



# **More Informative Untyped IntelliSense with Type Carriers**

Marios Ntoulas

Thesis submitted in partial fulfillment of the requirements for the

*Masters' of Science degree in Computer Science*

University of Crete

School of Sciences and Engineering

Computer Science Department

Voutes University Campus, Heraklion, GR-70013, Greece

Thesis Advisor: Prof. *Anthony Savidis*



University of Crete  
Computer Science Department

## More Informative Untyped IntelliSense with Type Carriers

Thesis submitted by  
**Marios Ntoulas**  
in partial fulfillment of the requirements for the  
Masters' of Science degree in Computer Science

### THESIS APPROVAL

Author:

*Marios Ntoulas*

Marios Ntoulas, Department of Computer Science

Committee approvals:

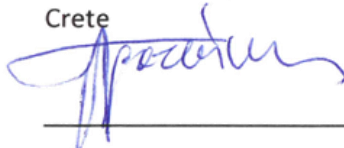
**Antonios  
Savvidis**

Digitally signed by  
Antonios Savvidis  
Date: 2021.05.30  
13:59:30 +03'00'

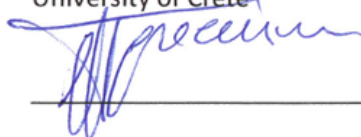
Anthony Savdis  
Professor, Computer Science Department, University of  
Crete, Thesis Supervisor



Evangelos Markatos  
Professor, Computer Science Department, University of  
Crete



Polyvios Pratikakis  
Assistant Professor, Computer Science Department,  
University of Crete



Department approval:

Polyvios Pratikakis  
Assistant Professor, Director of Graduate Studies,  
Computer Science Department, University of Crete

Heraklion, April 2021

# **More Informative Untyped IntelliSense with Type Carriers**

Marios Ntoulas

Master's Thesis

University of Crete  
Computer Science Department

## **Abstract**

IntelliSense has a major impact in the development process. The automations during source code editing assist developers in navigating, understanding, avoiding errors, and speeding-up the overall editing process. Currently, the use of untyped languages and the quantity of untyped code in software systems tends to escalate, including the deployment of third-party untyped code libraries. However, many errors are still shadowed by the dynamic nature of untyped code making semantic analysis a difficult and sometimes undecidable job. While there are typed language extensions, and sometimes newer versions introducing class-based constructs, not only there is still a lot of untyped legacy code, but many programmers prefer the abstraction flexibility and expressive economy of the untyped universe, although acknowledging this is traded for lack of type safety. In this context we believe there is a need for improved editing tools capable to analyze and evaluate incrementally the source code fragments of untyped languages while being edited, which eventually deliver more informative on-demand type feedback to programmers.

We present the techniques for the semantic analysis of untyped source code during editing focusing on the JavaScript language. Our system is implemented on top of the Visual Studio Code IDE and exploits the editor extension hooks offered by the Language Server Protocol for incremental parsing and editing automations. Our approach is based on the notion of type carriers which are associated to instructions that change the value or the type of a variable, and their chaining during editing in a way

enabling to precisely track from any given source location the stack of active type carriers per symbol, thus being able to on-demand tell its plausible context-dependent type.

# Βελτιωμένο IntelliSense με Μεταφορείς Τύπων

ΜΑΡΙΟΣ ΝΤΟΥΛΑΣ

Μεταπτυχιακή Εργασία

Πανεπιστήμιο Κρήτης  
Τμήμα Επιστήμης Υπολογιστών

## Περίληψη

Το IntelliSense έχει σημαντικό αντίκτυπο στη διαδικασία ανάπτυξης λογισμικού. Οι αυτοματισμοί κατά την επεξεργασία του πηγαίου κώδικα βοηθούν τους προγραμματιστές στην πλοήγηση, την κατανόηση, την αποφυγή σφαλμάτων και την επιτάχυνση της συνολικής διαδικασίας επεξεργασίας. Στις μέρες μας, η χρήση γλωσσών προγραμματισμού χωρίς τύπους και η ποσότητα του κώδικα χωρίς τύπους σε συστήματα λογισμικού τείνει να κλιμακώνεται, όπως και η χρήση βιβλιοθηκών κώδικα τρίτων. Ωστόσο, πολλά σφάλματα εξακολουθούν να επισκιάζονται από τη δυναμική φύση του κώδικα χωρίς τύπους που καθιστά τη σημασιολογική ανάλυση μια δύσκολη και μερικές φορές μη υπολογίσιμη εργασία. Ενώ υπάρχουν επεκτάσεις για τέτοιες γλώσσες και νεότερες εκδόσεις που εισάγουν κατασκευές βασισμένες στις κλάσεις, υπάρχει ακόμα πολύς κώδικας χωρίς τύπους που χρησιμοποιείται αλλά δεν υποστηρίζεται, αλλά και πολλοί προγραμματιστές προτιμούν την ευελιξία και την εκφραστική οικονομία του σύμπαντος χωρίς τύπους, αποδεχόμενοι την ελλειψη της ασφάλειας που θα τους προσέφεραν αυτοί. Σε αυτό το πλαίσιο πιστεύουμε ότι υπάρχει ανάγκη για βελτιωμένα εργαλεία επεξεργασίας, ικανά να αναλύουν και να αποτιμούν σταδιακά τα κομμάτια πηγαίου κώδικα των γλωσσών χωρίς τύπους κατά την επεξεργασία τους, τα οποία τελικά παρέχουν βελτιωμένη ανατροφοδότηση τύπων κατά απαίτηση στους προγραμματιστές.

Παρουσιάζουμε τις τεχνικές για τη σημασιολογική ανάλυση του πηγαίου κώδικα χωρίς τύπους κατά την επεξεργασία εστιάζοντας στη γλώσσα JavaScript. Το σύστημά μας

υλοποιείται πάνω από το Visual Studio Code IDE και εκμεταλλεύεται τα άγκιστρα επέκτασης προγράμματος που προσφέρει το Language Server Protocol για αυξητική επεξεργασία και αυτοματισμούς. Η προσέγγισή μας βασίζεται στην έννοια των μεταφορέων τύπων οι οποίοι, σχετίζονται με τις εντολές που αλλάζουν την τιμή ή τον τύπο μιας μεταβλητής και την σύνδεσή αυτών κατά την επεξεργασία με τρόπο που επιτρέπει να παρακολουθείται με ακρίβεια από οποιαδήποτε τοποθεσία προέλευσης στον κώδικα η στοίβα των ενεργών μεταφορέων τύπων ανά σύμβολο, ώστε να μπορεί κατά απαίτηση να αποφασιστεί ο εξαρτώμενος από τα συμφραζόμενα εύλογος τύπος.



## **Ευχαριστίες (Acknowledgements)**

Θα ήθελα να ευχαριστήσω όλους τους ανθρώπους χωρίς τους οποίους δεν θα ήταν δυνατή η εκπόνηση της μεταπτυχιακής μου εργασίας. Τον επόπτη μου, κ. Αντώνη Σαββίδη για την υποστήριξή του κατά τη διάρκεια των σπουδών μου, τους συναδέλφους μου για την συνεργασία μας, και την οικογένειά μου για την βοήθεια και την υποστήριξή τους όλα αυτά τα χρόνια.

# Table of Contents

Abstract .....	2
Περίληψη .....	4
Ευχαριστίες (Acknowledgements).....	6
Table of Contents .....	7
List of Figures .....	10
List of Tables .....	12
1. Introduction.....	13
1.1 Background .....	13
1.1.1 IntelliSense or Editing Automations .....	13
1.1.2 Typed vs Untyped IntelliSense .....	14
1.2 Motivation .....	15
1.2.1 Limitations of Untyped IntelliSense .....	15
1.3 Objectives.....	15
1.4 Outline.....	16
2. Related Work .....	17
2.1 Type-Annotation Based Systems .....	17
2.1.1 TypeScript.....	17
2.1.2 Flow .....	18
2.1.3 JSDoc .....	19
2.2 Evaluation Based Systems .....	19

2.2.1	Quokka.js .....	20
3.	Overview .....	21
3.1	Software Architecture .....	21
3.2	Language Server Protocol (LSP).....	22
3.3	Syntax Analysis.....	25
3.3.1	Error Tolerance .....	26
3.3.2	Incremental Document Synchronization.....	26
3.3.3	Incremental Parsing .....	27
4.	Type Carriers .....	29
4.1	Basic Idea .....	29
4.2	Type Information.....	29
4.3	Type Carrier .....	30
4.4	Binding Type Carriers to Symbols.....	32
4.4.1	Symbols.....	32
4.4.2	Type Binders .....	33
4.5	Type Computation.....	33
4.5.1	Symbol Look Up.....	34
4.5.2	Determination of Active Type Binders.....	35
4.5.3	Evaluation of Type Carriers.....	37
5.	Handling Variables and Expressions .....	39
5.1	Declarations.....	39
5.1.1	Declarations via var Keyword .....	40
5.1.2	Declarations via let Keyword.....	41
5.1.3	Declarations via const Keyword .....	42
5.2	Expressions.....	43

6.	Handling Assignments .....	44
6.1	Variables.....	44
6.2	Object Fields .....	44
7.	Handling Function Definitions .....	46
7.1.1	Parameter Type Prediction / Speculation.....	46
7.2	Plausible Return Types.....	47
8.	Function Calls (Call Sites) .....	49
8.1	Side Effects .....	49
8.2	Function Dependencies .....	50
8.2.1	Parameters .....	50
8.2.2	Free Variables .....	51
9.	Branches and Loops .....	52
9.1	If-Else Statement .....	52
9.2	Switch Statement.....	57
9.3	For Statement .....	58
9.4	While Statement .....	59
10.	Editing Automations .....	60
10.1	Code Completion.....	60
10.2	Parameter Help .....	62
10.3	Quick Information .....	65
10.4	Goto Definition .....	67
10.5	Document Symbols .....	68
11.	Conclusions and Future Work .....	70
	References.....	71

## List of Figures

Figure 1 – IntelliSense example (suggesting code completions).....	14
Figure 2 – TypeScript Type Annotations used by IntelliSense .....	17
Figure 3 – TypeScript IntelliSense (using type inference) .....	18
Figure 4 – Flow Type Information in Code completion (through type annotation) ....	19
Figure 5 – JSDoc Type Annotations used by IntelliSense.....	19
Figure 6 – Quokka.js Code Coverage in Action .....	20
Figure 7 – System Architecture .....	21
Figure 8 – No LSP vs LSP (A Language Server communicates with multiple IDEs)	22
Figure 9 - LSP in action (Host – Language Server communication).....	23
Figure 10 – LSP in action (Host communicating with multiple Language Servers) ...	23
Figure 11 – Incremental Parsing (Marking the affected AST nodes as volatile).....	28
Figure 12 – Symbol Look up Visualization .....	35
Figure 13 – Searching the active Type Binder of x .....	36
Figure 14 – Searching the active type binder of x inside a block.....	37
Figure 15 – AST for var declaration (after the analysis) .....	41
Figure 16 – AST for let declaration (after the analysis) .....	42
Figure 17 – Type Carriers in expression nodes .....	43
Figure 18 – AST after the analysis of Assignment Expression .....	44
Figure 19 – Search the Active Type Binders of x Inside an If Statement .....	55
Figure 20 - Search active binders of x starting inside a then/else statement .....	57
Figure 21 – Code Completion.....	62

