НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»
Факультет інформатики та обчислювальної техніки
Кафедра технічної кібернетики

«На правах рукопису»
УДК 004.4'6

«До захисту допущено»
Завідувач кафедри
_____ _Ткач М. М.__
(підпис) (ініціали, прізвище)
“___”_____ 2015 р.

Магістерська дисертація

зі спеціальності 8.05020102 «Комп’ютеризовані та робототехнічні_____
(код і назва спеціальності)
_____системи» _____

на тему:_____Дослідження статичного аналізу типів для мови з_____
_____динамічною типізацією JavaScript _____

Виконав: студент __VI__ курсу, групи ___ІК-31м___
(шифр групи)
_____Бабійчук Андрій Анатолійович _____ _____
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник _____ст. в. Сирота О.П._____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант __ОППЦБ__ _____доц. Праховнік Н. А.____ _____
(назва розділу) (науковий ступінь, вчене звання, , прізвище, ініціали) (підпис)

Рецензент __с.н.с., к.т.н., ФТІ НТУУ «КПІ» Родіонов А. М __ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2015 року

Пояснювальна записка
до магістерської дисертації

на тему: Дослідження статичного аналізу типів для мови з динамічною типізацією JavaScript

Київ – 2015 року

ЗМІСТ

ВСТУП

JavaScript – мова програмування для браузера, що була розроблена компанією *NetScape* в 1996 році. Ця мова була розроблена як мова сценаріїв для нескладних програм, для яких основною задачею є простота та компактність в програмуванні та зручність в роботі з елементами сторінки, які описуються в *DOM (Document Object model)*. Натомість мова є не придатною для опису складних і великих програм, так як її дуже важко було структурувати. А система типів взагалі виконує лише умовне розділення даних, так як ніякої перевірки на коректність програми не відбувається.

JavaScript з початку створення мав безліч недоліків. Чому ж проблема типізації в мові раптом стала такою важливою лише зараз коли пройшло 20 років після її створення, і чи проблема це взагалі так як з роками *JavaScript* ставав все більш популярним.

Однак за останні роки спостерігається значний розвиток он-лайн сервісів, все більше і більше програм з’являються в якості Інтернет ресурсу і або створюються версії існуючих програм для роботи через мережу. Це вимагає використання складного інтерфейсу та логіки на стороні клієнта, тобто браузера. До останні років досить популярним способом обходити недоліки в *JavaScript* були різні мови транслятори чи бібліотеки для існуючих мов, які перетворювали код в *JavaScript*. Це дозволило описувати непоганий функціонал на для клієнтської частини. При цьому можна використовувати одну мову для програмування як серверної так і клієнтської частини, зокрема існує багато засобів на *Java*, які дозволяють переводити *Java* код в *JavaScript*. Проте такий підхід накладає багато обмежень, як на програміста, так і на можливості кінцевого продукту. Зокрема, використовуючи мову транслятор, таку як *Java* складно

описувати великі структурні програми. А якщо це і вдається зробити, то ціною великих ресурсів.

Таким чином хочемо ми цього чи ні, але доводиться все більше і більше писати на *JavaScript*, якщо хочемо отримати гарний продукт. І, як наслідок, на даний момент *JavaScript* одна із найпопулярніших мов програмування і її використання з кожним роком зростає. Ще більшу популярність *Java*Sript має завдяки використанню «інтерфейсу однієї сторінки» *(single page application),* за допомогою таких бібліотек як *Emberjs, Angular* та ін. Це чудове, на мою думку рішення, дає незрівнянні переваги перед звичайним сайтом. Використовуючи цей підхід користувач лише раз завантажує весь код клієнтської частини сайту, а далі під час роботи з сервісом просто відбувається оновлення певних частин сторінки. Тобто на мові *JavaScript* тепер описується не просто візуальна частина сайту але й значна частина бізнес логіки. Крім того *JavaScript* також може використовуватись на стороні сервера.

Такі тенденції розвитку ставлять перед мовою *JavaScript* нові вимоги. А саме, більш структурований опис програм, більш строга система типів. За останні роки розробники зробили кроки в цьому напрямку і почали з’являтись системи, які дозволяються програмісту писати на *JavaScript* складні й масштабовані програмами, але все ж проблема далеко не вирішена.

1. ОГЛЯД ПРОБЛЕМИ

JavaScript (JS) – динамічна, об'єктно-орієнтована мова програмування. Реалізація стандарту *ECMAScript* [1]. Найчастіше використовується в браузерах, для виконання різних функцій, від маніпулювання елементами сторінки до обміну даними з сервером. Мова *JavaScript* також може використовується на стороні сервера проте така практика значно менш поширена, адже є інші мови такі як C# та *Java,* які краще підходять для такої роботи.

JavaScript класифікують як прототипну (підмножина об'єктно-орієнтованої) мову програмування для сценаріїв з динамічною типізацією. Окрім прототипної, *JavaScript* також частково підтримує інші парадигми програмування (імперативну та частково функціональну) і деякі відповідні архітектурні властивості, зокрема: динамічна та слабка типізація, автоматичне керування пам'яттю, прототипне наслідування, функції як об'єкти першого класу [1].

Розглянемо основні положення, переваги та недоліки системи типізації мови *JavaScript.*

У мові *JavaScript* будь яка змінна може мати безліч типів протягом роботи програми. При ініціалізації змінної в неї немає типу або точніше тип *undefinded,* невизначене значення. При присвоєнні змінній певного значення її тип змінюється, якщо протягом роботи програми ці значення різного типу, то і тип змінної може змінюватись. Тобто сам по собі тип не обмежує значення які може мати змінна і відповідно не служить засобом безпеки мови. Такий підхід можна класифікувати як слабка типізація. Інший момент на який варто звернути увагу — це виконання операцій з різними типами. В таких випадках мова автоматично виконує неявне приведення типів.

Такі особливості системи типів мають наступні переваги.

1. Спрощується написання нескладних програм, наприклад, сценаріїв чи для додавання невеликої динаміки до сторінки.

2. Код більш виразний. Тобто потрібно писати менше коду в порівнянні з більшістю мов для виконання одного і того ж функціоналу.

3. Спрощується робота з елементами сторінки *DOM (document Object model),* а для мови *JavaScript,* це основне призначення, чим, на мою думку, і зумовлений вибір даної системи типізації.

Однак, такий підхід має багатозначних недоліків.

1. Динамічна типізація не дозволяє без запуску програми виконати перевірку коду на помилки.

2. Неявне приведення типів, яке в *JavaScript* відбувається між будь якими типами часто дає неочікуваний чи неправильний результат.

3. В об'єктно-орієнтованих мовах не діє або діє з обмеженнями авто-доповнення: важко або неможливо зрозуміти, до якого типу належить змінна, і вивести набір її полів і методів.

4. Розвинена статична система типів відіграє значну роль в само-документуванні програми; динамічна типізація за визначенням не проявляє цієї властивості, що ускладнює розробку структурно складних програм.

5. Низька швидкість, пов'язана з динамічною перевіркою типу, і великі витрати пам'яті на змінні, які можуть зберігати «що завгодно».

При роботі з *JavaScript* ми стикаємось з серйозними проблемами безпеки, значну частину з яких можна було б виправити, за допомогою більш строгої статичної системи перевірки типів. А саме проблем описаних в пунктах 1 та 2 можна практично повністю уникнути. Також, це дозволить частково вирішити проблеми 3 та 4.

1.1. Типи даних в мові *JavaScript*

В мові *JavaScript* існує 5 примітивних типів даних, а саме *number, String, boolean, undefined, null;* також обєкти *(Object)* та функції *(Function),*

що є нащадками типу *Object*. Решта типів таких як масив *(Array)* дата *(Date)* наслідуються від об'єкта і при визначенні типу через оператор *typeof* повернуть «*Object*» (тип *null* також), що є неправильно.

Boolean (логічний тип). Може мати значення *true* або *false*.

Number (числовий тип). В мові *JavaScript* є лише один числовий тип *number* він замає 64 біти і може містити числа з плаваючою точкою. По суті цей тип є аналогом *double* в *Java*. В *JavaScript* немає окремого типу цілих чисел *(integer)*, що позбавляє різних помилок з переповненням максимального значення. Проте є й вагомі недоліки, по перше це марнування пам'яті (так як *integer* займає 16 чи 32 біти) зважаючи, що досить часто використовується саме *Integer*; по друге, не накладаються обмеження на значення змінних, що в деяких випадках може привести до некоректних результатів. До літералів типу *number* входять чисові значення такі як *10, -2, 100, 2.4*, значення з експонентою *1e2 (100)*, а також значення нескінченності та від'ємної нескінченності *Infinity* та *-Infinity*. І також літерал *NaN (Not a Number)* який означає, що це тип *number* з невизначеним значенням. *NaN*, зазвичай є результатом приведення типів.

String (строковий тип). Рядок може бути вкладений в одинарні або подвійні лапки (нічим не відрізняються). Він може містити нуль або більше символів. Символ *«\»* (зворотний слеш) це символ екранування. *JavaScript* не має символьного типу. Для представлення символу, можна використовувати рядок лише з одним елементом. До елементів строки можна звертатись як до елементів масиву, як показано нижче.

```
var s = "String";
s[0] –> "s";
```

Undefined (невизначений тип). Його будуть містити всі змінні які не об'явлені, або яким не присвоєно жодного значення. Його єдине значення *undefined*.

Null (порожнє значення). Окремий тип в *JavaScript* для опису порожніх значень. Його єдине значення *null*. Проте оператор визначення типу для значення *null* поверне *Object*.

Object: об’єкт, в *JavaScript* можна створити за допомогою літералу *«{key: value}»* описавши довільну структуру включно з функціями та об’єктами. Також за допомогою оператора *new* застосувавши його до функції конструктора. Для створення порожнього об’єкту потрібно виконати *new Object*().

Function: підтип типу об’єкт. функції в *JavaScript* подібно до примітивних змінних та об’єктів можна присвоювати іншим змінним, передавати як параметр у функцію та повертати як результат.

Array: масив, теж підтип об’єкта, але при визначенні типу за допомогою *typeof* поверне об’єкт, а не масив, для коректної перевірки на приналежність змінної до масиву потрібно писати власний метод. Масив може містити елементи різних типів. створюється наступним чином: *[«0»,1, {}, true]*.

1.1.1. Приведення типів

Умовно можна виділити три способи приведення типів.

1. Приведення за допомогою конструкторів типів *Boolean() Number() String() Array() Object() Function()* а також стандартний метод *.toString()*. Це найкращий спосіб приведення, який можна використовувати в системах с сильною типізацією. Для перших трьох типів це справді корисний інструмент, так як ми можемо контролювати приведення типів, а отже отримувати правильний результат. Єдине, на що варто звернути увагу це приведення до *boolean*, а саме, значення, які конвертуються у *false*: *null*, *undefined*, «» (порожній рядок) та числові значення *0* та *NaN*. Всі решту значень приводяться до значення *true* включно з строками *«0»*, «*false*» та будь які об’єкти.

2. Приведення за допомогою спеціальних операторів: *«+»* чи *«-»* виконують приведення до типу *number*; *«!»* виконує приведення до типу *boolean* з протилежним значенням, тобто для правильного переведення слід використовувати *«!!»*. Це можна вважати неявним приведенням, але часто це використовується для компактності, щоб не використовувати конструктори.

3. Неявне приведення типів у виразах. Найбільш слабке місце системи типізації в *JavaScript.* приклади описані в табл. 1.1.

Таблиця 1.1. Неявне приведення типів в *Java*Script

Вираз	Результат	Тип
1 + «1»	«11»	*String*
true + «1»	«true1»	*String*
true + 1	2	number
1 + null	1	number
true + null	1	number
«1» + null	«1null»	*String*
{} + []	0	number
[] + {}	«[Object Object]»	*String*
[] + []	«»	*String*
{} + {}	NaN	number
1 + {}	«1[Object Object]»	*String*
1 + []	«1»	*String*
«1» + undefined	«1undefined»	*String*
«1» + null	«1null»	*String*

1.2. Недоліки та проблеми в *JavaScript*

Розглянемо основні групи проблем та помилок, які можуть виникнути при програмуванні на *JavaScript.*

1.2.1. Неявне приведення типів у виразах

Ці помилки є причиною, в першу чергу, використання слабкої типізації в мові. Коли в залежності від операції значення одного типу може автоматично приводитись до іншого. Тобто коли відбувається операції множення з строками, логічно виконати приведення типу до числа. Але крім цього досить багато ситуацій приведення не продумані і не мають сенсу, що є наслідком поганої системи правил приведення типів. Тепер детальніше розглянемо основні групи неявного приведення типів.

Приведення до типу String

Будь який вираз чи змінна при операції додаванні до значення типу *String* поверне тип *String* (табл. 1.2).

Таблиця 1.2. Приведення до типу *String*

Вираз	Результат	Тип
1 + «1»	«11»	*String*
true + «1»	«true1»	*String*
«1» + {}	«1[Object Object]»	*String*
«1» + []	«1»	*String*
«1» + undefined	«1undefined»	*String*
«1» + null	«1null»	*String*
«1» + [«1»]	«11»	*String*

Як бачимо практично завжди це дає не той результат, який ми очікуємо. Для типів *number*, array, *boolean* це приведення інколи може мати сенс, а от для інших типів навряд чи це буде корисним.

Приведення до типу Number

Результатом арифметичних та бітових операцій буде тип *number*. Строки, що мають числове значення приводяться до відповідного числа, в іншому випадку до *NaN*, теж має тип *Number*. Суперечливе приведення типу *null* та *undefined*, так як *null* приводиться до *0*, а *undefined* до *NaN*. Також некоректне приведення виконується для масиву. У цьому випадку *undefined* приводиться до *0* (див табл. 1.3).

Таблиця 1.3. Приведення до типу *String*

Вираз	Результат
Number([])	0
Number([[[[]]]])	0
Number([«0.2»])	0.2
Number([[[[11]]]])	11
Number([undefined])	0
Number([null])	0
Number([1, 2])	NaN

Приведення для логічних операцій

Може стати дуже корисним засобом при правильному використанні, проте краще обходитись більш надійними засобами. Результатом логічних операцій *AND (&&)* чи *OR(||)* буде тип даних, що використовується у виразі. У випадку з *AND* це буде тип першого операнда, якщо він при приведенні до *boolean* поверне *false*, в іншому випадку тип другого операнда. У випадку з *OR* це буде тип першого операнда, якщо він при приведенні до *boolean* поверне *true* в іншому випадку тип другого операнда. Основна проблема, це те, що операнди можуть мати різний тип.

1.2.2. Невизначені значення: *null, undefined, NaN*

В *JavaScript* робота з невизначеними значеннями базується в більшості випадках на сильній логіці Кліні (виключення робота з числами та *NaN* і *Undefined*). Це непоганий підхід для мови зі слабкою типізацією. Проте є один великий недолік – в *JavaScript* існує три типи невизначених даних і кожен з них належить до різного типу. Це не зрозуміло і заплутує. Крім того таким чином немає єдиної перевірки не відсутність значення, тобто для того, щоб бути певним, що значення не існує потрібно виконувати перевірку 3-х умов: *!(variable === null || variable === undefined || isNaN(variable)).* В деяких випадках можна використовувати перевірку *if (!variable),* але така перевірка також виключить і значення *0, false* та *«»*, які можуть бути допустимими.

1.2.3. Перевизначення змінних

JavaScript дозволяє визначати змінну кілька раз. Враховуючи систему типів, це логічна поведінка, але може стати причиною різних помилок. Особливо коли змінна заново об’являється з іншим типом.

