Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Vizuální programovací jazyk pro univerzální použití

2024

Bc. Roman Wehmhőner

Vedoucí práce:
doc. RNDr. Jan Konečný, Ph.D.

Studijní obor: Informatika, prezenční forma

Bibliografické údaje

Autor:	Bc. Roman Wehmhőner
Název práce:	Vizuální programovací jazyk pro univerzální použití
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2024
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	doc. RNDr. Jan Konečný, Ph.D.
Počet stran:	49
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Bc. Roman Wehmhőner
Title:	Visual programming with universal usage
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2024
Study field:	Computer Science, full-time form
Supervisor:	doc. RNDr. Jan Konečný, Ph.D.
Page count:	49
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Cílem diplomové práce bylo prozkoumat problematiky vizuálního programování pro univerzální užití.

Synopsis

The goal of the master's thesis is to explore issues related to visual programming for universal use.

Klíčová slova: vizuální programování; vizuální jazyk; VPL

Keywords: visual programming; visual language; VPL

Tímto bych chtěl poděkovat vedoucímu diplomové práce panu doc. RNDr. Janu Konečnému, Ph.D. za hodnotné rady. Zároveň bych chtěl také poděkovat panu Janu Třískovi, Mgr. Ph.D. za neformální vedení práce.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

Seznam vět

1 Netextové programování

Koncept vizuálního programování existuje už dlouhou dobu. Jedním z prvních vizuálních jazyků je GRaIL (Graphical Input Language) který vznikl už kolem roku 1968.[1] Můžeme proto říct, že vizuální programování je zde velice dlouho. Navzdory dnešním v podstatě neomezeným zobrazovacím technologiím z hlediska jednoduché grafiky, jsou stále dominantní textové jazyky. Tato práce je motivovaná otázkou proč to tak je.

1.1 Moderní trendy zjednodušování vývoje

V současné době jsou běžně využívané takzvané no-code a low-code nástroje. Tyto nástroje umožňují práci nejen programátorům, ale také odborníkům bez hlubších programátorských schopností. Tyto nástroje se často nachází ve velice specifických prostředích s velice úzkým zaměřením. V důsledku toho může nástroj hladce pokrýt veškeré potřeby domény bez předávání zbytečné složitosti na samotného vývojáře. Tyto nástroje pak mohou být využívaný experty na danou doménu bez hlubších programátorských znalostí [2]. V moment, kdy ale začne být problematika nestandardní tak tyto nástroje začínají mít problémy které už bez klasického kódování vyřešit nejdou.

Mezi takovéto nástroje můžeme zařadit například CAD software, Editory formátovaného textu, Editory videí, Tabulkové procesory a další.

1.2 Vizuální jazyky

No-code nástroje cílí na kompletní odstranění nutnosti programování pro koncového uživatele, zatímco low-code se nachází někde na půli cesty. Mezi low-code nástroje můžeme zařadit i vizuální programovací jazyky neboli VPL *(Visual Programming Language)*.

Definice 1 (Vizuální programovací jazyk)
Vizuální programovací jazyk je takový programovací jazyk, ve kterém vývojář vytváří programy hlavně pomocí manipulace s vizuálními prvky.

VPL pak můžeme rozdělit na několik hlavních kategorií, z toho v této práci nás budou zajímat hlavně následující dvě.

Definice 2 (Blokový jazyk)
Blokový jazyk umožňuje uživateli umisťovat bloky do vývojového prostředí. Spojováním bloků dohromady jako stavebnice vzniká program. Umisťování a manipulace bloků je často prováděná pomocí drag-n-drop.[2]

Blokové jazyky mohou často připomínat přirozený kód, pokud si odmyslíme grafiku bloku a zanecháme pouze text. Jejich silnou stránkou oproti kódu je pak

prevence chyb a přiblížení fyzickému světu. Barevná a hravější forma pak může zvýšit zájem o programátorské vzdělávání, jako to například dělá Scratch.

Definice 3 (Diagramový jazyk)
Diagramové nebo také data-flow jazyky jsou charakteristické propojováním vizuálních prvků čárami, nebo šipkami. V důsledku tak mohou diagramové jazyky připomínat vývojové diagramy nebo jiné diagramy značící závislosti mezi jednotlivými komponenty programu. [2]

V mnoha systémech se diagramové jazyky využívají experty rovnou i jako náhled do zrovna běžícího systému. Proto jednou ze silných stránek diagramových jazyků je jejich vizualizační schopnost. Další výhodou je možnost přeskočení fáze nákresů a diagramů při vývoji. Namísto nákresu je možné vytvořit základ programu který je už začátek spustitelného programu.

2 Návrh vlastního jazyka

Pro návrh vlastního jazyka jsem se rozhodl zvolit diagramový typ s metodou vyhodnocování kde se každý uzel vyhodnotí maximálně jednou.

Jazyk byl vyvíjen nejdříve experimentálně. Při vývoji jsem se pokoušel najít různé způsoby řešení problematik. Po dokončení vývoje jsem posbíral všechny poznatky z hlediska interpretace a shrnul je v této kapitole.

2.1 Minimální VPL

Při navrhování diagramového jazyka můžeme vycházet z teorie grafů. Nalezením a přiřazením významů jednotlivým konstruktům lze vytvořit vizuální programovací jazyk. VPL je dobré následně rozšiřovat, aby byl schopen pojmout složitější konstrukty.

Pro usnadnění vyhodnocování a identifikaci konečného výsledku programu rozšíříme graf o množinu výstupních uzlů. Pro vyhodnocování jednotlivých uzlů programu musíme definovat funkci *eval*. Všeobecně *eval* nebudeme více zkoumat a pouze určíme vlastnosti, pokud budou potřeba.

Definice 4 (Minimální VPL)

$$VPL = \langle N, C, eval, OUT \rangle$$

- VPL je základní vizuální jazyk.
- N je konečná množina uzlů.
- C je konečná množina orientovaných propojení $C \subseteq N \times N$.
- $eval$ je funkce vyhodnocující uzel společně se vstupními argumenty $eval : N \times Args \to Res$.
- OUT je konečná množina výstupních uzlů $OUT \subseteq N$.

Už ve tvaru pouhého grafu je možné využít graf jako velice jednoduchý VPL. S takto navrženým jazykem jsme schopni vytvářet základní výrazy.

Příklad 5 (Základní aritmetický výraz)
Výraz $1 + 2$ lze reprezentovat pomocí VPL kde:

- $N = \{1, 2, +, out\}$
- $C = \{\langle 1, + \rangle, \langle 2, + \rangle, \langle +, out \rangle\}$
- $eval$:
  - $eval(x, \emptyset) = x \mid x \in \mathbb{N}$
  - $eval(+, args) = \sum_{x \in args} x$
  - $eval(out, args) = args_{\emptyset}$
- $OUT = \{out\}$
- $VPL = \langle N, C, eval, OUT \rangle$

Když se podíváme na příklad 5 můžeme najít nějaké paralely s textovými jazyky. Například každý uzel je funkcí, která může mít vstupy a má výstup. Propojení nám pomáhají vzít výstup nějakého uzlu a předat ho jako vstup jinému. Můžeme využít dvě propojení a vzít výstup jednoho uzlu a dát ho jako vstup dvou různých uzlů. Tyto dvě propojení bychom v textových jazyce označili za proměnnou. Zároveň ale také propojení určuje pořadí vykonání kódu, a tak bychom je mohli označovat třeba za čísla řádků. Proto koncept propojení nemá jednoznačný protiklad v textových jazycích a budeme ho brát jako tok dat.

2.2 Vyhodnocovací algoritmus

K vyhodnocování algoritmus 1 využívá struktury zásobníku a asociativního pole. V zásobníku se nachází vždy uložené uzly, které mají být vyhodnoceny. Vrchol zásobníku označíme jako pracovní anebo aktuálně vyhodnocovaný uzel. Asociativní pole bude obsahovat všechny výsledné hodnoty konkrétních uzlů. Pokud je uzel přítomen jako klíč v asociativním poli, označujeme ho jako vyhodnocený.

Vyhodnocovací postup začíná inicializací zásobníku *toEval* a asociativního pole *values*. Na zásobníku se na začátku nachází všechny uzly z OUT.

V hlavní smyčce program postupuje následovně. Pokud je pracovní uzel již vyhodnocený, program uzel odstraní. Takovýto uzel už byl vyhodnocený a výsledky, které potřebujeme pro další výpočty už máme. Pokud jsou všechny vstupní hodnoty uzlu známé, vyhodnotíme tento uzel a odebereme ho ze zásobníku. Výsledné hodnoty jsou uložené v asociativním poli. Pokud předchozí podmínka nebyla splněna, provede se přidání všech vstupních uzlů na zásobník. Tyto uzly musíme přidat na zásobník, protože pracovní uzel vyžaduje jejich hodnoty, aby se mohl sám vyhodnotit. Program končí, jakmile je zásobník prázdný.

Algorithm 1 Vyhodnocovací algoritmus

1: **Inicializace:**
2: $toEval := \emptyset$ ▷ prázdný zásobník
3: $values := \emptyset$ ▷ prázdné asociativní pole
4: **Přidání na zásobník:**
5: pushAll($toEval, VPL_{OUT}$)
6: **Hlavní smyčka:**
7: **while** $toEval \neq \emptyset$ **do**
8: $currentNode :=$ peek($toEval$)
9: **if** exists($values, currentNode$) **then**
10: pop($toEval$) ▷ uzel je už vyhodnocený
11: **else if** allInputsEvaluated($currentNode, values, VPL$) **then**
12: $inputs :=$ collectInputs($currentNode, values, VPL$)
13: $result :=$ eval($currentNode, inputs, VPL$)
14: set($values, currentNode, result$)
15: pop($toEval$)
16: **else**
17: pushAll($toEval$, inputNodes($currentNode, VPL$))
18: **end if**
19: **end while**
20: **Konec programu:**
21: $results :=$ collectResults($values, VPL$)
22: **return** *results* ▷ návratové hodnoty jsou klíče z množiny OUT

2.2.1 Pomocné funkce

Tento algoritmus je použitelný i pro VPL. Rozšíření VPL mění hlavně chování pomocných funkcí použitých v algoritmu ale málokdy algoritmus jako takový.

Jednoduché funkce jako jsou collectInputs a collectResults pouze vybírají data které zrovna potřebujeme z values abychom mohli s nimi dále pracovat. Každý uzel programu můžeme označit jako funkci kterou aplikuje eval. Každý uzel může mít svou vlastní logiku pro *eval* ale i naopak více různých uzlů může mít společnou logiku. Samotné fungování a tvar navrácených a vstupních dat evalu nás pro návrh jazyka nezajímá.

Definice 6 (pomocné funkce pro minimální VPL)

- $\text{inputsOf}(node, VPL) =$
  $\{x \mid \langle x, y\rangle \in VPL_C \wedge y = node\}$
- $\text{allInputsEvaluated}(node, values, VPL) =$
  $\forall x \in \text{inputsOf}(node, VPL)\, \exists kv\, (kv \in values \wedge kv_{key} = x)$
- $\text{collectInputs}(node, values, VPL) =$
  $\{kv_{value} \mid kv \in values \wedge kv_{key} \in \text{inputsOf}(currentNode, VPL)\}$
- $\text{collectResults}(values, VPL) = \{kv \mid kv \in values \wedge kv_{key} \in VPL_{OUT}\}$

Díky těmto pomocným funkcím už nám nechybí nic k vyhodnocení ukázkového programu.

Příklad 7 (Vyhodnocení příkladu 5)

1. $toEval = \langle out\rangle; values = \{\}$

   Všechny vstupní hodnoty pracovního uzlu nejsou vyhodnocené. Podle propojení $\langle +, out\rangle$ přidáme na zásobník $+$

2. $toEval = \langle out, +\rangle; values = \{\}$

   Pracovní uzel má dva vstupní uzly které nejsou vyhodnoceny. $+$ má dvě propojení $\langle 1, +\rangle$ a $\langle 2, +\rangle$. Na zásobník přidáme 1 a 2

3. $toEval = \langle out, +, 2, 1\rangle; values = \{\}$

   Pracovní uzel nemá žádné vstupy což znamená, že všechny vstupy jsou vyhodnocené. Pro uzel bez vstupních uzlů je vždy collectInputs roven $\emptyset$. Uzel 1 vyhodnotíme a odebereme ho z $toEval$. $\text{eval}(1, \emptyset) = 1$ a výsledek přidáme do $values$.

4. $toEval = \langle out, +, 2\rangle; values = \{\langle 1, 1\rangle\}$

   Stejně jako v předchozím kroku nemá pracovní uzel žádné vstupy tak jej vyhodnotíme a odebereme ho z $toEval$. eval$(2, \emptyset) = 2$ přidáme do $values$.