Figure 22 – Code Completion on objects .....	62
Figure 23 – Parameter Help (1/2) .....	64
Figure 24 – Parameter Help (2/2) .....	64
Figure 25 – Quick Information (Showing all plausible types after an if statement)....	66
Figure 26 – Quick Information (In Else Statement) .....	66
Figure 27 - Go to Definition (Before).....	67
Figure 28 - Go to Definition (After) .....	68
Figure 29 – Code Outline and Breadcrumb .....	69
Figure 30 - Go To Symbol Command .....	69

## List of Tables

Table 1 – List of currently supported language features in LSP .....	25
Table 2 – Type Info specification .....	29
Table 3 – Type Carriers for expressions .....	31
Table 4 – Symbol specification.....	32
Table 5 – Type Binder specification .....	33

# **1. Introduction**

## **1.1 *Background***

### **1.1.1 IntelliSense or Editing Automations**

On the early stages of software development there were only simple text editors. Later Integrated Development Environments (IDEs) came to increase productivity by providing tools not only to write code, but also compile, execute, debug and many more features all in one application. IDEs also provide another useful feature called IntelliSense [1] which intelligently helping the developers to write code.

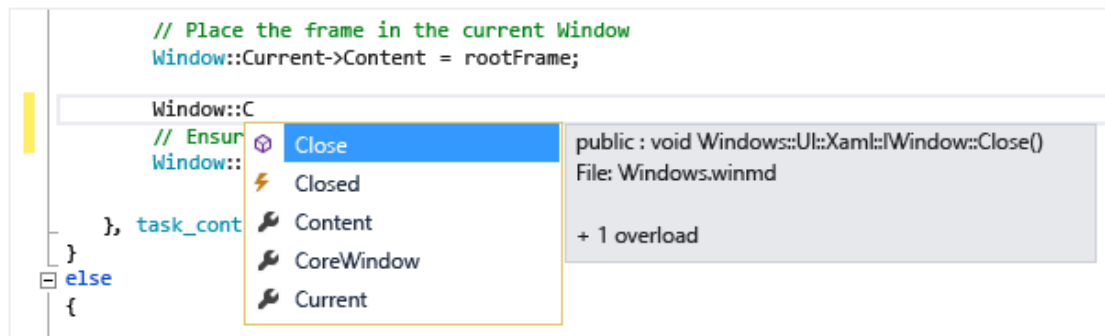
IntelliSense is a collection of different language specific features running in source code editors, assisting the developers in real time as they write code. There are numerous features under the IntelliSense term but the four basic are code completion, parameter help, quick information and goto definition.

The objective of these editing automations is to increase productivity and make the lives of the developers a lot easier. First of all, editing automations help the developers to avoid syntax errors that they would discover during the compilation. IntelliSense underlines the syntax errors during the editing and saves the time that they would have to come back and fix them.

Additionally, IntelliSense enables the developers to speed up the development process. It typically understands the programming language and knows the structure of the source code. Thus, it is able to suggest possible completions as the developer types. Otherwise, developers should have to memorize or spend time searching in the source code and documentations the names of classes, the names and the types of their members, the name of each function they use, the names and the types of their parameters, etc.



Moreover, there are editing automations that help the developers to refactor the source code quickly. IntelliSense suggests them possible recommendations to improve the quality of the code and it can apply them with a press of a button. It can restructure the code to improve maintainability without affecting the runtime behavior. There are editing automations to format code, rename symbols, move pieces of code to new functions, etc.



**Figure 1** – IntelliSense example (suggesting code completions)

### 1.1.2 Typed vs Untyped IntelliSense

The way IntelliSense works varies from language to language. Every programming language has its own characteristics and it is unique in its own way. The implementation of those features is different especially when we have to deal with a typed and an untyped language, typically with the latter being inferior.

Implementing all those editing automations in a typed language is not a big deal. The static type information carries everything someone needs to provide all those results. The fields of all the objects and their types, the return types of the functions and the types of their parameters. All this information is well known at compile-time.

On the other hand, implementing any editing automation in a language that has a weak dynamic typing discipline is non-trivial. A symbol can have different types on different locations in the source code which can also be different on each execution of the same program. However IntelliSense works with type information which in case of untyped languages is absent until runtime. Objects are typically populated during runtime and those languages usually support first-class functions. The analysis on these languages

has to compute the types and the possible values from the code semantics. But tracking the flow of data and control can be really hard.

## **1.2 Motivation**

### **1.2.1 Limitations of Untyped IntelliSense**

Most of the tools designed for untyped languages are lacking in comparison to the same tools for typed languages. The same applies to editing automations. The nature of untyped languages makes the development of tools for them difficult. In the meanwhile the use of those languages has sky-rocketed, from writing simple scripts to writing large-scale systems causing the need for such tools to increase.

There are many attempts to support editing automations by giving the ability to the developers to add type annotations on the source code. Many typed languages have been created on top of weakly typed languages to be able to provide such automations. We would like to see in real life automations for weakly typed languages, known only from strongly typed languages without requiring the developers to write any additional code, just by analyzing their existing code.

## **1.3 Objectives**

This work targets the field of editing automations focusing on weakly typed languages and explores the techniques that can be used to analyze the source code and extract type information from it. Specifically the objective of this work is to:

- Explore ways to make the IntelliSense more informative as the title of this thesis implies.
- Describe the source code analysis and the collection of the type information.
- Compose type information to create more complex type sets to provide all the different plausible types according to the different paths on the source code.

- Design algorithms that compute the type information of a symbol according to the context.
- Implement an IntelliSense system supporting the four basic editing automations.

The work in this thesis has been implemented for the JavaScript language. The most important reason we choose JavaScript is because it is an object based weakly typed language and we wanted to focus on such languages. Additionally it is the most popular untyped language for web-development which can also be used to develop back-end services.

## **1.4 Outline**

This thesis is organized as follows. Section 2 provides some background information on editing automations and discusses the related work, focusing on tools for the JavaScript programming language. Section 3 is an overview of the system we built and the technologies that we used to support this thesis. Section 4 introduces the basic structures we use during the analysis to collect information about the symbols of the source code. Sections 5 – 8 present how the structures described in Section 4 are created and stored during the analysis of the definitions and the expressions of the language. Section 9 focuses on how we use some of the statements of the language to compose more complex type information sets for the symbols. Section 10 shows how all the information we collect during the analysis is utilized during the editing of the source code to implement the basic editing automations. Finally, Section 11 summarizes the key points of this thesis, and discusses directions for future research.

## 2. Related Work

### 2.1 Type-Annotation Based Systems

One of the most common categories of tools that are enhancing the editing experience in untyped languages, is consisted of tools that let the developer to add Type Annotations on the source code. These Type Annotations could be part of the language, or it could be comments on the source code, without affecting the syntax of the language. In the former case, the source code is transpiled to the target language. IntelliSense parses the Type Annotations and uses this information to improve the editing experience. These systems grant the user the ability to have the benefits that statically typed languages typically offer during editing.

#### 2.1.1 TypeScript

TypeScript [2] is a programming language developed by Microsoft that extends JavaScript by adding static type definitions through Type Annotations. It also has a Type Checker that validates that the source code is working correctly. Moreover it supports declaration files, much like C++ header files, to let the developers have the editing experience of TypeScript without porting their projects to TypeScript.

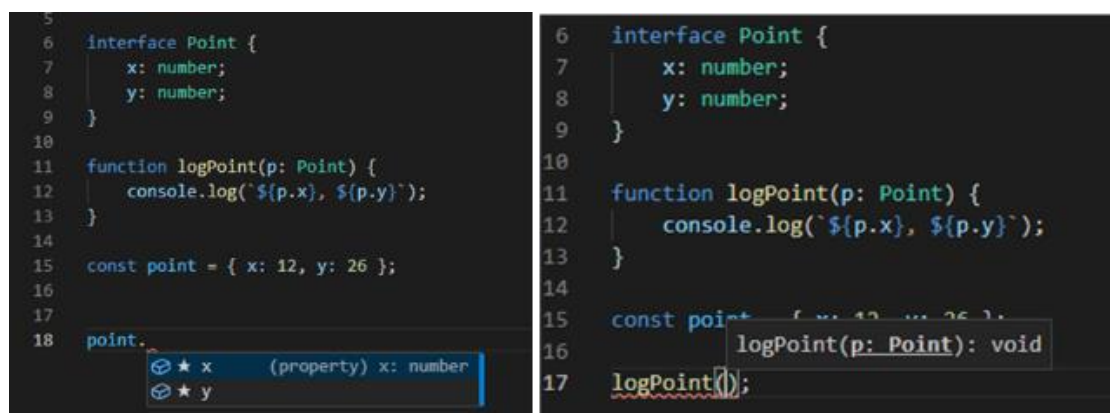


Figure 2 – TypeScript Type Annotations used by IntelliSense

TypeScript is a super set of JavaScript, thus besides Type Annotations, its IntelliSense can also provide type information for pure JavaScript source code by applying type inference on it. In this way developers are not required to write any additional code and still have a more precise but still limited experience.

```
1
2  function isToday(date) {
3      const today = new Date();
4      return date == today;
5  }
6      isToday(date: any): boolean
7  isToday()
8
```

