



**MATEMATICKO-FYZIKÁLNÍ
FAKULTA**
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Petr Martinek

Vizualizace změny kódu

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Vladan Majarech, Dr.

Studijní program: Informatika

Studijní obor: Obecná informatika

Praha 2022

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

Rád bych poděkoval Mgr. Vladanovi Majarechovi, Dr. za odborné vedení práce, za dobré rady, připomínky, trpělivost i podporu při vypracování práce. Dále bych rád poděkoval Pavlovi Dolečkovi z firmy Apify za návrh tématu, úvod do JavaScriptu a podporu během vývoje projektu.

Název práce: Vizualizace změny kódu

Autor: Petr Martinek

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Vladan Majarech, Dr., Katedra teoretické informatiky a matematické logiky

Abstrakt: Cílem této práce je navrhnout a implementovat program, který na základě dvou podobných verzí zdrojového kódu napsaného v JavaScriptu vygeneruje animaci ve formátu GIF, zobrazující postupný přepis první verze do druhé. Dále je cílem navrhnout tento program tak, aby byl rozšiřitelný o další jazyky.

K dosažení tohoto cíle je definována zjednodušená reprezentace zdrojového kódu, která je nezávislá na jazyce, a pro každou uvažovanou úpravu v rámci této reprezentace je definovaná cenová funkce. Na základě této cenové funkce je vyhledána nejlevnější posloupnost přepisů, podle které je konečná animace vygenerována.

Klíčová slova: gramatiky programovací jazyky stromová struktura

Title: Visualization of code changes

Author: Petr Martinek

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Vladan Majarech, Dr., Department of Theoretical Computer Science and Mathematical Logic

Abstract: The goal of this thesis is to design and implement a program, which, given two similar versions of source code written in JavaScript, generates a GIF animation showing a step-by-step process of rewriting the first version into the other. Another goal is to design the program in a way, so that the list of supported languages can be extended.

To achieve this goal, a simple, language-independent representation of source code is defined and for every considered transformation within this simplified representation, a cost function is defined. Based on this cost function, the cheapest sequence of transformations is found, which is then used to generate the final animation.

Keywords: grammars programming languages tree structure

Obsah

Úvod	3
Cíl a struktura práce	3
Inspirace a předcházející práce	3
1 Teoretický základ	6
1.1 Formální gramatiky	6
1.2 Struktura a překlad zdrojového kódu	6
1.2.1 Lexikální úroveň	6
1.2.2 Syntaktická úroveň	6
1.2.3 Sémantická úroveň	7
2 Použité technologie	8
2.1 Node.js	8
2.2 GraphicsMagick	8
2.3 JSDoc	8
2.3.1 Dokumentace tříd	9
2.3.2 Dokumentace vlastností třídy	9
2.3.3 Dokumentace metod a funkcí	10
2.4 Knihovny	10
2.4.1 Antlr4	10
2.4.2 Gm	10
2.4.3 Js-levenshtein	11
3 Analýza	12
3.1 Specifikace programu	12
3.2 Shrnutí specifikace	12
3.3 Jednotlivá rozhodnutí	13
3.3.1 Tokenizovaný text	13
3.3.2 Zjednodušená reprezentace kódu	13
3.3.3 Rozsah identifikátoru ve zjednodušené reprezentaci	17
3.3.4 Úpravy v rámci zjednodušené reprezentace	17
3.3.5 Cena úpravy	20
3.3.6 Průběh animace	21
3.3.7 Reprezentace úprav	21
3.3.8 Algoritmy nalezení posloupnosti úprav	24
3.3.9 Uživatelské rozhraní	24
3.3.10 Formát mezivýstupu	25
3.3.11 Rozšiřitelnost	25
3.3.12 Vzhled animace	26
4 Vývojová dokumentace	27
4.1 Samodokumentace	27
4.2 Přehled architektury programu	27
4.3 Tok dat	28
4.3.1 Režim úplného běhu	28

4.3.2	Režimy částečného běhu	29
5	Uživatelská dokumentace	31
5.1	Instalace	31
5.1.1	Program GraphicsMagick	31
5.1.2	Prostředí Node.js	31
5.1.3	Program Code-shift	32
5.1.4	Instalace knihoven	32
5.2	Omezení programu	32
5.3	Základní použití	33
5.4	Pokročilé použití	33
5.4.1	Mezivýstup	34
5.4.2	Více vstupních souborů	34
5.5	Úprava mezivýstupu	35
5.6	Význam jednotlivých argumentů	38
6	Testování programu	39
6.1	Testování běhu programu	39
6.2	Testování překladu	39
7	Budoucí práce	41
	Závěr	42
	Seznam použité literatury	43
	Seznam obrázků	44
A	Přílohy	45
A.1	Zdrojový kód	45
A.2	Specifikace	45

Úvod

Cíl a struktura práce

Cílem této práce je návrh a implementace programu Code-shift, který na základě dvou verzí podobného zdrojového kódu automaticky vygeneruje GIFovou animaci, která zobrazuje postupný přepis první verze zdrojového kódu do druhé. Výsledný program by měl umět porovnávat zdrojové kódy napsané v JavaScriptu a měl by být rozšiřitelný o další jazyky.

Hlavní myšlenkou pro vznik tohoto programu je potenciální využití v internetových výukových materiálech zaměřených na vývoj software. Doplnění návodu o názorné animace vyobrazující co se v každém kroku změnilo by pomohlo studentům následovat popsané změny lépe, s menší pravděpodobností omylného vynechání některého z kroků. Automatické generování těchto animací, docílené tímto programem, by pro změnu pomohlo tvůrcům těchto návodů.

V práci je nejprve představena použitá terminologie, která se převážně týká formálních gramatik a překladačů. Následně jsou představeny technologie použité k implementaci tohoto programu.

V třetí kapitole je problém podrobně rozebrán a jeho řešení je navrženo. Nej důležitější součástí návrhu je definice ceny úpravy zdrojového kódu. Tato definice umožní definici metriky na zdrojových kódech, kde metrika je daná nejlevnější posloupností přepisů, které kód přepíšou z jedné verze do druhé. Další důležitou součástí návrhu je definice zjednodušené reprezentace kódu. Použití zjednodušené reprezentace kódu dělá všechny hlavní algoritmy jazykově nezávislé a umožní rozšiřitelnost programu o další jazyky. Při návrhu řešení jsou diskutovány alternativní návrhy, které byly při vývoji historicky implementovány, ale které jsou ve výsledném programu nahrazeny.

Zbývající kapitoly se zabývají uživatelskou i vývojovou dokumentací a mírou, do jaké byl program testován.

Inspirace a předcházející práce

Námět pro tento projekt byl původně inspirován dvěma příspěvky na sociální síti Twitter. Oba příspěvky souvisely s animací strukturovaného textu.

Autorem prvního příspěvku je Omar Rizwan a příspěvek je dostupný na adrese <https://twitter.com/rsnous/status/1367305117293178881>. Příspěvek obsahuje animaci, ve které se text objektu reprezentovaného ve formátu JSON transformuje z neodsázeného textu na odsázený a naopak.

Autorem druhého příspěvku je Henry Zhu a příspěvek je dostupný na adrese https://twitter.com/left_pad/status/1367941962083471361. Animace

prezentovaná v příspěvku zobrazuje převod stejného objektu mezi dvěma formáty. Během této animace dochází k přesunu společného textu a odstranění a přidání ostatního textu.

Pro účel zachování jsou animace z původních příspěvků shrnuty níže na obrázcích 1 a 2. Na základě těchto dvou příspěvků vyvstal nápad, zda a do jaké míry je možné animovat změny ve strukturovaném textu, specificky v JavaScriptovém, či jiném zdrojovém kódu. Na tomto nápadu je založena tato práce.

Co se týče jiných, rozsáhlejších podobných prací předcházející této práci, jen stěží se dohledávají. Asi nejdůležitější částí této práce je hledání změn mezi zjednodušenými reprezentacemi kódu, tedy mezi stromovými strukturami. Jedním ze známých problémů týkajících se hledání rozdílů ve stromové struktuře je problém *tree-diff*. Pro tento problém existuje několik implementací v různých jazycích. Hlavní odlišností funkce hledání změn v Code-shiftu od problému *tree-diff* je skutečnost, že se Code-shift neomezuje pouze na přidávání a odebrání vrcholů stromu, ale připouští víc úprav týkajících se zdrojového kódu.

Z asi nejpodobnější dílo lze považovat program *git-diff-tree*. Jedná se o jednu z utilit spadající pod verzovací systém Git, která porovnává struktury adresářů a soubory obsažené v nich, včetně jejich obsahu. Tato utilita je velmi pokročilá, jelikož umí v adresářových strukturách detekovat přejmenování, přidání a odstranění souboru a změny v jednotlivých souborech na úrovni slov pomocí *word-diffu* či *line-diffu*. Funkce hledání změn v Code-shiftu také detekuje přidání, odstranění či přepis bloků kódu, což lze připodobnit operacím na adresářích a souborech. Hlavní odlišností je to, že Code-shift také detekuje přejmenování proměnných a funkcí (či jiných objektů s identifikátorem). Tato operace nemá žádný vhodný ekvivalent v *git-diff-tree*. Další odlišností je filozofie za zkonstruováním funkce určující vzdálenost zdrojových kódů. Odlišnosti obou děl jsou zřejmě dané případem použití. Mezitím co git počítá s libovolným typem souborů, Code-shift je vyvíjen specificky pro text strukturovaný jako zdrojový kód, dostatečně podobný jazyku C. Dokumentace programu *git-diff-tree* je dostupná na adrese <https://git-scm.com/docs/git-diff-tree>.

```

{"manifest_version":2,"name":"TabFS","description":"Mount your browser tabs as a filesystem","version":"1.0","permissions":["tabs","tabCapture","debugger","nativeMessaging","management","unlimitedStorage","<all_urls>"],"background":{"scripts":["vendor/browser-polyfill.js","background.js"],"persistent":true},"browser_specific_settings":{"gecko":{"id":"tabfs@rsnous.com"}}}

```

```

{
  "manifest_version": 2,
  "name": "TabFS",
  "description": "Mount your browser tabs as a filesystem",
  "version": "1.0",
  "permissions": [
    "tabs",
    "tabCapture",
    "debugger",
    "nativeMessaging",
    "management",
    "unlimitedStorage",
    "<all_urls>"
  ],
  "background": {
    "scripts": [
      "vendor/browser-polyfill.js",
      "background.js"
    ],
    "persistent": true
  },
  "browser_specific_settings": {
    "gecko": {
      "id": "tabfs@rsnous.com"
    }
  }
}

```

Obrázek 1: Shrnutí videa v tweetu od Omara Rizwana

```

const profile = (
  <>
    
    <h3>{[user.firstName, user.lastName].join(" ")}</h3>
  </>
);

```

```

const profile = /*#__PURE__*/_jsx(Fragment, {
  children: [/*#__PURE__*/_jsx("img", {
    src: "avatar.png",
    className: "profile"
  })],
  }, /*#__PURE__*/_jsx("h3", {
    children: [user.firstName, user.lastName].join(" ")
  })
]);

```

```

const profile = /*#__PURE__*/_jsx(Fragment, {
  children: [/*#__PURE__*/_jsx("img", {
    src: "avatar.png",
    className: "profile"
  })],
  }, /*#__PURE__*/_jsx("h3", {
    children: [user.firstName, user.lastName].join(" ")
  })
]);

```

Obrázek 2: Shrnutí animace v tweetu od Henryho Zhu

1. Teoretický základ

Tato kapitola představí terminologii nezbytnou pro plnohodnotné porozumění práci.

