podle jména proměnné. Pokud taková proměnná neexistuje, pak program vyhodí výjimku.

CreateVarNode<T> má jméno proměnné a podle tohoto jména a typu T požádá paměť o vytvoření proměnné.

Uzly **unárních operátorů (!, -)** vezmou z dítěte honotu a provedou na hodnotě změnu určenou oprátorem. Výslednou hodnotu poté vrátí.

Uzly **binárních operátorů (+, -, *, /, %, <, <=, =>, >, ==, !=, &&, ||)** vezmou hodnotu dětí a provedou operaci určenou operátorem a vrátí příslušnou hodnotu.

AssignNode<T> přiřadí proměnné v levém dítěti hodnotu podle pravého dítěte, resp. zažádá paměť o toto přiřazení. T je typ proměnné.

Uzly **GetAtIndex<T>** a **SetAtIndex** pracují na listech. Název třídy určuje, zda se jedná o *Getter* nebo *Setter*. T určuje typ proměné v listu a typ hodnoty, který se získá nebo se zapíše.

Uzly **GetFieldNode<T>** a **SetFieldNode<T>** pracují na proměnných typu *Vertex* a *Edge* (toto určuje T). Podle návrhu programu mají tyto třídy pole (*field*) typu *int*. Na tato pole se odkazujeme jejich jménem. *GetFieldNode<T>* získá hodnotu z pole a *SetFieldNode<T>* do daného pole zapíše hodnotu.

Další uzly jsou **CallActionNode<T>** a **CallFunctionNode<T, S>**. Začněme *CallActionNode<T>*. Tento uzel získá proměnnou typu T a pokusí se zavolat metodu podle jména funkce a podle seznamu proměnných. *CallFunctionNode<T, S>* získá proměnnou typu S a pokusí se zavolat funkci podle jména funkce a seznamu proměnných vracející hodnotu typu T. Rozdíl je tedy v tom, že *CallActionNode<T>* neočekává návratovou hodnotu metody a pokud se tam návratová hodnota nachází, tak ji ignoruje.

5.5.3 Rozbor řídicích uzlů

Další uzly jsou řídicí uzly a tedy určují průchod stromem a tedy běh algoritmu. Děti mohou být další sekvence nebo řídicí uzly. Některé uzly mají jako děti bloky kódu a tedy zažádají pamět o dočasné zvýšení úrovně. Výjimkou je *ForNode*, který zažádá o dvě úrovně, první pro vytvoření proměné na začátku *for* cyklu a druhou pro následující blok.

Začněme uzlem **Sequence**. Za standartních podmínek projde nejprve levým synem a následně projde pravým synem. Pokud však jedno z dětí vrátí nestandartní *BreakType* (tedy se objevilo *continue*, *break* či *return*), pak uzel ukončí co dělal a propaguje získaný *BreakType* rodiči.

Uzel **IfNode** má podmínku, výraz vracející *bool* a dvě sekvence: cesta, pokud podmínka byla pravdivá a cesta pro opačný případ. Tento uzel získá *BreakType* z cesty, která se provedla a tento *BreakType* pak propaguje rodiči.

WhileNode má opět podmínku a cestu, kterou jdeme, když je podmínka platná. Jako ve standartním *while* cyklu po ukončení cesty zkontrolujeme podmínku a pokud platí, vydáme se na cestu znovu. Toto ovšem může být přerušeno, pokud narazíme na *break* nebo *return* (continue ukončí průchod cestou, ale v iterování jinak pokračujeme). *Break* cyklus ukončí ale *while* stále vrací standardní *BreakType*. Pouze když získáme *return*, tak vracíme jiný *BreakType* (konkrétně *return*).

A nakonec máme uzel **ForNode**. Při vstupu do *for* cyklu provedeme tvorbu a přiřazení do proměnné. Poté zkontrolujeme podmínku. Pokud podmínka platí, vydáme se na cestu, tedy blok následující po *for* konstruktu. Po ukončení této cesty provedeme iteraci, zkontrolujeme podmínku a opakujeme. Stejně jako *while* může být cyklus a iterace přerušen pomocí *BreakType* a jaký *BreakType* vracíme je určeno stejně jako u *WhileNode*.