**Figure 3** – TypeScript IntelliSense (using type inference)

### 2.1.2 Flow

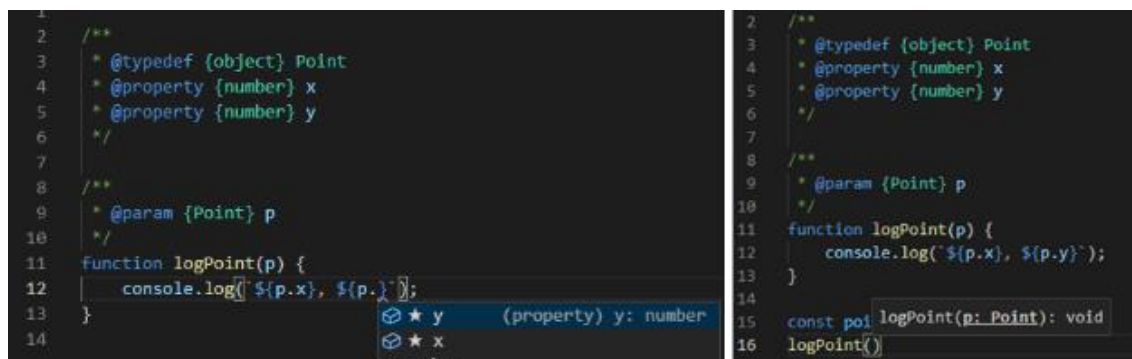
Flow [3] is a static type checker for JavaScript developed by Facebook. It is possible to work with type inference like TypeScript. Moreover it offers the developers the ability to add static type definitions through type annotations. It is pretty similar to TypeScript, even the syntax differences between them is not significant. They both provide a static Type Checker and tools that utilize the type annotations to enrich the editing experience. Although, Flow is less popular than TypeScript because there is tiny support from the most used JavaScript libraries.

```
8  let data: {
9      participants: [],
10     serialize: () => string
11 }
12
13 d
  [x] doc_method      doc_method
  [x] data            {participants: [], seriali ×
  [x] define          ze: () => string}
  [x] dowhile         Do-While Statement
```

**Figure 4** – Flow Type Information in Code completion (through type annotation)

### 2.1.3 JSDoc

JSDoc [4] is an open source Application Programming Interface (API) [5] documentation generator [6] for JavaScript. Developers use Type Annotations inside comments in their JavaScript source code and the tool generates an HTML website from the source files with the documentation. Many editing automation tools take advantage of this information and parse the comments to provide better code analysis. The difference between the previous systems and JSDoc is that JSDoc comments are standard JavaScript comments. Hence, there is no need for a build step to transpile the source code to JavaScript. The major drawback of all these systems is that they need the developer to annotate the source code, to work as expected.



The image consists of two side-by-side screenshots of a code editor with a dark theme. The left screenshot shows a JavaScript file with JSDoc annotations. Lines 2-6 define a `Point` type with properties `x` and `y`. Lines 8-10 define a `logPoint` function that takes a `Point` parameter. Line 12 shows the function call `console.log(p.x, p.y);`. An IntelliSense popup is visible below line 12, showing suggestions for `y` (property) and `x` (property). The right screenshot shows the same code with additional annotations. Line 14 adds a `const poi` declaration. Line 15 adds a `logPoint(p: Point): void` signature. Line 16 shows the `logPoint()` call. An IntelliSense popup is visible below line 15, showing the `logPoint(p: Point): void` signature.

**Figure 5** – JSDoc Type Annotations used by IntelliSense

## 2.2 Evaluation Based Systems

Another category of tools that are enhancing the source code editing experience is consisted of tools that actually execute the code as the developer is writing the code. These tools are less popular than the ones described in the previous section. They provide more precise results since they are executing the code, without requiring the users writing any additional code to work, but they have many limitations. The source

code must be syntactically correct for them to be able to provide any results and the delay could be huge making them impractical for large-scale projects.

### 2.2.1 Quokka.js

Quokka.js [7] is a rapid prototyping playground in the editor that offers inline reporting, code coverage and rich output formatting. Runtime values are updated and displayed in the IDE next to the source code, as the developer types. Quokka.js executes the source code during the editing and the results of the execution are displayed right in the editor.



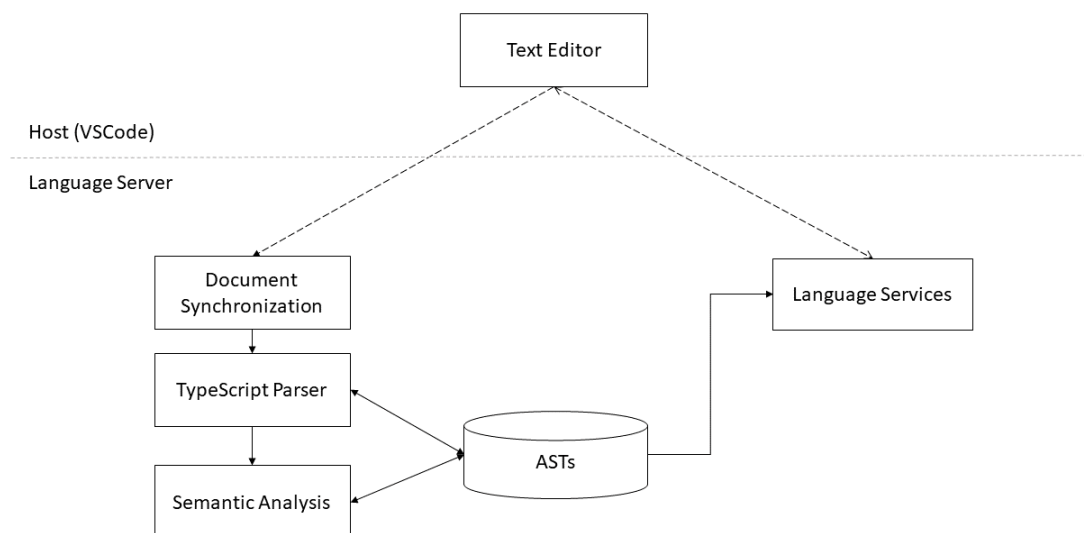
**Figure 6** – Quokka.js Code Coverage in Action

## 3. Overview

### 3.1 Software Architecture

To support this thesis we have built an extension for Visual Studio Code [8], and a JavaScript Language Server, which the extension communicates with to provide programmatic language features. Also Microsoft's TypeScript compiler was used by the Language Server to parse the source code to Abstract Syntax Tree (AST) [9]. The AST is analyzed to collect various information about the source code. The Language Server is listening specific notifications coming from Visual Studio Code editor and provides the following editing automations:

- Code Completion
- Parameter Help
- Quick Information
- Goto Definition
- Document Symbols



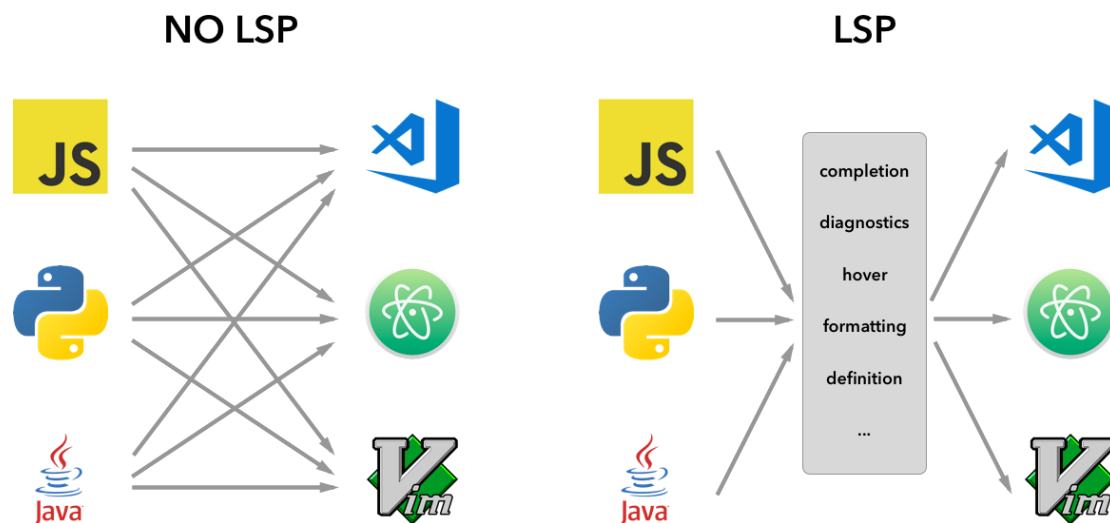
**Figure 7** – System Architecture



Microsoft Visual Studio Code is a powerful source code editor that is built with extensibility in mind. Almost every part of it, from the User Interface (UI) to the editing experience, can be enhanced or customized through its extension API. It comes with built-in support for JavaScript, TypeScript, and Node.js but it has a rich ecosystem of extensions for other programming languages and runtimes.

### 3.2 Language Server Protocol (LSP)

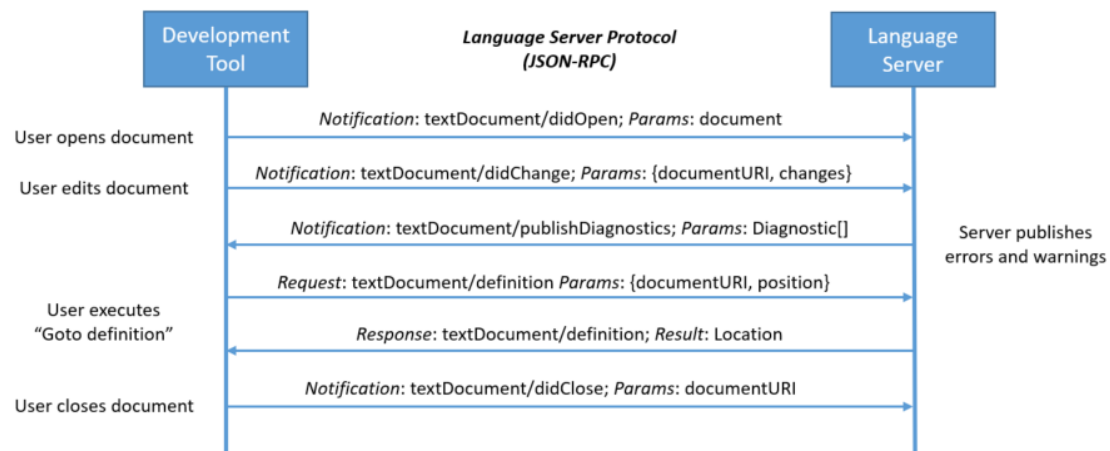
The Language Server Protocol (LSP) [10], was originally developed for Microsoft Visual Studio Code, and now it is a standard protocol for use between source code editors or IDEs and servers that provide language-specific programming features. The objective of the LSP is to allow the implementation and distribution of programming language support independently of any given editor or IDE.