5. $toEval = \langle out, +\rangle; values = \{\langle 1, 1\rangle, \langle 2, 2\rangle\}$

   Pracovní uzel má všechny vstupní hodnoty vyhodnocené.
   Získáme vstupní hodnoty collectInputs$(+, values, VPL) = \{1, 2\}$ a uzel vyhodnotíme eval$(+, \{1, 2\}) = 3$.

6. $toEval = \langle out\rangle; values = \{\langle 1, 1\rangle, \langle 2, 2\rangle, \langle +, 3\rangle\}$

   Jako poslední uzel vyhodnotíme out, který jen vrátí vstupní hodnotu eval$(out, \{3\}) = 3$.

7. $toEval = \emptyset; values = \{\langle 1, 1\rangle, \langle 2, 2\rangle, \langle +, 3\rangle, \langle out, 3\rangle\}$

   V tento moment je zásobník $toEval$ prázdný. Program nemá už další instrukce a skončil.

8. Výsledek volání programu je collectResults$(values, VPL) = \{\langle out, 3\rangle\}$.

2.2.2 Zacyklení

Vyhodnocovací algoritmus prochází program v podstatě metodou průchodu grafu do hloubky. Proto je důležité si dávat pozor na cykly v propojeních. Pokud by došlo k vytvoření cyklu, tak by program nikdy nedoběhl

Příklad 8 (Cyklický program)
Výraz $1 + 2$ lze reprezentovat pomocí VPL kde:

- $N = \{+_1, +_2, out\}$
- $C = \{\langle +_1, +_2\rangle, \langle +_2, +_1\rangle, \langle +_2, out\rangle\}$
- $eval(n, args) = \emptyset$ ▷ Nikdy neproběhne.
- $OUT = \{out\}$
- $VPL = \langle N, C, eval, OUT\rangle$

V tomto případě můžeme zvolit arbitrární definici pro $eval$, který nikdy není volaný.

Příklad 9 (Vyhodnocení příkladu: Cyklický program)

1. $toEval = \langle out\rangle$ $values = \{\}$

   Všechny vstupní hodnoty pracovního uzlu nejsou vyhodnocené. Podle propojení $\langle +_2, out\rangle$ přidáme na zásobník $+_2$

2. $toEval = \langle out, +_2 \rangle$ $values = \{\}$

   $+_2$ potřebuje ke svému vyhodnocení výsledek $+_1$ tak jej přidáme na zásobník.

3. $toEval = \langle out, +_2, +_1 \rangle$ $values = \{\}$

   $+_1$ potřebuje ke svému vyhodnocení výsledek $+_2$ tak jej přidáme na zásobník.

4. $toEval = \langle out, +_2, +_1, +_2 \rangle$ $values = \{\}$

   V tento moment již vidíme, že k vyhodnocení $+_2$ potřebujeme nejdříve vyhodnotit $+_1$. Abychom vyhodnotili $+_1$ potřebujeme nejdříve vyhodnotit $+_2$ Proto by v tomto příkladu došlo k nekonečnému přidávání $+_2, +_1, \ldots$ na zásobník a program by se nikdy nevyhodnotí

Díky tomuto algoritmu jsme schopni vyhodnocovat programy složené z konstant a z komutativních funkcí. Pokud chceme v programech pohodlně využívat nekomutativní funkce budeme muset jazyk rozšířit.

2.3 Porty

Jazyk rozšíříme o porty, které budou dodávat význam jednotlivým propojením. Takovýto port je buď zdrojový nebo cílový. Zdrojový port lze vnímat jako jeden z výsledků nějaké funkce, zatímco cílový jako argument funkce. Pokud nějaký uzel nevyžaduje rozlišení u vstupních nebo u výstupních hodnot, pak můžeme pro jednoduchost porty vynechat a označíme je za implicitní a na jeho významu ani pořadí nezáleží. Díky implicitním portům je minimální VPL zároveň VPL s porty. Jako poslední krok musíme změnit C na multimnožinu abychom umožnili many-to-many propojení mezi uzly.

Definice 10 (VPL s porty)

$$VPL = \langle N, C, eval, OUT, P \rangle$$

- VPL je základní vizuální jazyk který budeme rozšiřovat, aby splňoval naše aktuální požadavky.
- N je konečná množina uzlů.
- C je konečná multimnožina orientovaných propojení $C \subseteq N \times N$.
- $eval$ je funkce vyhodnocující uzel společně se vstupními argumenty $eval : N \times Args \to Res$.
- OUT je konečná množina výstupů
- $P \subseteq \{\langle c, \text{type}, \text{value} \rangle \mid c \in C, type \in \{source, target\}, value \in Text\}$

Díky tomuto rozšíření využívat nekomutativní funkce jako je například implikace

Algoritmus se nemění pouze některé jeho pomocné funkce. *values* je stále asociativní pole kde klíč je uzel stejně jako předtím. Hodnoty *values* jsou ale nově asociativní pole, kde klíč je text portu a hodnota je až výsledek vyhodnocení uzlu pro daný port. Implicitní port má klíč $\emptyset$.

Definice 11 (pomocné funkce pro VPL s porty)

- $\text{getPort}(c, type, VPL) = p_{value}$ pokud existuje p takové že:
  $p \in VPL_P \wedge p_c = c \wedge p_{type} = type$ jinak $\emptyset$
- $\text{connsWithPorts}(VPL) =$
  $\{\langle c_1, c_2, s, t\rangle \mid c \in VPL_C;$
  $s = \text{getPort}(c, source, VPL); t = \text{getPort}(c, target, VPL)\}$
- $\text{mapPortValues}(values, VPL) =$
  $\{\langle n, p, v \rangle \mid cp \in \text{connsWithPorts}(VPL) \wedge v = \text{get}(\text{get}(values, cp_1), cp_3);$
  $n = cp_2; p = cp_4\}$
- $\text{collectInputs}(node, values, VPL) =$
  $\{pv_p : pv_v \mid pv = \text{mapPortValues}(values, VPL) \wedge pv_n = node\}$

Většina původních pomocných funkcí není třeba měnit. Zásadní změnou prošla jen collectInputs funkce, aby vracela asociativní pole s porty místo samostatných hodnot. Funkce mapPortValues provádí nejdůležitější část nové logiky. Vezme hodnoty z výstupních portů uzlů a přiřadí je vstupním portům cílových uzlů. Výsledkem je pak trojice uzlu, portu a hodnoty se kterou už se velice snadno pracuje.

Příklad 12 (nekomutativní operace ve VPL s porty)
Výraz $a \rightarrow b$ lze reprezentovat pomocí VPL s porty kde:

- $N = \{a, b, \rightarrow, out\}$
- $C = \{\langle a, \rightarrow\rangle_1, \langle b, \rightarrow\rangle_2, \langle \rightarrow, out\rangle_3\}$
- $eval$:
  - $eval(x, y) = \{\emptyset : eval'(x, y)\}$
  - $eval'(x, \emptyset) = true$ pro $x \in \{a, b\}$
  - $eval'(\rightarrow, args) = args_{antacedent} \rightarrow args_{konsekvent} x$
  - $eval'(out, args) = args_{\emptyset}$
- $P = \{\langle C_1, \text{target}, \text{antacedent}\rangle, \langle C_2, \text{target}, \text{konsekvent}\rangle\}$

- $OUT = \{out\}$
- $VPL = \langle N, C, eval, OUT, P \rangle$

V příkladu 12 a následujících si pomůžeme funkcí $eval'$ která nám zjednoduší zápis pro návrat hodnoty v implicitním portu.

2.3.1 Více cílových propojení v jednom portu

V rozšířeném jazyku s porty může dojít k situaci která není pevně definovaná a přináší prostor pro různé interpretace. V případě že má jeden uzel v jednom svém cílovém portu více propojení, pak musíme s takovýmto propojením nějak pracovat. V tomto případě se nabízí několik možností:

1. Takovéto propojování zakázat a vynutit, aby každý port každého uzlu byl vždy unikátní
2. S takovýmto propojením nějak speciálně nepracovat a do vyhodnocovací funkce předat hodnoty tak jak jsou.
3. Určit funkci, která takovéto hodnoty spojí dohromady v jednu výslednou hodnotu

Projděme si jednotlivé možnosti.

V 1. případě nedochází všeobecně k žádným větším komplikacím. Velké množství funkcí má pevný počet argumentů a pokud bychom potřebovali neomezený počet argumentů. Pokud bychom potřebovali neomezený počet argumentů, tak můžeme jednoduše specifikovat, jak má jejich port vypadat.

Ve 2. případě může v některých situacích docházet k problému. Například pro implikaci nedává smysl mít dva antacedenty a stejně tak pro velké množství dalších funkcí nedává smysl mít více vstupních argumentů. V takovémto případě by bylo do budoucna komplikovanější využívat statickou analýzu a každý uzel by musel mít pevně určeno pro každý port, jestli podporuje více hodnot.

Ve 3. případě takovéto řešení může být často problém programátorem identifikovat co za implicitní operace se zde dějí.

Navzdory tomu, že 3. řešení může nabídnout zajímavé zkratky tak kvůli jeho negativům budeme v této práci využívat pouze 1. řešení, pokud nebude řečeno jinak.

2.3.2 Reprezentace pomocí minimálního VPL

Ve skutečnosti nejsou porty vůbec potřeba. Každý vstupní port můžeme reprezentovat jako uzel navíc který přidává konkrétnímu vstupu informaci o který vstup se jedná. To samé platí pro výstupy. Například implikaci bychom mohli reprezentovat takto.

Příklad 13 (nekomutativní operace v minimálním VPL)
Výraz $a \rightarrow b$ lze reprezentovat pomocí minimálního VPL kde:

- $N = \{a, b, antacedent, konsekvent, \rightarrow, out\}$
- $C = \{\langle a, antacedent\rangle, \langle b, konsekvent\rangle, \langle \rightarrow, out\rangle, \langle antacedent, \rightarrow\rangle, \langle konsekvent, \rightarrow\rangle\}$
- $\text{getInput}(args, port) = arg_2$ kde $arg \in args \wedge arg_1 = port$
- $eval$:
  - $eval(x, \emptyset) = true \mid x \in \{a, b\}$ ▷ pro jednoduchost
  - $eval(\rightarrow, args) =$ $\text{getInput}(args, antacedent) \rightarrow \text{getInput}(args, konsekvent)$
  - $eval(antacedent, args) = \langle a, args_\emptyset\rangle$
  - $eval(konsekvent, args) = \langle k, args_\emptyset\rangle$
  - $eval(out, args) = args_\emptyset$
- $OUT = \{out\}$
- $VPL = \langle N, C, eval, OUT\rangle$

Takto zapsaný program je ekvivalentní s předchozím s porty. A však koncept nějakého označování vstupů uzlu je tak častý, že by bylo nepraktické jazyk využívat takovýmto způsobem. Proto zůstaneme u verze s porty.

2.4 Roviny

Pro jednoduchý znovu použitelný kód zavedeme roviny. Každý program VPL s porty je nově zároveň rovinou neboli použitelným uzlem ve vpl. VPL s rovinami je nově dvojice z množiny rovin a vstupní roviny programu. Uzly, které jsou součástí nějaké roviny budeme označovat jako vlastní uzly. Celkovou množinu použitelných uzlů rozšíříme o symboly rovin. Pro jednoduchost necháme označení roviny jako VPL ale zaměnitelně můžeme používat i *plane*.

Definice 14 (VPL s rovinami)

$$VPL = \langle N, C, eval, OUT, P\rangle$$

- VPL je samostatně spustitelná rovina programu.
- N je konečná množina vlastních uzlů.
- N' je konečná množina vlastních uzlů rozšířená o symboly rovin $N \cup \{plane_x \mid x \in PL\}$.

- C je konečná multimnožina orientovaných propojení $C \subseteq N' \times N'$.
- *eval* je funkce vyhodnocující uzel společně se vstupními argumenty $eval : N \times Args \to Res$.
- OUT je konečná množina výstupů
- $P \subseteq \{\langle c, \text{type}, \text{value}\rangle \mid c \in C, type \in \{source, target\}, value \in Text\}$

$$program = \langle PL, entry \rangle$$

- PL je konečná množina rovin.
- $entry$ je vstupní rovina programu $entry \in PL$.

V definici je vidět že funkce *eval* je definovaná pouze pro vlastní uzly N. Ačkoliv se to může jevit jako chyba, jedná se o chtěné chování. Funkce *eval* bude vždy rozšířená o vyhodnocování rovin.

Definice 15 (Rozšíření *eval* o vyhodnocení rovin)
evalPlane($plane, values$) je vyhodnocovací algoritmus, který se inicializuje s předanými *values* namísto prázdného asociativního pole hodnot.
$eval(plane, Args)$ = evalPlane($plane, \{\langle IN_p, v\rangle \mid \langle p, v\rangle \in Args\}$) kde *plane* je uzel roviny.

Samotný program VPL s rovinami vyhodnotíme pomocí *eval* kterému předáme $plane_{entry}$ a zároveň můžeme předat nějaké argumenty jako vstup programu.