5.6 Krokování algoritmem

Než začneme s krokováním, vraťme se na chvíli k syntaktickému stromu, který jsme získali z kompilátoru. Máme tedy uložený jeho kořen. Syntaktický strom je složen z uzlů, kde každý uzel je nějaká operace nebo hodnota a nebo se jedná o nějaký řídicí uzel (viz. 5.5.3).

Řídicí uzly jsou důležitou částí pro krokování algoritmu. Každý řídicí uzel a uzel vracející *BreakType* implementuje interface *INodeSteppable* obsahující funkci *DoStep()*. Tato funkce vrací uzel s následujícím příkazem.

Řídicí uzly pak obsahují stavy, které reprezentují stav uzlu a krokování. Dále je vhodné podotknout, že tyto stavy, pomáhající řídit algoritmus, mohou být přebity typem *BreakType* (tedy např. *break*) Pojďme si tyto stavy rozebrat:

- Uzel **SequenceNode**, který vznikne jako spoj mezi dvěma příkazy, si pamatuje, zda jsme prošli levým a pravým synem.
- Uzel **IfNode**, který vznikne při použití konstruktu *if-else*, si pamatuje, zda jsme dělali podmínku a její výsledek. Podle výsledku pak určí, jakým blokem kódu se vydat.
- Uzel **WhileNode**, který vznikne při použití konstruktu *while*, si pamatuje tyto stavy:

  - Stav kontroly podmínky
  - Stav provedení bloku kódu
  - Stav opuštění cyklu

- Uzel **ForNode**, který vznikne při použití konstruktu *for*, si pamatuje tyto stavy:
  - Stav inicializace
  - Stav kontroly podmínky
  - Stav provedení bloku kódu
  - Stav iterace
  - Stav opuštění cyklu

Pokud ukončíme algoritmus, musíme stavy ve vrcholech vrátit do počátečního stavu. Vrcholy implementující funkci *DoStep()*, které nejsou řídicí, se pokusí provést akci podle *ReturnResult()*.

Kontrolu, kdy krok provedeme, děláme v hlavní smyčce programu. Máme nějaký časový interval. Řekněme, že jsme právě udělali nějaký krok. Program necháme běžet a v každé iteraci cyklu kontrolujeme uplynutý čas od posledního kroku. Pokud je čas čekání větší nebo roven danému intervalu, pak provedeme další krok. Pokud nějaký vrchol vrátí chybu, algoritmus zastavíme a odebereme odkaz na uzel pro krokování.