1.1 Formální gramatiky

Znalost formálních gramatik a s nimi spojené terminologie je pro tuto práci očekávaná. Terminologie týkající se gramatik je použita v úvodu terminologie týkající se struktury zdrojového kódu a je dále používána v textu. Důležité termíny jsou následující: Bezkontextová gramatika, terminál, derivační strom, jazyk, slovo a generování slova. Terminologie je používána tak, jak je definována v knize *Automaty a gramatiky*, Chytil (1984).

1.2 Struktura a překlad zdrojového kódu

Zdrojovým kódem ve většině programujících jazyků je strukturovaný text, který lze rozčlenit na jednotlivé souvislé úseky různými způsoby. Tato sekce představí lexikální a syntaktické členění zdrojového kódu a bude také diskutovat sémantické vlastnosti zdrojového kódu, které jsou pro tuto práci relevantní. Terminologie je přebrána z knihy *Compilers: Principles, Techniques, and Tools*, Aho a kol. (2006) s tím rozdílem, že místo termínu „lexikální analyzátor“ je použit zkrácený termín „lexer“.

1.2.1 Lexikální úroveň

Na lexikální úrovni je text členěn do takzvaných tokenů. Pojem token označuje terminál gramatiky uvažovaného programovacího jazyka. Většinou se jedná o minimální podřetězec znaků zdrojového kódu, ve kterém žádný z vlastních podřetězců tokenu sám o sobě význam nenesou, ale celý token už význam nese. Například v JavaScriptu klíčové slovo „var“ označuje deklaraci proměnné a jedná se tedy o token, zatímco podřetězce „v“, či „ar“ žádný význam nenesou.

Proces, při kterém je text zdrojového kódu rozčleněn na tokeny se nazývá lexikální analýza. Modul provádějící lexikální analýzu se nazývá lexikální analyzátor, zkráceně lexer.

Některé znaky, většinou bílé znaky a znaky které jsou součástí komentářů, se během lexikální analýzy vypouští a nejsou tedy součástí žádného tokenu. V této práci se bude při diskusi algoritmů na vypuštěné znaky také nahlížet jako na tokeny, jelikož jsou součástí výsledné animace a při zpracování zdrojových kódů je třeba s jejich přítomností počítat.

1.2.2 Syntaktická úroveň

Na syntaktické úrovni jsou jednotlivé tokeny zdrojového kódu uspořádány do stromové struktury. Tato stromová struktura přímo odpovídá derivačnímu stromu zdrojového kódu derivovaného podle gramatiky programovacího jazyka, ve kterém je daný kód napsán.

Proces tvorby tohoto derivačního stromu se nazývá syntaktická analýza a modul provádějící syntaktickou analýzu se nazývá parser. Pokud zdrojový kód není slovem generovaným bezkontextovou gramatikou daného programujícího jazyka řekneme, že zdrojový kód obsahuje syntaktickou chybu.

1.2.3 Sémantická úroveň

Na sémantické úrovni se pracuje s významem zdrojového kódu jako celku. Při běžném překladač zdrojového kódu do strojového kódu překladač provádí takzvanou sémantickou analýzu, jejíž cílem je odhalit sémantické chyby. Obecně řečeno, sémantické chyby ve zdrojovém kódu jsou takové chyby, které nelze odhalit v rámci syntaktické analýzy při použití bezkontextové gramatiky.

Code-shift sémantiku zdrojového kódu využívá při překladač zdrojového kódu do jeho zjednodušené reprezentace (viz str. 13). Hlavní sémantickou vlastností kódu je pro Code-shift rozsah identifikátoru (např. jména proměnné, třídy, či funkce). Rozsahem identifikátoru se rozumí část zdrojového kódu, ve které daný identifikátor označuje stejný definovaný objekt. Například v následujícím C# kódu oba výskyty identifikátoru `x` označují stejnou proměnnou a oba výskyty tedy patří do jednoho stejného rozsahu identifikátoru `x`:

```
int AddFive(int x)
{
    return x + 5;
}
```

V následujícím příkladu se vyskytují čtyři identifikátory `x` a dva různé rozsahy tohoto identifikátoru. Každý z obou rozsahů odpovídá právě jedné ze dvou funkcí.

```
int AddFive(int x)
{
    return x + 5;
}
int AddSix(int x)
{
    return x + 6;
}
```

Jedním z omezení programu Code-shift je nepřesná reprezentace rozsahu identifikátoru. Proto po uvedení tohoto omezení v podkapitole *Rozsah identifikátoru ve zjednodušené reprezentaci* (viz str. 17) bude jakékoliv použití termínu „rozsah identifikátoru“ označovat tento aproximovaný rozsah. Pojem „přesný rozsah“ bude označovat rozsah proměnné jak byl definován výše.

2. Použité technologie

2.1 Node.js

Program Code-shift je naprogramovaný v jazyce JavaScript. JavaScript je interpretovaný jazyk a Code-shift tedy potřebuje vhodný interpreter pro běh. K tomuto účelu je použit program Node.js.

Hlavním využitím tohoto programu je zprostředkování webových služeb na straně serveru. Alternativně Node.js slouží i jako program pro spouštění čistě lokálních programů napsaných v JavaScriptu. K běhu Code-shiftu je právě využita lokální varianta běhu.

S Node.js souvisí i správce balíčků Npm. Tento správce balíčků je instalován s instalací Node.js a umožňuje stažení knihoven pro JavaScriptové programy. Během vývoje programu dochází za předpokladu použití Npm k automatické aktualizaci souboru package.json, který uživatel vyvíjeného programu využije k snadné instalaci potřebných knihoven. Knihovny použité v programu Code-shift jsou instalovány právě prostřednictvím tohoto správce balíčků.

Node.js je dostupný pod specifickou licencí, popsané v repositáři tohoto projektu: <https://github.com/nodejs/node/blob/master/LICENSE>

2.2 GraphicsMagick

GraphicsMagick (dále jen GM) je program určený k práci s různými formáty obrázků. GM umí upravovat obrázky, a to aplikováním filtrů, či vykreslováním geometrických útvarů a textu. GM dále umí tvořit nové obrázky pomocí dříve zmíněných transformací a také umí převádět mezi formáty.

GM je dostupný pod licencí MIT, ale některé z knihoven, které GM používá jsou licencované licencemi GPL a LGPL. Instalační soubory programu jsou dostupné na adrese <https://sourceforge.net/projects/graphicsmagick/>. Oficiální webová stránka projektu je <http://www.graphicsmagick.org/>.

2.3 JSDoc

JSDoc je název dokumentačního jazyka JavaScriptového kódu a také název programu který umí na základě těchto komentářů automaticky generovat dokumentaci programu ve formátu HTML. Jazyk JSDoc je dokumentován na webové stránce <https://jsdoc.app/>. V dokumentačních komentářích Code-shiftu byl tento jazyk použit v následujícím rozsahu:

2.3.1 Dokumentace tříd

Dokumentací třídy je popis k čemu tato třída slouží. Tento popis je obsažen v dokumentačním komentáři. Dokumentační komentář je uvozen znaky „/**“ a končí znaky „*/“.

Pokud má být třída použita pouze jako abstraktní¹, dokumentační komentář navíc obsahuje tag `@abstract`.

2.3.2 Dokumentace vlastností třídy

Vlastnosti třídy jsou dokumentovány při jejich prvním výskytu v definici třídy. Protože JavaScript je dynamicky typovaný jazyk, součástí dokumentace je popis a volitelně i typ. Jméno dokumentované vlastnosti není součástí komentáře. Náležitost dokumentačního komentáře k dané vlastnosti je dána tím, že daná vlastnost následuje bezprostředně za daným komentářem. Dokumentační komentář vlastnosti má následující tvar:

```
/**
 * Popis.
 * @type {Typ}
 */
```

Při specifikaci typu je použita následující syntax:

- Výrazy „*“ a „any“ označují libovolný datový typ.
- Pokud se jedná o primitivní datový typ JavaScriptu (např. `number`, `string`, `object`), či v programu definovanou třídu, za „Typ“ v dokumentaci dosadíme název primitivního datového typu nebo třídy.
- Pokud se jedná o objekt datového typu `T` nebo `U`, jako typ v dokumentaci uvedeme „`T | U`“.
- Pokud se jedná o objekt datového typu `T`, pro přehlednost můžeme uvést typ „`(T)`“.
- Pokud se jedná o pole obsahující neznámý počet prvků datového typu `T`, v dokumentaci uvedeme typ „`T []`“.
- Pokud se jedná o prázdné pole, jako typ uvedeme „`[]`“.
- Pokud se jedná o pole obsahující předem známý počet prvků datových typů `T1`, `T2`, ..., `Tn`, v dokumentaci uvedeme typ „`[T1, T2, ..., Tn]`“.
- Pokud se jedná o poslední parametr metody, který zastupuje libovolné množství parametrů typu `T` (takzvaný „rest“ parametr), v dokumentaci uvedeme typ „`...T`“.²

Například výraz `[string, *] []` označuje pole neznámé délky, které obsahuje pouze dvojice, ve kterých je prvním prvkem textový řetězec a druhým prvkem je cokoliv.

¹V JavaScriptu neexistují modifikátory přístupu.

²Narozdíl od pole se známým obsahem jsou zde tři tečky součástí syntaxe.

2.3.3 Dokumentace metod a funkcí

Dokumentací metody či funkce (dále jen funkce) je popis toho, co daná funkce dělá, jaké jsou její parametry, jakého typu tyto parametry jsou a k čemu tyto parametry slouží. Dokumentace také zahrnuje návratovou hodnotu, její typ a popis, pokud funkce něco vrací. Dokumentační komentář předchází definici funkce a začíná popisem dané funkce. Po popisu funkce následují popisy parametrů a po nich následuje popis návratové hodnoty, pokud existuje. Dokumentace parametru a návratové hodnoty mají následující tvar:

```
@param {Typ} jmeno Popis.  
@returns {Typ} Popis.
```

Jméno parametru je od potenciálně víceslovného popisu parametru odděleno bílým znakem. Specifikace typu je v obou případech stejná jako při specifikování typu vlastnosti.

2.4 Knihovny

2.4.1 Antlr4

Antlr4 je nástroj pro automatické generování lexerů a parserů pro formální gramatiky. Kromě toho je Antlr4 také souborem knihoven pro různé programovací jazyky, poskytující funkce pro práci s vygenerovanými lexery či parsery.

Jako samostatný program Antlr4 dostává na vstupu soubor s definicí gramatiky a informaci o cílovém programovacím jazyce a na základě těchto informací vygeneruje lexer či parser pro danou gramatiku, napsaný v cílovém programovacím jazyce.

V programu code-shift byl využit parser a lexer pro JavaScript, vygenerovaný právě tímto nástrojem. Pro to, aby tento automaticky generovaný kód fungoval, je v programu zahrnuta i příslušná knihovna antlr4. Knihovna Antlr4 je dostupná pod licencí BSD.

Automaticky generovaný lexer a parser JavaScriptu jsou použity k vytvoření derivačního stromu, který je potom využit k překladu původního zdrojového kódu do zjednodušené reprezentace (viz str. 13).

Soubory gramatiky JavaScriptu jsou pod licencí MIT dostupné v repozitáři <https://github.com/antlr/grammars-v4/tree/master/javascript/javascript>.

