Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Superstrict mód pre Javascript

Bakalárska práca

2016
Matej Krajčovič

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Superstrict mód pre Javascript

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Tomáš Kulich, PhD.

Bratislava, 2016
Matej Krajčovič

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Matej Krajčovič
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Superstrict mód pre Javascript
Superstrict mode for Javascript

Cieľ: Implementovať direktívu "superstrict" analogickú k "use strict". Vytvoriť plugin do kompilátora Babel, ktorý v danom súbore rozpozná "superstrict" direktívu (analóg k "use strict") a následne pri preklade zapne (niektoré) runtime validácie:
- zabránenie prístupu k neexistujúcemu atribútu v objekte / mape
- zabránenie prístupu k neexistujúcemu atribútu v objekte / mape pomocou destructuring konštrukte
- zabránenie automatickej zmene typov (cast) primitív / objektov
- výsledok numerickej operácie nesmie byť NaN
- zabránenie tichému overflow

Vedúci: RNDr. Tomáš Kulich, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 30.10.2015

Dátum schválenia: 30.10.2015 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Abstrakt

V tejto práci predstavujeme vybrané časti sémantiky jazyka Javascript a porovnávam ich s analogickými časťami sémantiky jazyka Python. Na základe porovnania popisujeme spôsob, ako upravovať zdrojový kód v jazyku Javascript, aby bol bezpečnejší a zjednodušilo sa hľadanie chýb.

Kľúčové slová: Javascript, ECMAScript, Babel.js, striktný mód

Abstract

In this work we present selected parts of semantics of Javascript and compare them to semantics of Python. Based on this comparison we describe a way to change Javascript source code to be safer and easier to debug.

Keywords: Javascript, ECMAScript, Babel.js, strict mode

Obsah

Zoznam obrázkov

Úvod

Progamovací jazyk Javascript vytvoril Brendan Eich v roku 1995, keď pracoval vo firme Netscape Communications Corporation. Za 10 dní mal napísať prototyp programovacieho jazyka, ktorý by bol spustiteľný vo webovom prehliadači Netscape [17]. Už prvá verzia štandardu [14] obsahovala neintuitívne implicitné typové konverzie. Napríklad sčítanie prázdneho poľa a prázdneho poľa je prázdny reťazec. Tiež podľa štandardu prístup k neexistujúcemu prvku objektu nespôsobí chybu alebo výnimku, ale vráti hodnotu *undefined*, ale to môže byť aj valídna hodnota pristupovaného prvku [11].

Na rozdiel od Javascriptu, vývojári jazyka Python nemali limitovaný čas a nové vlastnosti pridávali postupne vo forme schvaľovaných PEPov (Python Enhancement Proposals) [13]. V PEP20 je spísaný 'Zen Pythonu', ktorého prvé tri riadky sú v preklade nasledovné [16]:

> Krásne je lepšie ako škaredé.
> Explicitné je lepšie ako implicitné.
> Jednoduché je lepšie ako komplexné.

V tejto práci skúmame spôsob, ako priblížiť tieto ideály Javascriptu a spraviť programovanie v ňom predvídateľnejšie a bezpečnejšie. Použijeme na to nástroj Babel.js, ktorý umožňuje definovať transformácie Javascriptového kód - v našom prípade pridanie testov.

V prvej kapitole predstavíme Babel.js, v druhej popíšeme striktný a nami navrhovaný superstriktný mód, v tretej sa zaoberáme spôsobom implementácie, v štvrtej vplyvu transformácií na veľkosť výsledného kódu a jeho rýchlosť a v poslednej kapitole spomíname nástroje, ktoré tiež riešia niektoré z problémov, ktorým je venovaná táto práca.

Kapitola 1

Babel.js

V nasledujúcom text popisujeme jednotlivé fázy transpilácie a podsystémy, z ktorých sa skladá Babel.

1.1 Fázy transpilácie

Babel pracuje v troch fázach:

- lexikálna a syntaktická analýza zdrojového kódu,
- transformácia abstraktného syntaktického stromu,
- vygenerovanie finálneho zdrojového kódu z AST.

1.1.1 Lexikálna a syntaktická analýza

Počas lexikálnej analýzy sa zo zdrojového kódu tvorí zoznam tokenov (kategorizovaných reťazcov viac nedeliteľných znakov): napríklad kľúčové slová, reťazce, otváracie a zatváracie zátvorky. Tiež sa kontroluje, či je zdrojový kód v súlade s gramatikou.

Výstup lexikálnej analýzy programu 1.1 je možné vidieť v 1.2 (parametre loc a type sú objekty obsahujúce ďalšie údaje, ale kvôli prehľadnosti nie sú celé vypísané):

```
1  function hello_world(name) {
2  console.log("Hello World " + name);
3  }
```

Listing 1.1: Ukážkový program

```
1  [
2    {type: {label: "function", keyword: "function"},
3      value: "function"},
4    {type: {label: "name"}, value: "hello_world",},
5    {type: {label: "("}},
```

```
6     {type: {label: "name"}, value: "name"},
7     {type: {label: ")"}},
8     {type: {label: "{"}},
9     {type: {label: "name"}, value: "console"},
10    {type: {label: "."}},
11    {type: {label: "name"}, value: "log"},
12    {type: {label: "("}},
13    {type: {label: "string"}, value: "Hello World "},
14    {type: {label: "+/-"}, value: "+"},
15    {type: {label: "name"}, value: "name"},
16    {type: {label: ")"}},
17    {type: {label: ";"}},
18    {type: {label: "}"}},
19    {type: {label: "eof"}
20  ]
```

Listing 1.2: Výstup lexikálnej analýzy

Následná lexikálna analýza na základe znalosti gramatiky jazyka vytvorí abstraktný syntaktický strom (AST) znázornený na grafe 1.1.