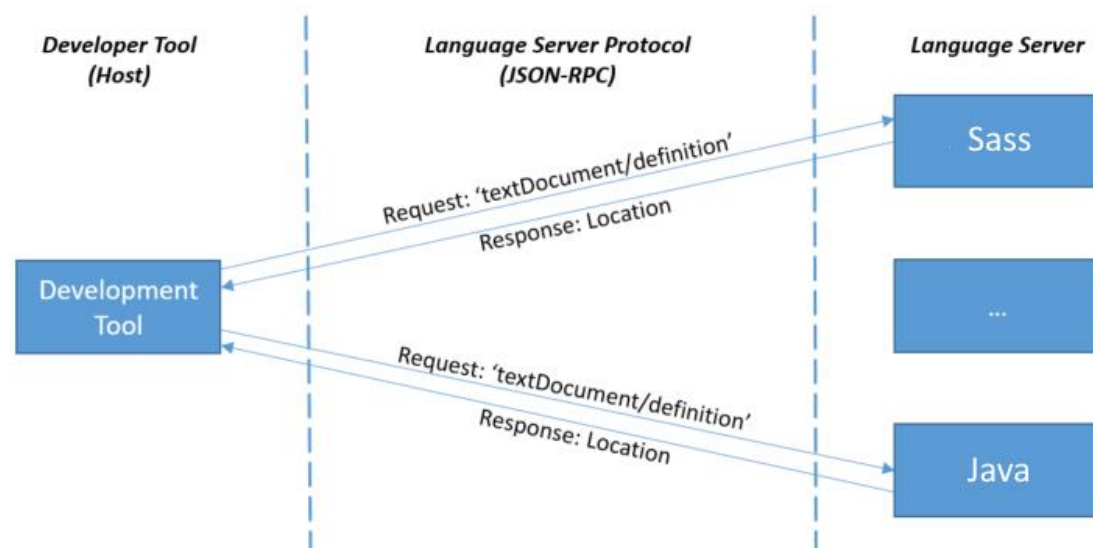
**Figure 8** – No LSP vs LSP (A Language Server communicates with multiple IDEs)

When a user edits a source code file using a tool that supports Language Server Protocol, the tool acts as a client that consumes the language services provided by a language server. Every action of the user during editing is sent by the client to the language server that stores the full state of the client, such as which documents exist in the workspace, the text that they contain, or which of them are currently open. Then, the client may request the server to perform a language service, e.g. when a character

is inserted in a specific position, the client will request the server to provide a list of possible words for auto-completion.



**Figure 9** - LSP in action (Host – Language Server communication)



**Figure 10** – LSP in action (Host communicating with multiple Language Servers)

Implementing editing features for a programming language could be hard and needs considerable effort. Also, each development tool usually provides its own API to implement those features, and it is written in different programming languages. Language Server Protocol came to unify that API by providing a Language Server and a common way for development tools to communicate with it. Without LSP, language providers should have to implement editing features for every development tool they would like to support. With LSP, they are able to reuse their implementation to any LSP-compliant code editor. In addition, editing features could be resource intensive,

causing significant CPU and memory usage. Language Servers run in their own process, avoiding this performance cost and let the performance of the development tools unaffected.

As described in the previous paragraphs, the Language Server Protocol provides some hooks for specific events, like document change, and hover, that may occur during the editing of a source code file. The Language Server must handle these events to update its state and provide language services like quick information and code-completion. The following table shows the language features that are currently supported in a language server.

Feature	Description
Document Highlighting	Highlights all symbols in a document
Hover	Provides hover information for a symbol selected in a text document
Completion	Provides a list of possible completions
Completion Resolve	Resolves additional information for a given completion item
Signature Help	Provides signature help for a symbol selected in a document
Goto Definition	Provides go to definition support for a symbol selected in a document
Goto Type Definition	Provides go to type/interface definition support for a symbol selected in a document
Goto Implementation	Provides go to implementation definition support for a symbol selected in a document

Find References	Finds all project-wide references for a symbol selected in a document
Document Symbols	Lists all symbols defined in a document
Workspace Symbols	Lists all project-wide symbols
Code Actions	Compute commands to run (typically beautify/refactor) for a given document and range
CodeLens	Compute CodeLens statistics for a given document
Document Formatting	This includes formatting of whole documents, document ranges and formatting on type
Rename	Project-wide rename of a symbol
Document Links	Compute and resolve links inside a document
Document Colors	Compute and resolve colors inside a document to provide color picker in editor

**Table 1** – List of currently supported language features in LSP

### **3.3 Syntax Analysis**

To be able to provide such language services, the Language Server needs to analyze semantically the source code and gather any related information about it. Hence, the source code should be in a form that someone could easily do some reasoning on it. A suitable form that is typically used is the Abstract Syntax Tree (AST). So, as the user edits the source code document, the language server must perform syntax analysis [11] on it to generate the AST and then perform semantic analysis [12] on the AST to collect the information it needs.

### 3.3.1 Error Tolerance

During development the code in the editor is usually incomplete and syntactically incorrect. But developers would still expect the editing experience to remain unaffected and code completion and other editing automations to work. Therefore, an error tolerant parser is necessary for a Language Server. To be able to provide editing features based on the AST, Language Servers should utilize a parser that should be able to generate meaningful AST from partially complete or invalid code.

This is usually achieved by introducing some additional error tokens to prevent the parser from stopping and making it continue the syntax analysis. One could be used for tokens that are missing, e.g. when a semicolon is missing, the parser inserts an imaginary semicolon. Another one could be used for pieces of text that makes no sense e.g. when a character exists in a place that should not be.

```
function parseIf($str, $parent) {  
    $n = new IfNode();  
    $n->ifKeyword = eat("if");  
    $n->openParen = eat("(");  
    $n->expression = parseExpression();  
    $n->closeParen = eat(")");  
    $n->block = parseBlock();  
    $n->parent = $parent;  
}
```

The above function parses an if statement and generates the appropriate AST node. But let's say we run it with the following if statement and input, which is missing a close parenthesis token. In this case, `eat(")")` will generate a Missing Token because the grammar expects a token to be there, but it does not exist.

```
if ($expression // ) <- MissingToken  
{  
}
```

### 3.3.2 Incremental Document Synchronization

As described above, Language Servers needs to keep the state of the documents in the development tool to be able to analyze them and provide the editing automations.

Language Server Protocol provides two options to synchronize the documents in the Language Server.

By default, the development tool will send the whole content of a document to the Language Server every time a change occurs in the document. Thus, lots of data is transferred to the server repeatedly. The other option that LSP supports is the incremental text document synchronization. By enabling it, the server can install a notification handler that is called when the content of a text document is changed. In this way, only an array that contains the content changes of the document is sent to the server. Each change is described by an object that contains the range of the text being replaced, and the replacement text.

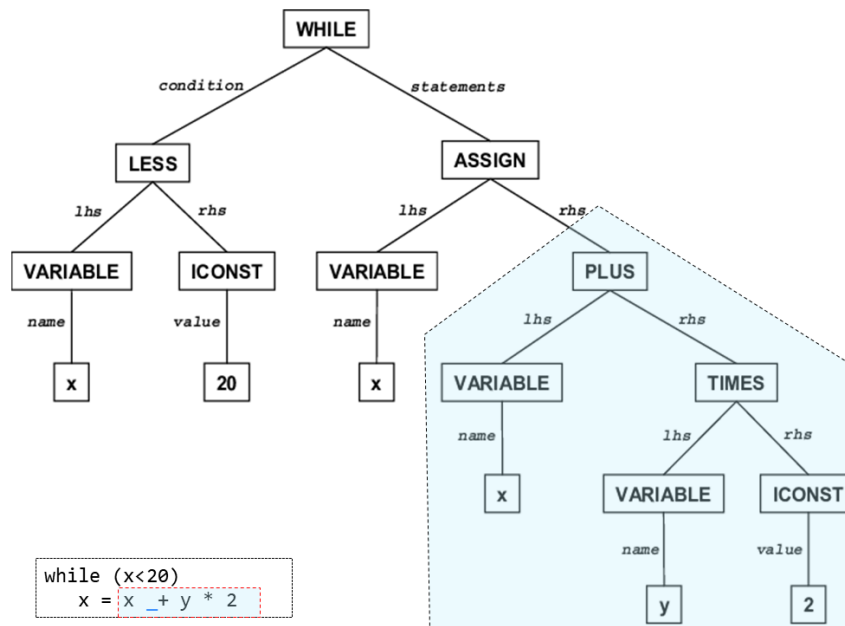
### **3.3.3 Incremental Parsing**

Parsing time is proportional to the source code size and the user changes to the source code are usually small and frequent. That means that performing syntax analysis on the whole source code on every change is impractical causing large delays since it may be executed multiple times per second. But execution time could be saved by reusing nodes from the old AST.

Instead of reparsing the whole document again and again, only the part of the corresponding edit range could be reparsed. In this way, the AST is generated incrementally [13] as the user edits the source code. When a piece of code is removed, the appropriate nodes are removed from the AST. When a piece of code is added, only that piece of code needs to be parsed and added to the AST.

To achieve this, a mapping between the AST and the source code is needed. This is usually done by storing to the AST nodes their position as an offset in the text. Additionally, we need to consider that after every change in the source code the stored position in each AST node after the edit needs to be adjusted. Those nodes may move

forward in cases that new characters were inserted or they may move backward in cases that characters were deleted.



**Figure 11** – Incremental Parsing (Marking the affected AST nodes as volatile)

To make a parser incremental, the AST nodes should be annotated with the original text per grammar symbol. When an editing change occurs, the smallest affected non-terminal node is located in the AST and its subtree is marked as volatile. Then the affected source text constitutes the input to the incremental parser. Finally, the incremental parsing is performed with a surgical parsing and an AST modification action that is very fast.

```

lex.Push(PARSE_EXPR_TOKEN);
lex.Input(<the text fragment>);
AST* ast = parser.Parse(lex);
<update the main tree>;

```

## 4. Type Carriers

### 4.1 *Basic Idea*

In a typed language, the type of variables is specified in their declaration. Their type is known at compile time and it is fixed in the whole program. In weakly typed languages, variables do not have types, instead their values do. Therefore, the type of a variable is dependent on the value that is assigned to it and may change during the execution of a program. In this system, we analyze the AST and we keep track of all the locations that the value of a variable may change. When information about a symbol is asked, based on the position in the document, we find the closest locations that a value was assigned to this symbol, and we can provide information about the type of its value.

### 4.2 *Type Information*

A *Type Information* object is used to describe the type of an expression. It can also provide information about its value. For example, the Type Information object for a numeric literal '2' will store that the type is number, that it has a value, and that the value is 2. The following table shows the structure of a Type Info object:

Property	Description
Type	The type of the expression.
Has Value	It stores whether or not it provides value information.
Value	Depending on the Type property it stores information about the value.

**Table 2** – Type Info specification



The type property of a Type Information object should be able to describe all the basic types of the language. But there are cases, such as the call of a variable that is not declared, that we have no type information. A compiler would terminate its execution providing an error, but the IntelliSense should work and provide information. Thus, the set of the types that a Type Information object can describe are the basic types of the language, plus one *any* type that indicates that we have no idea what the type could be. These types are:

- Class
- Function
- Number
- String
- Boolean
- Array
- Object
- Undefined
- Null
- Any

### **4.3 *Type Carrier***

In its pure notion, a *Type Carrier* is basically a set of instructions of how to find the type information for a symbol. They are generated during the semantic analysis, they are associated with the symbols through Type Binders, which will be discussed in the next section, and they are evaluated to gather all the type information of a symbol.