1.2.4. Нестроге порівняння

Результатом операцій з порівняння завжди буде *boolean*, проте при використанні нестрогого порівняння може не завжди повертати очікувані значення (табл. 1.4). Крім того порівняння, навіть строге не діє на *NaN* (табл. 1.5).

На жаль, система типів не може заборонити таке приведення, тому, що це дозволено мовою, і часто програмісти використовують саме таке порівняння. Використовувати чи ні таке порівняння вирішувати програмісту і в рамках системи, яка буде розроблена цей недолік мови буде ігноруватись.

Таблиця 1.4. Нестроге порівняння

Вираз	Результат
'0' == 0	true
'' == 0	true
'' == '0'	false
'false' == false	true
'0' == false	true

Таблиця 1.5. Порівняння для *NaN*

Вираз	Результат
NaN == NaN	false
NaN === NaN	false

1.2.5. Використання невизначених змінних

В *JavaScript* дозволяється використання невизначених змінних, в такому випадку вона буде мати значення *undefined*, таке ж саме, як і об’явлена змінна, якій не присвоєно жодного значення. Такі операції не можна допускати в жодному разі. На щастя *JavaScript* пропонує рішення таких проблем за допомогою *«use strict»*. Код який буде іти після цього рядка не буде дозволяти використання невизначених змінних.

1.2.6. Робота з функціями

Виклик неіснуючої функції

Перш за все це перевірка на виклик функції яка не була визначена, ця помилка так само як і в попередньому випадку буде спричиняти помилку та зупинку виконання програми, коли код з помилкою буде викликаний. Такі помилки можна і необхідно виявляти статично не запускаючи програми.

Перевірка параметрів

Другий тип помилки — виклик функції з неправильними параметрами. *JavaScript* дозволяє викликати функцію з будь якими параметрами не виконуючи перевірку на кількість та типи. Це може стати причиною багатьох помилок.

1.2.7. Робота з об’єктами

Звернення до неіснуючого об’єкту

Ця помилка виникає, коли в коді відбувається звернення до поля об'єкта який не визначений або не є об'єктом. Це досить поширена помилка і, на відміну від інших, *JavaScript* в такому випадку видає повідомлення про помилку і зупиняє виконання програми. Але така перевірка може вимагати великих затрат часу, адже вона відбувається лише коли *JavaScript* намагається виконати код з помилкою, тобто для перевірки всього проекту необхідно, щоб виконалась кожна частина програми.

Прості об'єкти

Це більш загальне рішення проблеми звернення до неіснуючих об’єктів, ми її розглядаємо як можливість системи перевірки надавати об'єктам типи у вигляді інтерфейсів. Тобто, якщо ми об'явимо об’єкт: *var o = {name: “”}*, то йому надається тип *{name: String}*, це дозволяє виконувати узагальнену перевірку типів об'єкта, а не часткову.

Використання функцій конструкторів

При використанні функцій конструкторів, які виконують роль класів (в *ECMAScript* 5) дуже важливо, щоб система перевірки могла визначити які поля і методи має екземпляр, ця проблема особливо яскраво проявляється при використанні наслідування, в цьому випадку система перевірки має аналізувати класи батьківського типу аж до типу *Object*.

Робота з класами

Нове розширення мови яке буде реалізоване в *ECMAScript* 6 (реліз запланований на середину літа 2015 року), що дозволяє описувати класи подібно як і в класичному ООП. Так як цей функціонал вже скоро буде доступний, необхідно враховувати його при розробці системи перевірки типів.

1.2.8. Явне присвоєння типу

В мові *JavaScript* немає способу явно вказати якісь параметри змінної, зокрема її тип. Це могло б значно допомогти при написанні більш стійких та надійних програм.

Типові анотації

В деяких випадках неможливо точно визначити тип змінної чи параметра для таких випадків доцільно застосовувати анотації, в яких ми будемо описувати інформацію про тип. Такі анотації модна описувати як через розширення синтаксису мови, так і через анотування в коментарях.

Не типові анотації.

Спосіб опису додаткових параметрів змінних полів та методів, такі як приватність чи заборона запису. Виходить за рамки системи типів, проте допомагає визначити деякі специфічні помилки, які не можна перевірити за допомогою лише типів.

1.3. Огляд існуючих рішень

На даних момент існує ряд рішень проблеми типізації в *JavaScript*. До них входять системи: *Flow (Facebook)*, та *TypeScript (Microsoft)*. Крім того, одним із найпростіших вирішень проблеми — сучасні редактори коду. Вони місять інструменти для аналізу якості мови і дозволяють одразу ж виявляти синтаксичні помилки і, навіть виконувати базову перевірку типів. Якщо програміст буде користуватись для опису програми *jsDoc*,

перевірка буде досить надійною, проте вона не дозволяє в повній мірі покладатись на код. Детальніше розглядати їх не будемо, так як вони мають ту ж саму систему типів, що і *JavaScript,* однак можуть виконувати часткову статичну перевірку.

Вище згадані системи по суті є новими мовами програмування, на основі *JavaScript.* Вони розширюють синтаксис мови для явного визначення типів, який в загальних випадках однаковий. Також кожна система має свої особливості. Для призначення змінній якогось типу потрібно після її об'явлення поставити *«:»* та написати тип який ми хочемо їй надати. Також можна присвоювати тип параметрам функцій та тип результату, що повертає функція. Приклад, який буде працювати в кожній з цих систем зображено нижче.

```
Function f (a: mixed, b: number): void { ... }
var x: boolean;
class Bar {
    y: String;
}
```

1.3.1. Система статичної перевірки типів *Flow*

Flow відносно нескладна система типізації. Підтримує перевірку стандартних типів, конструкторів та деякі особливі типи, про які буде іти мова далі. Проте в неї є одна дуже важлива перевага. *Flow* може аналізувати чистий *JavaScript,* тоді як *TypeScript,* змістовного аналізу коду не виконає. Для того, щоб виконати перевірку коду за допомогою *Flow* потрібно дописати коментар над кодом, який ми хочемо проаналізувати.

```
/* @flow */
Function add(num1, num2) {
    return num1 + num2;
}
var x = add(3, "0");
```

Розглянемо типи даних які використовуються системою *Flow*.

1. Базові типи *JavaScript*:

– *number*: цей тип буде практично завжди буде результатом арифметичних та бітових операцій;
– *string*: строковий тип;
– *boolean*: логічний тип – буде результатом логічний операцій.

2. Імена функцій-конструкторів: імена функцій конструкторів та класів (нова можливість в *ECMAScript* 6), можна теж використовувати як анотації для позначення типів.

3. *Flow* додав деякі свої типи, які можна використовувати для анотацій:

– *mixed:* супертип для всіх типів, кожен тип може приводитись до типу *mixed*.
– *any:* динамічний тип, кожен тип може бути приведений до типу *any* і навпаки.
– *void:* тип для вираження типу *undefined*.

Деякі правила системи типів

Класи *Number*, *String*, та *Boolean* можна використовувати як анотації, як і інші класи, проте так як вони описують об’єкти а не самі значення вони будуть не сумісні з *number*, *String*, and *boolean*. Методи для даних класів доступні і для відповідних примітивних типів.

Типи *Object* та *Function* також можна використовувати для анотацій, вони будуть позначати будь-який об’єкт та будь яку функцію.

В *JavaScript* є багато неявних перетворень до типу *boolean*. *Flow* розуміє та допускає таку поведінку. Наприклад, значення різних типів можуть сприйматись як *boolean* в умові *if*, або як операнд для операцій *OR* чи *AND*.

Оператор додавання допустимий для типів *number* і *string*. У випадку якщо в операції присутні обидва типи відбувається конвертація

числа до типу *string* за допомогою методу *toString()* і відбувається конкатенація рядків. *Flow* розпізнає та допускає таку поведінку.

Об’єкти типу *Date* конвертуються до числового типу при використанні в арифметичних операціях. *Flow* розпізнає та допускає таку поведінку.

Тип *void* як правило використовується для анотування типу, що повертає функція, якщо вона нічого не має повертати і краще не використовувати цей тип для інших цілей! Єдине значення що має цей тип, це *undefined,* і *Flow* строго перевіряє використання цього типу.

Для анотування типу, що може містити будь яке значення. Варто бути обережними з типами *any* та *mixed,* так як це може значно знизити ефективність перевірки і *Flow* буде пропускати більшість колізій як нормальну роботу програми.

Варто користуватись типом *any* через його особливість, а саме – це динамічний тип. Грубо кажучи, якщо Ви абсолютно впевнені в коректності роботи програми, а *Flow* з цим не погоджується, можете використати анотацію *any*. У цьому випадку будуть ігноруватись всі помилки.

Використовувати такий підхід небезпечно і не рекомендується для надійної роботи, але це необхідно, так як *JavaScript* мова з динамічною типізацією, тому важно змоделювати роботу програми базуючись лише на статичній типізації і використання всіх можливостей мови інколи варто звертатись до такого прийому.

Flow – система статичного аналізу типів, що дозволяє виконувати базову перевірку на типи, та пропонує гарну обробку невизначених значень. Її можна використовувати як окрему мову, описуючи типи за допомогою анотацій, так і на вже написаному коді на *JavaScript,* що є її гарною стороною.

1.3.2. Система статичної перевірки типів *TypeScript*

Це більш складна система, яка вносить безліч нових можливостей для написання програм. Вона містить більшість структур та функцій, що використовуються в класичному ООП та інші засоби для написання безпечних програм.

TypeScript має наступні типи даних.

1. Базові типи:

- *Boolean*: базовий тип який може мати значення *true*/*false;*
- *Number*: числовий тип;
- *String*: строковий тип;
- *Array*: *TypeScript*, так само як і *JavaScript*, дозволяє працювати з масивами даних. Може бути записаний одним з двох варіантів, як наведено нижче.

```
var list:number[] = [1, 2, 3];
var list:Array<number> = [1, 2, 3];
```

2. Нові типи.

Enum: Корисний додаток до стандартних типів в *JavaScript.* Як і у деяких інших мовах, як *C#*, *Enum* це спосіб давати більш інформативні імена числовим значенням.

```
enum Color {Red, Green, Blue};
var c: Color = Color.Green;
```

За замовчуванням *Enum* починає нумерацію своїх елементів з *0*. Але їх можна визначити довільним чином самому.

Any: Інколи потрібно описувати данні про тип який ми не можемо судити однозначно чи не бути впевненим в їх значеннях. В таких випадках можна використовувати динамічний тип *any*.

```
var notSure: any = 4;
notSure = "maybe a String instead";
```

```
notSure = false; // okay, definitely a boolean
```

Тип *any* добре використовувати для існуючого *JavaScript* коду, що дозволяє перевіряти чи не перевіряти типи під час аналізу коду.

Також добре використовувати тип *any* для масивів, що містять елементи різних типів, як приведено у прикладі нижче:

```
var list:any[] = [1, true, "free"];
list[1] = 100;
```

Void: протилежний до *a*ny тип, який вказує, що результат немає ніякого значення. Зазвичай це використовується для опису результату роботи функції, яка нічого не повертає.

3. Інтерфейси. Найкращий спосіб пояснити що це таке – навести простий приклад.

```
Function printLabel(labelledObj:{label: String})
{
    console.log(labelledObj.label);
}
var myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

Система перевіряє виклик функції *printLabel*. Вона має єдиний параметр, який має містити поле, що називається *label* типу *String*. Треба зауважити, що об’єкт має і інші поля, але система перевірки їх ігнорує, так як нас цікавить лише одне поле, яке має мати відповідну назві і тип.

Щоб спростити опис подібних ситуацій можна використовувати інтерфейси.

```
interface LabelledValue {
    label: String;
}

Function printLabel(labelledObj:LabelledValue) {
    console.log(labelledObj.label);
}
```

```
var myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

Інтерфейс також підтримує опис полів що є необов'язковими.

```
interface SquareConfig {
    color?: String;
    width?: number;
}
```

Це означає, що об'єкт який ми опишемо за допомогою інтерфейсу може мати дані поля, але якщо їх не буде це не викличе помилку.

```
Function createSquare(config:SquareConfig):
{color:String;area:number} {
var newSquare = {color: "white", area: 100};
if (config.color) {
    newSquare.color = config.color;
}
if (config.width) {
    newSquare.area = config.width * config.width;
}
    return newSquare;
}
```

4. Опис функцій. Ще одна можливість інтерфейсів – описувати функції.

```
interface SearchFunc {
    (source: String, subString: String): boolean;
}
var mySearch: SearchFunc;
mySearch = Function(source: String, subString:
String) {
    var result = source.search(subString);
    if (result == -1) {
        return false;
    } else {
        return true;
    }
}
```

Для інтерфейсів, що описують функції, співпадіння назв параметрів не обов’язкове.

5. Класичні інтерфейси. Основне застосування інтерфейсів, це описувати шаблони класів, тобто те ж саме, що і в класичному ООП в таких мовах як *Java* та *C#.*

```
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date);
}

class Clock implements ClockInterface {
    currentTime: Date;
    setTime(d: Date) {
        this.currentTime = d;
    }
    constructor(h: number, m: number) { }
}
```

6. Наслідування відбувається так само як і в класичному ООП.

```
interface Shape {
    color: String;
}
interface Square extends Shape {
    sideLength: number;
}
var square = <Square>{};
square.color = "blue";
square.sideLength = 10;
```

Інтерфейс також може наслідуватись одразу від кількох інтерфейсів.

```
interface Shape {
    color: String;
}

interface PenStroke {
    penWidth: number;
}
```

```
interface Square extends Shape, PenStroke {
    sideLength: number;
}
```

Фактично *TypeScript* – нова мова програмування, яка складається з розширеного синтаксису *JavaScript* та синтаксис для опису типів, добре продуманої системи типізації, а також розширення мови за допомогою класів та інтерфейсів. Таким чином *TypeScript* потужний засіб, що не тільки обходить проблеми, які присутні в мові *JavaScript,* а й надає вдосталь зручних можливостей для написання програм. Проте в нього є один вагомий недолік – він не буде працювати з вже написаним *JavaScript,* тобто не буде відбуватись ніякої перевірки. А так як є безліч потужних засобів та бібліотек написаних на чистому *JavaScript,* то *TypeScript* в об’єднанні з цими бібліотеками не зможе виконувати надійну перевірку. А для існуючих проектів він взагалі нічим не допоможе.

1.3.3. Порівняння існуючих систем

Розглянувши основні можливості систем статичної типізації для мови *JavaScript,* проведемо порівняльний аналіз.

Оцінювати їхні характеристики будемо на основі проблем типізації, що ми розглянули у підрозділі 1.2. Знак *«+»* будемо ставити якщо система в даному випадку буде сигналізувати про некоректну роботу; *«-»,* якщо система розглядає код як допустимий і не видає ніяких помилок; *«+/-»* у випадку, коли система у деяких випадках повідомляє про помилку. а у деяких пропускає код як коректний. Результати порівняння у табл. 1.6 та табл. 1.7.

Як можна судити з отриманих результатів *TypeScript* виконує досить надійну перевірку, по всім показникам, які ми розглядаємо в рамках роботи. *Flow* поступається за функціоналом і в деяких випадках може пропускати помилки. Однак *TypeScript* виконує перевірку,

використовуючи типові анотації, тобто використовуючи розширену мову, що компілюється потім в *JavaScript,* тоді як *Flow* покладається лише на свій власний аналіз. Таким чином нам слід розмежувати сфери застосування цих систем. Розділимо сфери застосування на 3 категорії: новий проект на *JavaScript* ; новий проект, з використанням стороннього коду; існуючий проект написаний на *JavaScript*.