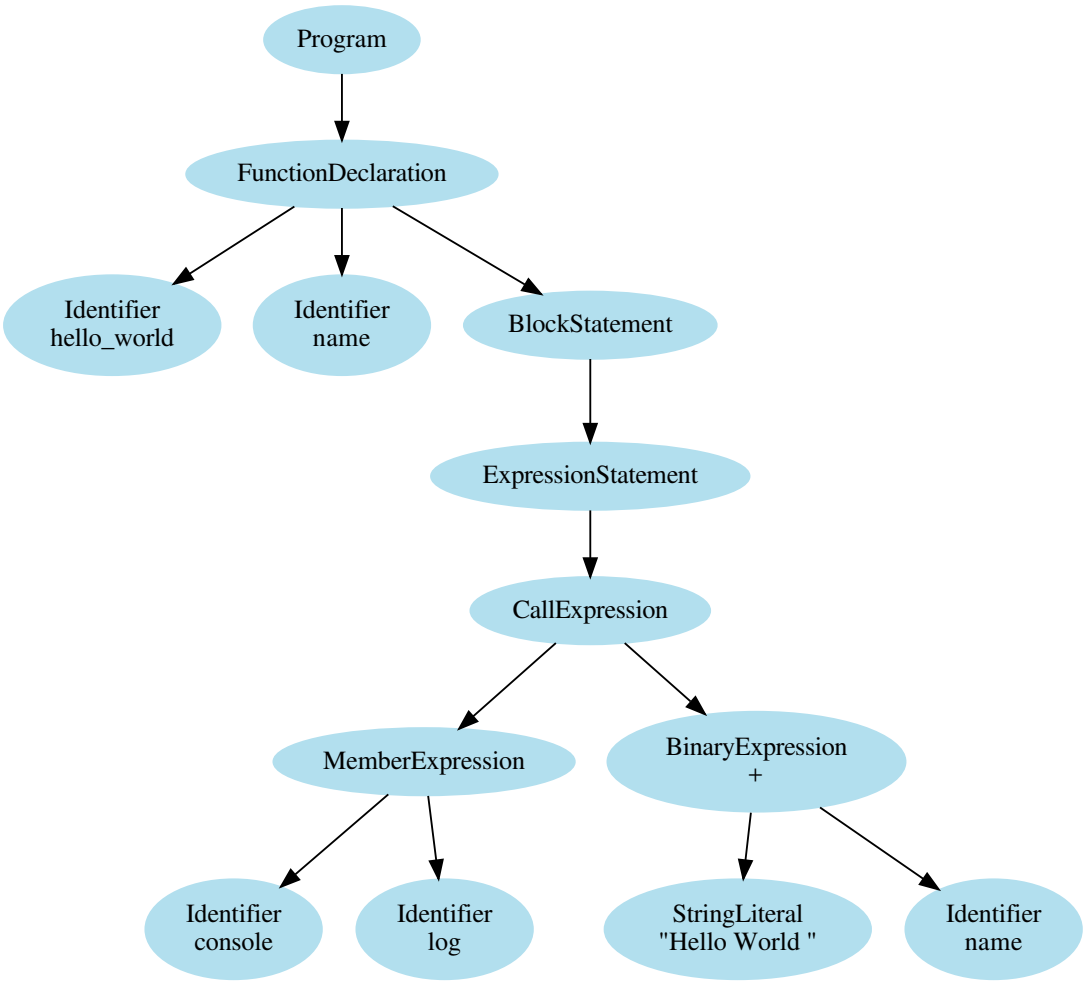
1.1.2 Transformácia

Transformácia AST je najvýznamnejšia fáza transpilácie, pri ktorej sa na strom aplikujú transformácie zadané tvorcom pluginu. Transformácia má formu návrhového vzoru visitor a udáva funkciu, ktorá sa vykoná pri navštívení vrcholu istého typu. Funkcia má prístup k parametrom daného vrchola a nadradeného vrchola a môže okrem iného:

- pridať nový vrchol pred alebo za navštívený vrchol,
- nahradiť alebo odstrániť navštívený vrchol,
- pridať premennú do "scope" (rozsah, v ktorom je platná),
- premenovať existujúcu premennú,
- spustiť nové prehľadávanie z navštíveného vrchola alebo niektorého zo synov.

1.1.3 Generovanie

Po vykonaní transformácií sa výsledný AST prehľadáva do hĺbky a jednotlivé vrcholy sa prepisujú do formy zdrojového kódu.



Obr. 1.1: Abstraktný syntaktický strom.

1.2 Podsystémy

Babel.js má modulárnu architektúru a skladá sa z viacerých balíčkov (npm packages), ktoré je možné použiť aj samostatne. Do veľkej miery kopírujú vyššie popísané fázy transpilácie.

1.2.1 babylon

Babylon je parser založený na pôvodnom parsere Acorn [4], ktorý podporuje najnovší štandard ECMAScriptu a tiež neštandardizované rozšírenia syntaxy ako napríklad JSX [10] alebo Flow [5]. Jeho výstupom je AST.

1.2.2 babel-traverse

Babel-traverse je modul implementujúci prehľadávanie AST a udržiavanie jeho stavu, vrátene pridávania a odoberania vrcholov.

1.2.3 babel-types

Babel-types je modul zodpovedný za vytváranie a validáciu vrcholov a ich následný prepis do zdrojového kódu. Okrem štandardizovaných prvkov jazyka umožňuje tiež vytváranie nových.

1.2.4 babel-generator

Babel-generator je modul zodpovedný za prepis AST do výsledného zdrojového kódu spolu so "source maps" (technológia podporovaná všetkými modernými internetovými prehliadačmi definujúca mapovanie medzi pôvodným a transpilovaným zdrojovým kódom a umožňujúca tak jednoduchšie ladenie transpilovaného kódu).

1.3 Ukážkový plugin

Plugin do Babel.js je Javascriptový modul, ktorý exportuje práve jednu funkciu, ktorá po zavolaní vráti objekt, ktorý obsahuje objekt *visitor*, v ktorom sú ako kľúče názvy vrcholov AST a k nim priradené hodnoty sú funkcie, ktoré sa zavolajú, keď sa *babel-traverse* pri prehľadávaní AST dostane na daný vrchol. Každý visitor môže definovať funkciu *enter*, ktorá sa zavolá pri vstupovaní do vrchola a funkciu *exit*, ktorá sa zavolá, keď sa prehľadali všetci jeho synovia a vrchol sa opúšťa. Nasledujúci príklad obsahuje visitor na unárne operácie, ktoré v prípade zhody operátora prepíše na volanie funkcie:

```
1  export default function ({types: t}) {
2    return {
3      visitor: {
4        UnaryExpression(path) {
5          if (['++', '--', '+', '-', '~'].indexOf(path.node.operator)
               !== -1) {
6            path.replaceWith(t.callExpression(
7              t.identifier('checkCastingUnaryPrefix'),
8              [
9                path.node.argument,
10               t.stringLiteral(path.node.operator)
11             ]
12           ));
13         }
14       }
15     }
16   };
17 }
```

Listing 1.3: Ukážkový plugin

Kapitola 2

Striktný a superstriktný mód

V tejto kapitole popisujeme striktný a superstriktný mód a programovací jazyk Python [12] a ich rozdiely.

2.1 Striktný mód

Striktný mód je vlastnosť pridaná vo verzii ECMAScript 5, ktorá obmedzuje použitie určitých vlastností jazyka, ktoré sú považované za nejasné a náchylné k chybám, a pridáva zmeny, ktoré robia kód v striktnom móde bezpečnejší.

Kontkrétne sa jedná o zmeny popísané v dodatku *ECMAScript 2015 Language Specification* [15]:

- Medzi rezervované slová sa pridávajú *implements*, *interface*, *let*, *package*, *private*, *protected*, *public*, *static* a *yield*, lebo sa očakáva, že budú použité v budúcich verziách a nemôžu byť preto použité ako názvy funkcií a premenných.

- Priradenie do neexistujúcej premennej nevytvorí danú globálnu premennú, ale spôsobí výnimku *ReferenceError*. Tým sa ľahšie odhalia chyby, keď programátor omylom spravil chybu v názve premennej a hodnotu priradí do odlišnej premennej.

```
1  var variable;
2  variable = 1; // ok
3  varaible = 2; // ReferenceError: varaible is not defined
```

Listing 2.1: Globálna premenná

- Nie je možné zadávať číselné literály v osmičkovej sústave. Ak literál začína nulou a obsahuje iba číslice menšie alebo rovné ako 7, nastane syntaktická chyba. Ak literál obsahuje aspoň jednu číslicu väčšiu ako 7, počiatočné nuly sa odignorujú a zvyšok literálu sa interpretuje ako číslo v desiatkovej sústeve.

```
1  0555 === 365;
2  0555; // SyntaxError: Octal literals are not allowed in strict
       mode.
3  0558 === 558; // ok
```

Listing 2.2: Číselné literály

- Nie je možné použiť výraz *with*. Tento výraz umožňuje pristupovať k vlastnostiam objektu, ktorý je argument výrazu, priamo bez nutnosti písať prefix objektu.

```
1  var a = 1;
2  var obj = {a: 2, b: 3}
3
4  with (obj) {
5    a = 47;
6  }
7
8  a === 1;
9  obj.a === 47;
10
11 // 'use strict';
12 with (obj) { // SyntaxError: Strict mode code may not include a
       with statement
13   a = 47;
14 }
```

Listing 2.3: With

- Identifikátory *eval* a *arguments* majú v ECMAScripte špeciálne postavenie, ale v striktnom móde sa správajú viac ako kľúčové slová. Identifikátor *arguments* obsahuje zoznam argumentov funkcie. Je podobný poľu, lebo má metódu *length* a je možné pristupovať k prvkom pomocou indexov, ale nie je to pole.
  - Nie je možné zmeniť hodnotu identifikátorov *eval* a *arguments*.

```
1  eval = myEval; // SyntaxError: Unexpected eval or arguments
       in strict mode
2  arguments++; // SyntaxError: Unexpected eval or arguments
     in strict mode
```

Listing 2.4: eval a arguments

  - Prístup k atribútom *caller* a *callee* identifikátora *arguments* vyhodí výnimku *TypeError*
  - Identifikátory *eval* a *arguments* nemôžu byť použité ako identifikátor *Catch* výrazu

- Hodnota *this* nie je pretypovaná na objekt a to ani ak je daná volaním funkcie *Function.prototype.apply* alebo *Function.prototype.call*. Ak je *null* alebo *undefined*, nenahradí sa globálnym objektom.
- Použitie operátora *delete* na premenné, funkčné argumenty a názov funkcie je *SyntaxError*.
- Použitie operátora *delete* na prvok objektu, ktorý má nastavený atribút *[[Configurable]]: false*, je *SyntaxError*.
- Je *SyntaxError* použiť identifikátor viac ako raz v zozname formálnych parametrov funkcie.