All the expressions of a program generate Type Carriers that are stored in the expression nodes. As an example, the number literal '2' will generate a Type Carrier. When this Type Carrier is evaluated, it will result to a Type Info Object that its type is Number. A variable reference x will generate a Type Carrier, which will search and return the type information of x when it is evaluated. The following table contains all the different kinds of Type Carriers that are generated and on which nodes:

Expression	Description	Evaluation
Literal	A Type Carrier is created that stores type information about the literal.	Just returns the stored type information.
Identifier	A Type Carrier is created that stores the referenced symbol and the Identifier node which is used to resolve the context of the reference when it is evaluated.	It finds the active type carriers of the referenced symbol evaluates them and returns the type information.
Binary Expression	A Type Carrier is created that stores the carriers of the operands.	It evaluates the carriers of the operands and tries to apply the operation if it is possible.
Unary Expression	A Type Carrier is created that stores the carrier of the operand.	It evaluates the carrier of the operand and returns the type information.
Call Expression	A Type Carrier is created that stores the call expression.	It evaluates the carriers returned by the callee and returns the type information. If there are no carriers, it returns undefined as type info. If the callee cannot be determined, it returns any as type info.
New Expression	Similar to call expressions, a Type Carrier is created that stores the new expression.	It evaluates the carriers returned by the constructor function and returns the type information.

**Table 3** – Type Carriers for expressions

## 4.4 Binding Type Carriers to Symbols

### 4.4.1 Symbols

IntelliSense needs to keep track of all the symbols in the source code to be able to provide information about them. Therefore the first step of the semantic analysis is to traverse the AST and spot nodes that they declare variables. Such nodes are:

- Function Declarations
- Variable Declarations
- Class Declarations
- Property Declarations
- Constructors
- Method Declarations
- Attribute Accessors
- Function Expressions
- Class Expressions

Like a compiler, for each declaration node met, a symbol object is created and stored in a symbol table. A symbol table is created for each scope in the source code and it is stored to the node that introduces the scope. Each symbol is stored in the symbol table of its corresponding scope. The structure of a symbol is shown in the following table.

Symbol Property	Description
Name	The name of the symbol
Declaration	A reference to the AST node that the symbol is declared
Type Binders	A list of all the Type Binders of the symbol

**Table 4** – Symbol specification

#### 4.4.2 Type Binders

Given that the type of a symbol may change on different parts of a program, IntelliSense should be able to answer questions like “What is the type of  $x$  in line 5”. Thus, a mechanism to tell a symbol type on a specific position in the program is required. For this purpose Type Binders are used. A Type Binder simply binds a symbol and a Type Carrier together. It is created and stored on every AST node that may change the type of a symbol. The structure of a Type Binder is shown in the following table.

Type Binder Property	Description
Symbol	The symbol that is referring
Type Carrier	The Type Carrier for that symbol

**Table 5** – Type Binder specification

Type Binders are context sensitive because they are stored on the AST. As an example, when an assignment  $x = 2$  is met, a new Type Binder for  $x$  is created and is stored on the Assignment node. The existence of the Type Binder on that Assignment node, signals that after this node, the type of  $x$  could be resolved by evaluating the Type Carrier which is stored on the binder.

#### 4.5 Type Computation

In the previous section all the basic structures of the system have been described. This section presents the usage of them, and the algorithms used to actually compute type information for a symbol. As said earlier, our system is able to provide different type information for a symbol in different locations of the program. Also, our worktable is the program’s AST, which is extended to store symbols and binders. Thus, all the algorithms are traversing the AST to find the information they need from it.

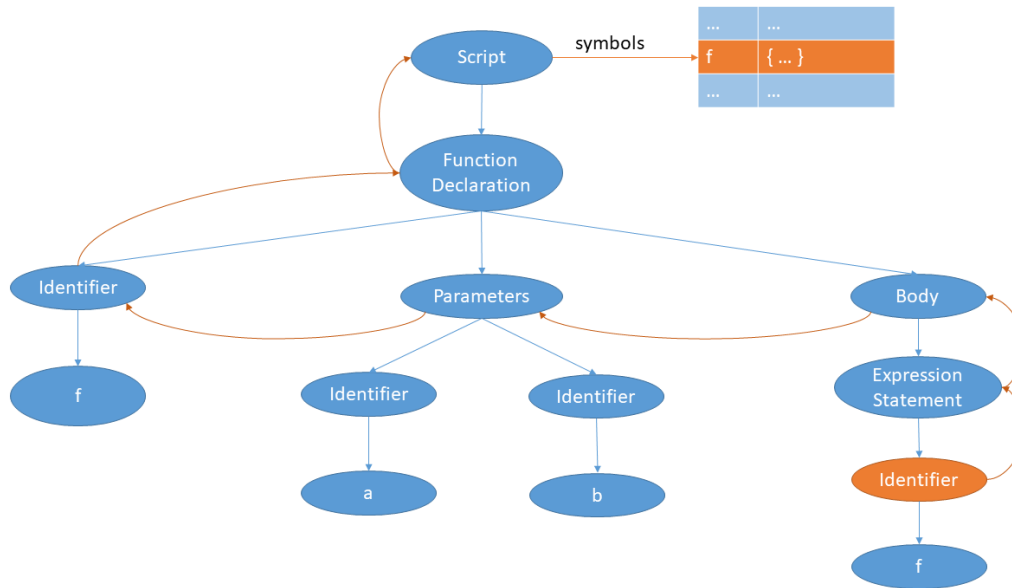
### 4.5.1 Symbol Look Up

To gather type information for a symbol the first step is to identify that symbol. The following algorithm shows how a symbol is retrieved from the AST.

```
function lookUp(node, name) {
  if(!node) { return; }
  const symbol = node.symbols.get(name);
  if(symbol) {
    return symbol;
  } else {
    const previousNode = getPreviousNode(node);
    return lookUp(previousNode, name);
  }
}

function getPreviousNode(node) {
  const leftSibling = findLeftSiblingWithoutScope(node);
  if(leftSibling) {
    return leftSibling;
  } else {
    return node.parent;
  }
}
```

If the left sibling of the current node is a node that introduces a new scope (e.g. Block node) it should be ignored. In that way we avoid finding symbols that are not visible in the current context.



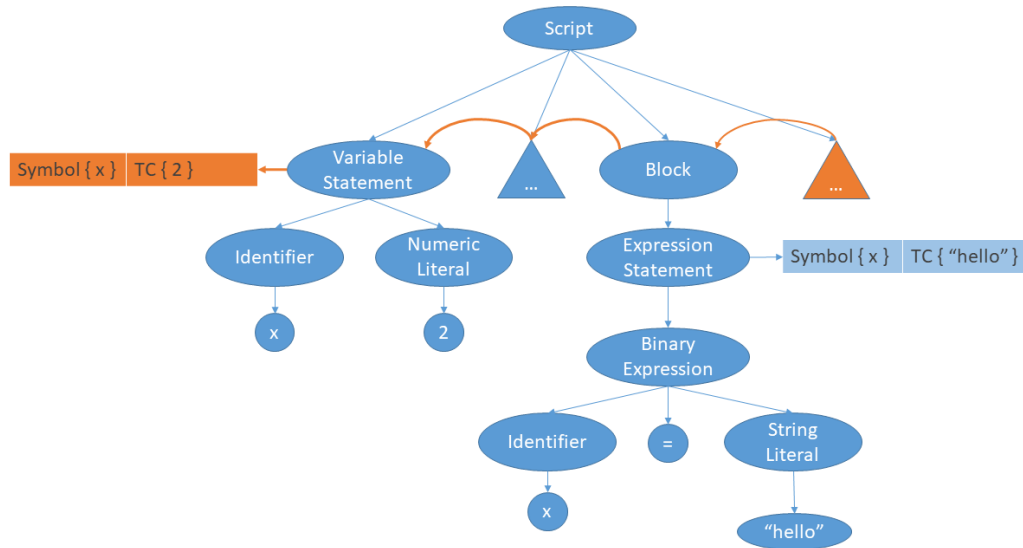
**Figure 12** – Symbol Look up Visualization

### 4.5.2 Determination of Active Type Binders

In a similar manner, the active type binders for a symbol are found. The AST is traversed and if a node contains a binder for that symbol, that binder is the active one. But some modifications are needed for this algorithm to work as expected. Consider the following case:

```
let x = 2;
// ...
{
  x = 'hello';
}
// ...
```

After the execution of this code, the type of `x` is string and its value is 'hello'. During our analysis two binders for `x` are generated in the two nodes that change its value. The second one is generated and stored in the assignment node which is descendant of the block node. Applying the previous algorithm to find the active binders of symbol `x` starting from an AST node subsequent to the code above, we would never find the correct binder because the children nodes of the block are never searched. Instead, the type binder in the declaration of `x` would be found.