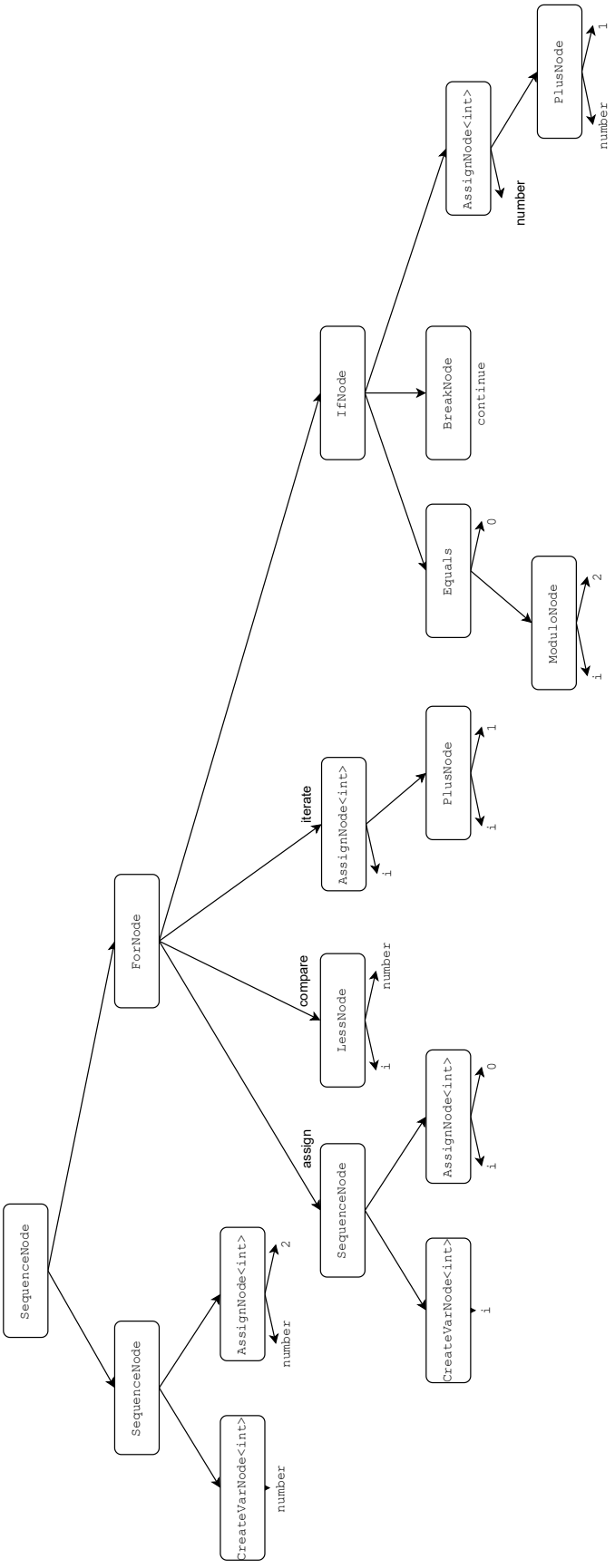
5.6.1 Ukázka

Pojďme si projít malým příkladem, abychom si lépe ukázali, jak funguje krokování. 5.6 Tento kód bude při iterování přičítat k *number* jedničku, je-li i liché. Toto děláme, dokud je i menší než *number*.

```
int number = 2;
for (int i = 0; i < number; i = i + 1)
{
    if(i % 2 == 0)
    {
        continue;
    }
    else
    {
        number = number + 1;
    }
}
```

Obrázek 5.6: Ukázkový kód pro krokování

V průběhu krokování se budeme odkazovat na tento syntaktický strom 5.7.



Obrázek 5.7: Zjednodušený syntaktický strom pro kód 5.6

1. Začínáme v kořeni, což je *Sequence*. Jdeme doleva, kde máme další *Sequence*, kde půjdeme opět doleva. Toto je počátek příkazu *int number = 2*. V tomto kroce tedy vytvoříme proměnnou *number*, vrátíme se do rodiče, půjdeme doprava a přiřadíme *number* hodnotu 2. Nakonec vrátíme rodiče.

2. V *Sequence*, kde právě jsme, jsme provedli oba příkazy. Vrátíme se tedy do rodiče, což je kořen. Jdeme doprava a přijdeme do *ForNode*, provedeme nejprve *assign* větev. Ta vytvoří proměnnou i a přiřadí ji hodnotu 0. Zůstáváme ve *ForNode*.

3. Nyní provedeme kontrolu podmínky větev compare. $i < number$ nám dá $0 < 2$ a to platí. Tedy půjdeme do bloku s kódem, kde máme *IfNode*.

4. Vstoupili jsme do *IfNode*, provedeme podmínku: $i\%2 == 0$, tedy $0\%2 == 0$. To je pravda a jdeme do pravdivé cesty s *BreakNode: continue*.

5. *BreakNode* vrátí continue a jdeme do rodiče.

6. *IfNode* splnil svůj účel, vrátíme se do *ForNode* a provedeme iteraci proměnné i.