2.5 Speciální forma IF

Díky rovinám jsme schopni lépe strukturovat kód a zároveň nám umožňují vytvářet rekurzivní volání, což je klíčový prvek tohoto rozšíření. Avšak vytvoření rekurzivního volání by znamenalo nekonečné vyhodnocování, protože nemáme způsob, jak vytvořit limitní podmínku s podmíněným vyhodnocením. K vytvoření limitní podmínky potřebujeme podmíněné vyhodnocování, které vytvoříme pomocí speciální formy IF. IF bude mít svůj vlastní vyhodnocovací postup v hlavní smyčce programu.

Algorithm 2 Vyhodnocovací algoritmus s IF

1: **Inicializace:**
2: $toEval := \emptyset$ ▷ prázdný zásobník
3: $values := \emptyset$ ▷ prázdné asociativní pole
4: **Přidání na zásobník:**
5: pushAll($toEval, VPL_{OUT}$)
6: **Hlavní smyčka:**
7: **while** $toEval \neq \emptyset$ **do**
8: $currentNode :=$ peek($toEval$)
9: **if** exists($values, currentNode$) **then**
10: pop($toEval$) ▷ uzel je už vyhodnocený
11: **else if** $currentNode = IF$ **then** ▷ IF může mít index
12: $condNode :=$ getNodeByPort($currentNode, target, cond, VPL$)
13: **if** $condNode \notin values$ **then** ▷ Podmínka není vyhodnocena
14: push($toEval, condNode$)
15: **else**
16: $condValue := get(values, cond)$ ▷ Určí která větev se vyhodnotí
17: $branchNode :=$
getNodeByPort($currentNode, target, condValue, VPL$)
18: **if** $branchNode \notin values$ **then**
19: push($toEval, branchNode$)
20: **else**
21: $result := get(values, branchNode)$
22: set($values, currentNode, result$)
23: **end if**
24: **end if**
25: **else if** allInputsEvaluated($currentNode, values, VPL$) **then**
26: $inputs :=$ collectInputs($currentNode, values, VPL$)
27: $result :=$ eval($currentNode, inputs, VPL$)
28: set($values, currentNode, result$)
29: pop($toEval$)
30: **else**
31: pushAll($toEval$, inputNodes($currentNode, VPL$))
32: **end if**
33: **end while**
34: **Konec programu:**
35: $results :=$ collectResults($values, VPL$)
36: **return** $results$ ▷ návratové hodnoty jsou klíče z množiny OUT

Definice 16 (Pomocná funkce getNodeByPort)

getNodeByPort$(node, type, value, VPL) =$
n kde $c \in VPL_C \wedge p \in VPL_P \wedge p = \langle c, type, value\rangle \wedge c = \langle n, node\rangle$

Teď když už máme možnost podmíněného vyhodnocování a rekurze tak si můžeme ukázat jednoduchý program pro výpočet faktoriálu.

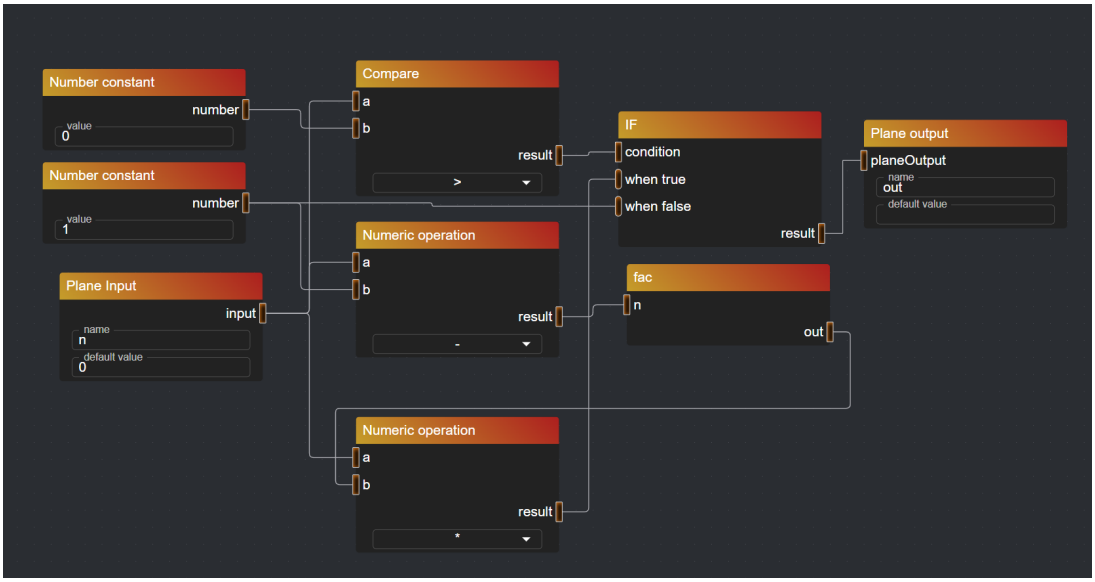
Příklad 17 (program s využitím speciální formy If)

$$fac(0) = 1$$
$$fac(n) = n * fac(n-1) \mid n \in \mathbb{N}$$

- $N = \{IN_n, IF_1, >, 0, 1, plane_{fac}, -, *, out\}$
- $C = \{$
  $\langle IN_n, >\rangle_1, \langle 0, >\rangle_2,$
  $\langle IN_n, -\rangle_3, \langle 1, -\rangle_4, \langle -, plane_{fac}\rangle_5, \langle plane_{fac}, *\rangle_6, \langle IN_n, *\rangle_7,$
  $\langle >, IF_1\rangle_8, \langle *, IF_1\rangle_9, \langle 1, IF_1\rangle_{10}, \langle IF_1, out\rangle_{11},$
  $\}$
- $eval$:
  - $eval(x, y) = \{\emptyset : eval'(x, y)\}$
  - $eval'(x, \emptyset) = x$ pro $x \in \mathbb{N}$
  - $eval'(>, args) = args_{levý} > args_{pravý}$
  - $eval'(-, args) = args_{menšenec} - args_{menšitel}$
  - $eval'(*, args) = \prod_{x \in args} x$
  - $eval'(out, args) = args_{\emptyset}$
- $OUT = \{out\}$
- $P = \{$
  $\langle C_1, \text{target}, \text{levý}\rangle, \langle C_2, \text{target}, \text{pravý}\rangle,$
  $\langle C_3, \text{target}, \text{menšenec}\rangle, \langle C_4, \text{target}, \text{menšitel}\rangle,$
  $\langle C_5, \text{target}, \text{n}\rangle,$
  $\langle C_8, \text{target}, \text{condition}\rangle, \langle C_9, \text{target}, \text{true}\rangle, \langle C_{10}, \text{target}, \text{false}\rangle$
  $\}$
- $plane_{fac} = \langle N, C, eval, OUT, P\rangle$
- $VPL = \langle \{plane_{fac}\}, plane_{fac}\rangle$

V obrázku 1 pak můžeme vidět tento program přímo v aplikaci.

Podobným způsobem bychom mohli umožnit vytváření podobných speciálních forem ovlivňujících vyhodnocování programu. Místo toho zkusíme prozkoumávat více intuitivní přístup.



Obrázek 1: Faktoriál v editoru

2.5.1 Problematika vedlejších efektů

Příklad 18 (program s vedlejšími efekty)
Jednoduchá kalkulačka na odečítání dvou čísel zadaných uživatelem.

- $N = \{Read_1, Read_2, -, Print, out\}$
- $C = \{\langle Read_1, - \rangle_1, \langle Read_2, - \rangle_2, \langle -, Print \rangle_3, \langle Print, out \rangle_4\}$
- $eval$:
  - $eval(Read_n, args) =$ Načti číslo z konzole $\mid n \in \mathbb{N}$
  - $eval(Print, args) =$ Vypiš argument do konzole
  - $eval(-, args) = args_{menšenec} - args_{menšitel}$
  - $eval(out, args) = args_{\emptyset}$
- $OUT = \{out\}$
- $P = \{\langle C_1, \text{target}, \text{menšenec} \rangle, \langle C_2, \text{target}, \text{menšitel} \rangle\}$
- $plane_{calc} = \langle N, C, eval, OUT, P \rangle$
- $VPL = \langle \{plane_{calc}\}, plane_{calc} \rangle$

V tomto příkladu není nijak zaručené pořadí vyhodnocení čtení z konzole. Navzdory tomu že prvky propojení jsou indexované, jedná se pouze o arbitrární indexaci, která usnadňuje odkazování na prvky množiny. Samotné indexy nijak neovlivňují pořadí vyhodnocení uzlů v algoritmu. Proto v tomto příkladu nemůžeme zaručit, že nejdříve načte menšenec a následně menšitel.

Tento program nic nevrací. Nedává tedy smysl mít nějaký výstupní uzel a je zde jen kvůli vyhodnocování. V tomto smyslu je využití print zvláštní. Celý program je jakoby psaný pozpátku a působí neintuitivně.

2.6 Toky

Se speciální formou IF je jazyk v podstatě kompletní. Problémy nastávají v moment, kdy dochází k využití zmíněných uzlů s vedlejším efektem. Uzly s vedlejším efektem jsou validní. Přitom samotná podstata není ve vedlejším efektu jako takovém. Na detailní znalost algoritmu, který bude daný kód vyhodnocovat bychom se neměli spoléhat. V budoucnu může kvůli optimalizacím docházet ke změnám algoritmu které by v důsledku mohli znamenat i kompletní změnu chovaní programů s vedlejšími efekty. Pro roviny využívající uzly s vedlejším efektem je potřeba mít nástroj který nám umožní explicitně určit pořadí vyhodnocování těchto uzlů.

Zavedeme toky, které jsou podobné propojením. Toky nám nabízí podmíněné vyhodnocování v opačném směru než propojení a tím i intuitivnější vyhodnocování programu. Ve standardním propojení algoritmus nejdříve označí k vyhodnocení cílový uzel a ten vyhodnotí až po vyhodnocení všech zdrojových uzlů. U toků uzlů je to naopak. Nejdříve algoritmus označí k vyhodnocení zdrojový uzel a na základě jeho výsledku se určí, jestli vůbec dojde k vyhodnocení cílového. Díky tomuto efektu jsme schopni jednodušeji řídit pořadí vyhodnocovaní uzlů, tak že je budeme řetězit za sebe.

Klasické propojení výpočtem připomíná vyhodnocování výrazu. Vyhodnocování toků připomíná spíše sekvenci příkazů.

Protože je výhodné, aby uzel mohl mít více různých výstupních toků, je dobré dodefinovat porty tak, aby mohli specifikovat i toky.

Definice 19 (VPL s toky)

$$plane = \langle N, C, F, eval, OUT, P \rangle$$

- *plane* je samostatně spustitelná rovina programu.
- N je konečná množina vlastních uzlů.
- N' je konečná množina vlastních uzlů rozšířená o symboly rovin $N \cup \{plane_x \mid x \in PL\}$.
- C je konečná multimnožina orientovaných propojení $C \subseteq N' \times N'$.
- F je konečná multimnožina orientovaných toků $F \subseteq N' \times N'$
- *eval* je funkce vyhodnocující uzel společně se vstupními argumenty $eval : N \times Args \to Res$.
- OUT je konečná množina výstupů

- $P \subseteq \{\langle cf, \text{type}, \text{value}\rangle \mid cf \in C \cup F, type \in \{source, target\}, value \in Text\}$

$$VPL = \langle PL, entry\rangle$$

- PL je konečná množina rovin.
- $entry$ je vstupní rovina programu $entry \in PL$.

Algorithm 3 Vyhodnocovací algoritmus s toky

Inicializace:
$toEval := \emptyset$ ▷ prázdný zásobník
$values := \emptyset$ ▷ prázdné asociativní pole
Přidání na zásobník:
pushAll$(toEval, VPL_{OUT})$
if $Init \in VPL_N$ **then**
    push$(toEval, Init)$ ▷ přidej úvodní tok
end if
Hlavní smyčka:
while $toEval \neq \emptyset$ **do**
    $currentNode :=$ peek$(toEval)$
    if exists$(values, currentNode)$ **then**
        pop$(toEval)$ ▷ uzel je už vyhodnocený
    else if $\neg$isEnabled$(currentNode, values, VPL)$ **then**
        pop$(toEval)$ ▷ uzel není určený k vyhodnocení
    else if allInputsEvaluated$(currentNode, values, VPL)$ **then**
        $inputs :=$ collectInputs$(currentNode, values, VPL)$
        $result :=$ eval$(currentNode, inputs, VPL)$
        set$(values, currentNode, result)$
        pop$(toEval)$
        pushAll$(toEval, \{t \mid \langle s, t\rangle \in F \wedge s = currentNode\})$
    else
        pushAll$(toEval,$ inputNodes$(currentNode, VPL))$
    end if
end while
Konec programu:
$results :=$ collectResults$(values, VPL)$
return $results$ ▷ návratové hodnoty jsou klíče z množiny OUT

V algoritmu 3 máme dvě změny. Po vyhodnocení uzlu se vždy přidají na zásobník vyhodnocení všechny cílové uzly toků pracovního uzlu. Při vybírání pracovního uzlu přeskočíme uzly, které nejsou určené k vyhodnocení. Kdy je uzel určen k vyhodnocení je detailněji popsané u pomocných funkcí.