Таблиця 1.6. Порівняння систем статичної перевірки

Система перевірки	*TypeScript* (з анотаціями)	*TypeScript* (без анотацій)	*Flow*
Приведення до *String*	+	-	-
Приведення до *Number*	+	+/-	+
Перевизначення змінних	+	+/-	-
Невизначені значення	+	+/-	+/-
Використання невизначених змінних	+	+	+
Звернення до неіснуючого об’єкту	+	+	+
Виклик невизначеної функції	+	+	+
Невірні параметри функції	+	+/-	+/-

Таблиця 1.7. Порівняння додаткових можливостей систем

Система перевірки	*TypeScript* (з анотаціями)	*TypeScript* (без анотацій)	*Flow*
Інтерфейси	+	-	+/-
Функції конструктори	+	-	+/-
Робота з класами	+	-	+/-
Типові анотації	+	-	+
Нетипові анотації	+	-	-

Перший пункт — новий проект, всі системи підходять і можуть використовуватись. Другий варіант припускає використання сторонніх бібліотек, які не підтримують *TypeScript* чи потребують значних затрат часу для того щоб використання *TypeScript* мало сенс. Для існуючих проектів *TypeScript* нажаль буде мало корисними, так як він в своїй системі перевірки покладається на типові анотації, які *JavaScript* на даний момент не підтримує. *Flow* ж може аналізувати код без анотацій, покладаючись на контекст виконання операції. Результати наведено у табл. 1.8.

Таблиця 1.8. Сфера застосування систем статичної перевірки

Сфери застосування	*TypeScript*	*Flow*
Новий проект на *JavaScript*	+	+
Новий проект та сторонні бібліотеки	+/-	+
Існуючий проект на *JavaScript*	-	+

1.4. Постановка задачі

З проведеного аналізу існуючих рішень можна зробити наступні висновки. Для написання нових проектів на мові *JavaScript* доцільно буде використовувати *TypeScript*, за допомогою якого можна здійснювати надійний аналіз коду на помилки. Він практично по всім аспектам виконує надійну перевірку, крім того має додаткові можливості для написання більш виразних та чітко структурованих програм. Однак для існуючих проектів чи проектів, що використовують значну частину стороннього коду (наприклад різні бібліотеки) він буде не надто корисним, адже без типових анотацій не зможе забезпечити надійну перевірку програм.

Flow краще працює без типових анотацій, та може виконати непоганий аналіз коду чим видається кращим варіантом для перевірки проектів, що вже частково чи повністю написані. Однак перевірка яку він здійснює не може забезпечити надійного результату. Так як *Flow*

покладається лише на власний аналіз, і в деяких випадках він пропускає код, що може містити помилки як коректний.

Отже, наша система буде розрахована здебільшого на існуючі проекти або на проекти в яких з певних причин не доречно використовувати *TypeScript*, так як *TypeScript* досить добре виконує задачу статичної перевірки типів. Крім того в системі буде враховуватись типові, та деякі нетипові анотації користувача для якіснішого аналізу. Для гарної інтеграції з існуючими проектами анотації будуть описуватись в коментарях, що дозволить без порушення структури програми дописувати анотації на існуючих проектах. Це дасть нашій системі перевагу над *Flow*, адже система при перевірці буде покладатись як на власний аналіз, так і на анотації програміста, що забезпечить надійний аналіз. Система перевірки буде називатись *StaticChecker*. *StaticChecker* має наступні складові: система типів, система обробки виключних ситуацій, система правил типізації, код *JavaScript* та синтаксис для опису анотацій в коментарях та система представлення коду у вигляді дерева.

Таким чином задачі, що будуть вирішуватись в даній роботі наступні: описати систему типів для системи перевірки, описати систему обробки виключних ситуацій, описати правила перевірки на основі певного мат-апарату, визначити синтаксис для опису анотацій. На основі результатів розробити систему. Практично підтвердити переваги розробленої системи.

Висновки до розділу

1. В розділі проаналізовано особливості мови, а також систему типів в *JavaScript*. Розглянути основні властивості системи приведення та перевірки типів. На результати проведеного аналізу будемо покладатись та враховувати особливості мови при проектуванні та розробці власної системи перевірки типів.

2. Приведено приклади некоректної роботи системи типів та неявного приведення типів в мові. Також описано основні проблемні архітектурні та типові рішення мови *JavaScript*, які є великим недоліком цієї мови. Приведено існуючі рішення, які в тій чи іншій мірі дозволяють писати програми, уникаючи цих помилок.

3. Виконано порівняння існуючих систем , а саме *Flow* та *TypeScript*, що виконують статичний аналіз типів для мови *JavaScript*. На основі порівняння можливостей цих систем визначено сферу застосування системи, *StaticChecker*. А саме, існуючі проекти написані на *JavaScript* та проекти де недоцільно чи по якимось причинам не використовується *TypeScript*. Враховуючи результати порівнянь існуючих систем та аналізу помилок мови JavaScript, виконано постановку задач.

2. ПРОЕКТУВАННЯ СИСТЕМИ ТИПІВ

Спершу опишемо основні принципи роботи системи *StaticChecker*. Основні принципи будуть служити загальним описом системи для ознайомлення з основними її можливостями це дасть змогу користувачам системи визначитись з доцільністю її використання та керуватись при виборі системи перевірки.

2.1. Принцип роботи системи

Будь які операції між значеннями чи змінними різних типів породжують попередження, в залежності від контексту воно може бути критичне чи попереджувальне.

Система аналізує весь код незалежно від наявних помилок (за виключенням синтаксичних). Тобто, якщо на початку програми виявлена помилка про несумісність типів – решта коду буде проаналізовано.

Кожен тип має ряд допустимих операторів з яким він працює, при використанні недозволених операторів генерується попереджувальне повідомлення, а результатом такої операції буде тип описаний в правилах.

Типи змінних не змінюються під час роботи програми, виключення динамічний тип *Any* та тип *Null*. Якщо змінній не присвоєно ніякого тип, тоді їй присвоюється тип *any*. При присвоєнні цій змінній якогось значення в ході роботи програми їй присвоюється тип цього значення.

Неявне приведення типів. Система розуміє неявне приведення типів у випадках з префіксами *«!»* та *«+»*. В інших випадках неявне приведення буде породжувати помилку.

Деякі помилки можуть ігноруватись системою на вимогу користувача, тобто за бажанням можна вимикати перевірку на деякі правила типізації, що не є критичними.

2.1.1. Правила перевірки типів

Опишемо загальні правила типізації на основі яких будемо описували конкретні правила.

Примітивні типи. Зберігається правило приведення до типу *String*, що є в *JavaScript*, але таке приведення буде генерувати помилку. Результатом решти операцій буде тип *Mixed*, так як важко передбачити який саме буде результат і, звісно, буде генеруватись помилка.

Типи *Object* та *Function*. Об’єкти чи функції не можуть бути використані в будь якій операції арифметичній чи бітовій, це призводить до помилки, результатом буде тип *Any*. За певних умов дозволяється використовувати Об’єкт у Логічних операціях *AND OR*. Це правило використовується зважаючи на специфіку роботи цих операторів в мові *JavaScript*. В цьому випадку система дозволяє такий вираз, якщо і перший і другий тип Об’єкт або один з операндів *Null*. Це ж саме стосується функцій. Виклик неіснуючої функції чи об’єкта призводить до помилки. Виклик функції з неправильною кількістю параметрів чи параметрів, що мають невідповідний тип призводить до попередження.

Null (*undefined*). Єдиний правильний варіант використання цього типу – це використання як результат, що поверне функція (*void*). Однак допускається також використання типу *Null* в логічних операція *AND* та *OR* тоді коли це має сенс, тобто тоді, коли ми не можемо бути впевнені в тому який тип буде мати змінна, і ці вирази використовуються для спрощення перевірки. В усіх інших випадках буде показано повідомлення про помилку (критичне чи попереджувальне), в залежності від умов. Та все ж тип, що буде результатом операції з типом *Null*, буде вираховуватись по певним правилам.

2.2. Типи даних

Система типів буде подібна до систем *Flow* та *TypeScript* і, так як розробляємо систему для вже написаної мови, а не нової, необхідно враховувати ряд обмежень при проектуванні систем.

1. Примітивні типи *JavaScript (Number, Boolean, String).* Представляють такі ж типи як і в мові *JavaScript.*

2. Невизначені значення (*Null*). Тип *Null* буде представляти значення *null* та *undefined*, так як вони несуть однакову інформацію і не має сенсу в їх розділенні.

3. Примітивні значення *(NotNull/Mixed).* Представляють всі значення, що відносяться до примітивних типів, тобто включають в себе *Number*, *Boolean, String*. Цей тип використовується тоді, коли змінна може мати значення різного типу і це точно не вплине на роботу програми. також цей тип може бути результатом операцій з примітивними типами, якщо вони різні.

4. Об’єкт *(Object).* Звичайні об’єкти в *JavaScript.* Однак цей тип буде автоматично розширюватись в залежності від контексту використання, тобто: якщо ми створюємо змінну: *var o = { name: «name»}*, то тип цієї змінної буде *{name: «String»}*, і коли далі в програмі буде відбуватись зміна цього типу, це буде вважатись помилкою.

5. Функція *(Function).* Теж саме, що і в *JavaScript.*

6. Масив *(Array).* Теж саме, що і в *JavaScript.* За виключенням того, що він має один тип, якщо потрібно мати масиви з різними типами можна використовувати тип *Mixed* чи *Any.*

7. Клас *(Class).* Нова можливість *ECMAScript* 6, за допомогою класів будуть створюватись нові типи об’єктів.

8. Будь який тип *(Any).* Цей тип використовується при неможливості визначити якийсь інший тип, зазвичай він буде результатом

помилки. Його теж можна буде явно вказати в анотаціях. Цей тип може динамічно переходити в інший, якщо він не вказаний явно.

2.3. Виключення

Основна задача системи повідомити програміста про можливі помилки в програмі. Досі ми розглядали формальний опис правил типізації, за допомогою цих правил, ми можемо вказати системі які операції з якими типами допустимі, таким чином, якщо програма написана правильно, система за допомогою правил зможе підтвердити це, проте, якщо в програмі наявні помилки то система дасть збій, без деталей про причину. За допомогою генерації виключень система зможе повідомити про причину та місце некоректного коду, а також зможе перевірити весь код не зупиняючись на помилках (за виключенням синтаксичних помилок, але їх можуть перевірити практично всі редактори коду).

На основі проблем, що описано в першому розділі сформуємо список можливих помилок на які буде реагувати наша система. Кожен тип помилки буде мати кодову назву, та повідомлення, яке в доступному для користувача вигляді буде пояснювати в чому проблема. Крім цього класифікуємо помилки за критерієм впливу на роботу програми. Буде три типи помилок: критичні помилки *(error),* будуть повідомляти про помилки, що можуть спричинити зупинку програми; важливі помилки *(warning),* повідомлятимуть про можливий неправильний результат роботи програми; та попереджувальні помилки *(notice)* будуть вказувати на незначні помилки, які імовірно будуть повертати небажаний результат. Крім інформації, що описана в таблиці до повідомлень буде додаватись текст, що описує рядок в якому виникли проблеми. Класифікація помилок наведена у табл. 2.1.

Таблиця. 2.1. Класифікація помилок.

Назва помилки	Тип	Повідомлення
UnknownFunction	*error*	*Function {FunctionName} is not defined.*
UnknownObject	*error*	*Object {ObjectName} is not defined.*
UnknownVariable	*error*	*Variable {variableName} is not defined.*
DuplicateVariable	*warning*	*Duplicate variable {variableName}.*
ObjectInExpression	*warning*	*Object can't be used in expression.*
NullInExpression	*warning*	*Null or undefined can't be used in expression.*
UnsupportedOperator	*warning*	*Type {TypeName} doesn't support operations with this operator.*
ParamsCountMissmatch	*warning*	*Function {FunctionName} expect {ExpNum} params, but {ActualNum} are given.*
ParamsTypeMismatch	*warning*	*Param {paramName} should have {ExpectedType}, not {ActualType}*
ConvertToMixed	*warning*	*{Operand1Type} and {Operand2Type} are used in one expression. This may be reason of unexpected result or data loss.*
TypeMismatch	*notice*	*{Type1} and {Type2} are used in one expression. May be a reason of collision.*
TypeMismatchObjectField	*notice*	*Property(Subproperty) {PropertyName} of Object {ObjectName} has type {PropertyType} trying to be used as {ExpressionType}*
ConvertToString	*notice*	*String and {OperandType} are used in one expression. This may be reason of unexpected result.*

Для генерації помилок ми будемо використовувати новий терм, *warning*, який буде являтись об'єктом помилки, тобто насправді замість *warning* будемо підставляти об'єкт конкретної помилки, відповідно до табл. 2.1.

Якщо при приведенні правил будуть виникати помилки система буде виконувати їх виводячи повідомлення про помилку в журнал.

Помилки, що будуть використовуватись в системі не будуть впливати на роботу системи, зупиняти чи змінювати її поведінку. Помилка в даному сенсі виступає лише в якості інформаційного повідомлення. Це зумовлено тим, що система виконує статичну перевірку, тобто програма не запускається, а отже і впливати на її роботу не може, лише повідомляти про помилки.

Таким чином правила, в яких будуть генеруватись помилки будуть виглядати наступним чином. Для прикладу візьмемо вираз *2 + «2»*.

$$x: Number, y: Sring \mid\!\!- x + y \longrightarrow ConvertToString \qquad (2.1)$$

Такий вираз можна розшифрувати так: *x* та *y* мають тип *String*, тоді вираз *x+y* буде повертати об'єкт помилки *ConvertToString*, що має тип *ConvertToString*. Для спрощення тип опускаємо, так як це очевидно.

2.3.1. Алгоритм обробки виключних ситуацій

Для того, щоб система коректно працювала потрібно певним чином обробити помилку, в залежності від типу, і в результаті повернути результуючий тип виразу. Для цього буде використовуватись наступна конструкція: *try t1 catch t2 final t3*, де *t1* початковий вираз, *t2* - вираз обробки помилки, а *t3* вираз завершення перевірки. Введемо додаткові функції для опису роботи обробки помилок терм *output* буде позначати функції виводу помилки користувачу; *next* перехід до наступного виразу; *typeof* — тип помилки *return* повернення результуючого типу функції. На рис. 2.1 приведено у вигляді виразу *case* обробку кожного типу помилки.

На основі такої нескладної схеми обробки буде працювати наша система. Таким чином при виникненні помилки буде генеруватись відповідне повідомлення, але в результаті виразу буде присвоєно деякий тип і система перевірки продовжить роботу.

```
try
    expression:T
catch exception:Exception
    case typeof exception == Error
        output exception
        expression: Any;
    case typeof exception == ConvertToString
        output exception
        expression: String
    case typeof exception == ConvertToMixed
        output exception
        expression: Mixed
    case typeof exception == UnsupportedOperator
        output exception
        expression: Mixed
    case typeof exception == TypeMismatch
        output exception
        expression: Any
    case typeof exception == paramWarning
        output exception
    break
finally
    next
```

2.4. Правила типізації на основі типізованого лямбда числення

Для створення системи перевірки типів нам необхідно мати певний математичний апарат такий, щоб ми могли записати за його допомогою правил типізації. Це дозволить нам коректно виконати реалізацію системи перевірки, яка буде використовувати спроектовану математичну модель, а також за його допомогою можна буде довести правильність та надійність даної системи.