2.1.1 Spôsob použitia

ECMAScript kód sa vykoná v striktnom móde, ak je splnená aspoň jedna z nasledujúcich podmienok:

- Globálny kód, ktorého úvodné direktívy obsahujú direktívu 'use strict'.
- Kód modulu sa automaticky vykonáva v striktnom móde.
- Všetky časti deklarácie triedy sa vykonávajú v striktnom móde a to aj v prípade výrazu.
- Kód spustený funkciou *eval*, ak jeho úvodné direktívy obsahujú direktívu "use strict", respektívne ak sa funkcia *eval* nachádza v kóde, ktorý sa vykonáva v striktnom móde.
- Kód funkcie, ak jej úvodné direktívy obsahujú direktívu "use strict", respektívne ak je funkcia deklarovaná v kóde, ktorý sa vykonáva v striktnom móde. Toto platí aj pre generátory, metódy a arrow funkcie.
- Kód funkcie, ak je vytvorená konštruktorom *Function* alebo *Generator* a úvodné direktívy tela funkcie obsahujú direktívu "use strict".

2.2 Superstriktný mód

Superstriktný mód mení sémentiku Javascriptu v troch oblastiach: overovanie prístupu k atribútom, zákaz implicitného pretypovania a overovanie numerických chýb pri práci s číslami s plávajúcou desatinnou čiarkou.

2.2.1 Prístup k atribútom

Pod prístupom k atribútom v tejto časti chápeme prístup nielen prístup k prvkom objektu, poľa alebo reťazca pomocou hranatých zátvoriek, ale aj prístup k metódam a premenným objektu pomocou bodky.

```
1  ['a', 'b', 'c'][0] === 'a';
2  ({a: 1, b: 2})['a'] === 1;
3  ({a: 1, b: 2}).a === 1;
```

Listing 2.5: Rôzne prístupy k atribútom

Sémantika v Javascripte

Gramatika Javascriptu popisuje prístup k prvkom pomocou neterminálu *MemberExpression*, pričom pre túto prácu sú relevantné práve tri pravidlá formálne definujúce 'prístup cez hranaté zátvorky' (bracket notation) a 'prístup cez bodku' (dot notation):

```
1  MemberExpression -> PrimaryExpression,
2  MemberExpression -> MemberExpression [ Expression ],
3  MemberExpression -> MemberExpression . IdentifierName
```

Listing 2.6: Gramatika MemberExpression

Štandard definuje dve abstraktné operácie pre prístup k prvkom: *Get(O, P)* a *GetV(V, P)*.

Operácia *Get* slúži na získanie prvku objektu. Skontroluje, či je O objekt, skontroluje, či P je *property key* a potom vráti hodnotu O.[[Get]](P, O). *[[Get]]* je interná metóda objektu, ktorá sa pomocou metódy *[[GetOwnProperty]]* pokúsi získať hodnotu priradenú k vlastnému prvok objektu, a ak neexistuje (*[[GetOwnProperty]]*, vráti *undefined*), inak sa rekurzívne zavolá na prototyp objektu a ak ani ten neobsahuje prvok, vráti *undefined*, inak vráti hodnotu priradenú k hľadanému prvku.

Operácia *GetV* je identická s operáciou *Get* s tým rozdielom, že prvý parameter nemusí byť objekt, ale ľubovoľná Javascriptová hodnota (undefined, null, boolean, number, string, symbol, object), ktorá sa operáciou *ToObject* pretransformuje na objekt na ktorom sú následne vykonané rovnaké kroky ako v operácii *Get*.

Operácia *ToObject* akceptuje jeden argument *argument*. Ak je typ argumenta Object, vráti ho, ak je Undefined alebo Null, vyhodí výnimku *TypeError*, a ak je Boolean, vráti objekt Boolean s danú hodnotu (v Javascripte existujú primitívne hodnoty *true* a *false* typu *boolean* a objekty Boolean(true) a Boolean(false) typu *object*, ktoré sa ale vedia priamočiaro pretypovať na primitívne hodnoty). Pre typy Number, String a Symbol analogicky vráti objekt s danou hodnotou.

Sémantika v Pythone

Python používa na prístup k prvkom dva neterminály *attributeref* a *subscription*:

```
attributeref -> primary "." identifier
subscription -> primary "[" expression_list "]"
```

Listing 2.7: Gramatika attributeref a subscription

V prípade *attributeref* sa najskôr pokúsi nájsť atribút na zvyčajných miestach - atribút inštancie, respektívne vyššie v hierarchii dedenia -, a v prípade neúspechu sa zavolá metóda objektu *__getattr__(self, name)*, ktorá buď vráti hodnotu priradenú k atribútu alebo vyhodí výnimku *AttributeError*. Funkciu *__getattr__(self, name)* objekt nemusí mať definovanú a vtedy nenájdenie atribútu na zvyčajných miestach hneď vyhodí výnimku *AttributeError*. V prípade, že je potrebná funkcia, ktorá sa zavolá pri každom prístupe k prvku, je možné implementovať funkciu *__getattribute__(self, name)*.

V prípade, že sa *subscription* aplikuje na vstavané objekty *list* alebo *dictionary*, pokúsi sa nájsť prvok priradený k danému kľúču alebo vyhodí výnimku *IndexError* (list) alebo *KeyError* (dictionary). Prípadne je možné objektu implementovať funkciu *__getitem__(self, key)*, ktorá tiež buď vráti správny prvok alebo vyhodí vhodnú výnimku.

Sémantika v superstriknom móde

Z vyššie uvedeného porovnania sémantiky Javascriptu a Pythonu sa ako najväčší rozdiel javí správanie v prípade nenájdeného prvku. V Javascripte sa vráti hodnota *undefined*, to je ale mätúce, lebo aj hodnota hľadaného prvku mohla byť *undefined*, a teda z testovania návratovej hodnoty nie je možné zistiť existenciu prvku. Druhý problém spočíva v tom, že ak programátor očakáva existenciu prvku a netestuje jeho prítomnosť, dostane hodnotu, s ktorou môže ďalej pracovať a problém sa prejaví až neskôr na inom mieste v kóde. V ukážke programátor očakával, že objekt obsahuje v oddelených atribútoch krstné meno a priezvisko, ale program pokračoval ďalej s nesprávnymi hodnotami bez vyhodenia výnimky. Podobná situácia by nastala, ak by programátor spravil preklep v názve atribútu.

```
function sayName(name) {
  console.log(name);
};

var person = {name: 'John Doe', email: 'john@doe.com', address:
    undefined};
sayName(person.firstName + ' ' + person.lastName);
```

```
7  // undefined undefined
```

Listing 2.8: Ukážka prístupu k neexistujúcemu prvku

Preto sa v superstriknom móde najskôr skontroluje existencia atribútu pomocou operátora *in* a ak neexistuje vyhodí sa výnimka *NonExistingPropertyException*, ktorá v prípade nezachytenia aj vypíše zoznam existujúcich atribútov objektu. V prípade, že atribút existuje, zavolá sa rovnaký kód ako v pôvodnej sémantike Javascriptu.