7. Jsme ve *ForNode*, provedeme podmínku a ta platí $(1 < 2)$.

8. Jdeme do *IfNode* a provedeme podmínku ($i\%2 == 0$). Jelikož $i == 1$, tak podmínka neplatí a jdeme do větve s *AssignNode<int>*.

9. Provedeme zápis do proměnné $number = number + 1$. Poté se vrátíme do *IfNode* a z něho do *ForNode*.

10. Jsme ve *ForNode*, provedeme iteraci $i = i + 1$.

11. Provedeme kontrolu podmínky $i < number$, $2 < 3$, podmínka platí.

12. Přejdeme do *IfNode* a provedeme podmínku $i\%2 == 0$.

13. Ta platí, jdeme do *BreakNode: continue* a vrátíme *continue*.

14. Vrátíme se zpátky do *ForNode* a provedeme iteraci $i = i + 1$

15. Provedeme kontrolu podmínky $i < number$. $3 < 3$ neplatí. Odcházíme z *ForNode* do kořene. V kořeni jsme prošli oběma dětmi a tedy jsme skončili s algoritmem.

6. Kreslení grafů

V této kapitole probereme, jak řešíme rozložení a kreslení grafů.

Definujme graf *G = (V, E)*, kde *V* je množina vrcholů a *E* jsou množina hran, kde hrana je dvojice vrcholů. Pro takto definovaný graf je našim problémem najít vhodné rozložení grafu, tedy chceme každému vrcholu předat pozici v nějakém prostoru.

V Eades a kol. (2009) a v Kobourov (2013) máme definovaná kritéria pro dobré rozložení grafu. Předtím, než projdeme kritéria, bych rád zmínil, že pro naše účely budeme kreslit graf pomocí úseček, tedy každé dva vrcholy jsou spojeny úsečkou. Toto zmiňuji, jelikož jde kreslit grafy i pomocí křivek nebo s hranami, které jdou pouze ve směru jedné souřadnice najednou.

Pojďme tedy projít kritéria kreslení:

- **Křížení hran:** v nákresu grafu typicky chceme omezit křížení hran v grafu, jelikož nadměrné křížení hran může způsobit ztrátu hrany a tedy špatné předání informací čtenáři. Pokud jde graf nakreslit bez křížení hran, pak je takový graf rovinný.
- **Symetrie:** symetrie může odhalit skrytou strukturu grafu a je pro člověka obecně příjemná.
- **Délky hran:** délky hran chceme mít stejně dlouhé nebo podobných délek.
- **Rovnoměrné rozložení vrcholů:** pro nakreslení grafu chceme využít co nejvíce daného prostoru a nemít moc velké shluky vrcholů.

Při vývoji programu jsme se zaměřili na kreslení grafů pružinkovým algoritmem. Jak funguje pružinkový algoritmus je popsáno v Eades (1984/5): „Každý vrchol si představíme jako ocelový kroužek a hrany mezi vrcholy budou pružinky. Vrcholy se mezi sebou odpuzují a pružinky, jsou-li moc natažené, se chtějí smršťit, a jsou-li moc zmáčknuté, se chtějí natáhnou. Při inicializaci vrcholy nějakým způsobem rozložíme a poté necháme pružinky pracovat, čímž nalezneme rozložení s minimální vnitřní energií."

6.1 Eadesův pružinkový algoritmus

Eades (1984/5) definuje odpuzující sílu a sílu pružinek takto: síla pružinek mezi vrcholy je definována jako

$$C_1 \cdot log\left(\frac{d}{C_2}\right), \tag{6.1}$$

kde C_1 a C_2 jsou konstanty a d je požadovaná délka hrany. Logaritmická síla pružinek pak byla zvolena, jelikož lineární pružinky jsou moc silné pokud jsou vrcholy daleko od sebe.

