# E. Код планеты Варгон (50 баллов)

На планете Варгон все жители помешаны на исследовании программирования. Настолько, что правительство стало выдавать разрешение на визит планеты только в том случае, если турист покажет им новую библиотеку. Нашему герою, вселенскому путешественнику Опенсорсному Лорду, жизненно необходимо попасть на планету. Откопав в своих архивах давно забытую, но очень полезную библиотеку ‘X‘, путник с радостью загрузил ее на таможенный компьютер. Библиотека была настолько полезной, что таможня тут же внедрила ее во все разработки планеты.

Однако в последний день пребывания Опенсорсного Лорда на замечательной планете программы начали ломаться. Путешественника арестовали, и ему готовится жестокий приговор, если он не починит всё. Потратив несколько часов на исследование проблемы, Опенсорсный Лорд дошел до причины ошибки. Всё дело оказалось в некорректном использовании компонента ‘n‘ этой библиотеки ‘X‘. Данный компонент является фабрикой объектов, куда передаётся начальное состояние. Полученным из фабрики объектам доопределяют начальное состояние либо сеттером ‘w‘, либо сеттером ‘av‘. Если в начальное состояние передать свойство ‘a‘ со строковым значением, то его нельзя доопределить с помощью сеттера ‘w‘, и нужно использовать ‘av‘.

Таможня успела внедрить код библиотеки в абсолютно все программы на планете, поэтому вручную все использования компонента ‘n‘ библиотеки ‘X‘ не перебрать. Нужно найти все пострадавшие места кода автоматически. Помогите написать проверку, которая находила бы в коде использование сеттера ‘w‘, работающего некорректно (то есть в тех местах, где в начальное состояние фабрики ‘X.n‘ было передано свойство ‘a‘, являющееся строкой).

Про код программ на планете Варгон известно, что:

* он написан на es3,
* переменные получают свои значения при декларации и не переписываются, т.е. в коде не будет подобного ‘var a = x; a = y;‘ и ‘var a = b = 1;‘,
* обращение к свойствам объекта возможно как через точку, так и через скобки (‘a.b‘ и ‘a[’b’]‘),
* часть выражения может быть сохранена в переменной, но никогда не передаётся в функцию параметром (‘a(x)‘ — запрещено),
* нет функций, которые возвращают часть искомого выражения,
* нет свойств объектов или элементов массивов, которые содержат часть выражения,
* при обращении к свойству объекта, название свойства может быть взято из переменной (‘a[x]‘, x — переменная).

Проверка должна быть оформлена в виде CommonJS-модуля, который экспортирует функцию, принимающую на вход абстрактное синтаксическое дерево (ast) проверяемого кода.

Функция должна вернуть массив из ast-узлов, которые соответствуют местам вызова сеттера ‘w‘. Порядок элементов в массиве не важен, дубли не допускаются.

module.exports = function (ast) {

...

return [...];

}

Ast получается из js-кода с помощью модуля @babel/parser версии 7.6.0.

const parser = require(’@babel/parser’),

ast = parser.parse(code);

## Пример

|  |  |
| --- | --- |
| **Ввод** | **Вывод** |
| {  "type": "File",  "start": 0,  "end": 63,  "program": {  "type": "Program",  "start": 0,  "end": 63,  "sourceType": "script",  "interpreter": null,  "body": [  {  "type": "ExpressionStatement",  "start": 0,  "end": 30,  "expression": {  "type": "AssignmentExpression",  "start": 0,  "end": 29,  "operator": "=",  "left": {  "type": "MemberExpression",  "start": 0,  "end": 24,  "object": {  "type": "CallExpression",  "start": 0,  "end": 12,  "callee": {  "type": "MemberExpression",  "start": 0,  "end": 3,  "object": {  "type": "Identifier",  "start": 0,  "end": 1,  "name": "X"  },  "property": {  "type": "Identifier",  "start": 2,  "end": 3,  "name": "n"  },  "computed": false  },  "arguments": [  {  "type": "ObjectExpression",  "start": 4,  "end": 11,  "properties": [  {  "type": "ObjectProperty",  "start": 5,  "end": 10,  "method": false,  "key": {  "type": "Identifier",  "start": 5,  "end": 6,  "name": "a"  },  "computed": false,  "shorthand": false,  "value": {  "type": "StringLiteral",  "start": 8,  "end": 10,  "extra": {  "rawValue": "",  "raw": "''"  },  "value": ""  }  }  ]  }  ]  },  "property": {  "type": "Identifier",  "start": 23,  "end": 24,  "name": "w"  },  "computed": false  },  "right": {  "type": "ObjectExpression",  "start": 27,  "end": 29,  "properties": []  }  }  },  {  "type": "ExpressionStatement",  "start": 32,  "end": 62,  "expression": {  "type": "AssignmentExpression",  "start": 32,  "end": 61,  "operator": "=",  "left": {  "type": "MemberExpression",  "start": 32,  "end": 56,  "object": {  "type": "CallExpression",  "start": 32,  "end": 44,  "callee": {  "type": "MemberExpression",  "start": 32,  "end": 35,  "object": {  "type": "Identifier",  "start": 32,  "end": 33,  "name": "X"  },  "property": {  "type": "Identifier",  "start": 34,  "end": 35,  "name": "n"  },  "computed": false  },  "arguments": [  {  "type": "ObjectExpression",  "start": 36,  "end": 43,  "properties": [  {  "type": "ObjectProperty",  "start": 37,  "end": 42,  "method": false,  "key": {  "type": "Identifier",  "start": 37,  "end": 38,  "name": "a"  },  "computed": false,  "shorthand": false,  "value": {  "type": "StringLiteral",  "start": 40,  "end": 42,  "extra": {  "rawValue": "",  "raw": "''"  },  "value": ""  }  }  ]  }  ]  },  "property": {  "type": "Identifier",  "start": 55,  "end": 56,  "name": "l"  },  "computed": false  },  "right": {  "type": "ObjectExpression",  "start": 59,  "end": 61,  "properties": []  }  }  }  ],  "directives": []  }  } | [  {  "type": "Identifier",  "start": 23,  "end": 24,  "name": "w"  }  ] |