**Figure 13** – Searching the active Type Binder of x

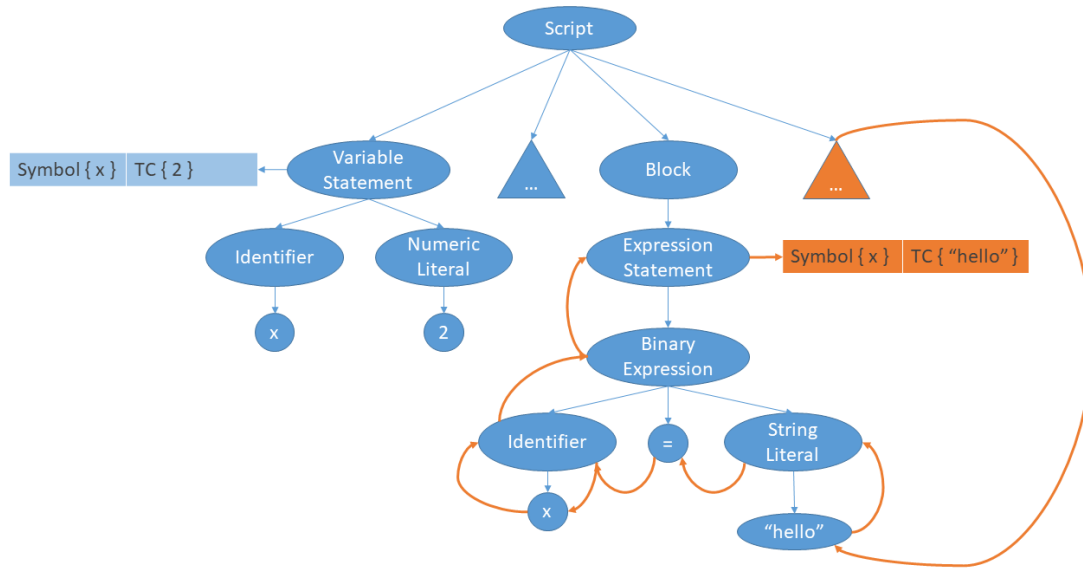
Therefore, the algorithm is modified to also search in the children nodes of the previous node, in cases where the previous node is the left sibling of the current node and its node type is Block. In this way any binders that live inside a block are exposed to the nodes that are subsequent to the block node.

```
function findActiveTypeBinder(node, symbol) {
  if(!node) { return; }
  const binder = node.binders.get(symbol);
  if(binder) { return binder; }
  const leftSibling = findLeftSibling(node);
  if(leftSibling) {
    return findActiveTypeBinderInLeftSibling(leftSibling, symbol);
  } else {
    return findActiveTypeBinder(node.parent, symbol);
  }
}

function findActiveTypeBinderInLeftSibling(node, symbol) {
  if(isBlock(node)) {
    node = findRightMostDescendant(node);
  }
  return findActiveTypeBinder(node, symbol);
}

function findRightMostDescendant(node) {
  if(!node.children.length) { return node; }
  return findRightMostDescendant(node.lastChild);
}
```

};



**Figure 14** – Searching the active type binder of x inside a block

But not only Block nodes are affecting this algorithm, there are also other kind of nodes that we need to consider to form the final algorithm. So the presented algorithm is not on its final form, other cases like this will be presented throughout this paper (e.g. calls, if statements, for statements, etc.) and it will be adjusted to solve all the problems that each case will generate. So this algorithm is just the base that we will work on.

Of course traversing all these nodes has an impact on performance and the IntelliSense is required to execute its operations quickly. Hence, we skip the traverse of nodes that it is certain that they cannot contain binders. Most of terminal nodes are skipped and the performance of this algorithm is boosted and it can run several times faster. Moreover, during the analysis we store the symbols for which binders exist on their scope nodes and when we search for active binders and a block is met we can quickly decide if we are going to search inside.

### 4.5.3 Evaluation of Type Carriers

To compute Type Information for a variable, given the symbol object and its active binder, the only thing left to do is to evaluate the Type Carrier that is stored on the



active binder. It is important to note that the evaluation of the Type Carriers is not performed during the analysis of the code. It is performed on demand when the IntelliSense is asked to provide information for a symbol. In this way, the analysis is more efficient because it only creates the instructions to get the Type Information and the evaluation of these instructions is postponed until it is necessary.

Additionally, in this way Type Carriers act as a graph that shows the path of how the Type Information is computed. Therefore, our system could be able to answer questions like “why the type of `x` is this?” and allow users to trace the respective path. But there is no support to do this using the LSP. This could be achieved by violating the LSP and sending custom messages between the host and the server, but this solution is not portable to other development tools.

The evaluation of Type Carriers is similar to the way an Interpreter would evaluate the AST. A dispatcher is used to call the right function depending on the kind of the carrier. The different types of Carriers and the steps to evaluate them have been presented in the beginning of this chapter. Thus, here are presented the general evaluate function and the evaluate functions for a constant, and a variable Type Carrier. The functions for the other kinds of Type Carriers are similar to these.

```
function evaluate(carrier) {
    const evaluateFunction = evaluateFunctions[carrier.kind];
    return evaluateFunction(carrier);
};

evaluateFunctions[TypeCarrier.Kind.Constant] = function(carrier) {
    return carrier.typeInfo;
};

evaluateFunctions[TypeCarrier.Kind.Variable] = function(carrier) {
    const binder = findActiveTypeBinder(carrier.node, carrier.symbol);
    return evaluate(binder.carrier);
};

// ...
```

## 5. Handling Variables and Expressions

### 5.1 *Declarations*

In JavaScript, since ECMAScript 2016 – ES6 specification, there are three keywords to declare a variable, each one providing different semantics. The first one is the `var` keyword which was in the specification before ES6. The two new keywords are the `let` and the `const` keywords. The problem with the `var` keyword is that it is not block scoped, but it is function / program scoped. Declaring a variable using the `var` keyword inside a block, is going to declare the variable as a global variable, unless a function declaration is mediated.

```
// ...
{
    var x = 2;
}
// ...
console.log(x); // 2
```

This was not a problem in JavaScript before ES6 since there was not a way to include / import files and all the source code was typically in a single file. But ES6 also brought the `import` and `export` keywords to support modules. Thus, the language should provide a way to declare block scoped variables. The two new keywords, `let` and `const`, were added to achieve this. In this way each module can have its own ‘private’ variables and export only the ones that are necessary to other modules.

Another thing that should be considered is hoisting. In JavaScript, a variable can be declared after it has been used. This is equivalent to moving the declarations to the top of their scope. But only the declarations are hoisted, not the initialization.

In declarations that use the `var` keyword, the variable is actually declared from the beginning of the scope and is initialized as `undefined`. The following example may clear the things up a bit.

```
console.log(x) // undefined
```

```
var x = 5;
```

The previous program is correct because of hoisting. The equivalent of this program after hoisting would be the following:

```
var x;  
console.log(x);  
x = 5;
```

In declarations using the `let` or `const` keywords, the declaration is hoisted to the beginning of the scope but it remains uninitialized until the actual declaration. That means that the name is reserved but it cannot be used until its actual declaration.

```
console.log(x) // Reference Error  
let x;  
console.log(x) // undefined
```

During our analysis for variable declarations, as described in the previous section, we need to:

- Create a Symbol for the declared variable
- Store the Symbol in the appropriate symbol table (Find its scope)
- Create a Type Binder for that Symbol.
- Store the Type Binder in the appropriate AST node.

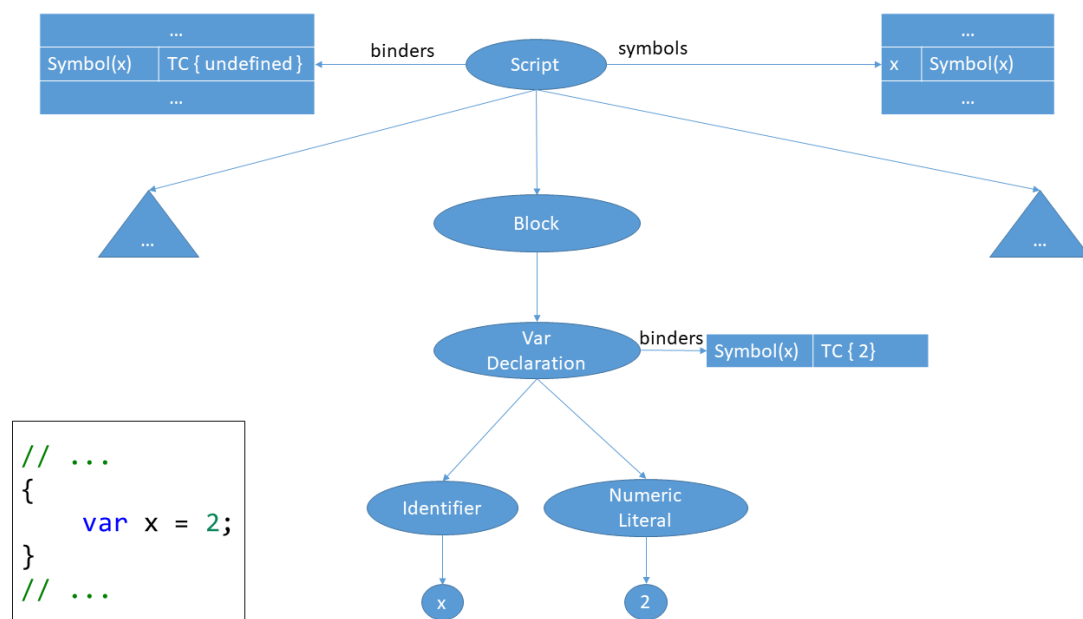
### 5.1.1 Declarations via var Keyword

During our analysis, when a declaration that uses the `var` keyword is met, a symbol for that variable is stored in the symbol table of the closest Function / Script ancestor. Also a Type Binder with the Type Carrier of the initialization node is created for that symbol and it is stored in the declaration node. In cases that initialization node is absent (e.g. `'var x;'`), a Type Binder is implicitly stored in the declaration node that binds symbol `x` with an undefined constant Type Carrier.

That should be enough, to be able to look up for this variable, and compute type information for it after its declaration. But as shown in the beginning of this chapter, the declaration of a variable with the use of the `var` keyword, actually initializes the

variable with undefined in the beginning of its scope. Therefore, a second Type Binder with a constant Type Carrier of undefined is created for that symbol and it is stored in the closest script, or function body ancestor node.

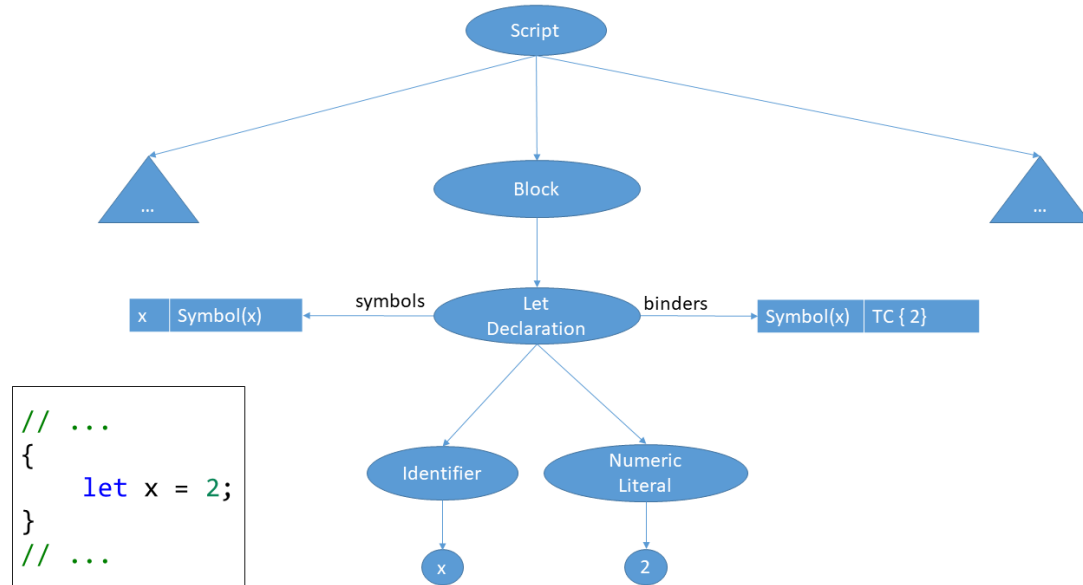
The following figure illustrates the AST after the analysis of the variable declaration node of `x`. It is clear now that a look up for `x` would find the symbol even when starting from nodes prior to the declaration. Also, the active binder of `x` after the declaration would be the binder generated from the declaration, and prior to the declaration would be the extra ‘undefined’ binder that was stored in its scope.