2.4.2 Gm

Gm je knihovna, pomocí které lze vygenerovat vstup pro samotný program GraphicsMagick za použití snadno srozumitelných funkcí. GraphicsMagick je v programu použit k vygenerování výstupního GIFu. Nejprve GraphicsMagick vygeneruje jednotlivé snímky animace prostřednictvím funkcí vykreslujících text a následně jsou všechny snímky spojeny do jedné animace pomocí funkce pro převod formátu. Knihovna Gm je dostupná pod licencí MIT.

2.4.3 Js-levenshtein

Js-levenshtein je JavaScriptová knihovna obsahující velmi efektivní implementaci výpočtu Levenštejnovy vzdálenosti. V programu je použita kdykoliv, kdy je třeba spočítat vzdálenost mezi dvěma textovými řetězci, které nenesou žádnou informaci navíc. Pro vizualizaci přepisu, při kterém je levenštejnova vzdálenost používána, tato funkce není využita, protože pro návrh animace je relevantní matice pocházející z výpočtu levenštejnovy vzdálenosti a vstupní text navíc nese informaci o barvě každého znaku (viz str. 26). Js-levenshtein je dostupný pod licencí MIT.

3. Analýza

3.1 Specifikace programu

Původním nápadem na projekt byla pouze vizualizace postupné změny JavaScriptového kódu. Projekt sám o sobě nebyl nějak specifikován. Z tohoto důvodu byla sepsána a odsouhlasena specifikace projektu, která je dostupná v souboru „Specifikace.pdf“ jako součást přílohy. Přiložený text je ve své první verzi, jelikož nikdy nebyl aktualizován. Původní specifikace obsahuje 4 nerozhodnuté položky, které byly následně rozhodnuty.

- Byly přidány dvě nové možnosti vstupu oproti původní specifikaci – možnost vypsaní nápovědy a možnost přepínající program do režimu testování překladu. Ve specifikaci je zmíněna možnost implementace vstupního argumentu, který určuje závažnost chyby, ze které se program ještě zotaví. Tento argument nebyl implementován.
- Jako formát mezivýstupu byl použit JSON.
- Program, který byl použit pro generování GIFů byl GraphicsMagick. S tímto programem byla použita i související JavaScriptová knihovna Gm.
- Vzhled výstupní animace byl rozhodnut a je popsán níže v této kapitole.

Oproti původní specifikaci došlo také k některým změnám.

- Vstupní přepínač „-n“ ve specifikaci je implementován jako „-f“.
- Část používané terminologie se změnila. Termín „sémantický strom“ či „sémantická reprezentace“ byl nahrazen termínem „zjednodušená reprezentace“.
- Ve specifikaci je zmíněna existence třech typů vrcholů ve zjednodušené reprezentaci kódu. V dokončené práci jsou čtyři typy vrcholů.
- Program nebyl otestován na stroji s operačním systémem Linux.
- Za dokumentaci díla jsou považovány části této práce a samodokumentace kódu.

3.2 Shrnutí specifikace

Z 4. kapitoly původní specifikace programu lze odvodit následující funkční požadavky:

1. Program umí načíst zdrojový kód a převést ho do zjednodušené reprezentace pomocí ručně psaných pravidel pro daný jazyk.
2. Program umí porovnat dva zdrojové kódy ve zjednodušené reprezentaci a odvodit seznam změn, které převedou první zdrojový kód na druhý.

3. Program umí vygenerovat a načíst JSON soubor se změnami, aby byl umožněn ruční zásah do algoritmu hledání změn.
4. Program umí nalezený seznam změn vizualizovat pomocí GIFové animace.
5. Program umí pracovat specificky s JavaScriptem.

Mimofunkční požadavky plynoucí ze specifikace jsou:

- Program musí být rozšiřitelný o další jazyky.
- Program je napsaný v JavaScriptu a spouští se prostřednictvím Node.js.

Důležité je také zmínit že uprostřed vývoje programu byl přidán další požadavek, který říká, že program musí podporovat obarvení syntaxe. (Funkční požadavek č. 6)

3.3 Jednotlivá rozhodnutí

3.3.1 Tokenizovaný text

Vždy když program ukládá text vstupního zdrojového kódu, tak tento text je reprezentován seznamem, obsahujícím instance třídy `TokenInfo`. Instance třídy `TokenInfo` odpovídá právě jednomu tokenu vstupního zdrojového kódu a ukládá informace o textu tokenu, skutečnosti, zda se jedná o identifikátor, a barvě použité pro obarvení výstupního textu podle syntaxe.

Při vývoji programu byl text zdrojového kódu dříve reprezentován přímo textovým řetězcem. Později se ukázalo, že tato reprezentace je nedostatečná. Důvodem k nedostatečnosti je detekce a aplikace přejmenování proměnné. Kdybychom ve zdrojovém kódu reprezentovaném řetězcem chtěli přejmenovat proměnnou, která se jmenuje stejně, jako klíčové slovo a použili bychom naivní přepis všech výskytů jména proměnné na jiný textový řetězec, došlo by i k přepisu některých klíčových slov, což zřejmě není záměrem. Jakékoliv sofistikovanější řešení vyžaduje porozumění struktuře kódu, či informaci o tom, co je identifikátor a co ne.

Program byl tedy přepsán tak, aby pracoval s tokenizovaným textem. Nejen že to umožnilo přejmenovávat proměnné bez problému, ale dále to umožnilo naplnění nového, 6. funkčního požadavku programu, tedy možnost obarvení textu podle syntaxe.

Jedno z alternativních uvažovaných řešení by bylo kód reprezentovat textově a udržovat si informaci o tom, jaké části tohoto textu jsou identifikátory. Toto řešení nebylo použito z důvodu přidání 6. funkčního požadavku během vývoje.

3.3.2 Zjednodušená reprezentace kódu

Existence zjednodušené reprezentace kódu je důležitou součástí původní specifikace. Zdrojový kód je ve zjednodušené reprezentaci reprezentován stromovou

strukturou, která přibližně odpovídá derivačnímu stromu zdrojového kódu v daném programujícím jazyce. Ve zjednodušené reprezentaci jsou definovány čtyři typy vrcholů, přičemž dva z nich jsou vždy listem a zbylé dva jsou vnitřními vrcholy. Listové vrcholy ukládají tokeny původního kódu, které reprezentují. Listy jsou buď typu „sémantická akce“, nebo „nesémantický text“. Ostatní vrcholy jsou typu „sémantická definice“ nebo „sémantické rozhodnutí“.

Překlad do zjednodušené reprezentace není z důvodu obecnosti jednoznačně definovatelný. Při psaní překladače reprezentací je doporučeno dodržovat následující pravidla, která se primárně řídí rozsahy identifikátorů.

Sémantická definice

Jako sémantická definice je označen libovolný podblok kódu, který definuje identifikátor s rozsahem v aktuálním uvažovaném bloku. Za vnitřní kód je považován veškerý text, včetně hlavy.

Definujeme-li funkci, vnitřním kódem je veškerý kód této funkce a hlavička funkce.

Definujeme-li třídu, vnitřním kódem jsou například klíčové slovo uvozující definici, jméno funkce, její metody, datové složky i složená závorka (či v závislosti na jazyce jiný token) na konci definice.

Jednotlivé vlastnosti sémantické definice jsou potom následující:

- Vnitřní kód. Jedná se o seznam synů tohoto bloku, což jsou opět bloky kódu v zjednodušené reprezentaci.
- Jméno definovaného objektu.
- Typ definice, což je libovolný textový řetězec, který nejlépe popisuje, co je definováno – např. „variable“, „function“, atd.
- Parametry definice. Jedná se o identifikátory, které mají rozsah uvnitř vnitřního kódu definice. Například při definici funkce jsou parametry definice právě formální parametry funkce.

Pro to, aby byla sémantická definice specifikována korektně, musí seznam tokenů získaný vyčtením tokenů všech listů této definice v pořadí zleva doprava odpovídat souvislé části původního zdrojového kódu, která zahrnuje původní definici a případně bílé znaky okolo, ale nezahrnuje jinou sémantickou část zdrojového kódu.

Sémantické rozhodnutí

Za sémantické rozhodnutí je považován jakýkoliv podblok kódu, který má strukturovaný vnitřní kód, ale který nic nedefinuje v rozsahu aktuálního uvažovaného bloku kódu. Většinou se jedná o bloky typu „if-else“, či „for“, ale ne nutně se musí jednat o blok, ve kterém se skutečně rozhoduje. Například v JavaScriptovém překladači, který je součástí této práce, je „with“ blok také považován

za sémantické rozhodnutí.

Jednotlivé vlastnosti sémantického rozhodnutí jsou potom následující:

- Vnitřní kód. Jedná se o seznam synů tohoto bloku, což jsou opět bloky kódu v zjednodušené reprezentaci.
- Typ rozhodnutí, což je libovolný textový řetězec, který nejlépe popisuje o jaký typ bloku se jedná – „if-else“, „for“ atd.
- Parametry rozhodnutí. Jedná se o identifikátory, které mají rozsah uvnitř vnitřního kódu rozhodnutí. Například při definici bloku „with“ je parametrem identifikátor definovaný v hlavičce tohoto bloku.

Podmínka na korektnost je stejná jako v u sémantické definice.

Sémantická akce

Za sémantickou akci je považovaný jakýkoliv blok kódu, který neobsahuje žádné další bloky kódu uvnitř. Většinou se jedná o příkaz nebo volání funkce.

Jednotlivé vlastnosti sémantické akce jsou:

- Seznam identifikátorů jejichž hodnota je změněna tímto příkazem. Například pro příkaz „ $x = f(x + y)$;“ je tímto seznamem $[x]$, protože hodnota proměnné x je tímto příkazem změněna.
- Seznam identifikátorů, které ovlivňují, jak je příkaz proveden. Identifikátory pokryté prvním případem se zde neuvažují. Například pro příkaz „ $x = f(x + y)$;“ je tímto seznamem $[f, x, y]$, protože i přes to, že proměnná x byla na levé straně přiřazení, tak se nachází i na pravé straně přiřazení.
- Seznam bezprostředně následujících tokenů původního kódu a případných bílých znaků okolo, které odpovídají skutečnému příkazu v původním zdrojovém kódu.

Nesémantický text

Za nesémantický text je považována libovolná část zdrojového kódu, která nenesé význam (z pohledu překladače). Za nesémantický text je navíc považován blok kódu, jehož pozice je strukturálně vyžadovaná, například hlavička funkce v rámci definice funkce. V druhém případě je třeba blok označit speciálním identifikátorem, aby při hledání nejlevnější posloupnosti úprav došlo k zachování smysluplné struktury kódu.