Podporované nie sú iba klasické Javascriptové objekty, ale všetky dátové štruktúry implementujúce protokol has/get, t.j. majúce funkciu *has* akceptujúcu kľúč a vracajúcu *true* práve vtedy, keď daný prvok existuje, a funkciu *get*, ktorá tiež akceptuje kľúč a vráti k nemu priradenú hodnotu. Týmto spôsobom je teda zahrnutá podpora pre objekt *Map* z knižnice *Immutable.js* poskytujúcej perzistentné dátové štruktúry a pre objekty *Map* a *WeakMap* definované v EcmaScripte 6.

Tiež je programátorovi umožnené pridať do objektu vlastnú funkciu, ktorá overí prítomnosť prvku a vráti ho.

2.2.2 Implicitné pretypovania

V tejto časti sa zaoberáme implicitným pretypovaním hlavne v kontexte aritmetických a logických operácii.

Sémantika v Javascripte

V prípade Javascriptu je správanie závislé od typu operácie.

Multiplikatívne operácie Medzi multiplikatívne operácie patria násobenie, delenie a modulo (zvyšok po celočíselnom delení) a sú popísané neterminálom *MultiplicativeExpression*.

```
1  MultiplicativeExpression -> UnaryExpression
2  MultiplicativeExpression -> MultiplicativeExpression
      MultiplicativeOperator UnaryExpression
```

Listing 2.9: Gramatika MultiplicativeExpression

Vyhodnotenie *MultiplicativeExpression* prebieha nasledovne: vyhodnotia sa oba operandy a na výsledné hodnoty sa zavolá abstraktná operácia *ToNumber*. Násobenie a delenie sa následne vykoná podľa štandardu *IEEE 754-2008* [3] definujúceho aritmetiku s pohyblivou rádovou čiarkou. Modulo sa počíta vzorcom $r = leftOperand - rightOperand \times n$, pričom n je výsledok delenia x/y, ale nie je zaokrúhlený k najbližšiemu číslu ako podľa štandardu *IEEE 754-2008* [3], ale zachová sa iba celá časť čísla.

Operácia *ToNumber* akceptuje jeden argument *argument*. Ak je typ argumentu Number, vráti ho, ak je Undefined, vráti NaN, ak je Null, vráti +0, ak je Boolean, pre true vráti 1 a pre false vráti +0. Ak je typ argumentu String, pokúsi sa ho interpretovať ako číslo na základe definovaj gramatiky a ak zlyhá, vráti NaN. Ak je typ argumentu Object, najskôr naň zavolá operáciu *ToPrimitive* a potom *ToNumber*.

Aditívne operácie Medzi multiplikatívne operácie patria sčítanie a odčítanie a sú popísané neterminálom *AdditiveExpression*.

```
1  AdditiveExpression -> MultiplicativeExpression
2  AdditiveExpression -> AdditiveExpression + MultiplicativeExpression
3  AdditiveExpression -> AdditiveExpression - MultiplicativeExpression
```

Listing 2.10: Gramatika AdditiveExpression

Vyhodnotenie *AdditiveExpression* prebieha nasledovne: vyhodnotia sa oba operandy a na výsledné hodnoty sa zavolá abstraktná operácia *ToPrimitive*. Ak je operácia sčítanie a jeden z operandov je typu String, obidva sa pretypujú na reťazce pomocou operácie *ToString* a výsledné hodnoty sa zreťazia (concatenation). Inak sa obidva operandy pretypujú na čísla pomocou operácie *ToNumber* a sčítajú, respektívne odčítajú, podľa pravidiel definovaných v štandarde *IEEE 754-2008* [3].

Operácia *ToPrimitive* slúži na konverziu hodnôt typu Object na primitívne hodnoty a akceptuje jeden argument *argument*. Ak je typ argumentu Undefined, Null, Boolean, Number, String alebo Symbol, vráti *argument*. Inak je typ argumentu Object a pokúsi sa použiť jeho metódy *@@toPrimitive*, *toString* a *valueOf*. Zavolá prvú, ktorá je definovaná a vráti jej výsledok, ak je to primitívna hodnota. Inak vyhodí výnimku *TypeError*.

Postfixové operácie Medzi postfixové operácie patria postfixové inkrementovanie a dekrementovanie a sú popísané neterminálom *PostfixExpression*.

```
1  PostfixExpression -> LeftHandSideExpression ++
2  PostfixExpression -> LeftHandSideExpression --
```

Listing 2.11: Gramatika PostfixExpression

Vyhodnotenie *PostfixExpression* prebieha nasledovne: vyhodnotí sa operand a výsledná hodnota sa pretypuje na číslo operáciou *ToNumber*. Odloží sa aktuálna hodnota, hodnota premennej sa zvýši, respektívne zníži, o 1 a vráti sa pôvodná hodnota.

Zvyšné operácie Zvyšné aritmetické, logické a binárne operácie nebudeme rozoberať do väčších detailov, nakoľko ich sémantika je do veľkej miery analogická s vyššie popísanými. Bežný vzor je vyhodnotiť operandy, zavolať na ne abstraktnú operáciu, ktorá ich skonvertuje na hodnoty vhodného typu a následne na nich vykonať danú operáciu.

Sémantika v Pythone

Python používa na definíciou aritmetických operácií dva neterminály *m_ expr* a *a_ expr*:

```
1  m_expr ->  u_expr
2  m_expr ->  m_expr "*" u_expr
3  m_expr ->  m_expr "@" m_expr
4  m_expr ->  m_expr "//" u_expr
5  m_expr ->  m_expr "/" u_expr
6  m_expr ->  m_expr "%" u_expr
7
8  a_expr ->  m_expr
9  a_expr -> a_expr "+" m_expr
10 a_expr -> a_expr "-" m_expr
```

Listing 2.12: Gramatika m_expr a a_expr

Operácie spomenuté vo vyššie uvedenej gramatike môžu byť vykonané iba na objektoch, ktoré implementujú metódy s danými názvami. Napríklad operácia sčítania zavolá metódu __add__ prvého operandu a ako parameter dostane druhý operand. V prípade, že hľadaná metóda nie je definovaná, vyhodí sa výnimka *TypeError*. Analogicky sú definované aj logické operácie (porovnanie $<$ je implementované ako metóda __lt__) a bitové operácie (bitový and je implementované ako metóda __and__). Tieto metódy môžu implementovať aj užívateľom vytvorené triedy.

Číselné typy tvoria hierarchiu dedenia - komplexné čísla (*numbers.Complex*) dedia od *numbers.Number*, reálne čísla (*numbers.Real*) dedia od *numbers.Complex*, racionálne čísla (*numbers.Rational*) dedia od *numbers.Real* a tak ďalej. Preto v prípade použitia operácie na dve hodnoty odlišných typov sa hodnota, ktorej typ je v hierarchii nižšie, implicitne pretypuje na typ, ktorý je v hierarchii vyššie. Napríklad v prípade sčítania reálneho a celého čísla sa celočíselné pretypuje na reálne a následne sa zavolá metóda __add__ prvého operandu.

Ale v prípade sčítania reťazca a celého čísla neexistuje žiady spoločný predok, preto výsledkom operácie je *TypeError*. Riešenie je explicitne pretypovať celé číslo na reťazec funkciou *str*.