**Figure 15** – AST for var declaration (after the analysis)

### 5.1.2 Declarations via let Keyword

Declarations that use the `let` keyword are handled in a similar way. Actually they are even simpler to analyze because the variable remains uninitialized until the declaration node. During the analysis, when a declaration node that uses the `let` keyword is met, a new symbol is stored in the closest block, or script ancestor node. Also a Type Binder is stored in the declaration node that binds the symbol with the Type Carrier of the initialization node.



**Figure 16** – AST for let declaration (after the analysis)

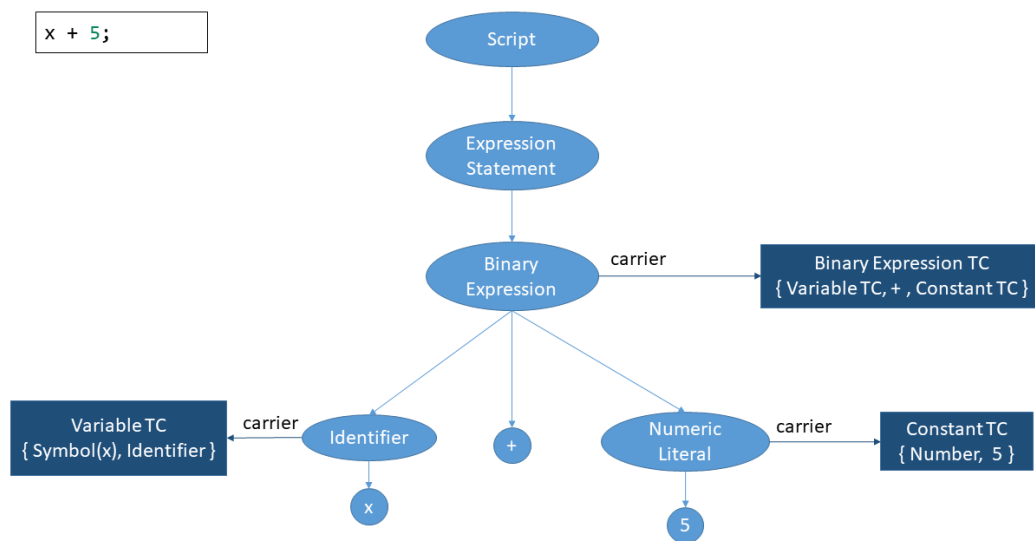
The differences between the analysis of var and let declarations are that on the latter we are storing the symbol in the closest block and not on the closest function body, although this block could be the closest function body, and that we don't have to add an extra Type Binder for this symbol before its declaration node, since the symbol cannot be used before its initialization.

### 5.1.3 Declarations via const Keyword

Actually there is no distinction between the declarations that use the let keyword and the declarations that use the const keyword. They are both handled the same way, the symbol is stored in the symbol table of the closest block / script ancestor node and the Type Binder for that symbol is stored in its declaration node. The only difference is that the initialization node is mandatory to exist on const declarations. But this is already handled by the TypeScript compiler.

## 5.2 Expressions

As described in the previous chapter each expression generates a new Type Carrier. During the analysis for every expression met, we create the appropriate Type Carrier and we store it on the expression node. Outer expressions may refer to inner expression's Type Carriers to generate their carriers. The following figure presents the AST of `x + 5;` along with its Type Carriers. It is important to note that the carrier of the Binary Expression references the carriers of its inner nodes.



**Figure 17** – Type Carriers in expression nodes

## 6. Handling Assignments

### 6.1 Variables

Handling variable assignments is pretty straightforward. During the analysis when an assignment is met, we just need to bind the Type Carrier of the right part of the assignment to the symbol of the left part. To achieve this we just create a Type Binder and store it in the assignment node.

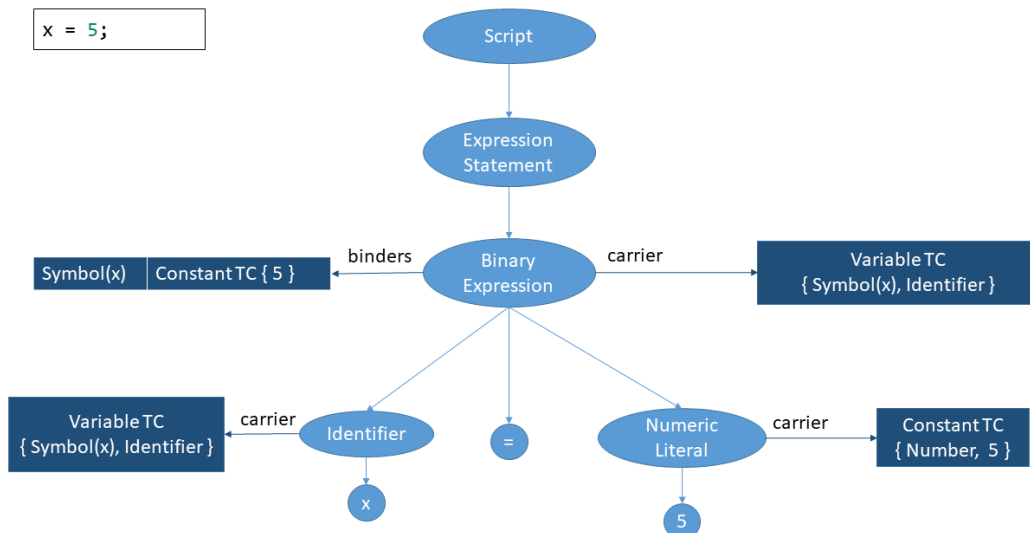


Figure 18 – AST after the analysis of Assignment Expression

### 6.2 Object Fields

Assigning a value to a field of an object is similar to variables, we just create the appropriate Type Binder and store it on the assignment node. But there are some tricky parts here that we would like to show. In weakly typed languages, it is possible to add or remove properties from an object at runtime. In JavaScript when a value is assigned to a property of an object, and this property does not exist, it is just created during the assignment. In the following example, an object is created at line 1 and it is assigned to

a variable `x`. The code in the second line, tries to assign the value 2 to `x.a`. Since the object assigned to `x` is empty, it does not have any properties, a new property for this object will be created.

```
let x = {};  
x.a = 2;  
console.log(x.a); // 2
```

During the analysis, the difference between the assignment to an object field and to a variable is that in the latter we already know the symbol of the variable. Instead when an assignment to a property is met, we must resolve the symbol of that property. This implies that we need, to gather all the properties of the object, decide if it already has that property, and if it is absent we need to extend its properties dynamically to contain the new property.

As we described in section 4.2, the Type Information is used to represent an object. Also, each object needs to have a database of its properties. Therefore our object representation keeps a symbol table for its properties. When a new property is added to the object, a new symbol is created and it is stored in its object's symbol table.

Unfortunately, in order to resolve the object properties and decide if the property already exists or it is a new one, we must evaluate the carrier of the left part of the property access expression. Evaluating the Type Carrier gives us its Type Information, which in case of objects it stores the symbol table of its properties. Since we have the symbol table, we can resolve the symbol of that property if exists, or extend it with a new symbol.

Other than resolving the symbol of the object field, every other part works as explained in the assignment of variables. Type Binders are created and stored on the AST to bind the symbol of each property to its corresponding Type Carriers.



## 7. Handling Function Definitions

A function definition in JavaScript is like a variable definition. A new symbol is generated and the developer is free to change its value later in the program. Functions are also hoisted to the closest function / program scope. Thus, the following program is correct.

```
foo(); // 2

function foo() {
    console.log(2);
}
```

During the analysis, when a function definition is met, we create a new symbol for it, and a binders for the symbol which they are stored in the closest function / program scope. Moreover, the function itself is analyzed since IntelliSense is required to provide all the language services when the function is edited.

### 7.1.1 Parameter Type Prediction / Speculation

During the editing of a function, its parameters have no values assigned to them, so the IntelliSense has no type information for them. Instead, a value is assigned to each parameter during the call of that function. However, during the analysis of the function body all the usages of these symbols are available. Therefore, we try to predict their plausible types by analyzing how they are used inside the function. This type information is used when the developer inspects a function parameter during the editing of the function and can also be composed to show the function signature when the developer calls that function.

Although JavaScript has *type coercion* which solves most of the representation mismatches inside the expressions with silent type conversions. Therefore, there are cases in which our type predictions for the function parameters are not correct, since

the actual arguments could be of any type and there would be no runtime error but still there are too close to reality.

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

Consider the above example. The developer could use the ‘add’ function and pass as arguments any expression and JavaScript would still execute the addition between them and return valid results. Even though, the type of those results would always be either number, or string depending on the type of the operands. Despite that we don’t know the actual type of ‘a’ and ‘b’ we are still able to tell that during the execution of this function they will be converted to either number or string if they are not already.

In order to predict the type of a parameter, we also need to consider the indirect usages of the parameter. In the following example, ‘a’ could still be converted to number or string even if it is not used directly as operand in the addition. Thus, when we meet the addition during the analysis we need to partially evaluate its operands to decide whether or not they are references to a parameter.

```
function add(a, b) {  
    const c = a;  
    return c + b;  
}
```

The concept described below could be used for most of the expressions in the source code. Even though that these predictions are not always correct, we believe that this information is still really useful to the developers when they edit the source code.

## **7.2 Plausible Return Types**

It is also possible to apply type inference to find the plausible return types of a function. The plausible return types of a function can be used to compose its signature. When a return statement is met during the analysis, its next statements, or in this case its right siblings, are marked as unreachable. If the current return statement is reachable it is stored in the function definition node. After the analysis, the function has stored in it all the return statements that it is possible to be executed when it is called. From the

return statements we have access to the returned expressions which store their type carriers. When they are needed later, we can compute the plausible return types by evaluating these type carriers.

Another thing to consider to form the plausible return types, is that in JavaScript when a function does not explicitly returns a value, it implicitly returns undefined. Therefore, the control flow needs to be analyzed to determine if there are any paths in the body of the function that do not return a value. In case such paths exist, we need to include a constant Type Carrier for the undefined value to the plausible return types of the function.