Для цієї мети ми будемо використовувати типізоване лямбда числення *(typed lambda-calculus)*. Лямбда-числення полягає у тому, що всі обчислення та операції, що виконує мова програмування зводяться до елементарних операцій, визначення функції та її виконання з певними параметрами, які в свою чергу теж є функціями як і результат виконання функції, що також є функцією. Важливість цього обчислення полягає в

тому, що його можна одночасно розглядати і як просту мову програмування на якій можна описувати розрахунки і як математичний апарат за допомогою якого можна доводити строгі твердження.

Як сказано вище лямбда числення використовується для опису функції, та за допомогою нескладних перетворень його можна збагатити кількома методами. Перш за все для зручності добавляють спеціальний синтаксис для примітивних типів, в нашому випадку це *Number, String, Boolean, Null*. Також необхідно розширити числення для використання об'єктів. І, так як, основна задача нашої системи повідомляти користувача про некоректні чи помилкові місця програми необхідно мати систему обробки виключних ситуацій.

2.4.1. Основні поняття

Синтаксис лямбда-числення складається з трьох видів термів. Змінна *x* сама по собі являється термом; абстракція змінної *x* (чи функція з параметром *x*) в термі *t1, (λx.t1),* — також терм. Її можна описати як функція, що отримує на вхід змінну *x* та повертає на виході терм *t1*. І, нарешті, застосування терму *t1* до терму *t2* (записується *t1 t2*) — третій вид термів. Ці способи конструювання термів виражаються наступною граматикою: *x* – змінна, *λx. t* – абстракція, *t t* – застосування.

Для розуміння подальших правил розглянемо основні положення лямбда числення на основі примітивних прикладів.

Запис *λx.y* означає, що *x* аргумент функції, а *y* її тіло, якщо бути точнішим – вираз, що представляє значення, що повертає функція.

Для наочного представлення приведемо приклад коду, який описує даний вираз:

```
function(x) {
    return y;
}
```

За допомогою лямбда числення можна описати будь які інші конструкції, зокрема об’єкти умовні вирази, класи та ін.

Безпека лямбда числення.

Основна властивість нашої системи типів — безпека (або коректність коректністю *(soundness)*): правильно типізовані терми «ніколи не ламаються». Система працює некоректно, якщо терм опинився в «тупиковому стані», в якому терм не є остаточним значенням, але правила обчислення нічого не говорять про те, що з ним робити далі. Отже, ми хочемо бути впевнені в тому, що правильно типізовані терми ніколи не виявляються в тупиковому стані. Ми доводимо це в два кроки, які традиційно називають теоремами просування *(progress)* і збереження *(preservation)*.

Просування: правильно типізований терм не може бути тупиковим (або це значення, або він може виконати наступний крок у відповідності з правилами обчислення).

Збереження: якщо правильно типізований терм робить крок обчислення, що виходить терм також правильно типізовані.

Разом ці дві властивості гарантують, що правильно типізований терм ніколи не потрапить в глухий кут в процесі обчислення [2].

Приклад використання лямбда числення.

Для прикладу опишемо перевірку на типи *number* та *boolean*, для цього вводимо наступні терми (в даному випадку терми і вирази це одне і теж саме):

t ::= терми:

n – будь який числовий літерал (окрім *NaN*);

t + *t* – оператор додавання.

Символ *t* в правій частині опису граматики називається мета-змінною *(metavariable)*. Це змінна в тому сенсі, що *t* служить в якості замінника якогось конкретного терму, але це «мета»-змінна, оскільки вона

не є змінною об'єктної мови *(Object language)* – тобто, самої мови програмування, синтаксис якої ми описуємо [2].

Твердження «терм *t* має тип *T*» (або «*t* належить типу *T*», або «*t* є елементом *T*») означає, що *t* «очевидно» дає при обчисленні значення потрібного типу — при цьому під словом «очевидно» ми маємо на увазі, що перевірити це можна статично *(statically)*, не виконуючи само обчислення *t* [2].

Результатом обчислення терму буде або певний тип, або помилка. У випадку, коли терм повертає помилку система має певним чином обробити результат, щоб повернути певний тип. Для цього будуть описані спеціальні правила.

Відношення типізації для виразів, що записується у вигляді *«t : T»*, визначається набором правил виводу, що задають термам типи.

Приклад правила:

$$\frac{n_1 : Number, n_2 : Number}{n_1 + n_2 : Number} \qquad (2.2)$$

цей запис означає наступне: якщо в нас є змінна *n1* та *n2*, які мають тип *Number*, то вираз *n1 + n2* також буде мати числовий тип.

Для перевірки виразу на коректність нам необхідно розбити його на елементарні вирази, створивши, таким чином, дерево виводу типів. Та перевірити вираз починаючи від елементарних складових (листя дерева) аж до даного виразу (корінь дерева). Дерево вивода типів *(typing derivation)* представляє собою дерево, що складається з екземплярів правил типізації. Кожній парі *(t, T)* у відношенні типізації відповідає дерево вивода типів з результатом *t: T* [2].

Приклад дерева виводу типів.

$$\frac{1: Number + \frac{2: Number + \ 3: Number}{(2+3): Number}}{1 + (2+3): Number} \tag{2.3}$$

Розшифрувати цей запис можна так: 1) якщо 2 та 3 мають тип *Number*, то 2+3 теж *Number*; 2) якщо 1 та (2+3) мають тип *Number*, то і *1+ (2+3)* теж має тип *Number*. Це приклад правильно типізованого виразу.

Таким чином ми можемо перевірити на коректність типів будь який вираз з простими типами, розбивши його на прості елементи для яких ми можемо виконати перевірку на правила. Звісно доповнивши числення іншими типами та правилами.

2.4.2. Типізоване лямбда-числення для опису складних типів

Функції

Так як в *JavaScript* типи визначаються динамічно, то ми не можемо знати який аргумент чекати, та який тип поверне функція. Якщо в коментарях описано тип аргументів будемо покладатись на нього, в іншому випадку буде відбуватись аналіз функції та перевірятись які операції будуть виконуватись над параметром. Наприклад якщо відбувається множення аргументу на число значить на вхід функція буде очікувати число. Якщо це визначити неможливо, значить тип *any*.

Для опису функцій ми будемо використовувати звичайні лямбда-числення. В нашу систему опису правил введемо новий тип — функцію «—>«, яка буде записуватись наступним чином T_1 —> T_2, що буде означати, що функція на вхід отримує тип T_1, а на виході повертає результат, що має тип T_2. Або у форматі лямбда обчислень:

$$\lambda x: T_1 . t_2: T_2 \tag{2.4}$$

Де *x* аргумент функції, T_1 – тип аргументу і T_2 результат повернення функції. Правила опису функцій право асоціативні, тобто T_1 —> T_2 —> T_3, теж саме, що і T_1 —> (T_2 —> T_3).

У повному вигляді це правило можна записати як:

$$\frac{x: T_1, t_2: T_2}{\lambda x. t_2: T_1 \to T_2} \tag{2.5}$$

Розшифрувати можна так: якщо *x* має тип T_1 *t2* має тип T_2, то функція (2.4) має тип T_1 —> T_2.

Контекст

Оскільки терми можуть мати вкладені лямбда абстракції, в загальному випадку нам доведеться говорити про кілька подібних припущень. Таким чином відношення типізації з двомісного стає трьохмісним *Г* |— *t : T*, де *Г* набір припущень про типи вільних змінних в *t*, тобто поточний контекст виконання.

Формально, контекст типізації (typing context) (чи середовище типізації, *typing environment*) *Г* представляє собою послідовність змінних і їх типів, а оператор «кома» розширяє *Г*, добавляючи до нього справа нове зв'язування. Порожній контекст інколи зображується символом *ø*, проте частіше ми просто опускаємо його і пишемо |— *t : T*, що означає: «Замкнутий терм t має тип T, виходячи з порожньої множини припущень». Тоді правило (2.2) можна записати так:

$$\frac{\Gamma, n_1: Number, n_2: Number}{\Gamma| - \ n_1 + n_2: Number} \tag{2.6}$$

Масиви

В *JavaScript* масив є набором елементів різних типів. Доступ до яких можна здійснювати за порядковим номером елемента починаючи з *0*. У нашій системі типів тип масив може містити один тип, якщо потрібно

використати масив, що містить різні типи можна використати батьківський тип, тобто для того, щоб визначити масив, що може містити типи *String, Boolean, Number* можна використати тип *Mixed*. Таким чином для позначення масиву достатньо використати один тип. У нашій системі ми будемо позначати його *{T}*. Так як масиви в *JavaScript* не мають сталої довжини ніякої прив'язки до розміру немає. Для позначення значення масиву одного елементу буде не достатньо. Тому в такому випадку будемо використовувати запис $\{t_1,t_2...\}$ або в загальному вигляді $\{t_i, i \in 1..n\}$.

Об'єкти

Дуже важливим типом в мові *JavaScript* є об'єкт. В об'єкті, на відміну від масиву кожне поле може мати різний тип, крім цього, поле має назву. Тому об'єкти будемо записувати у вигляді пари ключ-значення: *{key: «value»},* тип об'єкта – ключ: тип *{key: T}*. Труднощі може бути в тому, що однорідно описати структуру з довільним розміром та вмістом, включно з вкладеними об'єктами. Тому що треба буде шукати компроміс між читабельністю та точністю. Ми будемо використовувати наступне позначення. $\{t_i, i \in 1..n\}$.для об'єкта із n елементів від t_1 до t_n, і $\{T_i, i \in 1..n\}$ для його типу. Відмітимо, що при цьому n може дорівнювати нулю; в такому випадку, діапазон $1..n$ порожній, а $\{t_i, i \in 1..n\}$ рівне *{}*, пустому об'єкту. Вимога до об'єктів — кожне поле має мати унікальну назву. Тобто об'єкт *{x: 1, y: «2»}* буде мати тип *{x: Number, y: String}*.

Порядок полів не має значення, тому *{a: 1, b: «2»}* буде те ж саме, що і *{b: «2», a:1}* і, відповідно типи *{a: Number, b: String}* та *{b: String, a: Number}* будуть вважатись однаковими.

З об'єктами ми не можемо виконувати ніяких арифметичних операцій, лише операції *AND* та *OR*, і, крім цього потрібно виконувати перевірку чи правильно типи змінюють свої поля і стан. Ми будемо розглядати два випадки роботи з об'єктами. Перший це роботи з звичайними об'єктами без використання обмежень. В цьому випадку

дозволяється додавання та видалення полів об’єкта, зміна типу поля буде генерувати попереджувальне повідомлення. Другий тип це аналог інтерфейсів в *TypeScript*. Тобто ми будемо описувати структуру об’єкта всі його поля, їх типи, а також необов’язкові поля. В цьому випадку буде відбуватись строга перевірка і невідповідність шаблону буде вважатись помилкою. Такі структури можна об’єднувати в класи, тобто надавати їм імена, для зручності опису.

2.4.3. Типізоване лямбда-числення для опису підтипів

Система типів має однорідну структуру, тобто всі типи рівносильні, однак, для опису класів та підтипів необхідно використовувати ієрархічну структуру. Для цього нам потрібно ввести в систему підтипи.

Відношення підтипів формально описується через набір правил виведення, за допомогою яких можна отримати твердження вигляду $S <: T$, що читається як «S – підтип T» (або «T – над-тип S»). Ми розглянемо окремо кожен різновид типів (типи функцій, типи об’єктів та ін.), та сформуємо правила для кожного типу, у відповідності з якими можна буде використовувати один тип тоді, коли очікується інший.

Перш ніж приступити до правил, ми зробимо два загальних припущення: по-перше, відношення підтипів має бути рефлексивним: $S <: S$, а по друге воно має бути транзитивним: $S <: U \; U <: T \; S <: T$.

Ці правила прямо випливають з нашого інтуїтивного уявлення про безпеки підстановки.

У випадку, коли ми працюємо з простими об’єктами, тобто не класами, ми хочемо вважати тип $S \in \{k_1:S_1 ... k_m:S_m\}$ підтипом $T \in \{l_1:T_1 ... l_n:T_n\}$, якщо T містить менше полів, ніж S. зокрема, безпечно «забувати» останні поля в типах записів. Тобто $\{x: T, y:T\}$: буде підтипом $\{x: T\}$. Ця інтуїтивна ідея відображена в так званому правилі підтипів в ширину *(width subtyping)*: $\{l_i:T_i, i \in 1..n<k\} <: \{l_i:T_i, i \in 1..n\}$.

Може здатися дивним, що «менший» тип містить більше полів. Але тут головне не втратити інформацію при приведенні типів, тобто тип який є підтипом обов'язково має містити всі поля батька, хоча може мати скільки завгодно власних полів.

Правило підтипів в ширину застосовується лише до типів об'єктів, для яких типи загальних полів збігаються. Можна також дозволити відрізнятись типам окремих полів, якщо типи кожного відповідного поля є підтипом батьківського типу. Тобто, якщо *S* є підтипом T, то *{x: S, y: S}* є підтипом *{x: T, y: T}*. Ця інтуїтивна концепція виражається правилом підтипування в глибину *(depth subtyping)*: для кожного i, $S_i <: T_i$ $\{l_i: S_i\ i \in 1..n\} <: \{l_i:T_i, i \in 1..n\}$.

Тепер ми можемо об'єднати ці правила, отримавши більш гнучку систему не порушивши при цьому типову безпеку. Довівши при цьому, що тип *{x: {a: Number, b: String}, y: {c: Number}}* є підтипом *{x : {a: Number} y: {}}* за правилом підтипів в ширину: {a: *Number*, b: *String*} <: {a: *Number*}

{c: Number} <: {}, за правилом підтипів в глибину, так як відповідні об'єкта зліва мають типи, що є підтимами відповідних полів об'єкта справа то *{x: {a: Number, b: String}, y: {c: Number}}* є підтипом *{x: {a: Number} y: {}}*.

Останнє правило підтипів об'єктів виводиться з спостереження про те, що порядок полів у запису не робить ніякого впливу на те, як її можна безпечно використовувати, оскільки єдина операція, яку можна виконувати із записом після її створення – проекція її полів нечутлива до порядку полів. Тому, якщо $\{k_j:S_j j \in 1..n\}$ є перестановкою $\{l_i:T_i i \in 1..n\}$, то $\{k_j:S_j j \in 1..n\} <: \{l_i:T_i i \in 1..n\}$.

2.5. Опис правил типізації

Тепер перейдемо до опису правил згідно вимог до системи. Перш за все опишемо деякі позначення:

T – будь який тип;

P – будь який примітивний тип (*P ∈ {Sting, Number, Boolean};*

Class – будь який клас визначений через функцію конструктор. *{l_i:T_ii ∈ 1..n};*

Γ – контекст;

Γ |— *t* означає, що вираз t буде мати контекст *Γ*.

Множина типів системи:

{Number, Boolean, String, Mixed(NotNull), Null, Object, Class, Array, Function, Any}.

T<:Object – підтипи об’єкта, зазвичай нові класи описані в програмі.

Будь який клас є підтипом *Object*. Так само як і *Array* та *Function.*

Так як більшість операторів будуть мати однаковий значення (в контексті системи типів), тобто будуть при однакових операндах давати один і той самий тип, для зручності їх можна об’єднати в такі групи:

{}* – будь які оператори з двома операндами;

{!} – будь які оператори з одним операндом;

{a} – арифметичні операції, крім додавання, так як його можливості будуть дещо відрізнятись в деяких випадках. Це скорочення описує наступну множину операндів *{-, /, *, %};*

{i} — інкремет та декремент: *{++, --};*

{b} — бітові операції: *{^, |, , &, >>, <<, ~}.*

2.5.1. Примітивні типи

В табл. 2.2 описані правила для примітивних типів, коли *x* та *y* мають однаковий тип.