Sémantika v superstriknom móde

Z vyššie uvedeného porovnania sémantiky Javascriptu a Pythonu sa ako najväčší rozdiel javí, že Javascript sa snaží aspoň jeden z operandov operácie pretypovať, aby boli oba rovnakého typu a následne na nich vykoná operáciu. Python na druhú stranu v prípade základných typov definuje operácie pre určité "rozumné" kombinácie typov a pre tie zvyšné vyhodí výnimku *TypeError*. Výhodou je, že ak programátor naozaj potrebuje použiť operátor na nejakú exotickú kombináciu typov, musí ich explicitne pretypovať. Ak to ale nebol jeho zámer, ale omyl, dozvie sa to hneď na mieste vzniku,

čo je jednoduchšie na ladenie ako dostať implicitne pretypovanú hodnotu, s ktorou sa môže ďalej pracovať a problém sa odhalí neskôr a na inom mieste.

```
1  {} + {} === '[object Object][object Object]'
```

Listing 2.13: Ukážka implicitného pretypovania

Preto je v superstriktnom móde možné aritmetické operácia, bitové posuny, bitové operácie a porovnávania okrem rovnosti a nerovnosti vykonať iba na dvojici čísel, s výnimkou sčítania, ktoré je možné použiť aj na zreťazenie dvoch reťazcov.

Logické operátory úmyselne neobmedzujeme, lebo sú často používané na overenie, či premenné nemajú hodnotu *null* alebo *undefined*, respektívne či nie sú prázdny reťazec.

```
1  if (some_string && some_boject) { }
2
3  if (Boolean(some_string) && Boolean(some_boject)) { }
4  if ((typeof some_string === 'string') && (some_object !== null) &&
       (some_object !== undefined)) {}
```

Listing 2.14: Praktické využitie pretypovania na pravdivostné hodnoty

2.2.3 Numerické chyby

Sémantika v Javascripte a v Pythone

Obidva jazyky počítajú čísla s pohyblivou rádovou čiarkou podľa štandardu *IEEE-754*, preto v obidvoch môžu mať čísla hodnoty NaN, Infinity, -Infinity, +0, -0 a aritmetika s nimi je definovaná v štandarde: napríklad Infinity + Infinity = Infinity a 1 / 0 = Infinity.

Sémantika v superstriknom móde

V superstriktnom móde sa vyhodí výnimka *InvalidOperationResultException*, ak je výsledkom aritmetickej operácie NaN, Infinity alebo -Infinity, a v prípade delenia nulou sa vyhodí *DivisionByZeroException*.

2.2.4 Spôsob použitia

Plugin je v jednom z troch módov v závislosti od hodnoty parametru pluginu "directivePolicy". Ak je "opt in", prekladajú sa iba súbory, ktoré majú na začiatku direktívu "use superstrict", ak je "opt out", prekladajú sa naopak všetky súbory okrem tých, ktoré majú na začiatku direktívu "use !superstrict", alebo ak nie je nastavená alebo je "everything", prekladajú sa všetky súbory. Možnosť "opt in" je užitočná pri väčšom existujúcom projekte, ktorý je nepraktické celý naraz prispôsobiť superstriktnému módu. Možnosť "opt out" je zase užitočná v momente, keď je už takmer celý projekt

prispôsobený superstriktnému módu a je jednoduchšie označiť súbory, ktoré nechceme prekladať, ako označovať súbory, ktoré prekladať chceme.

Tiež je potrebné nastaviť parametre pluginu "safeGetFilePath" a "checkCastingFilePath" na cesty k súborom "safeGet.js" a "checkCasting.js", v ktorých sú pomocné funkcie superstriktného módu.

Kapitola 3

Implementácia

V tejto kapitole popisujem spôsob implementácie superstriktného módu, vrátane slepých uličiek.

3.1 Prístup k prvkom - neúspešná verzia

Myšlienka prvej verzie bola pred každý výraz *MemberExpression* pridať overenie existenciu prvku pomocou volania funkcie *assert*, ktorá vyhodí výnimku *AssertExpression*, ak nie je splnená podmienka. Volanie funkcie je v abstraktnom syntaktickom strome definované ako vrchol typu *CallExpression*. Pridať nový vrchol pred existujúci je možné pomocou funkcie *path.insertBefore*, pričom *path* je argument visitora a popisuje relatívnu pozíciu v abstraktnom syntaktickom strome. Ak sa pridá vrchol s volaním assertu priamo pred *MemberExpression*, v niektorých prípadoch dostaneme korektný kód ako v prípade (3.1), ale tiež môže vzniknúť AST, ktorý nie je odvoditeľný v gramatike Javascriptu ako v príklade (3.2) - pravá strana priradenia *AssignmentExpression* sa má skladať z jedného výrazu, ktorým je aktuálne *MemberExpression*, a preto nemá zmysel pridať pred neho nový výraz *CallExpression*.

```
1  b.x;
2
3  assert(b in x);
4  b.x;
```

Listing 3.1: Prvá verzia - fungujúci kód

```
1  a = b.x; // unknown: too much recursion
```

Listing 3.2: Prvá verzia - nefungujúci kód

To sa dá vyriešiť vytvorením visitora na *Statement*, z ktorého sa spustí prehľadávanie potomkov a v prípade nájdenia *MemberExpression* pridajú nový vrchol s asertom pred *Statement*. Nevýhodou tohto riešenia je nižšia efektivita, lebo nie je možné zmodi-

fikovať celý AST na jeden prechod kvôli vyššie spomenutému spusteniu prehľadávania v prehľadávaní - potomkovia *Statement* sa navštívia minimálne dvakrát.

Tiež je potrebné si uvedomiť, že očakávame, že superstriktný mód bude podporovať aj prístup k prvkom cez hranaté zátvorky a index teda môže byť výraz, napríklad volanie funkcie. Výraz sa nemusí vyhodnocovať deterministicky (napríklad závisí od globálnej premennej alebo komunikuje po sieti), preto by bolo potrebné si najskôr index vypočítať, uložiť do pomocnej premennej, otestovať existenciu a nahradiť ňou *MemberExpression*. Výsledný kód by mohol vyzerať nasledovne:

```
1  var global_var = 0;
2
3  var fun = function(a) {
4  global_var++;
5  return global_var + a;
6  };
7
8  a[fun(0)]; // a[1]
9  a[fun(1)]; // a[3]
```

Listing 3.3: Nedeterministický index - pôvodný kód

```
1  var temp_var_1 = fun(0);
2  var temp_var_2 = fun(1);
3
4  assert(temp_var_1 in a);
5  assert(temp_var_2 in a);
6
7  a[temp_var_1]; // a[1]
8  a[temp_var_2]; // a[3]
```

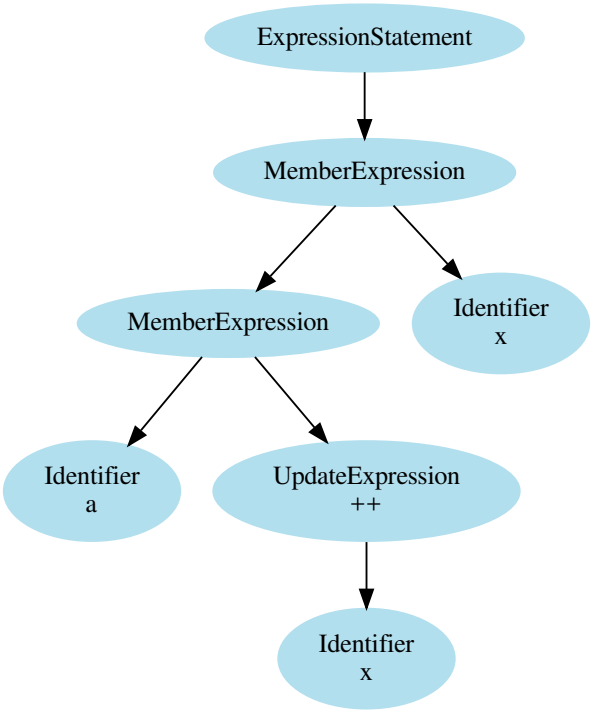
Listing 3.4: Nedeterministický index - po preložení posledných dvoch riadkov