## Примечания

Следующий код можно взять за основу для обхода дерева.

/\*\*

\* Функция обхода дерева. Выполняет обход дерева в глубину,

\* передаваяв callback-функции onNodeEnter (до посещения потомков)

\* и onNodeLeave (после посещения потомков) каждый узел дерева

\* и текущую область видимости (смотри определение Scope ниже)

\*

\* @param {object} ast Исходное ast

\* @param {Function} [onNodeEnter=(node, scope)=>{}] Вызывается для каждого узла до посещения потомков

\* @param {Function} [onNodeLeave=(node, scope)=>{}] Вызывается для каждого узла после посещения потомков

\*/

function traverse(

ast,

onNodeEnter = (node, scope) => {},

onNodeLeave = (node, scope) => {}

) {

const rootScope = new Scope(ast);

\_inner(ast, rootScope);

/\*\*

\* Определение области видимости узла.

\* Может либо вернуть текущий scope, либо создать новый

\*

\* @param {object} astNode ast-узел

\* @param {Scope} currentScope текущая область видимости

\* @return {Scope} область видимости для внутренних узлов astNode

\*/

function resolveScope(astNode, currentScope) {

let isFunctionExpression = ast.type === ’FunctionExpression’,

isFunctionDeclaration = ast.type === ’FunctionDeclaration’;

if (!isFunctionExpression &&

!isFunctionDeclaration) {

// Новые области видимости порждают только функции

return currentScope;

}

// каждая функция порождает новую область видимости

const newScope = new Scope(ast, currentScope);

ast.params.forEach(param => {

// параметры функции доступны внутри функции

newScope.add(param.name);

});

if (isFunctionDeclaration) {

// имя функции при декларации доступно снаружи функции

currentScope.add(ast.id.name);

} else {

// имя функции-выражения доступно только внутри неё

newScope.add(ast.id.name);

}

return newScope;

}

/\*\*

\* Рекурсивная функция обхода ast

\*

\* @param {object} astNode Текущий ast-узел

\* @param {Scope} scope Область видимости для текущего ast-узла

\*/

function \_inner(astNode, scope) {

if (Array.isArray(astNode)) {

astNode.forEach(node => {

/\* Рекурсивный обход элементов списков.

\* Списками являются, например, параметры функций

\*/

\_inner(node, scope);

});

} else if (astNode && typeof astNode === ’object’) {

onNodeEnter(astNode, scope);

const innerScope = resolveScope(astNode, scope),

keys = Object.keys(astNode).filter(key => {

// loc - служебное свойство, а не ast-узел

return key !== ’loc’ &&

astNode[key] && typeof astNode[key] === ’object’;

});

keys.forEach(key => {

// Обход всех потомков

\_inner(astNode[key], innerScope);

});

onNodeLeave(astNode, scope);

}

}

}

/\*\*

\* Представление области видимости

\*

\* @class Scope (name)

\* @param {object} astNode ast-узел, породивший эту область видимости

\* @param {object} parentScope Родительская область видимости

\*/

function Scope(astNode, parentScope) {

this.\_node = astNode;

this.\_parent = parentScope;

this.\_vars = new Set();

}

Scope.prototype = {

/\*\*

\* Добавление имени переменной в область видимости

\*

\* @param {string} name имя переменной

\*/

add(name) {

this.\_vars.add(name);

},

/\*\*

\* Была ли определена переменная с таким именем.

\*

\* @param {string} name имя переменной

\* @return {boolean} Встречалась ли переменная с таким именем в доступных областях видимости

\*/

isDefined(name) {

return this.\_vars.has(name) || (this.\_parent && this.\_parent.isDefined(name));

}

};