Definice 20 (pomocné funkce pro VPL s toky)

- getPort$(cf, type, VPL) = p_{value}$ pokud existuje p takové že:
  $p \in VPL_P \wedge p_{cf} = cf \wedge p_{type} = type$ jinak $\emptyset$
- flowsWithPorts$(VPL) =$
  $\{\langle f_1, f_2, s, t\rangle \mid f \in VPL_F;$
  $s = \text{getPort}(f, source, VPL); t = \text{getPort}(f, target, VPL)\}$
- mapPortEnables$(values, VPL) =$
  $\{\langle n, p, e\rangle \mid fp \in \text{flowsWithPorts}(VPL) \wedge e = get(get(values, fp_1), fp_3);$
  $n = fp_2; p = fp_4\}$
- isNodeDone$(node, values, VPL) = \text{exists}(values, node)$
  $\wedge (\neg\text{exists}(\text{get}(values, node), Done) \vee \text{get}(\text{get}(values, node), Done))$
- isFlowImplicit$(f, VPL) =$
  $\nexists p \in VPL_P\, p_{cf} = f \wedge p_{value} \neq Done$
- hasEnabledFlow$(node, values, VPL) =$
  $(\exists pe \in \text{flowsWithPorts}(values, VPL)\, pe_e \wedge pe_n = node)$
  $\vee \exists f \in VPL_F \text{ isFlowImplicit}(f, VPL) \wedge \text{isNodeDone}(f_1, values, VPL)$
- hasNoFlow$(node, VPL) =$
  $\nexists x, y \in VPL_{N'} \langle y, x\rangle \in VPL_F \wedge node = x$
- isEnabled$(node, values, VPL) =$
  $\text{hasNoFlow}(node, VPL) \vee \text{hasEnabledFlow}(node, values, VPL)$

Hodnoty používané pro toky jsou součástí hodnot *values*. Aby nedošlo ke kolizím se standardními hodnotami, tak budeme jako implicitní port pro toky používat klíč *Done*. *Done* budeme brát jako pravdu, pokud nemá vyhodnocený uzel jiný výsledek. Tohle chování je vidět ve funkci isNodeDone. Díky tomu můžeme za sebe řetězit volání jednodušeji. Zároveň nám to ale nechává prostor pro potlačení tohoto chovaní nastavením *Done* na nepravdu. Tímto můžeme například komunikovat chybové stavy a přerušit vyhodnocování dalších uzlů.

Díky používání funkce isEnabled můžeme zjistit, jestli je uzel určený k vyhodnocení. Pokud uzel nemá žádné vstupní toky, tak je určený k vyhodnocení vždy. Jedná se o uzel pouze s datovými propojeními. Uzly se vstupními toky musím mít ke svému vyhodnocení alespoň jeden tok pravdiví. Díky nastavení toků předchozího uzlu tak určujeme, jestli bude následující vyhodnocen.

Pokud rozšíříme funkci *collectInputs* o výsledky toků, stane se něco zajímavého. V podstatě nám tohle rozšíření umožní určit v *eval* odkud byl uzel volaný. Takovéto chování nemá moc výskyt v textových jazycích. Zároveň nám to ale může rozbít jeden z invariantů jazyka. Bez toků jako vstupu byly vždy vstupy jen datové propojení, které byly zaručeně vyhodnocené. Toky ale mají opačný

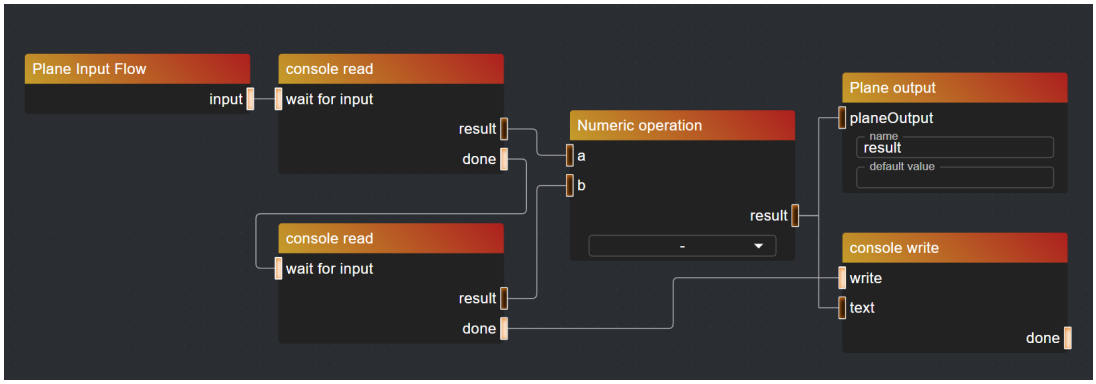
směr a může dojít k vyhodnocení předchozího uzlu po vyhodnocení aktuálního. Tím se ale změní jeho vstup což by mohlo změnit i výsledek samotného uzlu. Tak bychom museli uzel vyhodnotit znovu, a to by bylo porušení základního pravidla navrženého jazyku.

2.6.1 Vedlejší efekty a toky

Díky tomuto rozšíření máme možnost přesně vědět pořadí v jakém se vybrané bloky budou vyhodnocovat. Stačí jen procházet program po tocích. Další výhodou tohoto rozšíření je, že není třeba aby měl uzel výstup, aby se vyhodnotil. Díky tomu není třeba aby měl print výstup, který by stejně nedával moc smysl. Poslední charakteristikou, kterou toto rozšíření přináší je, že program nepotřebuje vracet žádnou hodnotu a jeho celá podstata tak může být ve vedlejších efektech.

Příklad 21 (Program s vedlejšími efekty pomocí toků)
Jednoduchá kalkulačka na odečítání dvou čísel zadaných uživatelem.

- $N = \{Init, Read_1, Read_2, -, Print, out\}$
- $C = \{\langle Read_1, - \rangle_1, \langle Read_2, - \rangle_2, \langle -, Print \rangle_3, \langle -, out \rangle_4\}$
- $F = \{\langle Init, Read_1 \rangle, \langle Read_1, Read_2 \rangle, \langle Read_2, Print \rangle, \}$
- $eval$:
  - $eval(x, y) = \{\emptyset : eval'(x, y)\}$
  - $eval'(Read_n, args) =$ Načti číslo z konzole $\mid n \in \mathbb{N}$
  - $eval'(Print, args) =$ Vypiš argument do konzole
  - $eval'(-, args) = args_{menšenec} - args_{menšitel}$
  - $eval'(out, args) = args_{\emptyset}$
- $OUT = \{out\}$
- $P = \{\langle C_1, \text{target}, \text{menšenec} \rangle, \langle C_2, \text{target}, \text{menšitel} \rangle\}$
- $plane_{calc} = \langle N, C, eval, OUT, P \rangle$
- $VPL = \langle \{plane_{calc}\}, plane_{calc} \rangle$



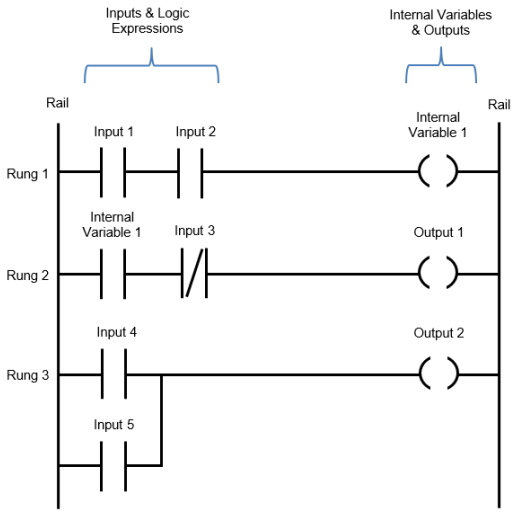
Obrázek 2: Příklad 21 v editoru

3 Přehled existujících technologií dle využití

Vizuální technologie se využívají v hodně odvětvích, kde i často bývají upřednostňované před textovým programováním. Některé z těchto odvětví si v této kapitole vyjmenujeme i s příklady. Je velice důležité zachytit jejich silné stránky abychom mohli zjistit co je potřeba k navržení kvalitního univerzálního VPL.

3.1 Průmyslová Automatizace

Časté využití vizuálních jazyků při programovaní PLC. V automatizaci se využívá více druhů vizuálních programovacích jazyků. VPL využívané pro automatizace jsou ladder diagram, function block diagram a sequential function chart. Důvodem pro využití VPL v tomto oboru je jejich jednoduchost a tedy nižší požadavky na programátorské schopnosti návrháře takovýchto systémů. Například v ladder diagramu (obrázek 3) jsou programy velice podobné elektrickým schématům. Tohle je značnou výhodou pro programátory PLC, kteří často mají elektrotechnické vzdělání.



Obrázek 3: Ladder Logic Diagram[3]

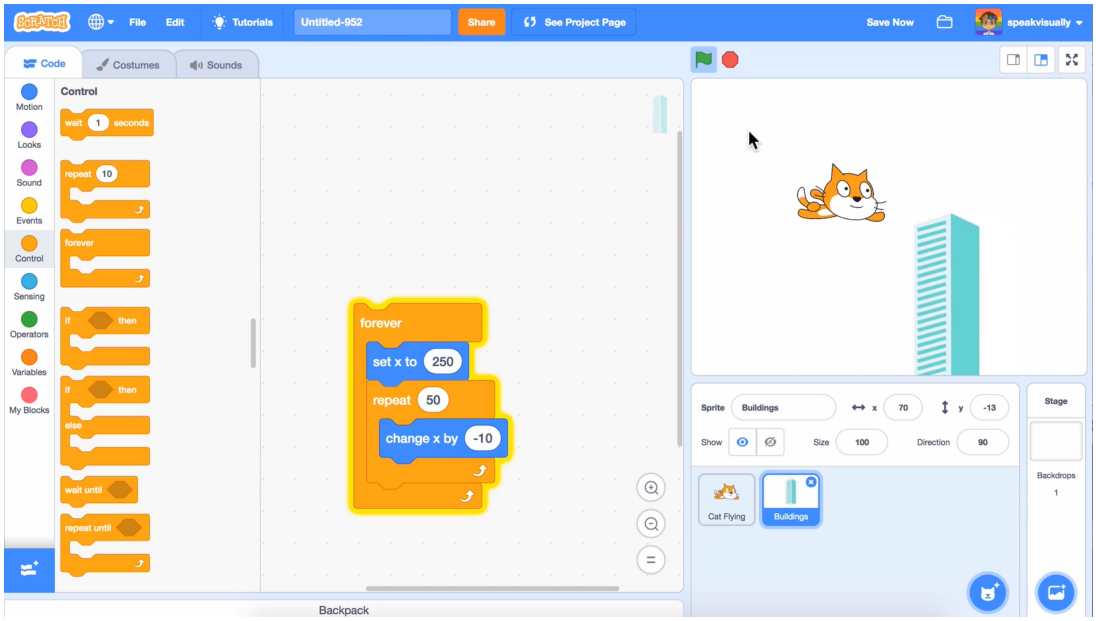
3.2 výuka programování

Abstraktní koncepty využívané při programování mohou být pro některé lidi a hlavně pro děti velice komplikované a těžce uchopitelné. Proto je velice dobré, že VPL nabízejí vytváření programů hravou formou propojování bloků, které jim připomíná hraní si se stavebnicemi.

3.2.1 Scratch

Scratch je populární blokový vizuální jazyk cílený pro výuku dětí. Už v březnu roku 2022 počet uživatelů dosáhl skoro 1 milionu a průměrný věk uživatelů se pohybuje kolem 12 let. [4] Vývojové prostředí se dělí na dvě části. První část je VPL, který využívá technologii blockly. V této části uživatel jednoduše pomocí drag-n-drop přetahuje bloky kódu z palety na okraji na pracovní plochu, kde z nich tvoří programy. Druhou část tvoří určitá forma želví grafiky. V této druhé části uživatel přidává sprity, které jsou v podstatě objekty z objektově orientovaného programovaní. V této části se také programy spouští.[5]

Scratch ani vzdáleně není využitelný pro praktické využití. Není zde možnost práce s poli a složitějšími datovými strukturami, nemá moc nástrojů pro snižování redundance kódu a tak dále. V tom, na co je ale Scratch vytvořený, vyniká dobře a je dobrým úvodem do programovaní mladým lidem.