Vlastnostmi nesémantického textu jsou:

- Seznam bezprostředně následujících tokenů původního kódu, které odpovídají nesémantické, nebo strukturálně vyžadované části v původním zdrojovém kódu.
- Typ nesémantického textu. Tato vlastnost je pouze přítomna, jedná-li se o strukturálně vyžadovanou část zdrojového kódu. Tato hodnota musí být unikátní v kontextu sourozeneckých bloků, tedy ve vnitřním kódu nadbloku.

```

class FirstClass {
  constructor(c, d) {
    this.c = c;
    this.d = d;
  }

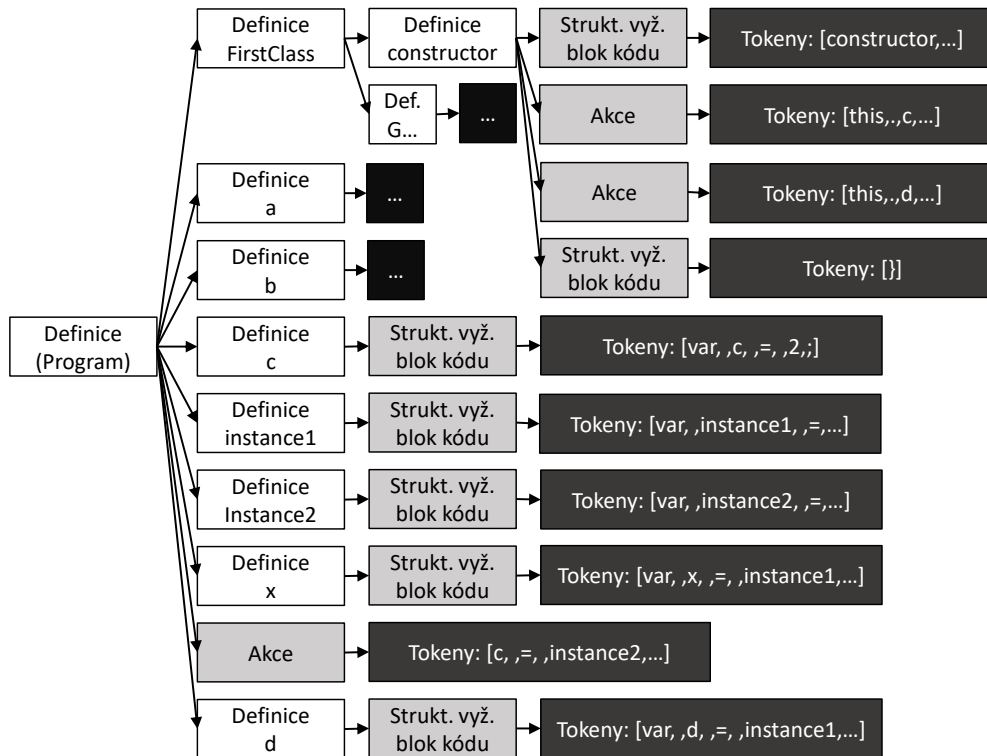
  GetFirstNElementsOfC(n) {
    var ret = [];
    // Debug:
    if (n % 2 == 0) {
      console.log('n is even')
    }
    else {
      console.log('n is odd');
    }
    for (var i = 0; i < n; i++) {
      ret.push(this.c[i]);
    }
    return ret;
  }
}

var a = 0;
var b = 1;
var c = 2;

var instance1 = new FirstClass([2,5,8,9,10,11], 'testInstance');
var instance2 = new FirstClass([1,3,3,4,5,6,7,7,8,9], 'testInstance');

var x = instance1.GetFirstNElementsOfC(3);
c = instance2.c[0];
var d = instance1.c[1];

```



Obrázek 3.1: Příklad JavaScriptového kódu a příslušné zjednodušené reprezentace

3.3.3 Rozsah identifikátoru ve zjednodušené reprezentaci

V předchozí sekci byla definována zjednodušená reprezentace. Protože část programu, která detekuje změny mezi zdrojovými kódy, pracuje výhradně s touto reprezentací a zároveň při uvážení sémantiky pracuje s rozsahem identifikátorů, je důležité se zamyslet nad tím, jak hodně lze reprezentovat rozsah identifikátoru v této reprezentaci.

Sémantická definice definovala identifikátor na úrovni sourozeneckých vrcholů a jejich podstromů. Efektivně se definice vztahuje na celý podstrom rodičovského bloku. Sémantická definice i sémantické rozhodnutí dále definovaly parametry, jejichž rozsahem byl podstrom daného bloku. Rozsahy identifikátorů tedy odpovídají podstromům ve stromu zdrojového kódu, což ne vždy odpovídá skutečnému rozsahu identifikátoru. Nadále budeme skutečnému rozsahu identifikátoru říkat přesný a rozsahu identifikátoru v rámci zjednodušené reprezentace budeme říkat nepřesný, či jednoduše „rozsah identifikátoru“.

V této reprezentaci zřejmě není možné dobře reprezentovat datové složky třídy a odkazy na ně z jiných částí kódů za identifikátory označující stejnou věc. Tato problematika v programu není nějak řešena a je považována za možné budoucí rozšíření. Jako důsledkem je například to, že je-li přejmenována datová složka třídy, přejmenování není aplikováno na odkazy na tuto složku. Tyto odkazy, místo toho, aby byly součástí přejmenování, jsou následně přepsány.

Další velkou limitací je použití dvou stejných identifikátorů, kdy jeden neoznačuje objekt definovaný v daném bloku, ani žádném rodičovském bloku, a druhý byl definován v daném bloku. Jako příklad uvažme následující úryvek JavaScriptového kódu:

```
function Example(a,b) {  
    return a + b.a;  
}
```

V tomto úryvku kódu identifikátor `b` reprezentuje instanci třídy s datovou složkou `a`. Kdyby byl parametr `a` přejmenován na `c`, správně by mělo dojít k přepisu pouze prvních dvou výskytů identifikátoru `a`. Protože ve zjednodušené reprezentaci bude přejmenování níže definováno na nepřesném rozsahu identifikátoru `a` bude obsahovat všechny výskyty identifikátoru v daném rozsahu, přejmenování zde by znamenalo přepis všech třech výskytů.

Z těchto pozorování lze shrnout, že modelování rozsahu identifikátoru v CodeShiftu není dostatečné. Modelování přesného rozsahu je v aktuální implementaci považováno nad rámec této práce. Při použití programu na malých příkladech za účelem výuky programování by se tyto problémy neměly moc projevit.

3.3.4 Úpravy v rámci zjednodušené reprezentace

K splnění druhého požadavku je nejprve třeba rozhodnout, jaké úpravy budou uvažované při výpočtu rozdílů mezi zdrojovými kódy. V programu jsou uvažovány následující úpravy:

- Nesémantický M:1 nebo 1:N přepis a přesun - Buď je několik bezprostředně následujících bloků přepsáno na jeden jiný blok, který je potom přesunut, nebo jeden blok je přepsán na více bezprostředně následujících jiných bloků, které jsou potom přesunuty.
- Sémantický přepis a přesun - Jeden blok je přepsán na druhý za předpokladu, že jsou stejného druhu. Od předchozí úpravy se liší v ohodnocení cenovou funkcí. Úpravy, které zohledňují sémantiku kódu jsou považovány za levnější než nesémantické úpravy. Po přepisu může být daný blok opět přesunut.
- Přidání bloku - Nový blok libovolné délky a libovolného druhu je přidán.
- Odebrání bloku - Libovolný blok libovolného druhu a délky je z kódu odebrán.
- Přejmenování identifikátoru - veškeré tokeny, které jsou identifikátory a označují stejný objekt (funkci, proměnnou, apod.), jsou přepsány. Přepis se provádí na podstromu celého stromu kódu, na kterém je daný identifikátor definován, tedy v nepřesném rozsahu identifikátoru.

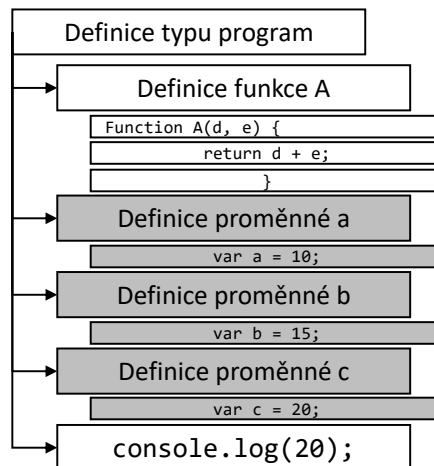
Tyto úpravy jsou uvažovány pouze na jedné úrovni kódu. V případě přesunutí větší části kódu do jiné úrovně, například přesunutí příkazů při definování funkce do těla dané funkce, je možné detekovat pomocí M:1 nesémantického přepisu. Původně byl uvažován i M:N nesémantický přepis, ale z důvodu náročnosti implementace vzhledem k stávajícímu kódu byly implementovány pouze 1:N a M:1 přepisy, které sami o sobě pokryjí velkou část praktických případů, jako například zmíněnou definici funkce.

Obrázek 3.2 znázorňuje příklad využití M:1 přepisu.

```
Function A(d, e) {
  return d + e;
}

var a = 10;
var b = 15;
var c = 20;

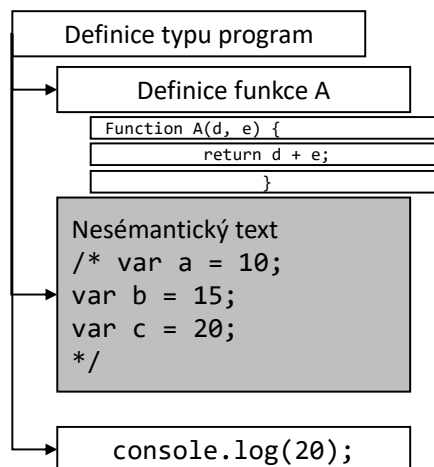
console.log(20);
```



```
Function A(d, e) {
  return d + e;
}

/* var a = 10;
var b = 15;
var c = 20;
*/

console.log(20);
```



Obrázek 3.2: Příklad využití M:1 přepisu pro přepis třech deklarácí na komentář

3.3.5 Cena úpravy

K tomu aby úpravy provedené ve výsledné animaci odpovídaly tomu, co by udělal skutečný programátor, bylo rozhodnuto, že cena úpravy bude přibližně založena na počtu stisků kláves, které je třeba provést v ideálním vývojovém prostředí k provedení úpravy. Tyto počty jsou dále přenásobeny konstantami, aby byly upřednostňovány „intuitivnější“ úpravy a aby, v rámci možností, byla zohledněna sémantika programu.

Definice ceny úprav umožní na transformace zdrojového kódu nahlížet jako na ohodnocené hrany grafu, jehož vrcholy jsou možné zdrojové kódy. Nyní můžeme definovat vzdálenost dvou zdrojových kódů jako délku nejkratší cesty mezi těmito zdrojovými kódy v tomto grafu.

Cena jednotlivých úprav je určena následovně:

- Nesémantický M:1 nebo 1:N přepis a přesun - K určení vzdálenosti bloků je použita Levenštejnova vzdálenost, která je potom přenásobena konstantou. K tomu je přičten počet znaků, přes který byl upravený blok přesunut, přenásobený jinou konstantou.
- Sémantický přepis a přesun:
Pokud se jedná o sémantickou akci, k určení vzdálenosti bloků je použita Levenštejnova vzdálenost přenásobena relativně nízkou konstantou a za každou proměnnou, kterou bloky nesdílejí je přičtena další vzdálenost.
Pokud se jedná o definici či rozhodnutí, nejprve je určena vzdálenost vnitřního kódu, která je přenásobena konstantou. Dále jsou porovnány seznamy definovaných proměnných, na kterých definice či rozhodnutí záleží a typy definice či rozhodnutí, a za jakoukoliv neshodu je přičtena další vzdálenost.
U nesémantického textu není sémantická vzdálenost uvažována, uvažuje se pouze nesémantický přepis a přesun.
- Přidání bloku - Cena přidání bloku je přímo úměrná jeho délce.
- Odebrání bloku - Z důvodu volby algoritmu pro nalezení provedených změn má odebrání bloku efektivně nulovou cenu. Volba tohoto algoritmu je motivována níže na základě vzhledu výstupní animace.
- Přejmenování identifikátoru - Cena přejmenování identifikátoru je přímo úměrná Levenštejnově vzdálenosti původního jména a nového jména. Toto je založené na skutečnosti, že většina vývojových prostředí nabízí možnost přejmenovat identifikátor, kdy stačí zadat nové jméno pouze jednou a dojde k přejmenování všech instancí tohoto identifikátoru.