Existujú ale aj programy, na ktorých tento prístup zlyhá. Z výrazu a[x++][x] sa vytvorí AST ako na grafe 3.1. Pomocné premenné sa vytvárajú v poradí navštívených vrcholov *ExpressionStatement*, teda najskôr x-tý prvok a[x++] a až potom x++-tý prvok a. Prvá sa vyhodnotí a uloží hodnota x a potom sa vyhodnotí a uloží hodnota x++. Správne sa ale najskôr malo vyhodnotiť x++ a až potom x.

```
1  a[x++][x]
2
3  var temp_var_1 = x;
4  var temp_var_2 = x++;
5
6  assert(temp_var_2 in a);
7  assert(temp_var_1_1 in a[temp_var_2]);
8  assert(temp_var_2 in a);
```

```
9  a[temp_var_2][temp_var_1];
```

Listing 3.5: Nedeterministický index - zlý vstup



Obr. 3.1: AST a[x++][x]

Ďalším problémom prvej verzie je, že aj *Statement* môže obsahovať *Statement*, napríklad potomkom podmienky *IfStatement* môže byť while cyklus *WhileStatement*, preto by bolo potrebné počas prehľadávania *Statement* ignorovať zvyšné *Statement*, čo by ešte viac komplikovalo kód.

3.2 Prístup k prvkom - úspešná verzia

Pôvodná verzia začala byť komplikovaná a nepokrývala všetky okrajové prípady, preto sme sa rozhodli zmeniť prístup. Namiesto vytvárania dočasných premenných sme nahradili *MemberExpression* volaním funkcie, ktorá vykoná overenie existencie prvku a buď vráti hľadanú hodnotu alebo vyhodí výnimku.

```
1  for (var i = 0; i < 10; i++) {
2    a[i];
3  }
4
5  for (var i = 0; i < 10; i++) {
6    safeGetAttr(a, i);
7  }
```

Listing 3.6: Druhá verzia - pole

To sa prejavilo zjednodušením nielen kódom pluginu, ale aj výsledného transpilovaného kódu. Overenie prítomnosti pomocou operátoru *in* bolo nedostatočné, lebo je definovaný iba pre hodnoty typu object, teda objekty a polia a nie pre reťazce. Prístup cez číselný index k znaku v reťazci by sa teda neoveroval pomocou operátora 'in', ale zistením, či je index kladný a či je menší ako dĺžka reťazca, ako v príklade 3.7. Ak by sme postupovali pôvodným spôsobom, pred každým prístupom k prvku by bola podmienka na základe typu hodnoty a minimálne dva spôsoby ako overiť prítomnosť prvku v rôznych objektoch.

```
1  'hello_word'[1]; // 'e'
2  1 in 'hello_world' // Cannot use 'in' operator to search for '1' in
       hello_world'
3  (1 >= 0) && (1 < 'hello_world'.length)
```

Listing 3.7: Druhá verzia - reťazec

Keďže sme rôzne metódy overovania skoncentrovali na jedno miesto do funkcie, bolo jednoduché pridať podporu aj pre perzistentnú dátovú štruktúru Map knižnice Immutable.js. Dátová štruktúra Map, teda asociatívne pole, umožňuje overiť prítomnosť kľúča pomocou metódy *has* a získať prvok priradený k danému kľúču pomocou metódy *get*, pričom rovnako ako bežné Javascriptové objekty vráti *undefined*, ak ku kľúču nie je priradený žiadny prvok. Zistili sme, že objekty *Map* a *WeakMap* z ES6 implementujú rovnaké metódy, preto neoverujeme typ objektu, ale iba to, či obsahuje tieto dve metódy.

Pre objekty podporujúce vyššie popísaný protokol umožňujeme prístup k prvkom pomocou hranatých zátvoriek, ale nie pomocou bodky. Problém by bol s kľúčmi, ktoré sa volajú rovnako ako metódy, napríklad 'get'. Ak by sa vrátil prvok priradený ku kľúču, nebolo by možné zavolať metódu. Ak by sa vrátila metóda, ale prístup pomocou kľúča 'get_2', ktorý nie je metóda *Map*, by vrátil k nemu priradený prvok, dostali by sme nekonzistentné a mätúce správanie. Preto sme sa rozhodli, že cez bodku bude možné pristupovať iba k atribútom a metódam objektu *Map* a k prvkom sa bude pristupovať iba cez hranaté zátvorky. Táto zmena v správaní si vyžiadala vytvoriť samostatné funkcie pre obidva spôsoby prístupu - *safeGetItem* pre prístup cez bodku a *safeGetAttr* pre prístup cez hranaté zátvorky.

Tiež je možné ľudovoľnému objektu deklarovať metódu *@@safeGetAttr* (symbol safeGetAttr), v prípade ktorej prítomnosti sa použije namiesto operátora *in* na overenie prítomnosti prvkov.

Ako problematické sa ukázali prístupy k metódam. Ak sa metóda zavolá cez bodku ako prvok objektu, nastaví sa kľúčové slovo *this* na daný objekt. Ak sa metóda uloží do premennej a následne zavolá, *this* sa nastaví na *undefined*. Preto sme vytvorili a aplikovali na funkcie *safeGetItem* a *safeGetAttr* dekorátor *conditionalBind*, ktorý zavolá danú funkciu a ak bol typ hľadaného prvku *function*, nastaví mu pomocou

funkcie *bind this* na objekt, ktorého je metódou.

Ak ani jedna metóda nenájde prvok, vyhodí sa výnimka *NonExistingPropertyException*.

Rozhodli sme sa tiež zmeniť sémantiku operátora *in*, aby v prípade zavolania na dva reťazce overil, či je prvý podreťazcom druhého, aby sa pokúsil použiť metódu *has*, ak ju objekt má definovanú a aby sa na poliach správal intuitívne a netestoval prítomnosť indexu, ale prvku.

```
1  1 in [1, 4, 9] === true;
2  4 in [1, 4, 9] === false;
```

Listing 3.8: Druhá verzia - operátor in

3.3 Implicitné pretypovania

Zabránenie implicitného pretypovania sme implementovali analogicky ako overenie prístupu k neexistujúcemu prvku. Vytvorili sme visitory na unárne operácie (*UnaryExpression*), binárne operácie (*BinaryExpression*) a inkrementovanie a dekrementovanie (*UpdateExpression*). V prípade nájdenia vhodného vrcholu overíme, či sa jedná o jeden z operátorov, ktorého operandom chceme zakázať implicitné pretypovanie, a nahradíme ho volaním jednej z funkcií, ktorá vykoná ešte dodatočné overenia a buď vráti vypočítanú hodnotu alebo vyhodí výnimku.

Binárne operátory overuje funkcia *checkCastingBinary* a jedná sa o sčítanie, respektívne zreťazenie (+), odčítanie (-), násobenie (*), delenie (/), zvyšok po delení (%), bitové posuny ($\ll$, $\gg$, $\ggg$), bitové operátory (&, ^, |), porovnania ($<$, $>$, $<=$, $>=$) a operátor in. Pre operátor sčítania môžu byť obidva operandy buď čísla alebo reťazce, zvyšné operátory vyžadujú, aby oba operandy boli čísla, a delenie aj zvyšok po delení vyžadujú, aby deliteľ nebol nula. Ak je to splnené, vráti očakávanú hodnotu alebo vyhodí výnimku *ImplicitCastingException* alebo *DivisionByZeroException*.

Unárne operátory sa spracujú odlišne podľa toho, či sú prefixové alebo postfixové. Prefixové najskôr zmenia svoju hodnotu a potom ju vrátia. Patria medzi ne inkrementovanie (++), dekrementovanie (–), plus (+), mínus (-) a bitový not (~). Overujú sa funkciou *checkCastingUnaryPrefix*, ktorá overí, či je operand číslo a vráti upravenú hodnotu alebo vyhodí výnimku *ImplicitCastingException*.