Odpudivá síla mezi vrcholy, které nesdílí hranu, je

$$\frac{C_3}{\sqrt{d}}, \tag{6.2}$$

```
Pružinky(graf G):
1. rozlož vrcholy z G náhodně
2. opakuj M-krát:
3.   vypočítej síly pro každý vrchol
4.   přemísti vrcholy podle sil * C4
5. Výstup: rozložení grafu pro vykreslení
```

Obrázek 6.1: Výsledný algoritmus podle Eades (1984/5)

kde C_3 je další konstanta.

V článku Eades (1984/5) je řečeno, že hodnoty $C_1 = 2$, $C_2 = 1$, $C_3 = 1$, $C_4 = 0.1$ jsou vhodné pro většinu grafů. Pro naše účely tyto hodnoty byly pozměněny na $C_1 = 1$, $C_2 = 20$, $C_3 = 100$.

6.2 Fruchterman a Reingold

Fruchterman a Reingold používají velmi podobný algoritmus jako Eades. Jak je psáno v Kobourov (2012), první způsob, kterým se liší, je, že síly jsou definovány pomocí požadované délky mezi vrcholy, konkrétně odpudivá síla mezi vrcholy je

$$-l^2/d \tag{6.3}$$

a přitažlivá síla pružin je

$$d^2/l, \tag{6.4}$$

kde d je vzdálenost a l je optimální vzdálenost definovaná jako

$$l = C \cdot \sqrt{\frac{\text{plocha}}{\text{počet vrcholů}}} \tag{6.5}$$

.

Algoritmus se dále liší využitím teplotního schématu. Tomuto říkáme simulované žíhání. Jedná se o pojem převzatý z metalurgie. Pokud zahřejeme kov, lze snadněji opracovávat, je kujný. S postupným chlazením se ale kov zpevňuje, je hůře opracovatelný, až nakonec teplota klesne natolik, že kov není možné opracovávat.

Tento princip je pak převeden do kreslení grafů takto: začneme s nějakou teplotou a tuto teplotu pak necháme nějakým způsobem klesat. Podle teploty pak můžeme říci jak moc mají síly velký efekt na vrcholy. Teplota klesá a my opakujeme simulaci dokud teplota není nízká (typicky blízko nuly).

6.3 Porovnání a implementace

Implementace obou algoritmů je v souboru GraphLayout.cs.

Eadsův i Fruchtenmanův a Reingoldův algoritmus je konceptuálně i implementačně jednoduchý. Oba algoritmy většinou dávají jako výsledek pěkný graf, rozložení vrcholu bývá dostatečně vyhovující.

```
FruchtenmanReingold(graf G):
// G = (V, E)
// vrcholy mají pole .pos a~.force (pos pro pozici, force pro sílu)
1. Vypočítej l
2. dokud teplota t > epsilon:
3.   pro každou uspořádanou dvojici vrcholů v a~u, kde v!=u:
4.     u.force += odpudivá síla mezi vrcholy
5.   pro každou hranu e (s vrcholy u, v):
6.     e.u.force += síla pružinky
7.     e.v.force -= síla pružinky
8.   pro každý vrchol v:
9.     v.pos += (v.force/|v.force|) * min(v.force, t)
10.    změň v.pos aby byl v okně
11.  změň teplotu t podle chladícího schématu
```

Obrázek 6.2: Pseudo-kód algoritmu Fruchtenman a Reingold jak popsáno Kobourov (2012)

```
1. Dej počáteční vrcholy na levou stranu (x = 0)
2. Dej koncové vrcholy na pravou stranu (x = počet vrcholů * 10)
3. Dej ostatní vrcholy náhodně do prostoru
   mezi počáteční a~koncové vrcholy
4. Spusť zvolený algoritmus
5. Přeškáluj hrany, aby vrcholy zaplnily okno
```

Obrázek 6.3: Pseudokód implementace pružinkového algoritmu

Někdy se ale stane, že graf je nevyhovující. Toto se stane, jelikož najdeme lokální energetické minimum. V takovém případě se může hodit posunout jeden nebo dva vrcholy ručně (uživatel tuto možnost má).