Таблиця 2.2. Правила приведення типів для примітивних типів

Умова	Правило
x: Number, y:Number	*x + y : Number*
	x {a} y : Number
	x {b} y : Number
	x {i} : Number
	{i} x : Number
	x || y : Number
	x && y: Number
	!x : Boolean
x: String, y: String	*x + y : String*
	x {a} y : UnsupportedOperator
	x {b} y :UnsupportedOperator
	x {i} : UnsupportedOperator
	{i} x : UnsupportedOperator
	x || y : String
	x && y : String
	+x : Number
	!x: z: Boolean
x: Boolean, y:Boolean	*x + y : UnsupportedOperator*
	x {a} y : UnsupportedOperator
	x {b} y : UnsupportedOperator
	{i} : UnsupportedOperator
	{i} x : UnsupportedOperator
	x || y: Boolean
	x && y: Boolean
	+x: Number

Тепер розглянемо правила перевірки типів, коли змінні мають різні примітивні типи, як правило це буде призводити до помилки.

Нехай *Γ, x:String y:T*, при чому *T ≠ String*, тоді:

$$\Gamma \vdash x + y : ConvertToString \quad (2.7)$$

Нехай *Γ, x:T1, y:T2*, де *T1 і T2* будь який примітивний тип або тип *Mixed*, при чому *T1≠T2*, тоді:

$$\Gamma \vdash x \{*\} y : ConvertToMixed \quad (2.8)$$

У випадку коли обидва оператори мають тип *Mixed. Γ x: Mixed, y: Mixed*, тоді:

$$\Gamma \vdash x \{*\} y: Mixed \quad (2.9)$$

2.5.2. Складні типи

З об’єктами чи функціями не може виконуватись жодна арифметична чи бітова операції, єдині допустимі операції, це логічне *OR* чи *AND*. У табл. 2.3 наведені правила роботи з об’єктами та функціями.

Ці помилки важливі, але зустрічаються вони не так часто, адже немає сенсу у використанні таких конструкцій, набагато важливіші помилки для об’єктів з використанням полів, яких не існує, а для функцій неправильні параметри.

Таблиця 2.3. Правила приведення типів для об’єктів та функцій

Умова	Правило
x: Object y: Object	x || y: x:Object
	x && y: x: Object
	x {a} y: ObjectInExpression
	x {b} y: ObjectInExpression

Продовження таблиці 2.3

	{!} x: ObjectInExpression
	x {i}: ObjectInExpression
	x + y: ObjectInExpression
x: Function y: Function	x || y: x: Function
	x && y: x: Function
	x {a} y: ObjectInExpression
	x {b} y: ObjectInExpression
	{!} x: ObjectInExpression
	x {i}: ObjectInExpression
	x + y: ObjectInExpression
x: Function y: Object	x || y: x: TypeMismatch
	x && y: x: TypeMismatch
	x {a} y: ObjectInExpression
	x {b} y: ObjectInExpression
	{!} x: ObjectInExpression
	x {i}: ObjectInExpression
	x + y: ObjectInExpression

Невизначені об'єкти та функції.

Нехай у нас є об'єкт $O1:\{x_i[i:1..n]:T_i[i:1..n]\ \}$ де x_i i-те поле об'єкта, а T_i тип відповідного поля, при чому T — будь який тип. Нехай z — назва поля, при чому z не належать $\{xi\ i:1..n\}$, тоді:

$$O1.z \text{ –> } Null:Null \tag{2.10}$$

$$O1.z.x : UnknownObject \tag{2.11}$$

У випадку, коли відбувається присвоєння полю об'єкта типу відмінного від поточного виникає помилка. $T1,T2$: будь які типи, $T1 \neq T2$.

$$O1.x: T1 = y: T2 : TypeMismatchObjectField \tag{2.12}$$

Розглянемо приклад неправильного використання функції. Для цього повернемось до вигляду лямбда числення, так як функціональні вирази описувати так простіше:

T1 і T2 будь який тип і *T1≠T2* тоді:

$$\lambda x{:}T1.y\ z{:}T2 : ParamsTypeMismatch \quad (2.13)$$

$$\lambda x{:}T1\ .\lambda y{:}T1.y\ z{:}T1 : ParamsCountMissmatch \quad (2.14)$$

Якщо λx не визначена в контектсі виконання, тоді

$$\lambda x : UnknownFunction \quad (2.15)$$

Узагальнити це правило можна наступним чином. Нехай у нас є функція F_1 параметрами якої є множина елементів $\{x_{1i}[i:1..n]:T_{1i}[i:1..n]\ \}$, де x_{1i} – назва *i*-того параметра, а T_{1i} його тип, в коді ця функція викликається з параметрами $\{x_{2i}[i:1..m]:T_{2i}[i:1..m]\}$, де x_{2i} — назва *i*-того параметра, а T_{2i} його тип, тоді, якщо $n \neq m$, результатом буде *ParamsCountMissmatch*; якщо $n=m$ для кожного і якщо $T2 \neq T1$ буде викликатись *ParamsTypeMismatch*.

2.5.3. Порожні значення *Null*

Порожні значення не можуть використовуватись в операціях, так як такі вирази будуть некоректними, хоча і в *JavaScript* можуть повертати очікуваний результат. Виключення логічні оператори *AND* та *OR*. Правила приведені в табл. 2.4.

Таблиця 2.4. Правила приведення типів для невизначених значень

Умова	Правило
x:*Null* y:*Null*	*x {*} y: NullInExpression*
x:*Null* y:T	*Γ |— x {*} y: NullInExpression*
	Γ |— !x: true: Boolean

Продовження таблиці 2.4

	Γ |— +x: Number
	Γ |— x || y : T
	Γ |— y || x : T
	Γ |— x && y : Null
	Γ |— y && x : NullInExpression

2.5.4. Динамічна зміна типу та робота зі змінними

Динамічна зміна типу можлива при присвоєнні змінній, що має тип *Any*, значення, вираз чи змінну, що має інший тип. Нехай *x* має тип *T1*, а вираз y повертає тип *T2*, такий, що *T1≠T2* і *T2* не є підтипом *T1* (а отже *T1* не може мати тип *Any*), тоді маємо наступні правила:

$$x{:}T1, y{:}\ T2 \vdash x = y : TypeMismatch \tag{2.16}$$

У випадку, якщо *T1=T2* або *T2<:T1* програма переходить до наступного кроку виконання.

У випадку, коли x має тип *Any*, а *T2* будь який інший тип, тоді відбувається зміна типу з *Any* до *T2*. Така ж зміна типу відбувається коли x має тип *Null*.

$$x{:}Any, y{:}\ T2 \vdash x = y \longrightarrow x{:}\ T2 \tag{2.17}$$

$$x{:}Null, y{:}\ T2 \vdash x = y \longrightarrow x{:}\ T2 \tag{2.18}$$

Коли в контексті вже визначена змінна, що має ім’я *x*, тоді виникає помилка дублювання:

$$x{:}\ T \vdash var\ x{:}T1 \longrightarrow DuplicateVariable \tag{2.19}$$

Коли в контексті не визначена змінна, що має ім’я *x*, але відбувається, що x зустрічається в коді у будь якому контексті чи операції, без попереднього об’явлення:

$$\varnothing \vdash x : UnknownVariable \tag{2.20}$$

2.6. Анотації

Для анотування типів та деяких нетипових параметрів буде використовуватись *«jsDoc»*. Це стандартна система для опису коду в *JavaScript*, що використовується для створення документації. *JsDoc* використовується в коментарях програми, що дозволяє не порушувати синтаксис *JavaScript*. Це дасть нашій системі перевагу над іншими, адже це дозволить користувачу описувати типи на свій розсуд, а не покладатись на аналіз системи і, при цьому, такі анотації без проблем можна впроваджувати в готові проекти.

Ось кілька прикладів опису типів за допомогою *jsDoc*.

Базовий приклад зображено нижче:

```
/**
 * @type {Number}
 */
var a;
```

В цьому випадку система буде працювати зі змінною a лише як з числом і код в програмі, що буде перевизначити її тип спричинятиме помилку.

Опис функції з параметрами зображено нижче:

```
/**
 * @param {Number} a
 * @param {Number} b
 * @param {String} c
 * @returns {String}
 */
```

```
Function f(a, b, c) {
 var sum = a + b;
 return c + sum.toString();
}
```

Як бачимо, без типових анотації не можна було б судити про тип параметрів, і якщо передати *a* чи *b* як *String* то результат буде неправильним. Анотація для функцій дуже корисна, так як не часто можна точно визначити тип параметрів чи результату покладаючись лише на аналіз операцій.

Також підтримуються деякі нетипові анотації, зокрема: *@private, @readonly, @const, @abstract.*

Приклад:

```
Function Person(name) {
    /** @private **/
    this.name = name;
    this.getName() {
        return this.name;
    }
}
```

В такому випадку звернення напряму до *this.name* буде вважатись помилкою.

2.7. Абстрактне Синтаксичне дерево

Аналізувати код такий яким його написав програміст складно, адже потрібно виконувати складний парсинг, для підстановки виразів у правила. Для цього потрібно привести конкретний синтаксис програми до абстрактного, який являє собою більш просте представлення програми у вигляді дерева *(abstract syntax trees).* Представлення у вигляді дерева робить структуру очевидною, що робить його зручним для аналізу. Перетворення з конкретного в абстрактний синтаксис відбувається в два етапи. Спочатку лексичний аналізатор *(lexical analyzer)* переводить

послідовність символів, написаних програмістом, в послідовність лексем *(tokens)* — ідентифікаторів, ключових слів, коментарів, символів пунктуації, та ін. Лексичний аналізатор прибирає коментарі, обробляє пробіли, вирішує питання з заголовними і строковими буквами, а також розпізнає формати числових і символьних констант. Після цього граматичний аналізатор *(parser)* перетворює послідовність лексем в абстрактне синтаксичне дерево. При граматичному розборі угоди про пріоритет *(precedence)* і асоціативності *(associativity)* операторів допомагають зменшити кількість дужок у програмі, явно вказують структуру складових виразів. Наприклад, оператор * має пріоритет вище, ніж оператор +, так що аналізатор інтерпретує вираз без дужок *1 + 2 * 3* як зображено на рис. 2.1.

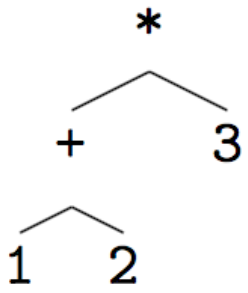


Рис. 2.1. Прилад абстрактного синтаксичного дерева.

2.8. Приклади

Розглянемо кілька прикладів перевірки виразів. Для підтвердження доцільності правил також буде приведено результат цих виразів.

Спершу розглянемо арифметичні вирази: *2 * 9 + 3 * + «2»*. Результат цього виразу: 24, коректний. Правила підтверджують це, перевірити можна за допомогою дерева виводу типів, яке буде мати наступний вигляд:

$$\frac{2: Number * 9: Number}{(2 * 9): Number} + \frac{3: Number * \frac{"2": String}{+"2": Number}}{(3 * +2): Number} \over (2 * 9) + (3 * +2): Number \quad (2.21)$$

Тепер приклад некоректної операції: *12 - «x»* Результат буде *«NaN»*, правило генерує помилку.

$$\frac{12: Number - "x": String}{12 - "x": ConvertToString} \quad (2.22)$$

Також помилка буде при виконанні операції з типом *Boolean: true - «x»*. Результат також «*NaN*», правило генерує помилку.

$$\frac{true: Bolean - "x": String}{true - "x": ConvertToString} \quad (2.23)$$

Конкатенація рядків – інший можливий випадок некоректної операції: *true + «x»*. Результат: *«truex»,* майже напевно, не те, що очікує програміст. правило генерує помилку.

$$\frac{true: Bolean + "x": String}{true + "x": ConvertToString} \quad (2.24)$$

Те ж саме при роботі конкатенації числа з рядком*: 1 + «1»*. Результат: *«11»,* більш очікуваний результат *2,* однак в деяких випадках необхідне саме приведення числа до строки. Тому помилка *ConvertToString* вважається попереджувальної і має бути виправлена на розсуд програміста.

$$\frac{1: Number + "1": String}{1 + "1": ConvertToString} \quad (2.25)$$

Висновки до розділу

1. На основі аналізу проблеми описано принципи роботи систем та основні вимоги до статичного аналізу типів. Також описано систему типів

StaticChecker, яка складається з стандартних типів мови *JavaScript* та деяких розширень, що дозволяють виконувати кращий аналіз коду.

2. На основі груп помилок, описаних в першому розділі, виділено три типи помилок, що описують рівні їх критичного впливу на результат програми. А також приведено алгоритм обробки помилок, що дозволить системі перевірки коректно опрацьовувати помилки в програмі.

3. Проведено дослідження математичного апарату, а саме, типізованого лямбда-числення, та розглянуто основні можливості такого числення. На основі проведеного дослідження створено систему правил, які будуть використовуватись при статичному аналізі коду.

4. На основі створених правил типізації приведено приклади роботи системи, що показують за допомогою дерева виводу типів як правила типізації застосовуються до різних виразів.

3. РОЗРОБКА СИСТЕМИ СТАТИЧНОЇ ПЕРЕВІРКИ

Система буде розроблятись на мові *JavaScript*. Так як система призначена для виконання перевірки мови *JavaScript* програмісту буде зручніше використовувати систему написану на тій самій мові, так як для цього не потрібно буде встановлювати різні додаткові додатки та служби це спростить процес перевірки. Для побудови дерева буде використовуватись бібліотека *ESPrima*. Для кращого розуміння роботи системи її можна розділити частини.

1. Система типів. Система, описана в попередніх розділах. складається з типів зв’язаних в ієрархічній структурі.

2. Правила. Набір правил що визначає всі допустимі комбінації типів та операторів між собою та визначає який тип буде результуючим. В тому числі які типи помилок будуть результатом.

3. Правила обробки помилок. Система обробки помилок, описана в попередніх розділах.

4. Синтаксичне дерево програми. Представляє програму в деревовидному вигляді, що дозволяє отримувати всю необхідну інформацію.

5. Область видимості. Змінні та функції програми. Також можна представити у вигляді дерева де вузлами будуть функції або об’єкти а листками змінні, що мають примітивні типи.

Пункти 1,2,3 — описують статичні частини програми, тобто не залежить від програми, що перевіряється та інших параметрів. Система типів може розширюватись програмістом, за необхідності. Для цього потрібно у певному форматі описати тип, функцію перевірки приналежності до цього типу та оператори, з якими він може працювати, а також батьківський тип. Якщо не задати батьківський тип. Система буде працювати некоректно.

Синтаксичне дерево (пункт 4) будується за допомогою спеціальної бібліотеки *ESPrima*. На вхід подається текст програми, а на виході отримується синтаксичне дерево і вся подальша робота системи буде вестись безпосередньо з отриманим синтаксичним деревом.

Область видимості (пункт 5), а точніше змінні та функції які описані у програмі будуються під час роботи системи паралельно з перевіркою на типи.

Коли у програмі іде об’явлення нової змінної система додає її в масив змінних поточної області видимості. При створенні об’єкта таким самим чином він додається до масиву змінних, об’єкт буде зберігатись у вигляді дерева; вузол буде коли поле буде об’єктом чи функцією, листком коли поле буде мати примітивний тип. Якщо йде об’явлення нової функції створюється нова область видимості і всі змінні, що описані в тілі функції будуть їй належати, та не будуть доступні ззовні (але зворотна видимість присутня, тобто функція бачить змінні, що описані за її межами).