Postfixové najskôr vrátia aktuálnu hodnotu a potom zmenia hodnotu premennej, ktorá je operandom. Javascript nepodporuje predať funkcii parametre referenciou, čo by umožnilo uložiť si pôvodnú hodnotu, upraviť premennú a potom vrátiť pôvodnú hodnotu. Namiesto toho sme sa rozhodli nahradiť postfixovú operáciu volaním funkcie *checkCastingUnaryPostfix*, ktorej prvým parametrom bude operand (vďaka čomu získame pôvodnú hodnotu), druhým postfixová operácia (ktorá zabezpečí upravenie pre-

mennej) a tretím operátor. Funkcia overí, či je typ operandu number a vráti pôvodnú hodnotu, teda prvý parameter, čím je zachovaná pôvodná sémantika postfixového operátora, ale je pridané overenie typu.

```
1  var a = 0;
2  a++ === 0;
3  a === 1;
4
5  checkCastingUnaryPostfix(a, a++, '+');
```

Listing 3.9: Implicitné pretypovanie - postfixové inkrementovanie

3.4 Pretečenia a numerické chyby

Pretečeniu a numerickým chybám je zabránené aplikovaním dekorátora *checkInvalidValues* na funkcie *checkCastingBinary*, *checkCastingUnaryPrefix* a *checkCastingUnaryPostfix*. Dekorátor zavolá danú funkciu a ak je jej návratová hodnota Infinity, -Infinity alebo NaN, vyhodí výnimku *InvalidOperationResultException*.

3.5 Testy

Počas implementovania vyššie spomenutej funkcionality sme pridávali testy na overenie funkčnosti a dokumentáciu očakávaného výsledkov. Celkovo ich je 147.

Všetky testy majú formu súboru, ktorý sa preloží vyvýjaným pluginom, vyhodnotí funkciou eval a testuje sa, či sa vyhodnotil na výraz ‘ok’. Testy sú preto navrhnuté tak, že buď sa očakáva vyhodenie výnimky a výraz ‘ok’ je v catch bloku, alebo sa očakáva vyhodnotenie bez výnimky a výraz ‘ok’ je na konci try bloku.

Kapitola 4

Overhead

V tejto kapitole sa snažím odhadnúť očakávaný nárast kódu a dopad na rýchlosť spustenia vygenerovanej verzie.

Ako ukážkový program sme vybrali jednoduchý program, ktorý vygeneruje dve náhodné štvorcové matice a vynásobí ich. Meranie dĺžky behu bolo realizované s veľkosťou matíc 200x200 a prvky matice boli medzi 10 a 99 vrátane.

4.1 Nárast kódu

Súbor	Počet riadkov	Počet slov	Počet bytov	Počet bytov po kompresii
original.js	48	143	1020	388
transpiled.js	67	208	2306	604
Nárast	19	65	1286	216
Nárast %	140	145	226	156

Za časť nárastu veľkosti je zodpovedné úvodné importovanie pomocných funkcií. V prípade tohto programu sa jednalo o 442 bytov. Toto množstvo je premenlivé kvôli nutnosti špecifikovať cesty k modulom safe_get.js a check_casting.js, ktoré môžu byť rôzne dlhé.

Za zvyšok nárastu je zodpovedné prepisovanie prístupov k prvkom a aritmetických operácii. Na príklade 4.1 vidíme, že z nárastu kódu o 24 bytov tvorí 18 bytov názov funkcie. Rozhodli sme sa použiť pre pomocné premenné dlhšie názvy, ktoré ale jasne pomenúvajú ich funkcionalitu a umožňujú tak pomerne ľahko zistiť význam kódu aj po transpilácii. Ak by ale mali funkcie kratšie názvy, ideálne jednoznakové, a zaistilo by sa, že dané identifikátory nebudú použité, nárast kódu by bol minimálny.

Otázka veľkosti Javascriptových súborov je najpálčivejšia v prípade kódu, ktorý sa má zobrazovať v rámci webovej stránky u klienta, lebo na rozdiel od serverového kódu si ho musí každý klient stiahnuť. V tom prípade je možné súbory skomprimovať metódou gzip, ktorá je široko podpovaná medzi webovými prehliadačmi aj servermi,

prípadne použiť nástroje ako UglifyJS [2], ktoré zo súboru odstránia komentáre, medzery a zbytočné nové riadky. Ako vidíme v tabuľke, skomprimovaním transpilovaného súboru sme dostali súbor takmer štvrtinovej veľkosti.

```
1  rowNum < size // 13 bytov
2  checkCastingBinary(rowNum, size, '<') // 37 bytov
```

Listing 4.1: Nárast kódu

4.2 Spomalenie kódu

Súbor	1	2	3	4	5	Priemer
original.js	0.17	0.19	0.18	0.18	0.18	0.18
transpiled.js	31.32	33.25	33.66	34.30	34.48	33.40

Spomalenie kódu sa javí ako vážnejší problém než nárast kódu. V tabuľke vidno výsledky piatich meraní dĺžky behu programu v sekundách. Transpilovaný kód bežal 186 ráz pomalšie ako pôvodný, čo nie je prekvapivé, ak si uvedomíme, že jednoduchý prístup k prvku objektu alebo aritmetická operácia, teda základná operácia, na ktorú je interpreter optimalizovaný, sa preloží na volanie funkcie, niekoľko porovnaní a až potom sa operácia vykoná, ak sa nevyhodí výnimka.

Javí sa, že superstrikný mód nie je pripravený na produkčné nasadenie, to ale nebráni použitiu transpilovanej a pomalšej verzie počas vývoja projektu na rýchlejšie a jednoduchšie odhalenie chýb a tam, kde je potrebný výkon, použiť pôvodný kód. Tiež je potrebné si uvedomiť, že nie je potrebné prekladať závislosti projektu, ale iba projekt samotný a závislosti považovať za odladené prvky.

Kapitola 5

Podobné nástroje

V tejto kapitole popisujeme nástroje a knižnice, ktoré sa rovnako ako *babel-plugin-superstrict* pokúšajú spraviť programovanie v Javascripte bezpečnejšie a predvídateľnejšie. Porovnávame ich výhody a nevýhody.

5.1 Flow

Flow [5] je opensource statický type checker pre Javascript vytvorený firmou Facebook a zverejnený v roku 2014. Umožňuje pridať typové anotácie premenným, parametrom funkcií, návratovým hodnotám funkcií a prvkom objektov a tiež sa ich snaží sám odvodiť, ak nie sú explictne zadané. Typové anotácie nie sú súčasťou štandardu ECMAScript a je potrebné ich odstrániť pred spustením kódu interpreterom. Slúži na to plugin systému Babel *babel-plugin-transform-flow-strip-types*.

```
1  var person = {
2    name: 'John'
3  };
4
5  console.log(person.firstName) // Property not found in object
       literal.
6  console.log('Hello ' + person.name) // ok
7
8  true + 'true'; // This type is incompatible with string.
```

Listing 5.1: Flow - odhalené chyby

```
1  function plus(first : number, second : number) : number {
2    return first + second;
3  }
```

Listing 5.2: Flow - ukážka typových anotácií

Podobne ako *babel-plugin-superstrict* používa direktívu "// @flow" na označenie súborov, na ktorých prebehne analýza. To znamená, že v prípade pridania Flowu do

existujúceho projektu nie je potrebné opravovať chyby a pridávať typové anotácie do celého projektu naraz, ale je možné postupne prepisovať podsystémy alebo súbory.

Veľkou výhodou je použitie architektúry klient-server. Pri spustení programu sa vytvorí server, ak neexistuje, a vykoná prvotnú analýzu projektu, v rámci ktorej zostrojí graf závislostí súborov, a vypíše výsledky. Server zatiaľ čaká na zmeny súborov a analyzuje už iba súbory dotknuté zmenou, teda samotné zmenené súbory a rekurzívne súbory, ktoré z nich importujú.