V dřívější implementaci byly místo ceny úpravy, která udává odlišnost zdrojových kódů před úpravou a po úpravě, uvažovány dvě vlastnosti úpravy – odlišnost a podobnost zdrojových kódů před úpravou a po úpravě. Úpravy, které měly relativně vysokou odlišnost oproti podobnosti nebyly uvažovány. Implementace byla změněna po vhodném nastavení konstant udávajících cenu úpravy. Úpravy, které by v dřívější implementaci nebyly uvažovány mají v aktuální implementaci vždy vyšší cenu než úpravy, které uvažovány byly.

3.3.6 Průběh animace

Pro průběh výsledné animace bylo rozhodnuto, že bloky kódu budou přesouvány a přepisovány v tom pořadí, v jakém se objeví na výstupu při čtení od shora dolů. Důvodem k tomuto rozhodnutí je existence souvislé zpracované části kódu, která začíná začátkem celého zdrojového kódu a která se v průběhu animace prodlužuje. Toto umožní jednodušší návrh algoritmů pro práci se zdrojovým kódem, který je právě upravován a také to zpřehlední animaci tím, že po zbytek animace je neměnný blok kódu čitelný od shora dolů, tedy v přirozeném směru čtení.

Protože přejmenování identifikátoru může změnit podstatnou část kódu, a ne vždy je nutné aby definice identifikátoru byla v kódu napsána před použitím daného identifikátoru (např. funkci v JavaScriptu je možné použít v kódu fyzicky výše, než je daná funkce definována), bude přejmenování provedeno vždy, kdy se zpracování animace dostane do rozsahu, pro nějž daná definice platí.

Pro urychlení animace bylo navíc rozhodnuto, že odebrání, přidání a přesun sousedících bloků kódu bude probíhat současně.

3.3.7 Reprezentace úprav

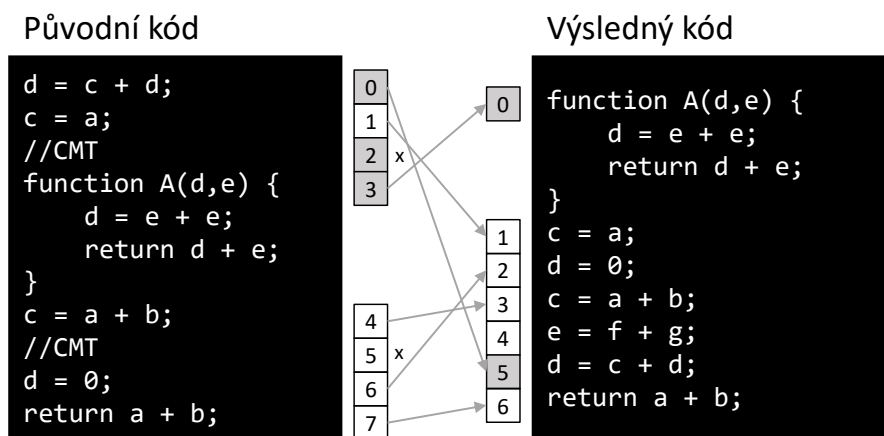
Pro reprezentaci úprav bude nejprve definována relace souvisejících bloků kódu. Bloky v původním a cílovém zdrojovém kódu spolu souvisí právě tehdy, když o původním bloku bylo rozhodnuto, že byl přepsán na cílový blok sémantickým přepisem, nesémantickým přepisem nebo oba bloky byly součástí 1:N přepisu či M:1 přepisu. Odebrané bloky původního kódu a přidání bloky cílového kódu nejsou v relaci s žádným jiným blokem. Přejmenování jsou uvažována zvlášť.

Seznam úprav bude reprezentován stromovou strukturou, která pro každý blok zjednodušené reprezentace původního kódu bude obsahovat instanci třídy `CodeChange`, která udává adresu souvisejícího bloku v cílovém kódu. Protože úpravy jsou uvažovány pouze na stejné úrovni kódu, adresou bloku je index bloku v kontextu sourozeneckých bloků. Pokud byl použit 1:N přepis, adresou místo jednoho bloku je pole bloků. Pokud je původní blok součástí M:1 přepisu, všechny zahrnuté bloky mají jako adresu uveden textový řetězec sestávající z písmene „M“ a adresy cílového bloku. Pokud byl blok původního kódu odebrán, jako adresa odpovídajícího bloku je uveden znak „x“.

Pro urychlení práce při vyhledávání informací v seznamu úprav je taktéž definován podobný seznam, který blokům cílového kódu přiřazuje bloky původního kódu. Tento seznam lze zkonstruovat z předchozího seznamu. Konvence při uvádění adresy původního bloku jsou stejné, jen v případě přidání bloku cílového kódu je použit znak „+“ místo „x“.

Kromě toho bude každý vrchol také obsahovat informaci o tom, k jakým přejmenováním došlo v rozsahu jeho vnitřního kódu.

Příklady reprezentací úprav jsou na obrázcích 3.3 a 3.4.



```

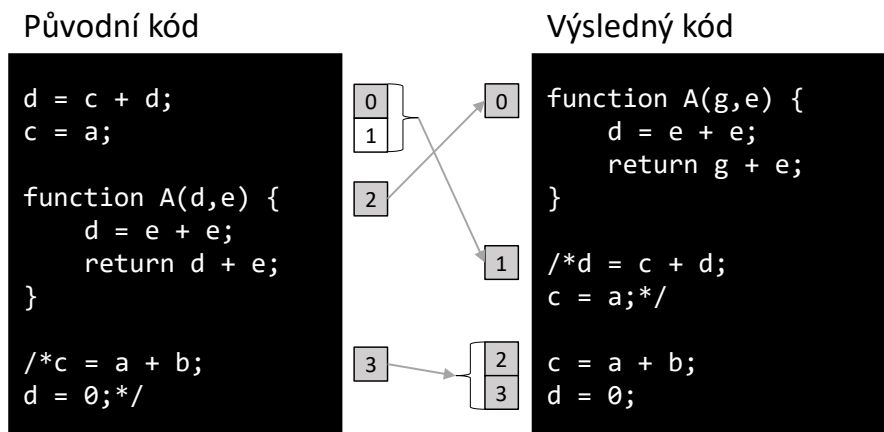
sourceChanges[0] = CodeChange {
    children : []
    renames: {}
    address: 5
}

sourceChanges[2] = CodeChange {
    children: []
    renames: {}
    address: 'x'
}

sourceChanges[3] = CodeChange {
    children: [CodeChange, ..., CodeChange]
    renames: {}
    address: 0
}

```

Obrázek 3.3: Příklad reprezentace souvisejících bloků



```

sourceChanges[0] = CodeChange {
    children : []
    renames : {}
    address : 'M1'
}

sourceChanges[2] = CodeChange {
    children: [CodeChange, ..., CodeChange]
    renames: {d: 'g'}
    address: 0
}

sourceChanges[3] = CodeChange {
    children: []
    renames: {}
    address: [2,3]
}

```

Obrázek 3.4: Příklad reprezentace přejmenování a souvisejících bloků ve vztazích 1:N a M:1

3.3.8 Algoritmy nalezení posloupnosti úprav

K diskuzi algoritmu nalezení posloupnosti úprav je třeba nejprve zmínit, že se jedná o 2 algoritmy – plnohodnotný a zkrácený. Plnohodnotný algoritmus detekuje přejmenování a vrací seznam úprav, zatímco zkrácený algoritmus přejmenování nedetekuje a pouze počítá vzdálenost zdrojových kódů. Zkrácený algoritmus je použit právě v části plnohodnotného algoritmu, ve které jsou testována přejmenování proměnných.

Plnohodnotný algoritmus nalezení posloupnosti úprav (dále jen algoritmus), podobně jako výsledná animace, si udržuje podmnožinu bloků výsledného kódu, které už byly pomocí úprav zdrojových bloků napsány a vhodně (podle minimální vzdálenosti) vybírá, zda má být další blok výsledného kódu přidán, či zda má být vytvořen přesunutím a přepisem jiného, doposud nepoužitého bloku z původního zdrojového kódu.

Před jakýmkoliv provedením přesunu či přepisu algoritmus nejprve provede detekci přejmenování tak, že pro každou dvojici identifikátorů platných v rozsazích porovnávaných kódů, uváží přejmenování identifikátoru z původního kódu na identifikátor v cílovém kódu a na původním kódu s přejmenovaným identifikátorem spustí zkrácený algoritmus. Pokud je vypočtená vzdálenost menší, než kdyby přejmenování použito nebylo, dané přejmenování je považováno za možné. Pokud alespoň jedno přejmenování je možné, to přejmenování s nejnižší vzdáleností od výsledného kódu je považováno za provedené. Pokud bylo provedeno alespoň jedno přejmenování, program pokračuje v hledání dalších provedených přejmenování obdobně.

Po detekci přejmenování následuje porovnávání a spojování souvisejících bloků v pořadí popsaném výše. Při sémantickém porovnání bloků s vnitřním kódem je instance algoritmu spuštěna na vnitřních blocích kódu těchto dvou bloků. Pokud je detekováno provedení 1:N přepisu (začínající v aktuálním zkoumaném bloku), iterátor bloků výsledného kódu zbývajících N-1 bloků přeskočí, jelikož provedením 1:N přepisu se jedná o použité bloky.

Aktuální použitý algoritmus lze označit jako hladový, jelikož bloky prochází v pořadí, v jakém jsou na výstupu a po přiřazení příslušného bloku na vstupu se k právě zpracovávanému bloku nevrací. Nalezený seznam změn tedy neodpovídá nejkratší cestě mezi původním a výsledným zdrojovým kódem, místo toho odpovídá určitému lokálnímu minimu. Pro nalezení nejkratší cesty by bylo třeba implementovat vhodný prohledávací algoritmus.

3.3.9 Uživatelské rozhraní

Třetí funkční požadavek vyžaduje, aby bylo možné upravit průběh animace pomocí volitelného mezivýstupu. V původní specifikaci to bylo řešeno tím, že by měly být implementovány tři běhové režimy. To bylo učiněno s tím rozdílem, že byl navíc přidán čtvrtý běhový režim. Režimy běhu Code-shiftu jsou následující:

- Režim úplného běhu, ve kterém je na základě vstupních souborů s kódem vygenerována GIFová animace.
- Režim částečného běhu s výstupem uprostřed, ve kterém je na základě vstupních souborů s kódem pouze vypočten seznam provedených úprav, který je následně uložen jako JSON soubor, který uživatel může upravovat.
- Režim částečného běhu s vstupem uprostřed, ve kterém je na základě vstupních souborů s kódem a JSON souboru se změnami vygenerována GIFová animace.
- Režim testování překladu. Účelem tohoto režimu bylo testování JavaScriptového překladáče, když byl vyvíjen. Tento režim je v programu ponechán pro případné rozšiřování o další jazyky.

Běhový režim se specifikuje při spuštění programu pomocí argumentů příkazové řádky. Protože podle požadavku rozšiřitelnosti je očekávaná podpora více jazyků, je možné jazyk vstupních souborů specifikovat při spouštění programu. Aby byl program uživatelsky přívětivější, použitý jazyk je odvozen z koncovky prvního souboru, pokud není uveden explicitně.

3.3.10 Formát mezivýstupu

Jak už bylo zmíněno na začátku kapitoly, formátem mezivýstupu je JSON. Důvodem k tomuto rozhodnutí je existence mnoha nástrojů pro úpravu souborů tohoto formátu, které zobrazují serializovaný objekt ve stromové struktuře. Jeden z takových nástrojů bude představen v uživatelské dokumentaci. Použití formátu JSON je snadné, protože JavaScript obsahuje vestavěné funkce pro převod do tohoto formátu, jelikož se jedná o formát určený právě k serializaci JS objektů.