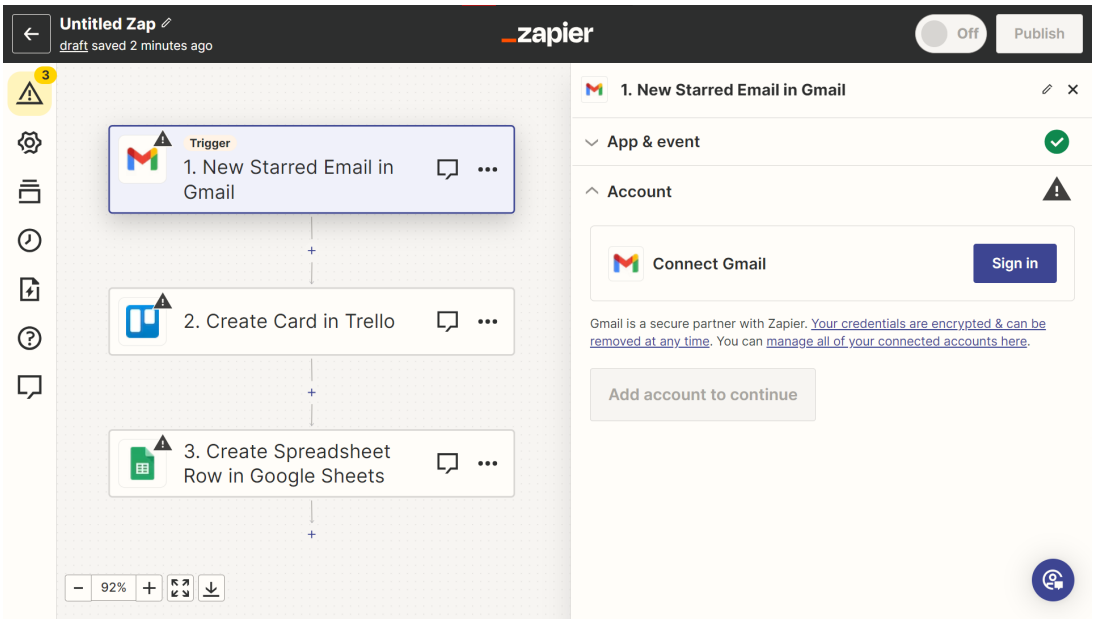
Obrázek 4: Scratch (tutorial Make it Fly) [5]

3.3 Skriptování a automatické procesy

Nároky normálních domácností ale i menších podniků neustále vzrůstají. Ačkoliv je dnes skoro na vše aplikace, tak často se objevují velice specifické požadavky lidí, které by se vývojářům nevyplatili do aplikací přidávat. Proto začínají vznikat různé nástroje, které pomáhají s drobnými automatizacemi jako můžou být vytváření poznámek, zálohování a práce se soubory, propojování webových služeb a více.

3.3.1 Zapier

Zapier je webový automatizační nástroj, který umožňuje propojení různých aplikací a služeb. Zapier zjednodušuje automatizaci opakujících se úkonů mezi různými online platformami, jako jsou e-maily, sociální sítě, cloudové úložiště, CRM systémy a další. Automatizace se provádí pomocí takzvaných zapů. Každý zap má trigger, který ho spustí, a akce, které po spuštění provede. Veškeré vytváření a editace zapů je pomocí vizuálního editoru a jedná se tak o no-code nástroj.



Obrázek 5: Zapier jednoduchý zap

3.4 Ve hrách

Videohry se staly rozsáhlým fenoménem dnešní doby, který oslovil velké množství lidí. Hry mají rozsáhlé množství žánrů. Jedním z takovýchto žánrů mohou být například logické hry, které mohou podpořit zájem o řešení logických a matematických problémů i v reálném životě. Jako jedním z podžánrů logických her bychom pak mohli označit hry s programováním v různých formách. Tyto hry pak mohou dát možnost hráčům v kontrolovaném prostředí objevovat algoritmizaci a problematiky s ní spojené. Hráči pak mohou získat díky takovýmto hrám zájem o technické obory a mají tak popularizační potenciál. Nejedná se však pouze o logické a programovací hry které mohou mít takový efekt. Existují i hry které nabízí programovací prvek jako vedlejší herní mechaniku které není důležitá pro dokončení hry.

Takto můžeme hry s určitým popularizačním efektem rozdělit na programovací a algoritmické hry a Hry s možností programování. Programovací a algoritmické mají cíl kterého se dosáhne přímo nějakou formou programovaní nebo

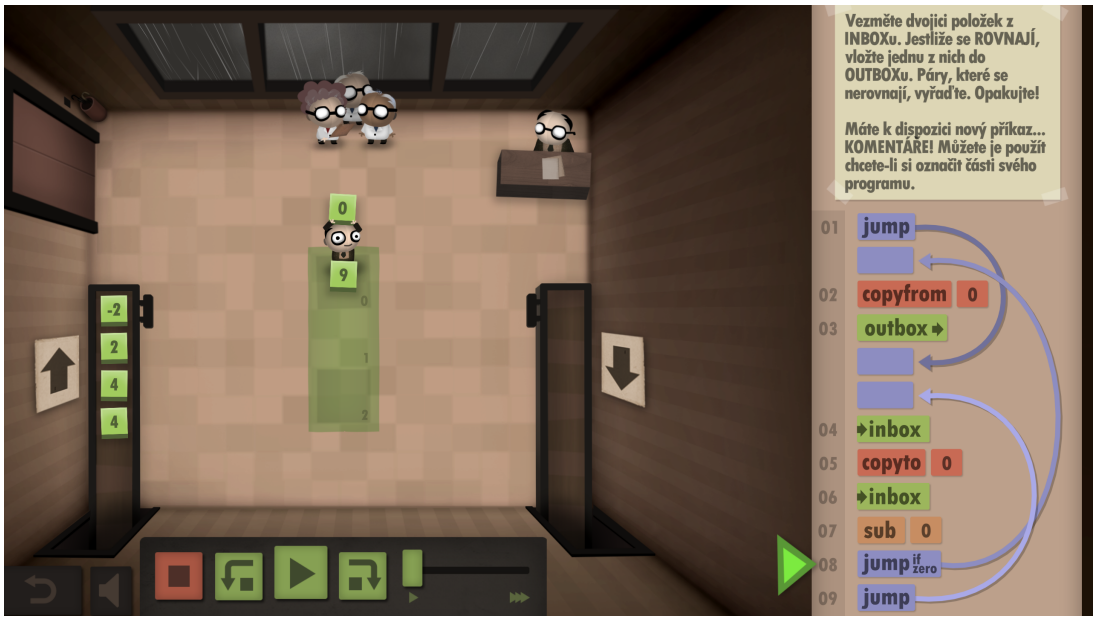
algoritmizace. Pro hráče to znamená že už musí mít nějaký zájem o řešení hádanek, aby si hru chtěl zahrát. Mezi tyto hry patří například Human resource machine, Opus Magnum, Infinifactory, TIS-100 a Autonauts. Hry s možností programování pak nemají primární herní zážitek spojený s programováním. Při hraní těchto her se často objeví nějaká možnost zpříjemnění herního zážitku, která už ale vyžaduje naučení se alespoň základů algoritmizace. V důsledku tak může díky tomu hráč objevit svůj zájem o programování. Nejznámější hrou v této kategorii je Minecraft.

3.4.1 Human resource machine

Human Resource Machine je počítačová hra, kterou tvoří jednoduché až složitější programovací výzvy. Můžeme ji tedy zařadit do kategorie programovací a algoritmické hry. Využívá vizuálního blokového kódu, ze kterého hráč skládá kód tak, aby splnil zadání úkolu. Kód vykonává pracovník podatelny, který podle něj má zpracovat příchozí balíky. Hra nabízí možnost krokování a komentování kódu. Zároveň pro každé řešení spouští na pozadí více různých testovacích scénářů. Díky tomu není možné, aby hráč vytvořil hardcode řešení pro aktuální vstup.

Popularizační potenciál této hry spočívá v líbivé grafice a jednoduchých prvních úrovních hry. Takto hra zjednodušuje zejména mladým lidem intuitivně začít vnímat algoritmizační problémy krůček po krůčku. Hra naučí hráče v abstraktním smyslu pochopit základy cyklů, podmínek a práce s pamětí.

Hra má také pokračování 7 Billion Humans. V tomto pokračování hráč vytváří program, který už ale vykonává větší množství zaměstnanců naráz. Díky tomu je pokračování velice zajímavým úvodem do paralelizace.



Obrázek 6: Human resource machine

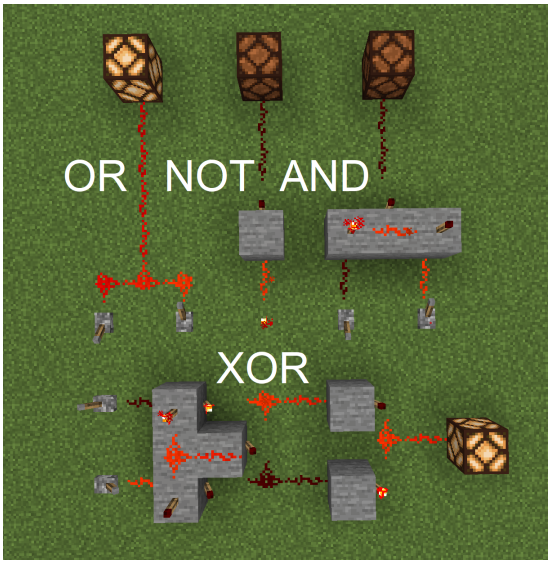
3.4.2 Minecraft

Celosvětově známá hra s otevřeným procedurálně generovaným světem s možností programování. Hra sama o sobě nemá cíl jako takový a jedná se o takzvané pískoviště. Neboli hráč si své cíle ve hře určí sám. Celá hra je tvořená kostkami, které se nazývají bloky. Hráč může bloky libovolně zvedat a pokládat a tvořit tak různé stavby a podobně.

Už v roce 2011 začali být do hry přidávané různé Redstone předměty a mechanismy. Redstone si můžeme představit jako nějaký drát který je základem pro logické obvody. Ze začátku nešlo dělat s redstonem mnoho. Šly jen vytvářet logické brány a otevírat dveře. Postupem času se ve hře začali objevovat další předměty až do fáze kdy jde ve hře vytvořit cokoliv. Lidé vytváří z redstone různé MIDI přehrávače, mikropočítače nebo dokonce velice minimalistickou ale hratelnou verzi samotného minecraftu.

Důležitý je fakt, že redstone není hlavním prvkem hry. Hráč hru objeví a chce si ji zahrát, protože si chce postavit dům, farmařit a podobně. Nemusí mít vůbec žádný zájem o objevování číslicové techniky. Při stavbě svého domečku může ale hráč chtít mít třeba dveře, které jdou otevírat z obou stran pomocí tlačítka. Zkoumání takových problémů může pokládat hodně zajímavých otázek, které eventuálně mohou vyústit v hráčův zájem o programování.

Jako bonus je pak neustále zjednodušovaná modovatelnost hry. Díky tomu může s touto hrou hráč projít celou cestou z neznalosti základních logický funkcí až po psaní skutečného kódu pro modování hry.

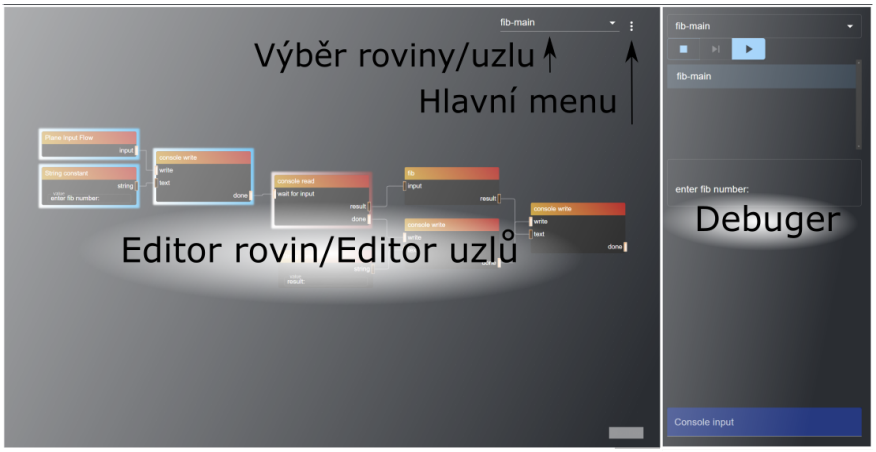


Obrázek 7: Minecraft jednoduché obvody

4 Editor

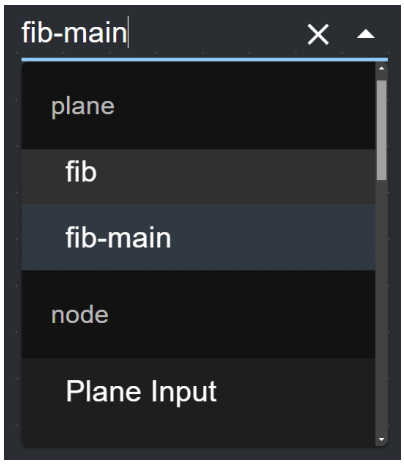
Editor je velice jednoduchý. Jeho cílem bylo co nejjednodušší prototypování samotného jazyka. Má tři důležité části editor rovin, editor uzlů a debuger jak je vidět na obrázku 8. Fungování odpovídá definici VPL s toky a se speciální formu IF.