Nevýhodou je, že nakoľko sa jedná o statický type checker, môže pracovať iba s informáciami dostupnými počas kompilácie a napríklad nevie overiť prítomnosť prvkov v objektoch, ktoré sa prijali po sieti alebo zo súborového systému, alebo prístupom k prvkom poľa ako v príklade 5.3. Tiež nevie odvodiť všetky typové anotácie a je potrebné ručne pridať tie neodvodené, čo je práca navyše oproti superstrict módu.

```
1  for (var i = 0; i < 10; i++) {
2    arr[i];
3  }
```

Listing 5.3: Flow - pole

5.2 Typescript

Typescript [6] je programovací jazyk založený na Javascripte s typovými anotáciami vytvorený firmou Microsoft zverejnený v roku 2014. Kód v Typescripte sa podobá kódu napísanému s použitím Flowu - typové anotácie sa píšu za dvojbodkou za premennou, parametrom funkcie alebo návratovou hodnotou funkcie. Nakoľko ale Typescript používa vlastný kompilátor *tsc*, nie je možné použiť Babel, ktorý má väčšiu podporu posledného štandardu ECMAScriptu. Podľa [1] podporuje Babel 6.5 74% ES6 a Typescript 1.8 60%.

V kontexte funkcionality superstriktného módu je Typescript schopný odhaliť porovnateľné typy chýb ako Flow. Chybovou hláškou upozorní na prístup k neexistujúcemu prvku, ak neexistuje v čase prekladu, síce nepovažuje za chybu sčítanie pravdivostnej hodnoty a reťazca ako Flow alebo superstriktný mód, ale sčítanie dvoch prázdnych objektov už chyba je.

```
1  var person = {
2    name: 'John'
3  };
4
5  console.log(person.firstName) // Property not found in object
       literal.
6  console.log('Hello ' + person.name) // ok
7
```

```
8  true + 'true'; // // ok
9  ({} + {}) // Operator '+' cannot by applied to types of '{}' and
       '{}'
```

Listing 5.4: Typescript - odhalené chyby

```
1  function plus(first : number, second : number) : number {
2    return first + second;
3  }
```

Listing 5.5: Flow - ukážka typových anotácií

5.3 babel-plugin-check-data-access

babel-plugin-check-data-access [7] je plugin do systému *Babel.js*, ktorý transpiluje ES6 kód s cieľom zjednodušiť *deep access* (bezpečný prístup k atribútom objektov objektov). Rovnako ako *babel-plugin-superstrict* prekladá prístup k prvkom objektov a polí cez index aj atribút a v prípade neexistujúceho prvku vyhodí výnimku.

Na rozdiel od *babel-plugin-superstrict* je schopný viacero postupných prístupov zlúčiť do jedného volania funkcie, ktorá overí existenciu celej prístupovej cesty. V tomto prípade superstrict preloží kód na tri volania funkcie *safeGetItem/safeGetAttr*, čo je jednoduchšie na implementovanie, ale zrejme pomalšie na spustenie kvôli réžii spojenej s volaním funkcie. Tejto téme sa viac venujem v kapitole Overhead.

```
1  obj.a.b.c
2  deepAcc.get(obj, ['a', 'b', 'c'])
```

Listing 5.6: babel-plugin-check-data-access

Okrem funkcie *get* implementuje plugin aj funkcie *exists* (overí, či prvok s danou cestou existuje), *set* (nastaví prvok na danej ceste na danú hodnotu a vytvorí cestu, ak je to potrebné), *replace* (podobne akofunkcia *set*, ale hodí výnimku, ak prvok ešte neexistuje), *del* (odstráni prvok), *toggle* (ak prvok existuje, odstráni ho, inak ho nastaví na danú hodnotu), *update* (zavolá funkciu s daným prvkom ako parametrom a nastaví prvok na jej výsledok), *push* (ak je prvok pole, pridá na jeho koniec novú hodnotu), *patch* (pridá do objektu nové prvky a nahradí hodnoty existujúcich.).

Funkcia *collect* umožňuje syntax známu z programovacieho jazyka *Ruby*: na prístup k prvku poľa pomocou hranatých zátvoriek je možné okrem nezáporného celého čísla možné použiť aj záporné čísla, ktoré vrátia prvok smerom od konca poľa (teda *array[-1]* vráti posledný prvok poľa), a funkciu, v ktorej prípade sa vrátia prvky poľa, pre ktoré vráti *true*, teda identické správanie ako funkcie *filter*.

Záver

Javascript je jediný programovací jazyk spustiteľný v rámci webových stránok, ak nerátame jazyky, ktoré sa doň prekladajú. Nanešťastie bol navrhnutý za 10 dní a niektoré jeho oblasti majú značne odlišnú a neintuitívnu sémantiku.

Cieľom tejto práce bolo vytvoriť nástroj na pridanie runtime validácií do Javascriptového kódu, ktoré majú zabrániť vzniku chýb, ktoré zostanú nepostrehnuté a prejavia sa až neskôr a na inom mieste v kóde. Zamerali sme sa na pristupovanie k prvkom objektov a polí, implicitné pretypovania a numerické chyby ako delenie nulou a pretečenia. Nie sme si vedomí iných projektov, ktoré ponúkajú všetky z týchto vlastnosti.

Výsledkom je plugin do transpilátora Babel, ktorý dané ciele spĺňa a sú jeho súčasťou aj testy pre všetky podporované funkcionality. Do budúcnosti sa dá vylepšiť zverejnením na repozitári *npm.js*, aby bol dostupnejší pre zvyšných programátorov. Tiež bude zaujímavé zistiť, nakoľko sa dá kód zrýchliť, aby mohol byť bez problémov použiteľný aj v produkcii.

Literatúra

[1] Ecmascript compatibility table. http://kangax.github.io/compat-table/esintl/. Accessed: 2016-05-16.

[2] Uglifyjs. http://lisperator.net/uglifyjs/. Accessed: 2016-05-16.

[3] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[4] Acorn. https://github.com/ternjs/acorn/, 2012. Accessed: 2016-03-07.

[5] Flow. https://flowtype.org/, 2014. Accessed: 2016-03-07.

[6] Typescript. http://www.typescriptlang.org/, 2014. Accessed: 2016-03-07.

[7] babel-plugin-check-data-access. https://github.com/fab1an/babel-plugin-check-data-access, 2015. Accessed: 2016-03-07.

[8] Babel plugin handbook. https://github.com/thejameskyle/babel-plugin-handbook, 2015. Accessed: 2015-12-07.

[9] Babel.js. https://babeljs.io/, 2015. Accessed: 2015-12-07.

[10] Jsx. https://facebook.github.io/jsx/, 2015. Accessed: 2016-03-07.

[11] Gary Bernhardt. Wat. https://www.destroyallsoftware.com/talks/wat, 2012. Accessed: 2016-05-16.

[12] Python Software Foundation. The python language reference¶. https://docs.python.org/3/reference/, 2016. Accessed: 2015-05-16.

[13] David Goodger. Pep 0 - index of python enhancement proposals. https://www.python.org/dev/peps/, 2000. Accessed: 2016-05-16.

[14] Ecma International. Ecmascript 1st edition. http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,\%201st\%20edition,\%20June\%201997.pdf, 1997. Accessed: 2015-12-07.

[15] Ecma International. Ecmascript 2015 language specification. `http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf`, 2015. Accessed: 2015-12-07.

[16] Tim Peters. Pep 20 - the zen of python. `https://www.python.org/dev/peps/pep-0020/`, 2014. Accessed: 2016-05-16.

[17] Charles Severance. Javascript: Designing a language in 10 days. *Computer*, 45(2):7–8, 2012. Accessed: 2016-05-16.