Крім цього для наочності будуть застосовуватись сторонні бібліотеки. Вони не є частиною система, а служать лише для візуалізації результатів. Для візуалізації абстрактного синтаксичного дерева буде використовуватись *«JavaScript AST visualizer»,* для візуалізації області видимості програми буде використовуватись бібліотека *JSTree*.

3.1. Алгоритм роботи системи

Тепер розглянемо алгоритм роботи системи. Спершу загальний, потім розбивши на блоки розглянемо кожен блок детальніше.

1. Першим кроком в роботі системи буде побудова абстрактного синтаксичного дерева. Приведемо простий приклад:

```
Function sq(x) {
 return x*x;
}
var c = sq(10);
```

Даний код буде виглядати наступним чином (рис 3.1).

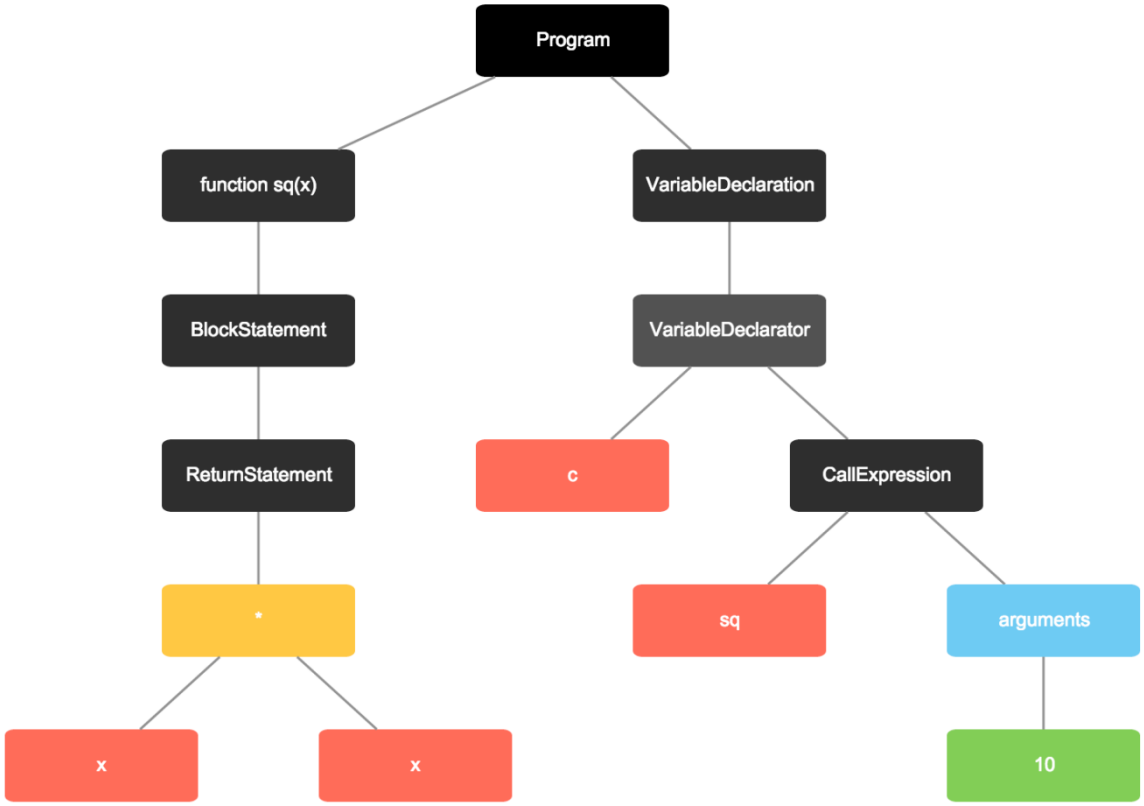


Рис. 3.1. Приклад АСД, що використовувався у *StaticChecker*

Тепер отриманий результат передаємо в систему перевірки. Система не працює з початковим кодом програми, лише з синтаксичним деревом.

2. Наступний крок — створення глобального області видимості програми в якому будуть зберігатись усі змінні та функції.

3. Далі програма працює в одному великому циклі в якому на кожному наступному кроці на перевірки перевірятись наступний вираз (рис. 3.2).

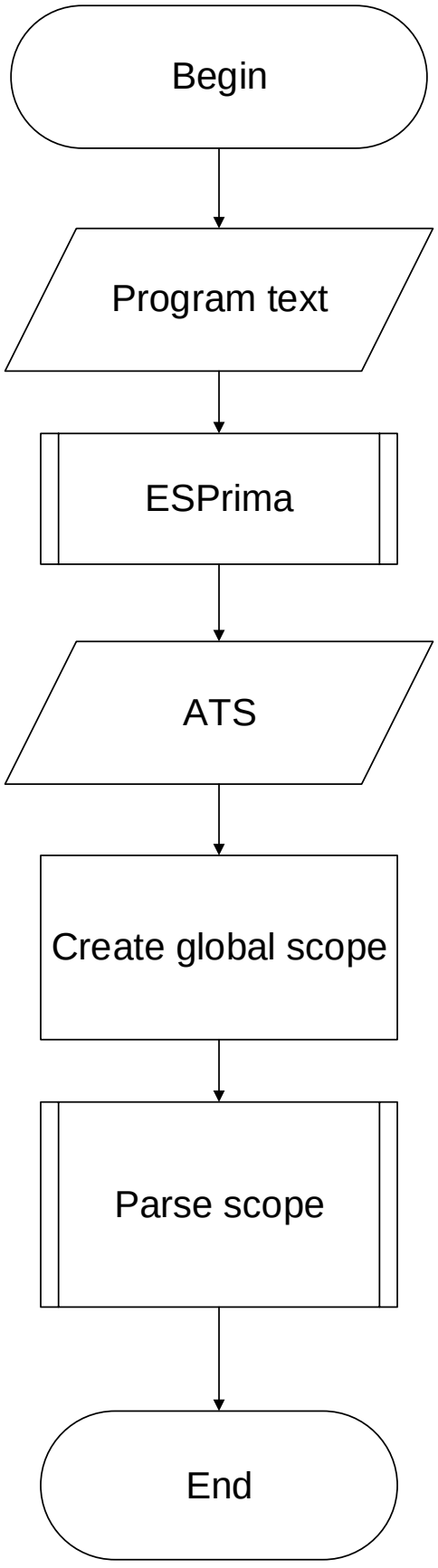


Рис. 3.2. Алгоритм роботи програми.

Таким чином виконується прохід по кожному виразу в програмі, якщо це ініціалізація змінної, виконується додавання її у скоуп, якщо це вираз виконується перевірка на основі правил описаних у попередньому розділі. Цей крок можна об’єднати як окремий сервіс *«Parse Scope»*, який зображено в додатку В. На цьому програма перевірки завершує свою роботу.

3.2. Приклади роботи програми

Розглянемо як працює система на реальних прикладах. Будемо розглядати примітивний код для перевірки на правила, які описано вище.

3.2.1. Критичні помилки *(error)*

Приклад роботи програми при виявленні необ’явлених змінних (функцій/об’єктів) рис. 3.3.

```
1  /** UnknownFunction **/
2
3  f1();
4
5  /** UnknownObject **/
6
7  var o = null;
8
9  var res = o.o;
10
11 /** UnknownVariable **/
12
13 var v2 = x;
14
```

Рис. 3.3. Критичні помилки

Вивід в консоль буде наступний:

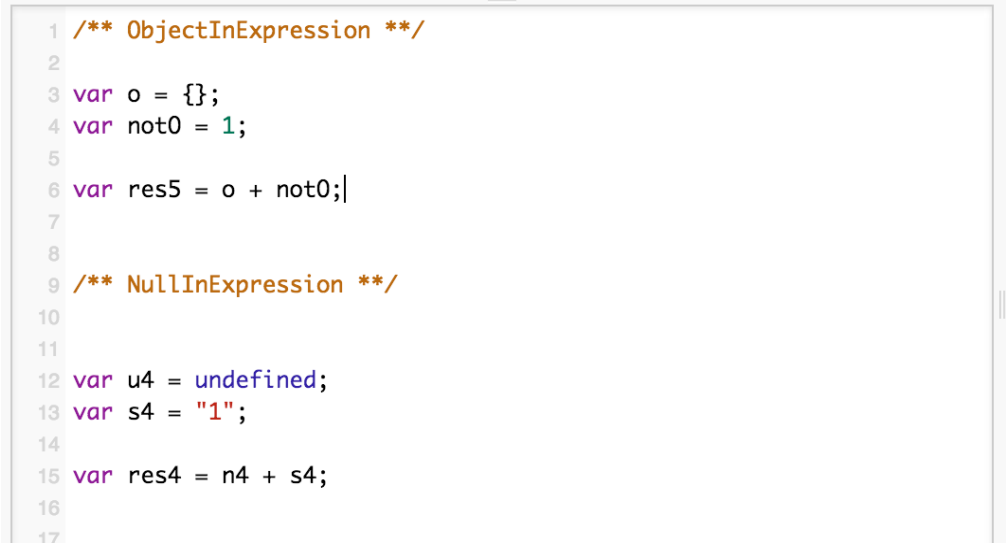
Error at line 3: Function f1 is not defined.

Error at line 7: Object o is not defined.

Error at line 13: Variable v2 is not defined.

3.2.2. Звичайні помилки *(warning)*

Наступна група помилок буде описувати некоректні вирази в яких об’єкти чи невизначені значення будуть використовуватись в операціях арифметичних чи бітових (рис.3.4).



```
/** ObjectInExpression **/

var o = {};
var not0 = 1;

var res5 = o + not0;


/** NullInExpression **/


var u4 = undefined;
var s4 = "1";

var res4 = n4 + s4;
```

Рис.3.4. Звичайні помилки

Вивід в консоль:

Warning at line 6: Object can’t be used in expression.

Warning at line 15: Null or undefined can’t be used in expression.

3.2.3. Помилки-попередження *(notice)*

Ці помилки виникають при роботі з примітивними типами (рис. 3.5).

Вивід в консоль:

Notice at line 6: String and Number are used in one expression. This may be reason of unexpected result.

Notice at line 14: String is used in arithmetic expression. This may be reason of unexpected result or data loss.

```
/** ConvertToString **/

var n1 = 1;
var s1 = "1";

var res1 = n1 + s1;

/** ConvertToMixed **/

var n2 = 1;
var s2 = "1";

var res2 = n2 * s2;

```

Рис. 3.5. Помилки-попередження

3.2.4. Типові анотації

У даному випадку перевірка покладається на типові анотації, описані в коментарях. Тоді не обов’язково присвоювати змінним початкові значення (рис. 3.6).

```
/**
 * @type {string}
 */
var a;

/**
 * @type {number}
 */
var b;

var c = a + b;
```

Рис. 3.6. Типові анотації

Вивід в консоль буде:

Notice at line 6: String and Number are used in one expression. This may be reason of unexpected result.

Описувати типи таким чином дуже корисно для функцій, адже автоматично визначити тип системі не завжди вдається. Наприклад у випадок описаному на рис 3.7. Система не може автоматично визначити

тип параметрів, так як оператор порівняння допустимий для всіх примітивних типів.

```
/**
 * @param {number} n1
 * @param {number} n2
 * @returns {number}
 */
function max(n1, n2) {
    return n1 > n2 ? n1 : n2;
}

max("1","2");
```

Рис. 3.7. Типові анотації для функції

У такому випадку вивід в консоль буде:

Warnign at line 10: Param n1 should be Number not String.

Warnign at line 10: Param n2 should be Number not String.

3.2.5. Нетипові анотації

Інколи для забезпечення безпеки не достатньо обходитись лише типами і потрібно додаткові параметри такі як приватність чи інші. Система типів не може автоматично визначати такі параметри системи, однак вміє розпізнавати їх з анотацій в коментарях.

Наступний приклад на рис 3.8 ілюструє використання анотацій для опису приватного поля.

```
function Car(name) {
    /** @private **/
    this.name = name;

    getName() {
        return this.name;
    }
}

var bmw = new Car("BMW");
bmw.name = "Ford";
```

Рис. 3.8. Нетипові анотації

Такий запис згенерує наступну помилку:

Warning at line 11: private field "name" trying to be used as public.

Також досить поширена проблема – константи рис. 3.9.

```
/** @readonly **/
const projectName = "StaticTypeChecker";

projectName = "DynamicTypes";
```

Рис. 3.9. Константа

Такий запис згенерує наступну помилку:

Warning at line 4: const "projectName" is trying to be changed.

Треба відмітити, що перевірку на *jsDoc* мають деякі сучасні редактори коду та деякі додаткові плагіни. Однак не завжди є можливість такої перевірки. Тому в системі підтримується перевірка найбільш поширених структур *jsDoc*.

3.3. Порівняння розробленої системи з існуючими

Для порівняння можливостей систем перевірки в цьому розділі будемо використовувати два невеликих проекти. Розмір близько 5 000 рядків коду. Один з них написаний на чистому *JavaScript*, а інший з

використанням *Ember.js*. Для першого будемо порівнювати розроблену систему з *Flow* та *TypeScript*, для другого проекту лише *Flow*.

3.3.1. Перевірка коду написаного на чистому *JavaScript*

Проект написаний на чистому *JavaScript*, з частковим використанням бібліотеки *jQuery*. При перевірці з використанням *TypeScript* використовувались типові анотації, також більшість методів та змінних описані за допомогою *jsDoc*. Тобто максимально строго описана структура. Перевірка дала такі результати. Помилки класифікуються відповідно до попередньої структури. В таблиці також подається кількість помилок які дійсно призводять чи можуть призводити до некоректного результату (табл. 3.1).

Таблиця 3.1. Порівняння існуючих систем з *StaticChecker* на основі коду на чистому *JavaScript*

Тип помилок	Помилки	*TypeScipt*			*Flow*			*StaticChecker*		
		В	З	П	В	З	П	В	З	П
Критичні помилки	19	19	0	0	19	0	0	19	0	0
Важливі Помилки	82	80	0	3	72	3	8	80	7	8
Попереджувальні помилки	17	15	1	3	3	0	12	13	1	6
Пропущено помилок	-	6			20			14		
Зайві помилки	-	1			3			13		

Як бачимо з результатів. *TypeScript* однозначно дає найкращий результат, однак для переводу проекту на *TypeScript* та опису його за допомогою анотації займає досить багато часу, а при швидкому темпі розробки цей перехід буде тривати або довго і поступово. *Flow* показав

гарні результати однак пропустив кілька помилок, однак він покладався лише на власну систему аналізу, тобто вимагає найменшої витрати часу на впровадження системи. Власна система перевірки додатково використовувала *JsDoc*, який також вимагає витрат часу. Однак *TypeScript* інколи надто строго перевіряє код, що в деяких випадках призводить до генерації зайвих повідомлень. Одна з причин це неоднозначність коду в деяких місцях, що за наявності анотації буде генерувати зайві помилки, а без них пропускатиме деякі колізії без помилок. *Flow* також інколи генерує зайві помилки, але він покладається лише на власний аналіз, тому очікувано, що інколи він буде неправильно оцінювати код, зважаючи що система вийшла лише пів року тому. Власна система дещо поступається *TypeScript* в перевірці, однак досить незначно. проте генерує найменше зайвих повідомлень, так як використовує комбінований аналіз власний та *JsDoc*.