Každý uzel je buď rovina anebo vychází z editovatelné definice, které obsahuje informace o vstupech, výstupech, implementaci a další. Speciální jsou uzly jádra. Tyto uzly nejsou editovatelné a mají speciální význam. Jedná se o IF který je speciální forma IF. Input flow který je Init toků. A pak jsou to plane input/output které přidávají rovině (a jejímu uzlu) vstupy a výstupy.



Obrázek 8: Rozložení editoru

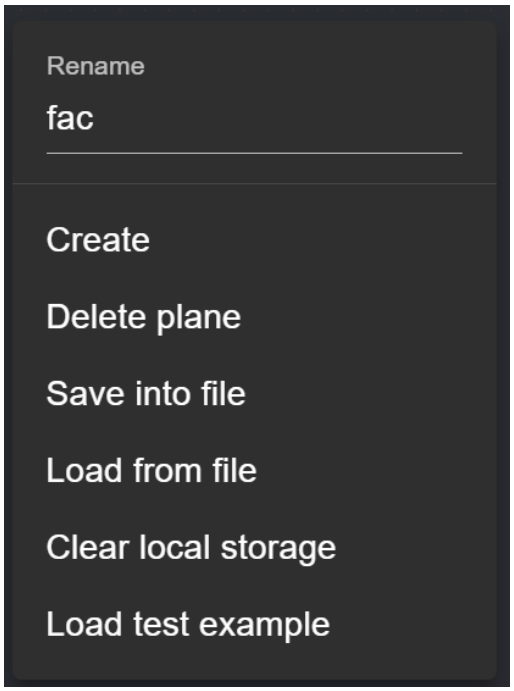
Výběr roviny (obrázek 9) a uzlu určuje které zobrazení bude editor zrovna mít. Výběr umožňuje textové vyhledávání. Při zrušení výběru křížkem se dostaneme do menu, která umožňuje vytvořit nový uzel nebo rovinu.



Obrázek 9: Výběr roviny/uzlu

V hlavní menu (obrázek 10) se nachází dvě kontextové akce k aktuálnímu vybranému uzlu/rovině a ostatní akce nezávislé na kontextu. Tyto akce jsou:

- *Rename* ... Ihned přejmenuje aktuální rovinu/uzel
- *Create* ... Zavře aktuální rovinu/uzel a zobrazí tak menu k vytvoření nové roviny nebo uzlu
- *Delete plane/node* ... Smaže aktuální rovinu/uzel. Vyžaduje potvrzení.
- *Save into file* ... Stáhne celý program ve formátu json.
- *Load from file* ... Umožní nahrát stažený program ve formátu json.
- *Clear local storage* ... Vymaže aktuální uložená data. Po znovunačtení stránky bude editor prázdný.
- *Load test example* ... Přepíše aktuální program s ukázkovým programem.



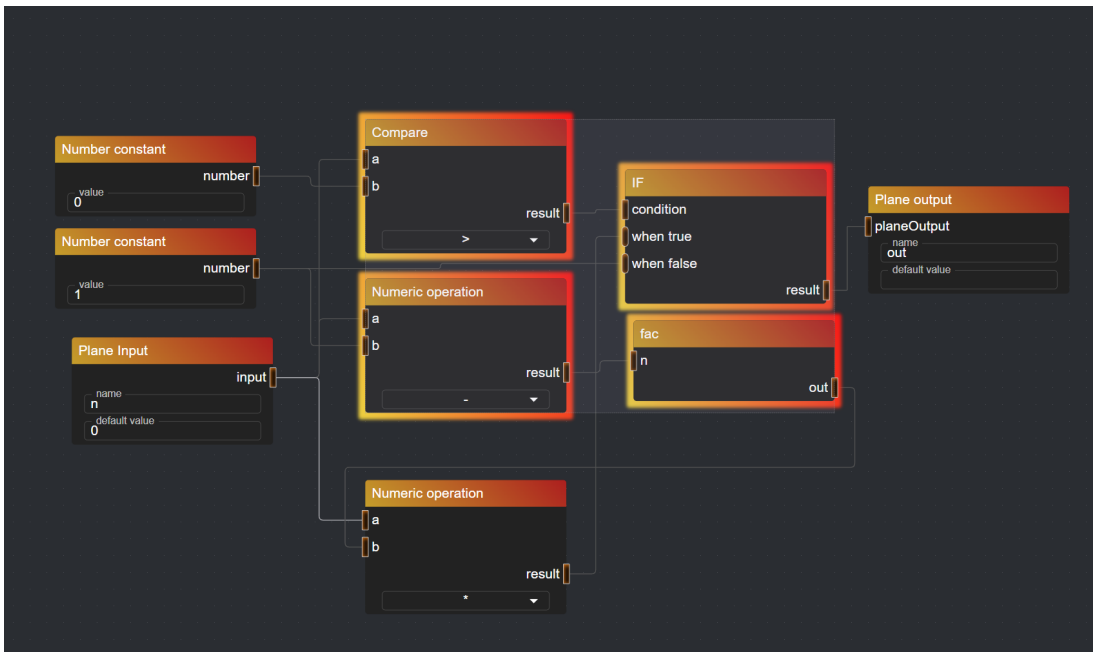
Obrázek 10: Hlavní menu

4.1 Editace rovin

Nové uzly je možné přidávat kliknutím na prázdné místo v editoru rovin. Následně se zobrazí kontextové menu s vyhledáváním, kde si uživatel vybere, který uzel bude následně vložen. Uzly i propojení je možné vybírat a vybrané uzly

a propojení jsou zvírazněné. Vybrané uzly je pak možné přetahováním myši přesouvat. Smazání vybraných uzlů a propojeních je možné pomocí klávesy *backspace*. Uzly jsou tvořené vstupy, výstupy a textem nebo výběrem. Výstup a vstup uzlu je možné propojit tažení z jednoho konce s ukončením tahu v druhém. Texty a výběry jdou normálně vyplňovat jako ve standardních formulářích.

Veškeré změny provedené v programu se vždy hned uloží do localStorage prohlížeče. To znamená že po načtení stránky jsou všechny změny zachované.



Obrázek 11: Editace rovin

4.2 Editace uzlů

Na obrázku 12 je editor uzlů, v kódu označený jako architect, je nástroj pro přidávání nových uzlů které jsou primitiva jazyku. Jeho části jsou:

- *ID* ... Needitovatelné ID pro odkazování na uzel. Nemá v podstatě význam pro uživatele.
- *Name* ... Jméno uzlu viditelné v nadpisu v editoru rovin.
- *Docs* ... Dokumentace uzlu. Aktuálně se nikde nezobrazuje.
- *Interpret* ... Logika uzlu.
- *Interpret compiled* ... Ukázka jak vypadá celý kód který je pak vyhodnocený funkcí eval.

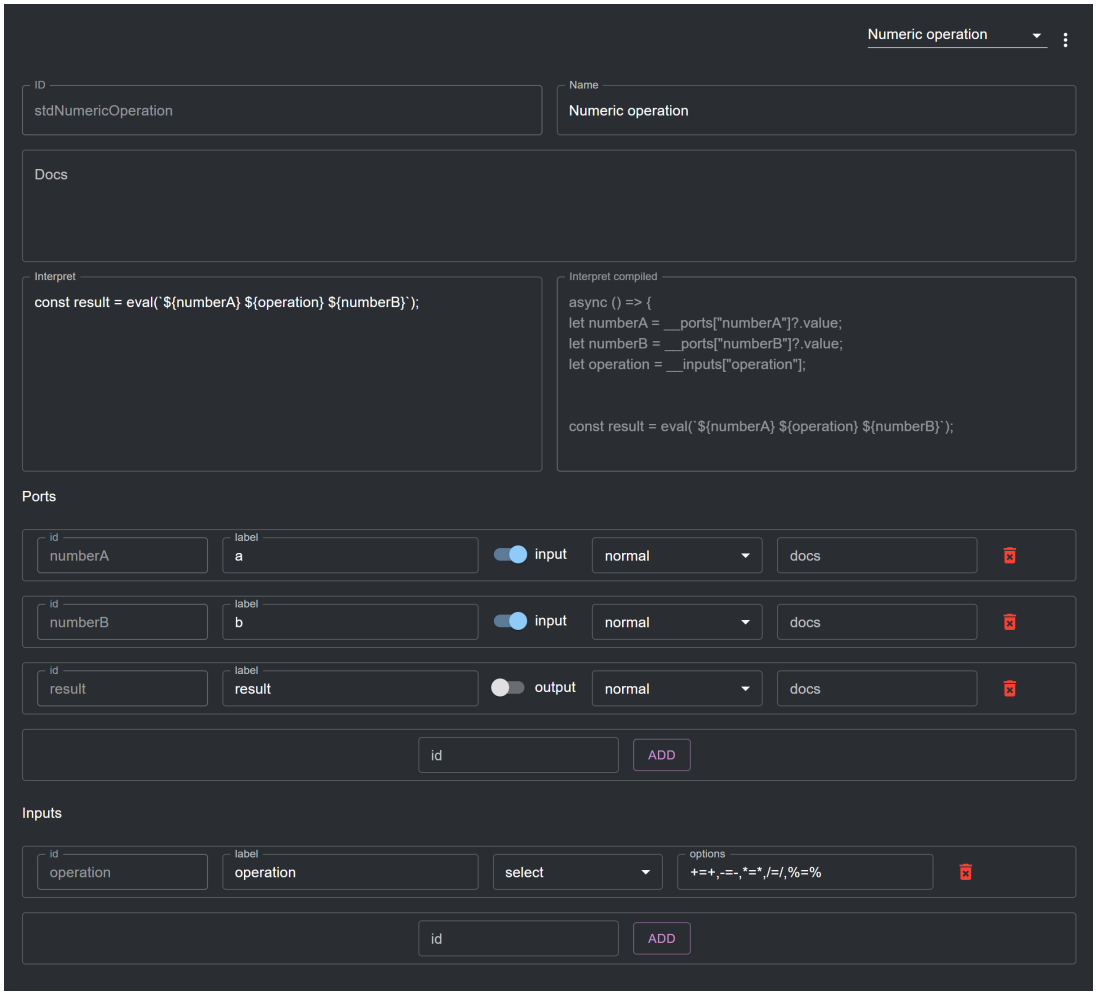
- *Ports* ... Porty s následujícím částmi
  - *id* ... Needitovatelné ID pro odkazování na port. Využívá se při interpretaci.
  - *label* ... Popis portu viditelný v editoru roviny.
  - *output/input* ... Určuje, jestli se jedná se o vstup nebo výstup.
  - *typ* ... Typ portu. Normal je klasický typ, flow je tok.
  - *docs* ... Dokumentace portu. Aktuálně se nezobrazuje.
  - *delete* ... Tlačítko na odstranění portu.
  - *Poslední řádek* ... Možnost přidání dalšího portu s daným ID.
- *Inputs* ... Uživatelské vstupy s následujícím částmi
  - *id* ... Needitovatelné ID pro odkazování na vstup. Využívá se při interpretaci.
  - *label* ... Popis vstupu viditelný v editoru roviny.
  - *typ* ... Typ vstupu. Text je libovolný text, select je výběr z možností.
  - *options* ... Pouze pro select. Možnosti zobrazované v selektu. Možnosti jsou ve formátu "popis=hodnota" oddělené čárkou.
  - *delete* ... Tlačítko na odstranění vstupu.
  - *Poslední řádek* ... Možnost přidání dalšího vstupu s daným ID.

Při vyhodnocování je kód zadaný v *interpret* zabalen v asynchronní funkci, která je pak volána pomocí eval. Při pohledu do kompilované funkce je možné vidět jak se vstupy a porty dekonstruují v lokální proměnné a následně konstruují. Kód počítá s tím, že veškeré výstupní proměnné budou vytvořeny.