Jako alternativou bylo dlouho zvažováno použití formátu XML. Hlavním důvodem byla opět existence nástrojů pro práci s tímto formátem. Hlavní nevýhodou tohoto formátu byly některé konvence, jako například skutečnost, že jména tagů nesmí začínat číslem. Tyto konvence ztěžovaly přímou serializaci, kterou formát JSON nabízel. Výsledná serializační funkce byla svou strukturou složitější a úpravy souboru by byly potenciálně uživatelsky méně přívětivé i za použití editoru.

I když je XML široce podporovaným formátem, JSON byl nakonec upřednostněn kvůli tomu, že se jedná o hlavní serializační formát v JavaScriptu.

3.3.11 Rozšiřitelnost

Jedním z cílů programu bylo napsat obecný algoritmus, který je nezávislý na programujícím jazyce, a umožnit rozšiřitelnost o další jazyky. Pro tento účel bylo definováno jazykové rozhraní, které definuje tři vlastnosti:

- Funkce, která převede textový řetězec zdrojového kódu napsaného v daném jazyce do zjednodušené reprezentace.
- Jména jazyka.

- Koncovka souborů se zdrojovým kódem v daném jazyce.

Druhá a třetí vlastnost rozhraní souvisí s odkazováním se na daný jazyk pomocí argumentů příkazové řádky, jak bylo zmíněno výše při diskusi uživatelského rozhraní.

3.3.12 Vzhled animace

Přesuny, přidání a odebrání jsou animovány tak aby se jednotlivé tokeny, které jsou zachovány, přesunuly z jejich počáteční pozice do jejich koncové pozice. Tokeny které jsou přidávány nebo odebírány zůstávají na místě a buď se postupně zviditelní nebo zneviditelní.

Přepisy a přejmenování jsou animovány na základě Levenštejnovy vzdálenosti. Účelem je odstranit a přidat co nejméně znaků a co nejvíc znaků přesunout bez výměny pořadí. To, jaké znaky mají být přesunuty a které mají být přidány nebo odebrány je odvozeno z matice výpočtu Levenštejnovy vzdálenosti.

Jednotlivé tokeny ve výsledné animaci jsou obarveny podle jejich typu. Původně bylo plánováno, že text, který se mění, bude zvýrazněn. Tato funkce nebyla nakonec implementována. Zvýraznění měnícího se textu nebylo součástí původní specifikace, ale implementace této funkce by výslednou animaci určitě zpřehlednila.

4. Vývojová dokumentace

Podstatná část architektury programu byla nastíněna již v sekci *Jednotlivá rozhodnutí* v kapitole *Analýza* (viz str. 13). Tato kapitola se zabývá architekturou programu podrobně.

4.1 Samodokumentace

Hlavní součástí vývojové dokumentace je samodokumentace programu. Samodokumentace programu je napsána v jazyce JSDoc, který byl popsán v kapitole *Použité Technologie* (viz str. 8).

4.2 Přehled architektury programu

Architektura programu je podle funkčnosti rozčleněna do následujících souborů:

- Zpracování vstupních argumentů a běh (index.js)
- Jazykové rozhraní:
 - Definice jazykově nezávislých tříd (languageInterface.js)
 - Definice implementací jazykového rozhraní (languageDefinitions.js)
 - Překladač JavaScriptového kódu do zjednodušené reprezentace (javaScriptTranslator.js)
- Výpočet provedených změn (distance.js)
- Animování:
 - Vytvoření seznamu animací (animationSequence.js)
 - Realizace seznamu animací a enumerace mezivýsledků (animationEnumerator.js)
- Generování souborů GIF (gifWriter.js)
- Ostatní:
 - Mezivýstup (intermediateOutput.js)
 - Výpočet animace založené na Levenšteinově vzdálenosti (levenAnimator.js)
 - Pomocná metoda pro hledání identifikátorů v daném rozsahu (identifierFinder.js)

4.3 Tok dat

Tato sekce představí tok dat v programu Code-shift. Obrázek 4.1 znázorňuje tok dat pro tři hlavní běhové režimy. Tok dat bude popsán nejprve pro režim úplného běhu programu a následně budou popsány rozdíly pro ostatní režimy běhu. O režimu běhu je rozhodnuto v souboru *index.js* po zpracování vstupních argumentů. V tomto souboru se nachází i definice funkcí, které vykonávají daný režim běhu.

4.3.1 Režim úplného běhu

V režimu úplného běhu jsou vstupem dva soubory se zdrojovým kódem a případná informace o jazyce. V případě, že informace o jazyce není uvedena, je použita funkce `DetermineLanguage` v souboru *languageDefinitions.js* k jejímu zjištění.

Na základě uvedeného či zjištěného jazyka jsou oba vstupní soubory s kódem přeloženy do zjednodušené reprezentace za použití funkce překladač pro daný jazyk. Podporované jazyky a jejich překladačové funkce jsou uvedeny v souboru *languageDefinitions.js*. Pro JavaScript je překlad implementován funkcí `JSToTree` v souboru *javaScriptTranslator.js*.

Zdrojové kódy jsou následně porovnány funkcí `FindCodeChanges` v souboru *distance.js*. Výsledkem porovnání je seznam provedených úprav. Ze seznamu provedených úprav je za použití funkce `GetAnimationSequence` v souboru *animationSequence.js* vygenerován seznam animačních příkazů¹.

Animační příkazy jsou nadále zpracovávány třídou `AnimationEnumerator` v souboru *animationEnumerator.js*. V této třídě je definována generující metoda `EnumerateAnimations`, která si udržuje aktuální pořadí bloků kódu pro výsledný GIF a na základě seznamu animačních příkazů ho mění a po každé změně, která se projeví změnou textu nebo pořadí jednoho či více bloků, tento generátor vrátí instanci jedné z tříd reprezentující příkaz pro výstup².

Příkazy pro výstup jsou posílány do instance třídy `GIFWriter` ze souboru *gifWriter.js* za použití metody `ApplyAnimation`. `GIFWriter` posléze volá příslušné funkce definované v souboru *gifWriter.js* pro vykreslení jednotlivých snímků GIFové animace. Na závěr je použitím metody `End` vygenerován výsledný GIF, který je na závěr přesunut na výstupní adresu.

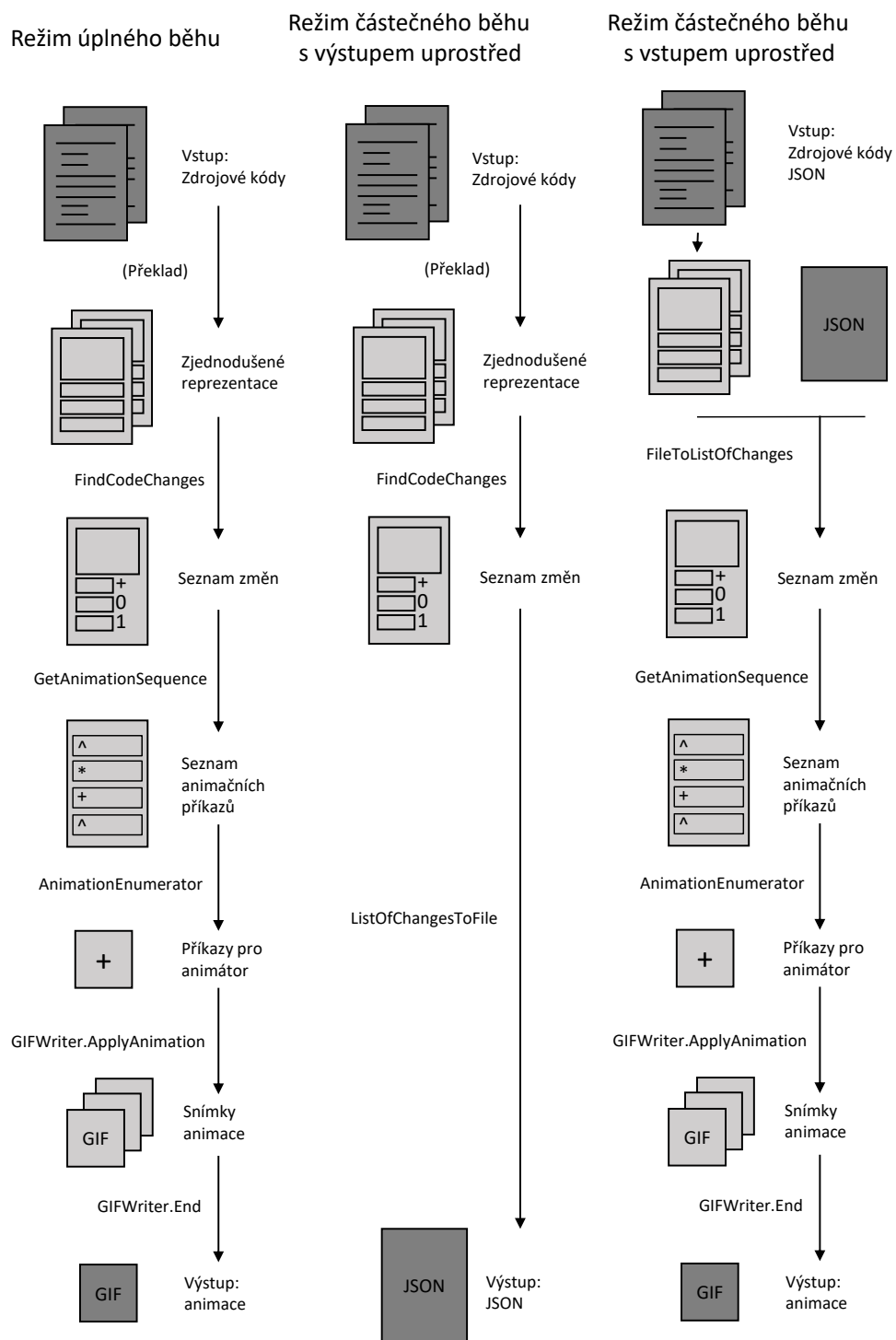
¹Animačními příkazy jsou instance tříd tvaru `*Command`, které jsou definované v souboru *animationSequence.js*

²Příkazy pro výstup jsou instance tříd tvaru `*Animation`, které jsou definované v souboru *animationEnumerator.js*

4.3.2 Režimy částečného běhu

V režimu částečného běhu s výstupem uprostřed začíná tok dat stejně, jako v případě režimu úplného běhu. Potom, co je zjištěn seznam provedených úprav, je tento seznam serializován za použití funkce `ListOfChangesToFile` definované v souboru *intermediateOutput.js*.

Režim částečného běhu s vstupem uprostřed je téměř shodný s režimem úplného běhu s tím rozdílem, že místo porovnání zjednodušených reprezentací zdrojového kódu k vytvoření seznamu úprav dojde k získání tohoto seznamu deserializací vstupního JSON souboru. JSON soubor je deserializován použitím funkce `FileToListOfChanges` definované v souboru *intermediateOutput.js*.



Obrázek 4.1: Tok dat ve třech hlavních běhových režimech

5. Uživatelská dokumentace

5.1 Instalace

Tato sekce popisuje instalaci programu Code-shift. Některé kroky instalace jsou popsány specificky pro operační systém Windows.

5.1.1 Program GraphicsMagick

Aby mohl být generován grafický výstup programu, je třeba nejprve nainstalovat program gm (GraphicsMagick) do složky, která je zahrnuta v proměnné prostředí PATH. V případě že jsme program dříve instalovali, tento krok může být vynechán. Případnou předchozí instalaci ověříme otevřením terminálu a zadáním příkazu „gm“. Pokud se vypíše nápověda k použití programu GraphicsMagick, přesuneme se do další podkapitoly.