3.3.2. Перевірка коду написаного з використанням *Ember.js*

Для цього проекту не проводилась перевірка на *TypeScript*, так як перепрограмування коду для нього надто затратне по часу, враховуючи досвід переходу чистого *JavaScript* коду на *TypeScript*. Крім цього частину проекту не можна коректно описати через типові анотації. Результат перевірки коду подано в табл. 3.2.

Як бачимо обидві системи добре перевіряють критичні помилки, однак для решти *Flow* частіше допускає помилки хоча і досить добре виконав перевірку. Проте і власна система частіше пропускає колізії при роботі з бібліотеками так як деякі вирази не можна анотувати чи перевірити належним чином. Для надійної перевірки в таких випадках необхідно використовувати юніт-тести, щоб запобігти виникненню помилок.

Таблиця 3.2. Порівняння існуючих систем з *StaticChecker* на основі коду на чистому *JavaScript*

Тип помилки	Помилки	Flow			StaticChecker		
		В	З	П	В	З	П
Критичні помилки	12	12	0	0	12	0	0
Важливі Помилки	67	61	4	10	65	1	3
Попереджувальні помилки	17	8	1	10	16	2	3
Пропущено помилок	-	20			6		
Зайві помилки	-	5			3		

Висновки до розділу

1. Визначено засоби розробки системи перевірки, а саме мова *JavaScript*, так як програмісту буде простіше користуватись такою системою, яка не вимагає встановлення додаткових компіляторів. Також розроблено алгоритм роботи програми *StaticChecker* на основі правил.

2. Приведено приклади роботи системи перевірки з різними групами помилок. А саме критичних, звичайних та попереджувальних помилок. Також приведено приклад роботи з нетиповими анотаціями. Що дозволяють продемонструвати можливості системи *StaticChecker*.

3. Проведено порівняльний аналіз розробленої системи з існуючими (*Flow* та *TypeScript*). Порівняння відбувалось на двох невеликих проектах, один з яких написано на чистому *JavaScript*, інший з використанням бібліотеки *Ember.js*. На основі порівняння доведено належний рівень роботи системи, що може виконувати статичну перевірку краще за існуючі системи.

4. ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАЗВИЧАЙНИХ СИТУАЦІЯХ

В даному розділі аналізується робоче місце інженера-програміста, що працює із ПК. В даній дипломній роботі виконується розробка апаратного комплексу, який буде використовуватися в ЕОМ. Потрібно проаналізувати можливі небезпечні та шкідливі виробничі фактори та навести рекомендації щодо мінімізації їх впливу на людину, розглянути питання безпеки робочого місця, освітлення, пожежної безпеки, мікроклімату, шуму та електробезпеки.

4.1. Характеристики приміщення

Схема приміщення та розташування персональних комп'ютерів зображено на рис. 4.1, а характеристики кімнати в табл. 4.1.

Таблиця 4.1. Характеристики кабінету

Довжина приміщення L, м	6
Ширина приміщення W, м	5
Висота приміщення H, м	3
Об'єм приміщення V,	90
Площа приміщення S,	30
Кількість працюючих, чол.	4
Об'єму приміщення на 1 працівника,	22,5
Площі приміщення на 1 працівника,	7,5
Висота першого вікна HB1, м	2
Ширина першого вікна W B1, м	1,5
Висота другого вікна HB2, м	2
Ширина другого вікна W B2, м	1,5
Висота дверей HД, м	2,4
Ширина дверей W Д, м	1,25

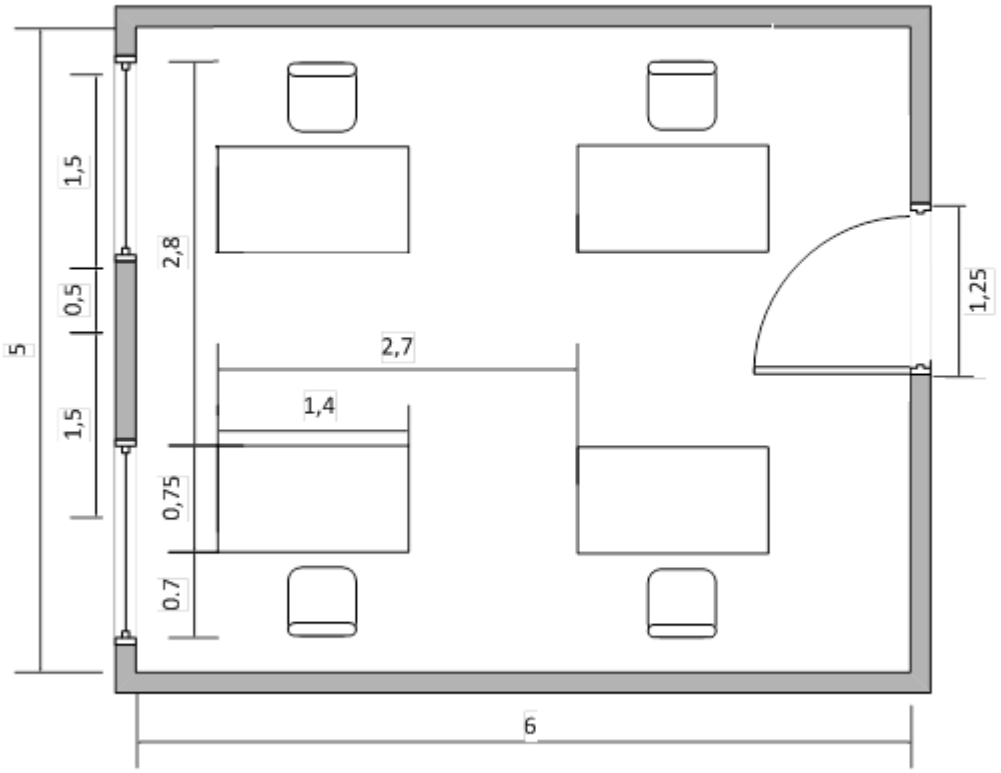


Рис. 4.1. Схема приміщення

Так, як приміщення обладнане комп’ютерами, то потрібно враховувати норми згідно [9], якого площа на одне робоче місце має становити не менше ніж 6.0 м^2, а об’єм не менше ніж 20.0 м^3. У даному приміщенні на кожну людину відводиться 7,5 м^2 площі та 22,5 м^3 об’єму, тобто приміщення відповідає вимогам. Відстань між бічними поверхнями екранів повинна бути не менше 1,2 метра, між тильними – 2,5 м.

4.2. Аналіз шкідливих виробничих факторів

4.2.1.Мікроклімат

У даному приміщенні використовується центральне водяне опалення низького тиску. Температуру також регулює канальна загально-обмінна припливна та витяжна вентиляція.

Роботи, що виконуються персоналом, відносяться до фізичних робіт категорії «легка 1а» за [10], так як вони виконуються сидячи та не потребують фізичного навантаження.

Так як характер роботи – постійний, то встановлюються оптимальні умови мікроклімату.

Згідно таблиці з [10], в холодний період року температура повітря повинна становити 22-24 ^{0}С, відносна вологість 60-40%, а швидкість руху повітря до 0,1 м/с. У теплий період року: температура 23-25 ^{0}С, відносна вологість 60-40%, швидкість руху повітря до 0,1 м/с.

4.2.2. Освітлення

В приміщенні виконується робота за монітором комп'ютера. Роботу можна віднести до розряду високої точності «1а». Відносна тривалість зорової роботи в напрямку зору на робочу поверхню не менше 70%.

В приміщенні використовується бокова система природного освітлення та загальна система штучного освітлення.

В приміщені 2 вікна, загальною площею 6 м^2. Вікна зорієнтовані на північ. Необхідна площа вікон згідно [15] для цього приміщення 4,2 м^2. На вікнах повинні бути встановлені пристрої для регулювання освітленості, такі як жалюзі, штори і т. п.

Стіни вкриті матовою фарбою, світло-блакитного кольору, що полегшує роботу зору під час використання ЕОМ.

4.2.3. Рівень шуму

Джерелом шуму в приміщенні виступають ЕОМ, а саме їхні компоненти: вентилятор охолодження, CD-дисковід.

У цьому випадку рівень шуму охолоджувачів на всіх машинах становить 35 дБА та дисководів – 46 дБА. Загальний рівень шуму складає 49,8 дБА. Допустимий еквівалентний рівень звуку згідно [12] наступні: для

програміста ЕОМ нормований звуковий тиск повинен бути не вище 50 дБА. Тобто наявний рівень шуму відповідає вимогам.

4.3. Інженерне рішення

Приміщення освітлюється за допомогою 10 дволампових світильників типу PL-L40W, які розміщені у два ряди і в кожному з яких знаходяться люмінесцентні лампи типу ЛБ потужністю 40 Вт. Згідно [11] рівень освітленості на робочому столі в зоні розташування документів має бути в межах 300-500 лк. План освітлення кімнати зображено на рис. 4.2.

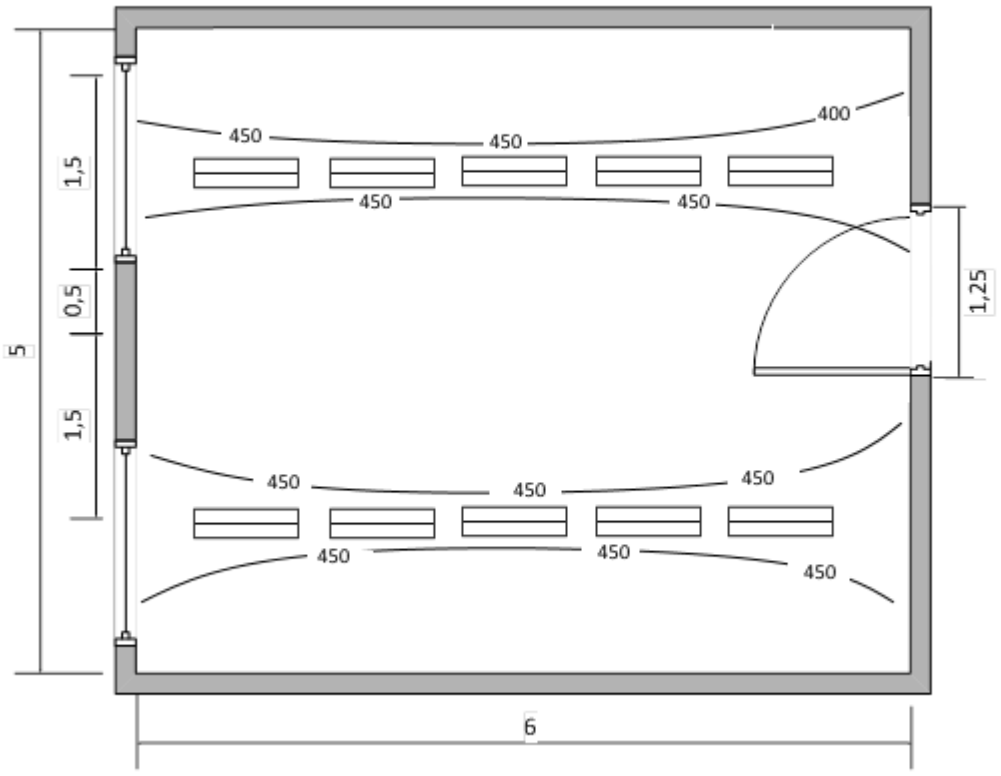


Рис. 4.2. Схема освітлення приміщення

Для розрахунку освітлення в приміщенні використовувався програмний продукт DIALux 4.9 з встановленим плагіном для систем

освітлення Philips. Штучне освітлення створює освітленість 478 лк, що задовольняє нормативне значення.

4.4. Електробезпека

Згідно [13] за ступенем небезпеки враження електрострумом приміщення відноситься до приміщення без підвищеної небезпеки. У приміщення проведено 3-х фазне електроживлення напругою 220 В, частотою 50 Гц та з максимальним струмом у 32 А.

Електроустановки підключені до групового щитка, що знаходиться в приміщенні. Використовуються кабелі та проводи з мідними жилами перерізом не менше 2,5 мм^2. Застосована відкрита електропроводка в пластмасових коробах із важкогорючих матеріалів з помірною димоутворювальною здатністю.

Електричну енергію споживає штучне освітлення та чотири комп'ютери.

Технічні захисні заходи електробезпеки: виконана робоча ізоляція струмопровідних частин (поліетилен, пластмаса), а також застосовані пластмасові коробки з важкогорючих матеріалів з помірною димоутворювальною здатністю. Електромагнітне блокування безпеки перемикачів виконана за допомогою стержневого магніту.

Електричний роз'єм (розетка, вилка) у разі підключення до електричної мережі гарантує з'єднання корпусу з заземленням, а в разі відключення гарантує від'єднання корпусу від заземлювача в останню чергу.

4.5. Пожежна безпека

В приміщенні знаходяться пожежонебезпечні матеріали: папір та дерево. Вибухонебезпечні матеріали у приміщенні відсутні. Виходячи з цього приміщення за [14] відноситься до категорії «В» пожежної безпеки.

Для забезпечення пожежної безпеки у приміщенні заборонено паління. Окрім того, в якості попереджувальних заходів періодично проводиться інструктаж з правил пожежної безпеки. Для вчасного попередження про пожежу у приміщенні встановлена порогова пожежна сигналізація з пасивними сенсорами диму. У разі спрацювання системи сигнал йде до чергового у корпусі.

Відповідно до [15] у приміщенні повинно бути встановлено по 2 вогнегасники на кожні 20 м^2. Відстань між місцями розташування вогнегасників не повинна перевищувати 15 м. В приміщенні знаходяться чотири вуглекислотні вогнегасника типу ВВ-2, які підходять для тушіння невеликих джерел займання, а також електроустаткування.

4.6. Безпека в надзвичайних ситуаціях

На відстані 22км від будівлі підприємства, в якій знаходиться розглянуте приміщення, розташовано АЕС, що є потенційно небезпечним промисловим об’єктом. У разі аварії на АЄС будівля може потрапити в зону радіаційного зараження (РЗ). Необхідно оцінити радіаційну обстановку, що може скластися в районі будівлі та вибрати найбільш ефективний спосіб захисту робітників і службовців об’єкту. Для цього необхідно визначити наступні дані.

1. Визначити, в яку зону РЗ потрапив об’єкт.

2. Визначити, які дози радіації отримають люди, якщо вони:

- залишаться працювати на робочому місці;
- укриються в захисних спорудах (ЗС);
- евакуюються із зони зараження.

3. Визначити допустиму тривалість праці на робочому місці при установленій дозі радіації.

4. Визначити, коли об’єкт зможе почати працювати повними робочими змінами.

Відомі наступні дані:

- відстань до АЕС: 22 км;
- швидкість вітру: 3 м/с;
- перший вимір рівня радіації: 10 Р/год;
- коефіцієнт ослаблення захисних споруд: 120;
- довжина маршруту в зоні РЗ: 5 км;
- середня швидкість автоколони: 60 км/год;
- установлена доза радіації: 3 Р.

4.6.1. Визначення зони ураження

Визначення зони РЗ, в яку потрапив об’єкт. Для визначення зони нам потрібні наступні дані:

R = 22 км — відстань від об’єкта до АЕС;

V_C = 3 м/с — середня швидкість вітру;

$P_П$ = 10 Р/год — рівень радіації на об’єкті на початку його зараження.

Етапи визначення.