4.3 Debuger

Na pravé straně editoru se nachází debuger. Debuger tvoří následující části:

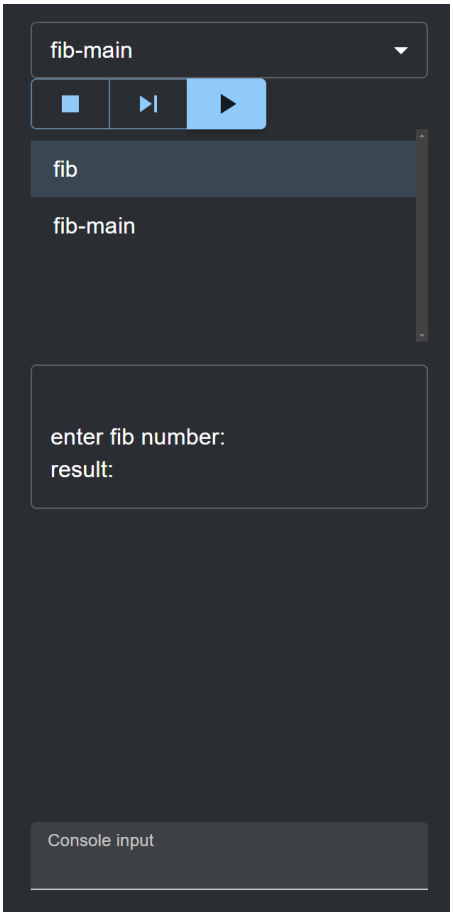
- Výběr roviny ke spuštění
- Tlačítka:
  - *Stop* ... Zastavení aktuálního krokování.
  - *Další krok* ... Další krok, nebo začít krokování.
  - *Start* ... Spustit program bez krokování.
- Zásobník vyhodnocovaní rovin pro krokování
- Výstup konzole
- Vstup konzole



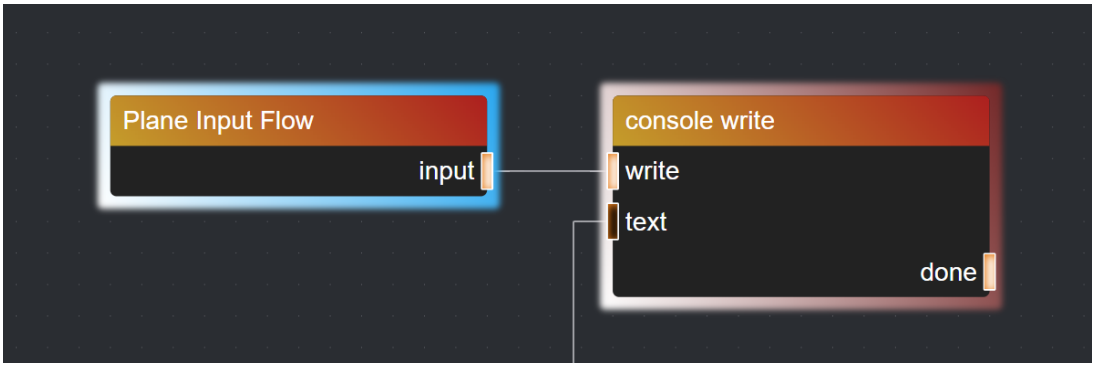
Obrázek 12: Editace uzlů

Program se dá spustit buď v režimu krokování nebo spustit celý naráz. Pro jednoduchost je možné v programu využívat konzoli debugeru pomocí uzlů v příkladech console write a console read. Vsup konzole se očekává, když je Console input modrý a má nadpis *waiting for input*. Vstup do konzole se potvrdí klávesou enter.

Při krokování se zobrazuje na zásobníku postupné vyhodnocování rovin. Překlikáváním je možné vybrat konkrétní instanci roviny a prozkoumávat hodnoty jaké má. Najetím na konkrétní uzly během krokování se zobrazí jejich aktuální hodnoty. Při vyhodnocování jsou vyhodnocené uzly zvýrazněné světle modrou barvou (levý uzel na obrázku 14). Uzly určené k vyhodnocení jsou zvýrazněné tmavě červenou (pravý uzel na obrázku 14).



Obrázek 13: Debuger



Obrázek 14: Zvýraznění uzlů při krokování

5 Požadavky na VPL a jeho editor

V rámci experimentu vytvoření vlastního univerzálního jazyka jsem se pokusil prozkoumat jednu z možných cest pro vizuální jazyky. Vytvořený editor cílil na prozkoumávání možnosti návrhu jazyka, aby byl jednoduchý a úplný. V praxi jsou ale jazyky textové i vizuální posuzované nejen podle jazyku samotných, ale i podle jejich dostupných nástrojů. Mezi největší takovýto nástroj patří hlavně editory, které často spojují další nástroje dohromady.

Ačkoliv jsme si prokázali schopnost vytvořit jednoduché programy s využitím navrženého (jazyka a) editoru, tak je tento editor stále daleko od použitelnosti v praxi. V této kapitole bude zmíněné alespoň některé z aspektů, které se objevují v současných textových jazycích a mají vliv na jejich použitelnost. Výčet těchto vlastností pak může být jistým návrhem kam s editorem a jazykem dál a ve kterých směrech je možné jej zlepšovat.

5.1 Modularita

V dnešní době je i jednoduchý program složený z komplikovaných částí. Proto je důležité, aby měl jazyk schopnost nějakým způsobem vytvářet abstrakce a shlukovat kód.

Shluky podobného kódu můžeme pak nazvat knihovnami nebo moduly. Takový modul může být například sada matematických funkcí, nebo práce s DOM elementy. Takto není potřeba řešit pořád dokola stejné problémy a můžeme místo toho můžeme v programu využít modul, pokud existuje.

Dalším prvkem modularity je abstrakce. Může se jednat o různé prvky jazyka umožňující nám skrýt nějaké implementační detaily a používat nějaké složitější funkcionality bez potřeby hlubší znalosti. Mezi takovéto abstrakce můžeme zařadit například funkce, makra, třídy nebo v případě vizuálního jazyk uzly, bloky, roviny a další.

V Případě editoru máme možnost přidávat vlastní uzly se svojí logikou. Z těchto uzlů pak můžeme vytvářet roviny které jsou taky znovupoužitelné. Díky tomuto máme abstrakci umožněnou a je to prvním krokem k rozšíření editoru o možnost vytváření a práci s moduly.

5.2 Flexibilita

Pod flexibilitou jazyka si můžeme představit přizpůsobivost jazyka k řešení problému různými způsoby. Každý problém je pohodlnější řešit jiným přístupem. Proto když chceme univerzální programovací jazyk, který řeší různé sady problémů, tak je dobré, aby umožňoval tyto problémy řešit co možná nejvíce způsoby. Díky tomu si může programátor vybrat způsob který je vyhovující aktuálnímu problém

Jedním z hlavních způsobů flexibility dnešních jazyků je možnost programovat ve více paradigmatech. Velice časté jsou dnes objektově orientované jazyky, které zároveň umožňují funkcionální přístup. Tyto jazyky často mají preferované

paradigma a ty ostatní paradigmata nemusí být tak pohodlné na užívání, a tak dobře optimalizované. Navzdory tomu si může programátor zvolit z více přístupů.

Další možný způsob, jak může být jazyk flexibilní je schopnost vytvářet DSL jako součást hlavního kódu. DSL neboli *domain specific language* je jazyk zaměřený na řešení specifických problémů. Jako DSL můžeme označit například HTML, různé modelovací jazyky, účetní systémy a podobné. Univerzální jazyk pak může tvořit moduly které se často dají označit za DSL. Takovéto moduly poznáme tak, že řeší velice specifický problém velice pohodlným způsobem nepoznatelným od skutečného DSL.

Navržený jazyk umožňuje vytváření výrazů což podporuje deklarativní přístup, ale i vytváření příkazů čímž podporuje i imparativní přistup. Pro podporu funkcionálního přístupu potřebuje navržený jazyk přistupovat k uzlům jako k hodnotám. Umožňovat předat uzel jako hodnotu anebo vrátit volatelný uzel jako hodnotu. Pro podporu objektového přístupu pak jazyk potřebuje určitou koncepci zapouzdření. K tomu potřebujeme objekty, které jsou v podstatě jen spojení dat a kódu.

K rozšíření navrženého jazyka o funkcionální nebo objektové programování je potřeba zodpovědět velké množství otázek. Jakým způsobem se budou předávat uzly jako vstup? Jak by vypadalo, kdybychom chtěli vyhodnotit výstupní uzel? Co je objekt a jak by měl vypadat? Jak reprezentovat stav objektu, tak aby se s objektem dalo přehledně pracovat? Tyto a další otázky mají různé odpovědi které provází hodně zajímavé důsledky. Dává proto smysl se těmito otázkami také zabývat.

Schopnost navrženého jazyku k vytváření DSL také stojí za další zkoumání.

5.3 Robustnost

Robustnost jazyka nemusí znamenat pouze stabilitu programu jako takového. V důsledku jsou často robustní jazyky zároveň i efektivnější pro vývoj. Velkou část robustnosti jazyka tvoří statické analýzy. Díky ní můžeme odhalovat chyby ještě dřív, než spustíme samotný program. Statická analýza prováděna nějakým programem, a proto se může zdát, že je možné takovýto program napsat pro libovolný jazyk. Přesto je dobré, aby jazyk šel naproti možnostem provádění statické analýzy a umožnil tak tyto analýzy co nejjednodušeji provádět.

Typové systémy můžou být velkou podporou pro detekci chyb pomocí statické analýzy. Staticky typované jazyky potřebují mít své typy správně aby byl schopný z nich kompilátor určit výslednou podobu kódu. Dynamicky typované jazyky jsou na druhou stranu flexibilnější a rychlejší pro prototypování. Na druhou stranu se dynamicky typované jazyky otevírají možným chybám, které by se díky kontrole typů odhalili. Proto je dnešním trendem rozšiřovat dynamicky typované jazyky o typový systém. Takovéto rozšíření umožňuje dřívější detekci chyb, ale často nemá žádný vliv na běh programu.

Dalším aspektem robustního kódu je způsob práce s chybovými stavy. Velice častý je přistup výjimek s try catch bloky. Výjimku může způsobit jakékoliv vo-

lání kódu. Na programátorovi pak je, aby kód zaobalil v try bloku a vyřešil eskalaci chyby v catch části. Další častý způsob je pomocí návratových hodnot často s podporou monád. V tomto případě si v kódu vždy vymezíme speciální hodnoty pro chybové stavy. Pokud funkce vrátí jednu z těchto hodnot, pak víme že došlo k chybě. Tuto chybu pak můžeme jednodušeji propagovat pomocí monád. Ve výsledku může být pak kód s monádami docela podobný přístupu s výjimkami.

Navržený jazyk je konstruovaný tak, aby byl, pokud možná platformě a jazykově agnostický. Proto by bylo dobré najít takový typový systém, aby měl, pokud možná co největší synergii s existujícími textovými jazyky. Jedním z možných přístupů může být zkoumání nástrojů na převod kódu mezi jazyky. Díky tomu si pak budeme moci vyvodit závěry ohledně podobností dnešních jazyků a vyvodit tak vhodný typový systém, aby vyhovoval nejvíce z nich.

Řešení chybových stavů v navrženém jazyku je pouze formou návratové hodnoty. Vývojář sám zajistí že uzel doběhne správně. Zde by mohlo dojít k problému, když se nějaký uzel nevyhodnotí správně. Jedním z řešeních je přidat chybový výstup všem rovinám a uzlům. Pokud nějaký uzel nebude mít ošetřený chybový stav a dojde k chybě, tak ukončí vyhodnocování roviny a vypropaguje chybu dál. V takovémto případě může být ale více zkomplikované řízení vyhodnocování programu. Tento přístup je podobný výjimkám. Alternativě přístup podobný monádám by mohl vypadat následovně. Uzel, který se má vyhodnotit se vyhodnotí vždy. Pokud dojde k chybě při jeho vyhodnocování na výstupech se objeví speciální hodnota. Na tuto hodnotu buď může uzel reagovat anebo ji přepošle dál na své výstupy místo svého standardního vyhodnocení. V tomto případě nám budou dělat problémy toky. Tok může být buď pravda pokud se má napojený uzel vyhodnotit nebo nepravda pokud ne. Tímto ale přidáváme třetí hodnotu, která není specifikovaná a musíme se rozhodnout, jak by se jazyk na takovouto hodnotu choval.

Posledním významným aspektem robustonsti v kontextu VPL je testovatelnost kódu. Vzhledem k tomu, že textové jazyky často nemusí moc dělat pro to, aby byly testovatelné, tak budeme předpokládat, že nebudou žádné specifické požadavky ani pro jazyky vizuální.