V případě, že program nainstalovaný není, provedeme instalaci následovně:

1. Navštívíme internetovou stránku obsahující instalační soubory programu GraphicsMagick,
<https://sourceforge.net/projects/graphicsmagick/files/>.
Následující kroky budou popisovat instalaci specificky pro Windows.
2. Pro instalaci na Windows navigujeme do složky „graphicsmagick-binaries“ a posléze do složky s nejnovější verzí. Vybereme instalační soubor nejlépe odpovídající naší platformě.
3. Po stažení instalačního souboru tento soubor otevřeme a řídíme se pokyny. Důležité je před započítím instalace zaškrtnout možnost přidání cesty instalace do proměnné prostředí PATH, případně cestu instalace do PATH dodat posléze.
4. Po skončení instalace otevřeme terminál a spuštěním příkazu „gm“ instalaci ověříme.

5.1.2 Prostředí Node.js

K spuštění programu Code-shift je nezbytné nejprve nainstalovat běhové prostředí Node a s ním související správce balíčků npm. V případě, že jsme Node a npm dříve nainstalovali, tento krok přeskočíme. K ověření, zda jsme Node a npm nainstalovali do adresáře zahrnutého v proměnné prostředí PATH otevřeme terminál a zadáme příkazy „node -v“ a „npm -v“. V případě, že je daný program nainstalován, dojde k vypsání jeho verze. V případě, že alespoň jeden z programů nainstalovaný není, instalaci provedeme následovně:

1. Na internetové stránce projektu Node.js, <https://nodejs.org/en/>, přejdeme do sekce „Downloads“ a v této sekci klikneme na položku, která odpovídá operačnímu systému a platformě (případně procesoru) našeho stroje. Kliknutím na danou položku by mělo dojít k stažení instalačního souboru programu Node.js.

2. Stažený instalační soubor spustíme a řídíme se pokyny v něm. Ujistíme se, že v instalaci je zahrnut správce balíčků npm a že cesta instalace bude přidána do proměnné prostředí PATH, případně změníme některá nastavení, nebo složku dodáme do proměnné prostředí PATH po dokončení instalace.
3. Potom, co je instalace dokončena, otevřeme terminál a spuštěním příkazů „node -v“ a „npm -v“ instalaci ověříme.

5.1.3 Program Code-shift

Archiv obsahující program Code-shift je dostupný v souborech přiložených k této práci, alternativně ho lze stáhnout z repositáře na adrese <https://github.com/Pet297/Code-shift/releases/tag/v1.1.2>. Archiv přesuneme do libovolné složky a rozbalíme.

Dříve, než program spustíme, musíme nainstalovat potřebné knihovny.

5.1.4 Instalace knihoven

Pro instalaci knihoven otevřeme terminál a přesuneme se do složky, která obsahuje zdrojový kód Code-shiftu. V této složce spustíme příkaz „npm install“. Protože zdrojový kód obsahuje soubor „package.json“, instalace všech potřebných knihoven v očekávaných verzích proběhne automaticky prostřednictvím správce balíčků npm. Pokud nedošlo k žádné chybě, program Code-shift je připraven k použití.

5.2 Omezení programu

Během testování i používání programu je třeba brát ohled na následující omezení programu:

- Cesty vstupních a výstupních souborů, se kterými může program pracovat, jsou omezené. Tato omezení jsou dána omezeními běhového prostředí Node.js. Vstupní soubory je před zpracováním doporučeno přesunout do adresáře který obsahuje složku s kódem a odkazovat je pomocí relativní cesty „../“.
- Oddělovače řádek použité v přiložených testovacích souborech jsou CRLF. Protože Code-shift používá systémové oddělovače mezer při zpracovávání zdrojových kódů místo toho, aby zohlednil oddělovače použité ve zpracovávaném souboru, použití přiložených testovacích souborů na stroji s Linuxem může potenciálně vrátit nevhodný výsledek.
- Code-shift není stavěn na souběžné spuštění více instancí. Důvodem je to, že Code-shift používá pouze jednu složku pro uchovávání individuálních snímků výsledné animace. Snímky pro jednotlivé instance programu nejsou nijak odlišeny, takže při spuštění více instancí programu by mohlo docházet ke vzniku animací, které kombinují snímky z různých běhů programu.

- Vzhled výstupní animace není konfigurovatelný bez zásahu do zdrojového kódu.

5.3 Základní použití

Při základním použití program porovná 2 soubory se zdrojovým kódem a vytvoří GIF, vyobrazující postupnou transformaci prvního zdrojového kódu do druhého.

V případě, že porovnávané JavaScriptové soubory obsahují koncovku „.js“, program spustíme příkazem:

```
> node index.js -i [vstup1.js] -i [vstup2.js] -o [vystup.gif]
```

Pokud vstupní soubory koncovku „.js“ neobsahují, ale jsou napsány v JavaScriptu, k vstupním argumentům přidáme informaci o jazyce:

```
> node index.js -l JS -i [vstup1.js] -i [vstup2.js] -o [vystup.gif]
```

Jako příklad program spustíme na testovacích souborech:

```
> node index.js -l JS -i tests/test_F1_0 -i tests/test_F1_1 -o ../out.gif
```

Běh programu může trvat pro soubory s vysokým počtem úprav i několik minut. V praxi je generování GIFu výrazně pomalejší, než analýza provedených změn.

5.4 Pokročilé použití

Program Code-shift nabízí čtyři režimy běhu. Jedná se o režimy

1. úplného běhu
2. částečného běhu s výstupem uprostřed
3. částečného běhu s vstupem uprostřed
4. testování překladače

Dříve jsme si představili režim úplného běhu. Níže si představíme jak změnit seznam provedených úprav ve výsledné animaci pomocí režimů částečného běhu. Poslední čtvrtý režim si představíme v kapitole *Testování programu* (viz str. 39), jelikož jeho cílem je testování překladače reprezentací při jeho vývoji.

5.4.1 Mezivýstup

Přesto, že cílem programu Code-shift je automatické generování animací vyobrazující provedené změny mezi dvěma verzemi zdrojového kódu, není vždy možné, aby seznam provedených změn odpovídal očekávání uživatele. Právě z tohoto důvodu jeden z požadavků na program byla možnost volitelného mezivýstupu.

Princip mezivýstupu spočívá v tom, že nejprve je program spuštěn v režimu částečného běhu s výstupem uprostřed. Při tomto prvním běhu výstupem programu není GIFová animace, ale JSON soubor reprezentující seznam provedených úprav. Tento soubor uživatel upraví a program spustí opět, nyní v režimu částečného běhu s vstupem uprostřed. Program v tomto běhu vygeneruje GIFovou animaci na základě úprav popsanych v JSON souboru.

K spuštění programu v režimu částečného běhu s výstupem uprostřed při spuštění programu použijeme spínač „-f“. Vstupem jsou opět soubory se zdrojovým kódem a výstupem je nyní JSON soubor. Volitelně zahrneme informaci o jazyce, zde „JS“.

```
> node index.js -f -l JS -i [vstup1.js] -i [vstup2.js] -o [vystup.json]
```

K spuštění programu v režimu částečného běhu s vstupem uprostřed při spuštění programu zahrneme argument „-c [zmeny.json]“ udávající cestu k vstupnímu souboru se změnami. Kromě souboru se změnami jsou i vstupem dva soubory se zdrojovým kódem a výstupem je soubor GIF. Volitelně zahrneme informaci o jazyce, zde není zahrnuta.

```
> node index.js -i [vstup1.js] -c [zmeny.json] -i [vstup2.js] -o [vystup.gif]
```

Struktuře JSON souboru se změnami se věnuje další sekce. Pro ověření funkčnosti obou dvou režimů použijeme následující 2 příkazy:

```
> node index.js -f -l JS -i tests/test_F1_0 -i tests/test_F1_1 -o ../out.json  
> node index.js -l JS -i tests/test_F1_0 -c ../out.json -i tests/test_F1_1 \  
-o ../out.gif
```

Protože jsme JSON soubor nějak neupravovali, výsledek bude stejný, jako při použití režimu úplného běhu.

5.4.2 Více vstupních souborů

Ve všech předchozích případech jsme jako vstup používali 2 soubory se zdrojovým kódem. Program je také možné spustit na více vstupních souborech. Pro N vstupních souborů je vygenerováno $N - 1$ výstupních souborů popisující změny mezi bezprostředně následujícími vstupními soubory. Spuštění s více vstupními soubory je dostupné ve všech doposud probraných režimech běhu.

V režimu úplného běhu je pro N vstupních souborů vygenerováno $N - 1$ GIFů. Syntax pro 3 vstupní soubory je následující:

```
> node index.js -i [vstup1.js] -i [vstup2.js] -i [vstup3.js] \  
-o [vystup12.gif] -o [vystup23.gif]
```

V režimu částečného běhu s výstupem uprostřed je pro N vstupních souborů vygenerováno $N - 1$ JSON souborů se změnami. Syntax pro 3 vstupní soubory je následující:

```
> node index.js -f -i [vstup1.js] -i [vstup2.js] -i [vstup3.js] \  
-o [vystup12.json] -o [vystup23.json]
```

V režimu částečného běhu s vstupem uprostřed je pro N vstupních souborů s kódem a $N - 1$ vstupních souborů se změnami vygenerováno $N - 1$ GIFů. Syntax pro 3 vstupní soubory je následující:

```
> node index.js -i [vstup1.js] -c [zmeny12.json] -i [vstup2.js] \  
-c [zmeny23.json] -i [vstup3.js] -o [vystup12.gif] -o [vystup23.gif]
```

Jako příklad tuto funkci programu otestujeme následujícím příkazem:

```
> node index.js -i tests/test_F3_0 -i tests/test_F3_1 -i tests/test_F3_2 \  
-o ../out1.gif -o ../out2.gif
```

Program doběhne ve chvíli, kdy jsou oba výstupní soubory vygenerovány.

5.5 Úprava mezivýstupu

Jak bylo popsáno v sekci *Reprezentace úprav* (viz str. 21), k reprezentaci změn v kódu je používána dvojice seznamů typu `CodeChange[]`, které určují, jaké bloky původního a cílového kódu spolu souvisí. Protože je možné jeden z těchto seznamů sestavit na základě druhého, JSON soubor se změnami obsahuje serializovanou pouze jednu proměnnou typu `CodeChange[]`, reprezentující bloky původního zdrojového kódu. Úpravou tohoto souboru a následným načtením v režimu částečného běhu se vstupem uprostřed dojde k vygenerování jiného seznamu změn, než kdyby změny byly hledány automaticky, což změní průběh animace.

K upravování souboru typu JSON je vhodné použít editor, který obsah těchto souborů vyobrazí ve stromové struktuře. V příkladu je použit editor na adrese <https://jsoneditoronline.org/>.

Jako příklad vygenerujeme JSON soubor pro test F1 použitím příkazu

```
> node index.js -f -l JS -i tests/test_F1_0 -i tests/test_F1_1 -o ../out.json
```

Otevřeme-li vygenerovaný JSON soubor ve vhodném editoru, vidíme, že celý blok původního programu (jediný prvek serializovaného pole) souvisí s celým blokem cílového programu, protože hodnota `address` je 0. Rozbalíme-li seznam dětí bloku celého programu (`children`), vidíme například, že 0. blok uvnitř původního programu souvisí se 7. blokem uvnitř cílového programu a že 3. blok původního

programu byl odstraněn.