Pokud povolíme opravy grafů takovýmto posunutím jednoho nebo dvou vrcholů, má Eades dle mého názoru o něco hezčí výsledky. Toto ale může spadat pod několik úprav, které byly provedeny oběma algoritmům.

Začněme s úpravami Eadsova algoritmu. Krom změn konstant byl Eades převeden na teplotní schéma. Teplotní schéma je nelineární, začínáme s nějakou hodnotou menší než 1 a tu při každé iteraci násobíme samu sebou. Touto hodnotou pak násobíme síly při každé iteraci.

```
temp *= temp
```

Dále (pro případy, kde je to možné) zamkneme počáteční a koncové vrcholy grafu. Toto pak zajistí, že ostatní vrcholy nemohou jít moc daleko od zbytku grafu, budou vždy přitaženy počátkem a cílem.

Změna provedená u algoritmu Fruchtenmana a Reingolda byla odstranění odmocniny ve vzorečku 6.5.

Dodatečná implementace je pak v obrázku zde 6.3

Ve výsledku jsem použil Eadsův algoritmus. Implementace Fruchtenmana a Reingolda je v kódu zanechána, ale není nikdy volána.

Závěr

Ve výsledku jsme tedy vytvořili funkční program, který animuje uživatelem definované algoritmy na grafech. Pro tyto účely jsme sestrojili nový jazyk Cb inspirovaný C#, ve kterém se algoritmy budou psát a pro tento jazyk máme v programu vestavený kompilátor, který algoritmus zkompiluje do syntaktického stromu. Grafy kreslíme pomocí pružinkového algoritmu, konkrétně používáme Eadsův pružinkový algoritmus.

Dalším krokem výzkumu by mohlo být vyzkoušení programu při výuce a získání dat, zda takováto vizualizace vskutku pomůže studentům při pochopení grafových algoritmů nebo zda by vyučujícím pomohla při výkladu.

Přestože program pracuje pouze na grafech a grafových algoritmech, program by mohl být rozšířen o další druhy algoritmů (například třídicí). Toto by vyžadovalo novou definici vstupů a nový způsob animování, ale jazyk a kompilátor by měly být dostatečně obecné pro takové účely. A pokud by nebyly, rozšíření jazyka by nemelo být obtížné.

Samotný jazyk Cb by mohl být rozšířen o definici funkcí na začátku souboru. Dále by koncovka souboru *.ces* a jazyk Cb mohly dostat rozšíření například pro Visual Studio ke zvýraznění klíčových slov a typů.

Seznam použité literatury

Aho, A. V., Lam, M. S., Sethi, R. a Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Pearson, Harlow, 2nd edition. ISBN 0-321-48681-1.

Capellman, J. a Salin, L. (2020). *MonoGame Mastery: Build a Multi-Platform 2D Game and Reusable Game Engine*. Address L. P., Berkeley, CA, 1st edition. ISBN 9781484263082.

Eades, P. (1984/5). A heuristic for graph drawing. *Congressus numerantium*, **11**, 147–160.

Eades, P., Gutwenger, C., Hong, S.-H. a Mutzel, P. (2009). Graph drawing algorithms. In CMikhail J. Atallah, M. B., editor, *Algorithms and Theory of Computation Handbook*, chapter 6. Chapman & Hall/CRC, New York, 2nd edition. ISBN 9780429144660.

Kobourov, S. G. (2012). Spring embedders and force directed graph drawing algorithms. *CoRR*, **abs/1201.3011**.

Kobourov, S. G. (2013). Force-directed drawing algorithms. In Tamassia, R., editor, *Handbook of Graph Drawing and Visualization*, chapter 12, pages 383–407. Chapman & Hall/CRC, 1st edition. ISBN 978-1-58488-412-5.

MonoGame Team (2012). Monogame documentation. `https://docs.monogame.net/articles/index.html`. Accessed: 2024-03.

Seznam obrázků