5.4 Čitelnost

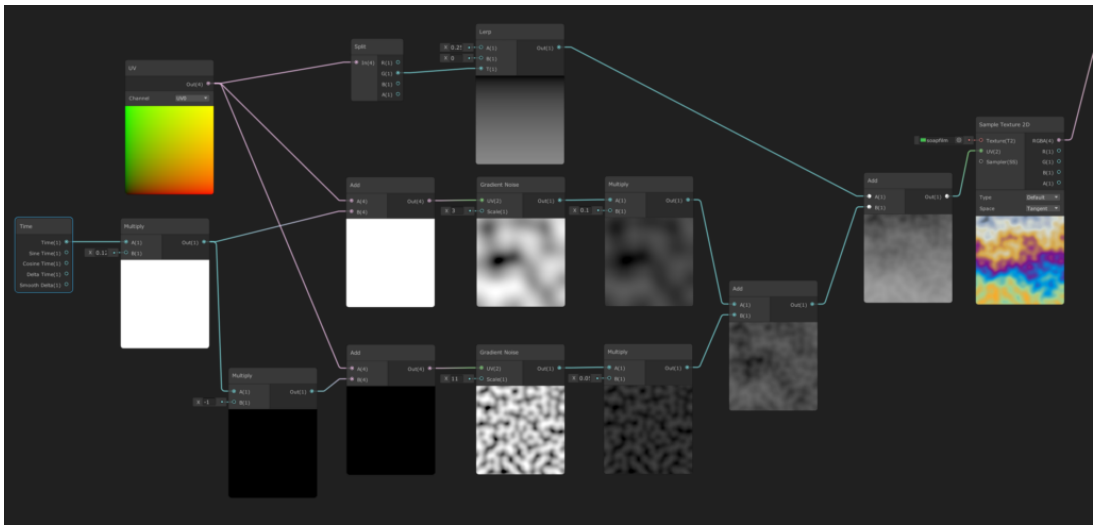
Všechny předchozí vyjmenované aspekty mají výrazný vliv na čitelnost jazyka. Například typový systém nám může hodně prozradit o tom, jak nějakou funkci používat. Za dobu používání textových jazyků neustále vznikají různé tipy a konvence, jak udržovat kód čitelný. Naproti tomu vizuální jazyky jsou v některých směrech komplikovanější. Jedná se o umisťování bloků v, dvojrozměrném prostoru. Samotné umístění nemá na vyhodnocování vliv, ale ovlivňuje orientaci v kódu. Pro diagramové jazyky se může stát, že propojení nebudou vždy vidět anebo jdou špatně rozpoznat.

Navržený jazyk nemá zatím žádné nástroje, které by pomáhali zvýšit čitelnost. Čitelnost můžeme zvýšit například umožněním manuálního určení kudy

vedou propojení. Značením propojení podle významu, například barva a další vlastnosti podle typu. To samé platí pro uzly.

5.5 Debuging

Jednou z nejsilnějších stránek VPL pochází z jejich vizuální podstaty. Díky tomu dokážou být dobré k zobrazovaní hodnot při krokování. Při vývoji můžeme využít pozicování bloků a využít jejich rozmístění. Pak nám může samotný kód posloužit jako dashboard. Tento efekt je velice silný například u vytváření shaderů, jak je vidět na obrázku 15.



Obrázek 15: Nástroj Shader Graph v herním enginu Unity[6]

5.6 Ovladatelnost

Poslední ale velice důležitá vlastnost se týká pouze editoru. Editory nabízí často mnoho akcí, co je možné provádět. Jsou to různé editace, spouštění programu nebo přepínání mezi různými soubory. S takovým množstvím akcí je důležité mít propracovaný systém, jak se uživatel k těmto akcím dostane. Jedním takovým způsobem mohou být klávesové zkratky. Mimo to také dnešní editory nabízí například i možnost využívat pouze klávesnici bez potřeby myši. Díky tomu můžou vývojáři zvýšit svou efektivitu a nemusí kvůli žádné akci opouštět prostor klávesnice.

Prvním krokem k vylepšení editoru ve směru ovládání by mohla být integrace do existujícího textového editoru jako je například VSCode. Jednotlivé roviny mohou být jednotlivé soubory. Tím dostaneme základní rozložení a ovládání rozhraní a mnoho už existujících výhod hotového editoru. Dále by bylo dobré se zaměřit na ovládání editoru pouze klávesnicí.

Závěr

Cílem diplomové práce bylo prozkoumat svět vizuálního programování. Podařilo se experimentálně vytvořit jednoduchý univerzální programovací jazyk, který pomohl osvětlit problematiku univerzálních vizuálních jazyků jako celku. Výsledkem je zjištění, že vizuální jazyky sdílí velké množství problematik s textovými jazyky. Mimo to mají navíc vizuální jazyky své vlastní další problémy. V důsledku bych do budoucna přistupoval k problematice z opačného pohledu. Místo vytváření vizuálního jazyka jako samotného by bylo možná lepší pokusit se jej vytvořit jako nádstavbu nad existujícím textovým jazykem. Tak se dá vyhnout velkému množství problematik které mají už textové jazyky dávno vyřešené a při vývoji se dá soustředit pouze na problémy spojené s VPL.

Všeobecně vizuální jazyky nejsou zdaleka tak populární jako ty textové. Naopak v některých odvětvích jsou silně preferované doménově specifické jazyky. Silné zastoupení mají například v automatizaci a výuce. Další formy vizuálního programování a algoritmizace můžeme najít v počítačových hrách. V tomto případě může jít o určitou formu popularizace programovaní.

Conclusions

The aim of this thesis was to explore the world of visual programming. I have succeeded in experimentally creating a simple universal programming language that helped to find out some of the issues of universal visual languages as a whole. The result of experiment was discovery, that visual languages share a large number of issues with textual languages. Visual languages have their own additional problems. As a result, in the future I would approach the problem from the opposite perspective. Instead of creating a visual language as a language on its own, it might be better to try to create it as an extension over an existing textual language. This way, we can avoid a large number of issues that text languages have already solved and focus only on VPL-related problems.

In general, visual languages are not nearly as popular as textual ones. On the contrary, domain-specific languages are strongly preferred in some industries. Vizual languages are strongly represented in automation and education for example. Other forms of visual programming and algorithms can be found in computer games. In this case, it may be a form of popularisation of programming.

A Použité technologie

Pro vývoj jsem se rozhodl vybrat webové technologie. Díky jejich flexibilnímu zobrazování jsou vhodné pro vývoj VPL. Jako bonus může pak být využívání VPL v prohlížeči při připojení na vzdálený server.

A.1 Node.js

Node.js je opensourcové multiplatformní běhové prostředí. Je postavené na na V8 JavaScript enginu který je i jádro Google Chrome. Umožňuje tak vytvářet aplikace v JavaScriptu s jinými cíli, než jen prohlížeč jako jsou například servery, desktopové a mobilní aplikace a dále. Hlavní výhodou node.js je, že umožňuje psát kód který poběží jak v browseru, tak na serveru.

A.2 TypeScript

TypeScript je nadmnožinou JavaScriptu, což znamená, že veškerý validní JavaScriptový kód je i validním TypeScriptovým kódem. TypeScript přidává do Javascriptu typový systém, díky kterému je možné provádět statické analýzy a je tak vhodnější na tvorbu středních a velkých projektů.

A.3 Vite

Vite je sada nástrojů pro vývoj frontendu. Jeho hlavními vlastnostmi jsou rychlý developement server s hot-reloadem a bundlování.

A.4 React

React je populární JavaScriptová knihovna pro vývoj webových aplikací. Kód v Reactu se snaží přiblížit podobě html které pak vykresluje. V Reactu je možné si vytvářet vlastní komponenty, ve kterých se snadno propojí logika s vykreslenými elementy.

A.5 React flow

React flow je knihovna pro vytváření interaktivních diagramů a editorů v Reactu. Knihovna je plně přizpůsobitelná a je vhodná pro vytváření statických diagramů až po složitější editory. Obsahuje základní akce jako drag-n-drop přesouvání a spojování uzlů, zoomování, výběr a další. Ostatní akce mohou být tvořeny pomocí proměnných a callbacků předávaných jako argument ReactFlow komponenty.

A.6 Zustand

Zustand je JavaScriptová knihovna pro práci se stavem aplikace. Z podstate může být zpráva stavů react aplikace dost komplikovaná. Například můžeme narazit na problém kdy chceme data z komponenty využívat v hodně zanořeném potomku komponenty nebo v sourozencích. V takových případech je potřeba aby každý React komponent ve struktuře, který je mezi zdrojem a cílem dat, data předával dál. Takto může kód strašně rychle narůstat, navíc je pak komplikované měnit strukturu komponent.

Zustatnd umožňuje vytvořit globální stav. K tomuto globálnímu stavu můžeme přistupovat a měnit ho odkudkoliv. K přístupu z komponenty má zustand speciální funkci, kterou je možné vybrat data, které komponenta využívá a při jejich změně se komponenta vždy překreslí.

A.7 Pkg

Pkg je balíčkovací systém. Umožňuje zabalit node.js aplikaci do jednoho spustitelného souboru. Díky tomu není potřeba sdílet zdrojové kódy ani cokoliv dalšího instalovat na stroji uživatele. Ke spuštění zabalené aplikace není potřeba ani samotný node.js.

B Vývojářská příručka

Tato kapitola obsahuje základní informace k vývoji a sestavení aplikace.

Pro spuštění a kompilaci ze zdrojových kódu je potřeba:

1. nainstalovat node podle webu https://nodejs.org
2. nainstalovat yarn pomocí `npm install -g yarn`
3. spustit v kořenovém adresáři projektu `yarn install`

Alternativně jsou součástí elektronických dat instalátory a zálohy knihoven.

1. rozbalit `node_modules.zip` do kořenového adresáře projektu
2. nainstalovat node ze složky `dev_install/node/`
3. následně místo `yarn` používat `npm run yarn` - Tento přístup může být velice nestabilní proto doporučuji instalovat yarn globálně.

B.1 Spuštění pro vývoj

Díky hot-reloadu se všechny změny v kódu ihned objevují v aplikaci. Stačí mít spuštěný vývojový server který se spouští pomocí příkazu `yarn dev`. Server pak běží na `http://localhost:5173/`. V případě že by byl port 5173 už zabraný, vite zvolí jiný a adresu vypíše do terminálu.

B.2 Kompilace

Kompilace je plně automatizovaná. Stačí spustit příkaz `yarn dist`. Výsledné spustitelné soubory se nachází ve složce `./dist/`.

C Obsah elektronických dat

`bin/`
: Spustitelné soubory aplikace pro windows, linux a mac. Nevyžaduje žádné dodatečné závislosti.

`docs/`
: Text práce ve formátu PDF, včetně všechny souborů potřebných pro bezproblémové vygenerování PDF dokumentu textu

`src/`
: Zdrojové kódy VPL editoru. Podrobněji v příloze C.1.

`scripts/`
: Pomocné skripty. Spouštěné pomocí programu yarn. Obsahuje skript pro sestavení spustitelného souboru VPL editoru pomocí knihovny pkg.

`readme.txt`
: Instrukce pro instalaci a spuštění VPL editoru, včetně všech požadavků pro jeho bezproblémový provoz.

`dev_install/`
: Obsahuje instalátory a zálohy knihoven pro sestavení programu.

C.1 Struktura zdrojových kódů

`server/`
: Zdrojový kód pro minimalistický server pro spouštění z jednoho souboru.

`src/components/`
: Zdrojové kódy komponent pro knihovnu reactjs. Obsahuje veškerou logiku vykreslování a zobrazování.

`src/store/`
: Zdrojové kódy akcí a struktury globálního stavu aplikace. Využívá knihovnu zustand. Globální stav je rozdělený ve složce slices. Každý slice obsahuje definici části stavu a logiky akcí k němu patřící. Hlavní slice je code který obsahuje editace kódu VPL a editor slice který provádí akce editoru jako je například výběr zobrazené roviny.

src/logic/

Zdrojové kódy hlavní logiky editoru. Mají minimální závislost na store a components a externích knihovnách. Hlavní algoritmus podobný vyhodnocovacím algoritmům 3 a 2 se nachází v `/interpreter/interpreter.ts`. Definice základních uzlů a typů VPL se nachází v `/vpl/`. Zde se dá podle typů odvodit struktura uloženého JSON programu. Poslední podstatnou částí je soubor `/examples.json` který obsahuje ukázkový program.

Literatura

[1] HOSICK, Eric (ed.). *Visual Programming Languages - Snapshots*. 2014. 103 s. Dostupný z: ⟨\http://blog.interfacevision.com/design/design-visual-progarmming-languages-snapshots/⟩.

[2] KUHAIL, Mohammad Amin; FAROOQ, Shahbano; HAMMAD, Rawad; BAHJA, Mohammed. Characterizing Visual Programming Approaches for End-User Developers: A Systematic Review. *IEEE Access*. 2021, roč. 9, s. 14181–14202. Dostupný také z: ⟨\http://dx.doi.org/10.1109/ACCESS.2021.3051043⟩.

[3] *Ladder Logic Programming*. Dostupný také z: ⟨\https://ladderlogicworld.com/ladder-logic-programming/⟩.

[4] *Scratch statistics*. 2023. Dostupný také z: ⟨\https://scratch.mit.edu/statistics/⟩.

[5] *Scratch*. 2023. Dostupný také z: ⟨\https://scratch.mit.edu/⟩.

[6] SMEENK, Roland. *Blowing bubbles with shader graph*. 2020. Dostupný také z: ⟨\https://smeenk.com/blowing-bubbles-with-shader-graph/⟩.