1. Визначаємо час початку зараження території об’єкта:

$$t_{\text{П}} = \frac{R}{V_{\text{С}}} = \frac{22000}{3} = 7333 \text{ с} = 2{,}04 \text{ год.}$$

2. Розраховуємо рівень радіації на 1 годину після початку аварії:

$$P_1 = P_{\text{П}} * K_{t\text{П}} = 10 * 1{,}321 = 13{,}21 \; P/\text{год.}$$

Значення $K_{tП}$ для $t_П$ = 2,04 год. взято з таблиці

3. Об’єкт потрапив в зону РЗ - В.

4.6.2. Визначення доз радіації

Визначення доз радіації, які можуть отримати працівники. Для розв’язання цієї задачі нам потрібні наступні дані:

P_1=13,21 Р/год — рівень радіації на 1 годину після аварії;

$Д_у$= 3 Р — доза радіації, що установлюється на одну добу (робочу зміну) для виробничого персоналу;

$K_{осл\ буд}$ = 6 — коефіцієнт ослаблення будівель, де працюють люди;

$K_{осл\ зс}$ =120 — коефіцієнт ослаблення захисних споруд, де можуть укритися люди;

$K_{осл\ тт}$ = 2 — коефіцієнт ослаблення транспортного засобу, на якому евакуюються люди;

$t_{р\ max}$ = 8 год — максимальна тривалість однієї робочої зміни;

L = 5 км — довжина маршруту евакуації по зараженій території;

$V_{рух\ тр}$ = 60 км/год — середня швидкість руху автоколони по зараженій місцевості;

$V_{рух}$ = 5 км/год — середня швидкість руху пішою ходою по зараженій місцевості.

Етапи визначення:

1. Визначаємо дозу радіації, яку можуть отримати робітники за повну робочу зміну (8 год), якщо почнуть роботу з початку зараження об'єкта (це найгірший випадок з можливих):

$$Д_{буд} = \frac{Р_1 * (t_К^{0,6} - t_П^{0,6})}{0,6 * K_{осл\ буд}} = \frac{13,21 * (10,04^{0,6} - 2,04^{0,6})}{0,6 * 6} = 9,02\ Р.$$

Значення $t_К$ знаходимо за умов тривалості робочої зміни у 8 год:

$$t_К = t_П + t_{р\ max} = 2,04 + 8 = 10,04\ год.$$

2. Визначаємо дозу радіації, яку можуть отримати люди за час перебування в захисній споруді протягом 24 годин (доби):

$$Д_{буд} = \frac{Р_1 * (t_К^{0,6} - t_П^{0,6})}{0,6 * K_{осл\ буд}} = \frac{13,21 * (26,04^{0,6} - 2,04^{0,6})}{0,6 * 120} = 1,02\ Р.$$

Значення $t_К$ знаходимо за умов перебування в захисній споруді протягом 24 годин (доби):

$$t_K = t_П + t_{зс} = 2{,}04 + 24 = 26{,}04 \text{ год.}$$

3. Визначаємо дозу радіації, яку можуть отримати люди під час евакуації, якщо вони почнуть рух з моменту початку зараження об'єкту:

(a) під час маршу пішою колоною:

$$Д_{піш} = \frac{Р_{осp} * t_{св}}{K_{осл}} = \frac{Р_П * L}{2 * K_{осл} * V_{рух}} = \frac{10*5}{2*1*5} = 5 \text{ Р.}$$

(b) під час маршу автомобільною колоною:

$$Д_{авто} = \frac{Р_П * L}{2 * K_{осл} * V_{рух}} = \frac{10*5}{2*2*60} = 0{,}208 \text{ Р.}$$

Підведемо підсумки.

1. При установленій дозі радіації 3 Р працювати на робочих місцях повну зміну, а також евакуюватись із зони РЗ пішки є недопустимо.

2. Можливими способами захисту людей є укриття в захисних спорудах або евакуація на автомобілях.

4.6.3. Визначення допустимої тривалості перебування у зоні РЗ

Визначення допустимої тривалості перебування людей у зоні РЗ при установленій дозі радіації. Для цього необхідні наступні дані:

P_1=13,21 Р/год — рівень радіації на 1 годину після аварії;

$Д_у$ = 3 Р — доза радіації, що установлюється на одну добу (робочу зміну) для виробничого персоналу;

$K_{осл\ буд}$ - коефіцієнт ослаблення будівель, де працюють люди;

$t_П$ = 2,04 год - час початку роботи робочої зміни.

Етапи розв'язання:

1. Розрахуємо допоміжний параметр «а»:

$$\alpha = \frac{P_1}{Д_у * K_{осл\ буд}} = \frac{13{,}21}{3*6} = 0{,}734 \ \frac{1}{\text{год}}.$$

2. Визначаємо допустиму тривалість роботи у будинку цеху:

$$t_{\text{доп}} = \left(t_{\text{П}}^{0,6} + \frac{0,6}{\alpha}\right)^{\frac{1}{0,6}} - t_{\text{П}} = \left(2,04^{0,6} + \frac{0,6}{0,734}\right)^{\frac{1}{0,6}} - 2,04 = 2,1 \text{ год.}$$

Підведемо підсумки.

1. Допустима тривалість роботи в приміщенні в умовах РЗ становить 2,1 год, за цей час люди отримають установлену дозу радіації 3 Р.

2. У разі потреби роботи можна виконати скороченими змінами.

4.6.4. Визначення часу відновлення повноцінного режиму роботи

Необхідно визначити через який проміжок часу, підприємство зможе відновити роботу у нормальному режимі. Для цього необхідні наступні дані:

$\alpha = 0,734 \frac{1}{\text{год}}$ — параметр, що розрахований у попередній задачі;

$t_{p\,max} = 8$ год — максимальна тривалість однієї робочої зміни.

Етапи розв'язання:

1. Для $t_{p\,max} = 8$ год і $a = 0,734$ знаходимо $t_{П} = 77$ год.

В табл. 4.2 приведені підсумкові дані.

Таблиця 4.2. Підсумкова таблиця

Р1, Р/год	Доза радіації, Р				tдоп	Початок роботи у звичайному режимі, год
	Дбуд8	Дбуд24	Дпіш	Давто		
13,21	9,02	1,02	5	0,208	2,1	77

4.6.5. Загальна оцінка

1. Під час аварії на АЕС об'єкт потрапляє в зону В — небезпечного зараження.

2. Аналіз варіантів дій виробничого персоналу показує, що найбільш доцільним способом захисту людей є евакуація на автомобілях.

3. У разі потреби продовження безперервного виробничого процесу потрібно розрахувати режими роботи скороченими змінами.

4. Об’єкт може почати працювати повними змінами через 77 год після аварії на АЕС.

4.6.6. План дій персоналу при аварії

Відповідно до розглянутого вище, однією з можливих аварій є аварія на атомній електростанції. В цьому випадку необхідно дотримуватись наступного плану дій:

1. Отримавши повідомлення про очікування випадання радіоактивних опадів проінформувати працюючих про аварію та можливе радіоактивне забруднення місцевості, про необхідність герметизації приміщень.

2. Всі працівники, присутні у лабораторії, мають вжити негайних заходів для запобігання шкоди життю та здоров’ю.

3. Провести часткову санітарну обробку співробітників.

4. Організувати роботу постів радіохімічного спостереження; вимірювати рівень радіації на території підприємства за допомогою приладів і визначити режим захисту та поведінку співробітників.

5. Організувати у відділах і службах виготовлення марлевих пов’язок для захисту співробітників від радіоактивного пилу.

6. Виконати роботи по герметизації всіх приміщень.

7. Спеціальні служби приймають участь у евакуації людей, встановлюють забороняючі знаки тощо. За можливості, знаходячись у безпечній зоні, необхідно надавати першу допомогу постраждалим.

8. Організувати у відділах і службах виготовлення марлевих пов’язок для захисту співробітників від радіоактивного пилу.

9. Через встановлений час спеціальними службами проводиться обеззараження місцевості шляхом дезактивації.

10. Проводиться експертиза щодо безпечності продовження роботи на обеззараженій території. У випадку позитивного результату, допускається продовження робіт.

Висновки до розділу

1. У даному розділі проведено детальний аналіз умов праці у робочому приміщенні, в якому розміщено 4 робочих місць. Внаслідок проведеного аналізу санітарно-гігієнічних умов праці, умов електробезпеки і пожежної безпеки приміщення, де виконуються роботи з використанням ЕОМ, зроблено висновок про відповідність переважної більшості чинників нормативним вимогам.

Таким чином, умови роботи з відео термінальними пристроями відповідають майже всім нормам, не виконуються лише норми штучного освітлення, тобто необхідно вжити заходів по покращенню штучного освітлення приміщення, а також слід приділити увагу організації відпочинку та перерв працюючого персоналу. Об’єм приміщення, з розрахунку на одну людину відповідає нормативному значенню. Фактичні показники мікроклімату цілком відповідають допустимим значенням. Параметри природного освітлення відповідають нормі. Рівень електробезпеки та пожежної безпеки знаходиться у відповідності нормативним вимогам.

Загалом незначна кількість невідповідностей фактичних умов праці нормам дозволяє зробити висновок, що умови праці в приміщенні, що розглядалося, є задовільними. Рекомендації щодо показників, які не відповідають нормативам, наведені у наступному підрозділі.

2. Визначено, що характер робіт складності є допустимим рівнем напруженості і рекомендовано робити перерви по 10 хв після кожної години роботи.

3. Визначено, що дане приміщення за групою електронебезпечності відноситься до приміщень без підвищеної небезпеки ураження струмом. Зазначено, що для пожежної безпеки використовуються вогнегасники – ВВ-2. Детальний аналіз показав, що всі показники відповідають встановленим нормам згідно з чинним законодавством.

ЗАГАЛЬНІ ВИСНОВКИ

1. Даній магістерській дисертації проаналізовано та описано особливості мови, а також систему типів в *JavaScript*. Виявлено та приведено приклади некоректної роботи системи типів та неявного приведення типів в мові. В зв'язку збільшенням використання мови в складних проектах проблеми системи типів стають більш критичними.

2. Розглянуто існуючі системи, що виконують статичний аналіз типів для мови *JavaScript*, а саме: *Flow* та *TypeScript*. Виконано порівняння цих систем, що включало перевірку на знаходження виявлених помилок приведення типів. На основі порівняння можливостей цих систем визначено сферу застосування системи *StaticChecker*. А саме, існуючі проекти написані на *JavaScript* та проекти де недоцільно чи по якимось причинам не використовується *TypeScript*.

3. На основі виявлених проблем мови та аналізу існуючих рішень виконано постановку задач. Виділено наступні задачі: описати систему типів для системи перевірки, описати систему обробки виключних ситуацій, описати правила перевірки на основі певного мат-апарату, визначити синтаксис для опису анотацій. На основі результатів розробити систему. Практично підтвердити переваги розробленої системи.

4. На основі виявлених помилок, виділено три типи помилок: критичні, звичайні та попередження; що описують рівні їх критичного впливу на результат програми. Враховуючи таке розділення класифіковано помилки на які буде відбуватись перевірка системою. А також приведено алгоритм обробки помилок, що дозволить системі перевірки коректно опрацьовувати помилки в програмі.

5. Проведено дослідження математичного апарату, а саме, типізованого лямбда-числення, та розглянуто основні можливості такого числення. Описано систему типів для статичної перевірки, яка включає

існуючі типи *JavaScrip* та деякі доповнення для кращого аналізу. Також, На основі типізованого лямбда-числення описано правила перевірки типів які є основою статичного аналізатора.

6. Для розробки системи вибрано мову *JavaScript,* так як перевірятись буле саме ця мова, то програмісту буде зручніше використовувати систему, що не вимагатиме додаткового встановлення компіляторів. Для побудови синтаксичного дерева, важливої складової процесу аналізу, вибрано систему *ESPrima,* яка представляє код програми у зручному для перевірки вигляду. Представлено блок схему алгоритму роботи системи.

7. Проведено порівняння розробленої системи з існуючими. Порівняння відбувалось на двох невеликих проектах, один з яких написано на чистому *JavaScript,* інший з використанням бібліотеки *Ember.js*. На основі порівняння доведено належний рівень роботи системи, що може виконувати статичну перевірку краще за існуючі системи.

8. Проведено детальний аналіз умов праці у робочому приміщенні, в якому розміщено 4 робочих місць. Внаслідок проведеного аналізу санітарно-гігієнічних умов праці, умов електробезпеки і пожежної безпеки приміщення, де виконуються роботи з використанням ЕОМ, зроблено висновок про відповідність переважної більшості чинників нормативним вимогам. Визначено, що характер робіт складності є допустимим рівнем напруженості і рекомендовано робити перерви по 10 хв після кожної години роботи. Визначено, що дане приміщення за групою електронебезпечності відноситься до приміщень без підвищеної небезпеки ураження струмом. Детальний аналіз показав, що всі показники відповідають встановленим нормам згідно з чинним законодавством.

ПЕРЕЛІК ПОСИЛАНЬ

1. JavaScript [Електронний ресурс]: Режим доступу: https://uk.wikipedia.org/wiki/*JavaScript* Назва з екрану. - JavaScript Дата перегляду – 3.12.2014 р.

2. Бенджамин Пирс. Типы в языках програмирования. 14.10.2010

3. Абстрактне синтаксичне дерево [Електронний ресурс]: Режим доступу: http://uk.wikipedia.org/wiki/Абстрактне_синтаксичне_дерево Назва з екрану. - Абстрактне синтаксичне дерев. Дата перегляду – 3.02.2015 р.

4. The Lambda-calculus and the Javascript [Електронний ресурс]: Режим доступу: http://www.slideshare.net/normanrichards/the-lambda-calculus-and-the-javascript Назва з екрану. - The Lambda-calculus and the Javascript. Дата перегляду – 3.04.2015 р.

5. Strong and weak typing [Електронний ресурс]: Режим доступу: https://en.wikipedia.org/wiki/Strong_and_weak_typing. Назва з екрану. - Strong and weak typing. Дата перегляду – 3.12.2014 р.

6. Динамическая типизация [Електронний ресурс]: Режим доступу: https://ru.wikipedia.org/wiki/Динимическая_типизация. Назва з екрану. - Динамическая_типизация. Дата перегляду – 3.12.2014 р.

7. Система типов [Електронний ресурс]: Режим доступу: https://ru.wikipedia.org/wiki/Система_типов. Назва з екрану. - Абстрактне синтаксичне дерев. Дата перегляду – 3.12.2014 р.

8. *TypeScript* [Електронний ресурс]: Режим доступу: http://www.*TypeScript*lang.org/Handbook#basic-types. Назва з екрану. – *TypeScript* Basic Type. Дата перегляду – 3.12.2014 р.

9. ДСанПіН 3.3.2-007-98. Гігієнічні вимоги до організації роботи з візуальними дісплейними терміналами електронно-обчислювальних машин\МОЗ України-К.:1998.18с.

10. ДСН 3.3.6.042-99 Санітарні норми мікроклімату виробничих приміщень - Київ, 2000.

11. ДБН-В.2.5-28-2006 – Природне і штучне освітлення.

12. ДСН 3.3.6.037-99 Санітарні норми виробничого шуму, ультразвуку та інфразвуку.

13. Правила улаштування електроустановок ПУЕ-2009

14. НАПБ Б.03.002-2007. Нормы определения категорий помещений, зданий и наружных установок по взрывопожарной и пожарной опасности.

15. НПАОП 0.00-1.28-10 Правила охорони праці під час експлуатації електронно-обчислювальних машин.

16. Flow [Електронний ресурс]: Режим доступу: http://Flowtype.org/docs/base-types.html Назва з екрану. – Flow. Дата перегляду – 3.12.2014 р.