Nyní změním všechny adresy kromě té na pozici 0 na „x“. Nová interpretace souboru změň říká, že definice funkce `A` v původním zdrojovém kódu (0. blok) odpovídá definici funkce `A` v novém zdrojovém kódu (7. blok) a že všechny ostatní bloky původního kódu byly odebrány a tím pádem že všechny ostatní bloky cílového kódu byly přidány. Upravený JSON soubor uložíme jako „in.json“. Tato úprava JSON souboru je znázorněna na obrázku 5.1.

Použitím příkazu níže dojde k vygenerování GIFu, v němž je průběh animace založený na popisu výše a je tedy jiný než kdyby byl použit režim úplného běhu.

```
> node index.js -l JS -i tests/test_F1_0 -c ../in.json -i tests/test_F1_1 \  
-o ../out.gif
```

Při upravování JSON souboru je třeba aby soubor reprezentoval validní serializaci objektu typu `CodeChange[]`. Serializovaný objekt typu `CodeChange[]` by sám o sobě měl být korektní podle specifikace v sekci *Reprezentace úprav* (viz str. 21).

```

[ 1 item
  0 : {
    address : 0
    children : [ 9 items
      0 : {
        address : 7
        children : [ 5 items ]
        renames : { 0 props }
        tokens : [ 41 items ]
      }
      1 : {
        address : 0
        children : [ 1 item ]
        renames : { 0 props }
        tokens : [ 10 items ]
      }
      2 : {
        address : 1
        children : [ 1 item ]
        renames : { 0 props }
        tokens : [ 10 items ]
      }
      3 : {
        address : x
        children : [ 0 items ]
        renames : { 0 props }
        tokens : [ 5 items ]
      }
      4 : {
        address : 2
        children : [ 0 items ]
        renames : { 0 props }
        tokens : [ 8 items ]
      }
      5 : { 4 props }
      6 : { 4 props }
      7 : { 4 props }
      8 : { 4 props }
    ]
    renames : { 0 props }
  }
]

[ 1 item
  0 : {
    address : 0
    children : [ 9 items
      0 : {
        address : 7
        children : [ 5 items ]
        renames : { 0 props }
        tokens : [ 41 items ]
      }
      1 : {
        address : x
        children : [ 1 item ]
        renames : { 0 props }
        tokens : [ 10 items ]
      }
      2 : {
        address : x
        children : [ 1 item ]
        renames : { 0 props }
        tokens : [ 10 items ]
      }
      3 : {
        address : x
        children : [ 0 items ]
        renames : { 0 props }
        tokens : [ 5 items ]
      }
      4 : {
        address : x
        children : [ 0 items ]
        renames : { 0 props }
        tokens : [ 8 items ]
      }
      5 : {
        address : x
        children : [ 0 items ]
        renames : { 0 props }
        tokens : [ 8 items ]
      }
    ]
    renames : { 0 props }
  }
]

```

Obrázek 5.1: Úprava JSON souboru se změnami

5.6 Význam jednotlivých argumentů

V programu code-shift jsou dostupné následující vstupní argumenty:

- -i [vstup]: Cesta k vstupnímu souboru s kódem.
- -o [výstup]: Cesta k výstupnímu souboru s animací, či změnami podle běhového režimu.
- -c [vstup]: Cesta k vstupnímu souboru se změnami. Použití tohoto argumentu přepne program do režimu částečného běhu s vstupem uprostřed.
- -l [jazyk]: Jméno programovacího jazyka, ve kterém jsou vstupní soubory napsány. Pokud tento argument není zahrnut, jazyk vstupních souborů je odvozen z koncovky prvního souboru. Pokud koncovka prvního souboru není zahrnuta v definici z žádných implementovaných jazyků, program končí chybou.
- -t [vstup]: Cesta k vstupnímu souboru s kódem. Použití tohoto argumentu přepne program do režimu testování překladu.
- -h: Vypíše nápovědu k použití programu.
- -f: Přepne program do režimu částečného běhu s výstupem uprostřed.

Při spuštění s nesprávnou kombinací argumentů program vypíše chybovou hlášku a ukončí se. Nesprávné kombinace argumentů zahrnují:

- Použití nesprávného počtu vstupů a výstupů. Počet vstupů (argumentů -i) musí být o 1 větší, než počet výstupů (argumentů -o). V režimu částečného běhu s vstupem uprostřed je třeba, aby počet argumentů -c byl stejný, jako počet argumentů -o.
- Spuštění programu v dvou režimech zároveň – například, je-li použit argument -f a argument -c. Pokud režim běhu programu není nějak upraven žádným argumentem, dojde k běhu v režimu úplného běhu.

6. Testování programu

Při vývoji programu Code-shift byla použita varianta metody programování řízeného testy. Testování se zaměřovalo na 2 aspekty program – Spouštění programu v různých běhových režimech na sadě vstupních souborů a testování JavaScriptového překladače reprezentací. Z důvodu podstaty testovaných aspektů nebyly zřízeny automatické testy.

6.1 Testování běhu programu

Účelem testování běhu programu bylo testovat funkci pro hledání seznamu provedených změn a také testování GIFového výstupu. Jelikož výstup programu je závislý na konstantách, a ne vždy lze říct, co je správný výstup, testy byly navrhované tak, aby seznam provedených úprav v daném testu byl více méně jednoznačný. Správnost výstupu byla potom prozkoumána ručním zkontrolováním výstupního GIFu. Vždy, když měla být naimplementována jedna z funkcí programu bylo přidáno několik testů.

Soubory určené pro testování běhu programu jsou dostupné v složce „tests“ a jejich názvy mají strukturu „test_??_?“, kde prvním a druhým neznámým znakem je identifikátor testu a třetí neznámý znak je číslo 0 nebo 1 udávající zda se jedná o zdrojový kód před úpravou, či po úpravě. V případě testu „F3“ jsou dostupné tři verze zdrojového kódu. Tento test sloužil od otestování funkce spuštění programu na více vstupech.

6.2 Testování překladače

Účelem testování překladače bylo odladění JavaScriptového překladače reprezentací. Testy byly napsány pro různé neterminály jazyku JavaScript a různorodé příkazy derivované z těchto neterminálů. JavaScriptový parser pro Antlr4, který byl v projektu použit definuje 75 neterminálů, které jsou indexované čísly 0 až 74. Soubory pro testování překladače jsou také dostupné ve složce „tests“. Názvy souborů s testy pro překladač mají strukturu „test_T??“, kde „??“ je index testovaného neterminálu.

Narozdíl od testování běhu by tyto testy šlo automatizovat, jelikož překlad do zjednodušené reprezentace, i když není jednoznačně definovaný, lze před vývojem překladače jednoznačně rozhodnout a na základě tohoto rozhodnutí jednoznačně testovat. Testy nebyly automatizovány, jelikož návrh automatizovaných testů by nejspíš byl časově náročnější, než samotná implementace překladače.

Pro testování překladače byl naimplementován čtvrtý běhový režim. Chceme-li otestovat správnost překladače souboru „tests/test_T20“ do zjednodušené reprezentace, použijeme následující příkaz:

```
> node index.js -l JS -t tests/test_T20
```

Protože program neskončí chybou, a dojde k výpisu zjednodušené reprezentace kódu obsaženého v daném souboru, usoudíme, že překlad použitých pravidel přepisu je implementovaný. Správnost zjednodušené reprezentace je třeba ověřit ručně.

7. Budoucí práce

Jak bylo předchozích částech textu ukázáno, program Code-shift má v aktuální implementaci některé nedostatky. Řešení netriviálních nedostatků by mohlo být námětem pro budoucí práce.

Asi největším nedostatkem programu jsou problémy spojené s rozsahem identifikátorů, kterým se věnovala podkapitola *Rozsah identifikátoru ve zjednodušené reprezentaci* (viz str. 17). Asi nejsnazším řešením by bylo vhodným způsobem zahrnout myšlenku datové složky v rámci zjednodušené reprezentace a algoritmy pracující s identifikátory této změně přizpůsobit.

Jedním z budoucích rozšíření programu by také mohla být podpora více programovacích jazyků. I přesto, že program byl navržen na to, aby byl rozšiřitelný, aktuálně je podporován pouze jeden jazyk a to JavaScript. Zjednodušená reprezentace kódu je inspirována jazykem C a jemu podobnými jazyky, takže pokrytí alespoň této třídy jazyků by mělo být možné.

Pro uživatelskou přívětivost by navíc bylo vhodné, kdyby program měl grafické uživatelské rozhraní, ve kterém by bylo možné upravovat průběh animace bez nutnosti použití souborů s mezivýstupem. Uživatelské rozhraní by také mohlo zahrnovat náhled animace bez nutnosti zdlouhavého generování GIFů. Kromě toho v programu chybí možnost konfigurace výstupu a konstant pro hledání změn. Tyto konstanty lze změnit pouze zásahem do zdrojového kódu.

Na závěr, program byl pouze testován na malé sadě testů. Ideálně by, kromě použití větší sady testů, byla otestována i použitelnost programu prostřednictvím ankety.

Závěr

Navrhli a naprogramovali jsme program Code-shift, který na základě dvou vstupních souborů se zdrojovým kódem umí vygenerovat animaci postupného přepisu první verze do druhé. Za účelem jazykové nezávislosti jsme definovali zjednodušenou reprezentaci kódu a převod do této reprezentace jsme implementovali pro JavaScript. Pro usměrnění algoritmu hledání provedených změn jsme definovali cenu úpravy kódu, která byla posléze použita v tomto algoritmu. Vývoj programu byl řízený testy, zaměřenými na běh programu a testování překladač do zjednodušené reprezentace.

Slabé stránky aktuální implementace byly diskutovány. Vyšetření těchto nedostatků a následné rozšíření programu by mohlo být námětem pro budoucí práce.

Seznam použité literatury

AHO, A., LAM, M., SETHI, R. a ULLMAN, J. (2006). *Compilers: Principles, Techniques, and Tools*. Addison Wesley, USA, 2nd edition. ISBN 0321486811.

CHYTL, M. (1984). *Automaty a gramatiky*. SNTL, Praha.

JSDOC 3 DOCUMENTATION PROJECT (2017). @use jsdoc. URL <https://jsdoc.app/>. [cit. 2022-05-09].

SOFTWARE FREEDOM CONSERVANCY (2022). git-diff-tree 2.36.0. URL <https://git-scm.com/docs/git-diff-tree>. [cit. 2022-05-09].

Seznam obrázků

1	Shrnutí videa v tweetu od Omara Rizwana	5
2	Shrnutí animace v tweetu od Henryho Zhu	5
3.1	Příklad JavaScriptového kódu a příslušné zjednodušené reprezentace	16
3.2	Příklad využití M:1 přepisu pro přepis třech deklarácí na komentář	19
3.3	Příklad reprezentace souvisejících bloků	22
3.4	Příklad reprezentace přejmenování a souvisejících bloků ve vztazích 1:N a M:1	23
4.1	Tok dat ve třech hlavních běhových režimech	30
5.1	Úprava JSON souboru se změnami	37

A. Přílohy

A.1 Zdrojový kód

Zdrojový kód programu Code-shift se nachází ve stejnojmenné složce. Alternativně ho lze stáhnout z repositáře na adrese <https://github.com/Pet297/Code-shift/releases/tag/v1.1.2>.

A.2 Specifikace

Soubor „Specifikace.pdf“ obsahuje původní specifikaci programu, na základě které byl Code-shift implementován. Tato specifikace je zmíněna na začátku kapitoly *Analýza* (viz str. 12).