We are able to tell whether or not a function returns a in every control flow path, by applying the following logic:

- A set of statements returns on all control paths if at least one of its statements returns on all control paths.
- A return statement returns on all control paths.
- An if statement returns on all control paths if both its then and else statements return on all control paths.
- A switch statement returns on all control paths if all of its clauses return on all control paths or any of them but the last falls-through.

## 8. Function Calls (Call Sites)

Function calls provide various information about the program. First of all, in cases that the result of the function is stored, we care about the plausible return types of the called function to provide more precise results. Additionally, we need to consider the side-effects of a call. The body of the called function may contain assignments to symbols defined out of the function and thus mutate their type. Finally, the plausible types of the actual arguments could be used to provide more information on future calls of the same function during the editing.

### 8.1 Side Effects

When a function is invoked, it may change the type of variables declared out of it. Such functions are called *non-pure*. In the following example the `f` has an assignment to `x` in its body. Before the call of `f`, the type of `x` is undefined. After the call of `f`, `x` is a string since the body of `f` mutates it. Therefore `f` is a non-pure function.

```
let x;  
function f() {  
    x = 'a';  
}  
f();
```

Also a function may mutate, or add new object fields in global variables or parameters. In case that it mutates an object field a new binder is created for it and in case of new field, we also create a symbol for it. Considering all these cases, during the search of the active binders of a symbol, we need to also search inside the body of any function that is called when the function mutates that symbol.

Thus during the analysis, when a call is met, we evaluate the called expression to resolve its AST node and its subtree is replicated to be able to store the binders of the affected symbols. Its body is analyzed and the new binders are added to the AST. When we search for active type binders and a call is reached, we also search inside the

replicated function body. To achieve this we modify the way we search the active type binders when the next node is the parent node.

```
function findActiveInParent(node, symbol, startNode, stopNode) {
    if(node === stopNode) { return; }
    // ...
    if(isCallLikeExpression(node) && node.callee) {
        return findActiveInCallee(node, symbol, startNode, stopNode);
    }
    // ...
    return findActive(node, symbol, startNode, stopNode);
};
```

## 8.2 *Function Dependencies*

### 8.2.1 Parameters

As described in Section 7, during the analysis of function declarations, we have no type information for its parameters. Instead their type information is available during the call of a function where the actual arguments are passed in. Thus, when a call is met during the analysis, we generate new type binders for the parameters of the called function and we store them in the call node. In cases that arguments are omitted, we implicitly add an undefined binder for each omitted argument.

When the type information of a parameter is requested, its type binders can be found in the appropriate call node. Thus, function nodes are extended to also store their call sites. Also, to be able to reach their binders, during the search when we reach a function body as a parent we need to continue the search in the appropriate call node.

```
function findActiveInParent(node, symbol, startNode, stopNode) {
    if(node === stopNode) { return; }
    // ...
    if(isFunctionLike(node) && node.call) {
        return findActiveInCallSite(node, symbol, startNode, stopNode);
    }
    // ...
    return findActive(node, symbol, startNode, stopNode);
};
```

### 8.2.2 Free Variables

Functions may have access to variables that are neither locally declared nor passed as parameters. Such variables are called free variable. Free variables can be mutated by code written between different calls of a function. So, free variables may have different values and thus types between different calls of a function. In the following example, during the first call of `f`, symbol `x` is type of number, but during the second call of `f`, `x` is type of string.

```
let x;  
function f() {  
    console.log(x);  
}  
x = 10;  
f();    // 10  
x = 'a';  
f();    // 'a'
```

Therefore, when the type information of a symbol inside the body of a function is requested and there is no type information in the function, we need to continue the search in its call node.

Although this information is used by our system to gather more precise results for other symbols, it could also be used during the inspection of a function. When the users aim to inspect during the editing the types of free variables, they could be presented with choices of the call sites of the function if such exist. Also, when irrespective of the call sites, the symbol types remain the same, there is no need to actually request the call site selection.

## 9. Branches and Loops

Branches and loops introduce a major problem. How could we resolve the Type Information of a symbol that may have different outcomes in a conditional statement? We could probably try to evaluate the Type Carrier condition and search for Type Binders in the block that is going to be executed. But in real life it is almost impossible to know what the value of a condition is going to be. The condition could be an expression that has a random factor in it, or its origin could be from somewhere around the network. Even if we could actually execute the program, the results could be different on every run. Thus we compose Type Carriers as needed to identify more composite Type Carrier sets and provide all the plausible outcomes.

### 9.1 *If-Else Statement*

Since JavaScript is an untyped language, a variable could hold values of different type on different paths of execution. After the execution of the following source code, given that we don't know anything about the condition of the if statement, the x could be a number or a string. Therefore that's exactly what we are aiming to do, we are going to provide all the plausible types of a variable. We are going to retrofit our system to provide that information to the developer.

```
let x;  
// ...  
if(e) {  
    x = 2;  
} else {  
    x = 'hello';  
}  
// ...
```

The system in its current state is providing type information for a symbol based on the last statement that changes its value. Each possible change of a value is marked on the

AST with a Type Binder. Then depending on the position on the AST we search for the closest Type Binder which is called active.

In the previous example, there are two Type Binders for `x` inside the `if` statement, one inside its `then` statement and one inside its `else` statement. Thus, in order to deliver to the users multiple plausible types for a symbol, we just need to have more than one active Type Binder. Each active Type Binder gives a plausible type to a symbol.

So, the algorithm to find the active Type Binder presented in section 4.5.2 needs to be modified to return multiple Type Binders. When an `if` statement is met, it should return all the individual active binders found in each one of its possible paths. Also in some cases, an `if` statement is not changing the value of a variable definitely and we also need to include the active binders in its conditions and in statements prior to the `if` statement if there are no binders in the conditions.

```
function findActive(node, symbol, startNode, stopNode) {
  if(!node) { return; }
  const binder = node.getBinder(symbol);
  if(binder) { return [ binder ]; }
  if(node == stopNode) { return ; }
  const leftSibling = findLeftSibling(node);
  if(leftSibling) {
    return findActiveInLeftSibling(leftSibling, symbol, startNode,
      stopNode);
  } else if(node.parent) {
    return findActiveInParent(node.parent, symbol, startNode,
      stopNode);
  }
}

function findActiveInLeftSibling(
  node,
  symbol,
  startNode,
  stopNode
) {
  if(node == stopNode) { return; }
  return findActiveInStatement(node, symbol, startNode, stopNode) ||
    findActive(node, symbol, startNode, stopNode);
}

function findActiveInStatement(node, symbol, startNode) {
  switch(node.kind) {
```



```

        case Block:
            return findActiveInBlock(node, symbol, startNode);
        case IfStatement:
            return findActiveInIfStatement(node, symbol, startNode);
    }
}

```

The function `findActiveInStatement` searches for binders in the statements of a block and stops when it reaches the block. In this way, when we search for active binders in if statements, we avoid searching outside of then-else blocks multiple times and end up with duplicities.

```

function findActiveInBlock(node, symbol, startNode) {
    const lastStatement = findLastStatement(node);
    if(!lastStatement) { return ; }
    return findActiveInStatement(lastStatement, symbol,
        startNode, node);
}

function findActiveInIfStatement(node, symbol, startNode) {
    const statements = findThenElseStatements(node);
    const binders = [];
    const conditionsToSearch = new Set();
    for(const s of statements) {
        const sBinders = findActiveInStatement(s, symbol, startNode);
        if(sBinders) {
            binders.push(sBinders);
        } else {
            conditionsToSearch.add(s.parent.expression);
        }
    }
    if(!hasElse(node)) {
        conditionsToSearch.add(statements.last.parent.expression);
    }
    if(!conditionsToSearch.isEmpty) {
        binders.push(
            findActiveOutOfIfStatement(conditions, symbol, startNode)
        );
    }
    return binders;
}

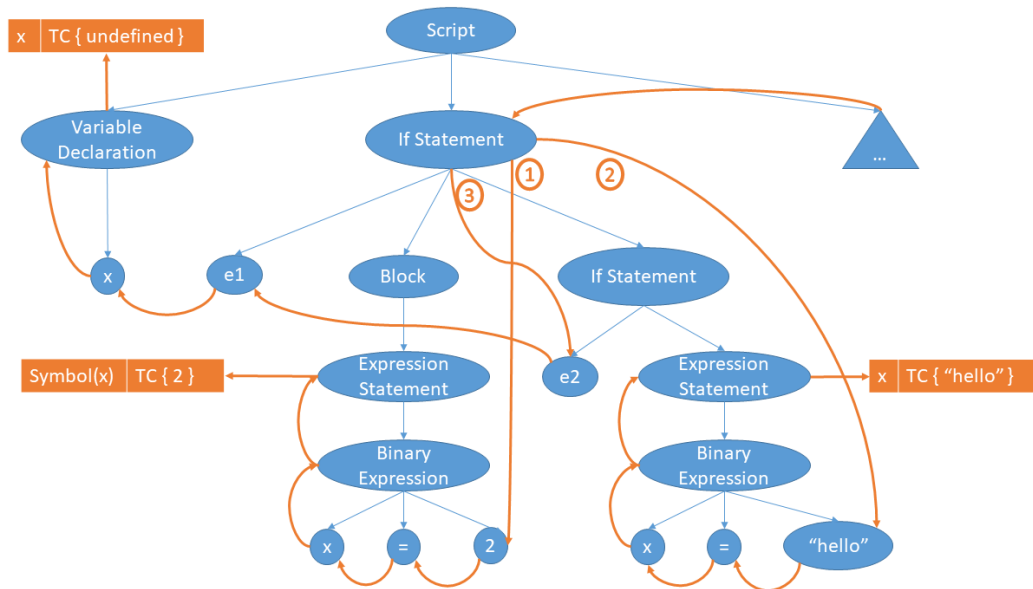
function findActiveOutOfIfStatement(conditions, symbol, startNode) {
    const topLevelIfStmt = findTopLevelIfStatement(conditions);
    const binders = findActiveInIfConditions(

```

```

        conditions, symbol, startNode
    );
    if(conditions.length) {
        binders.push(
            findActive(ifStatement, conditions, startNode)
        );
    }
    return binders;
}

```



**Figure 19** – Search the Active Type Binders of x Inside an If Statement

As we described previously, there are cases that if statements do not change the type of a symbol definitely and we need to identify those cases and include the active binders prior to the if statement. The first case is when the if statement does not have an else statement:

```

x = true;

if(e1) {
    x = 2;
} else if(e2) {
    x = '';
}

```

In this case, it is possible that both e1 and e2 evaluate to false and thus the type of x after the if statement remains boolean. The second case is when the if statement contains

at least one path that does not change the type of the symbol that we are searching for its active binders. This case also covers the case in which the if statement does not change the type of the symbol at all:

```
x = 1;

if(e) {
    x = ''
} else {
    // ...
}
```

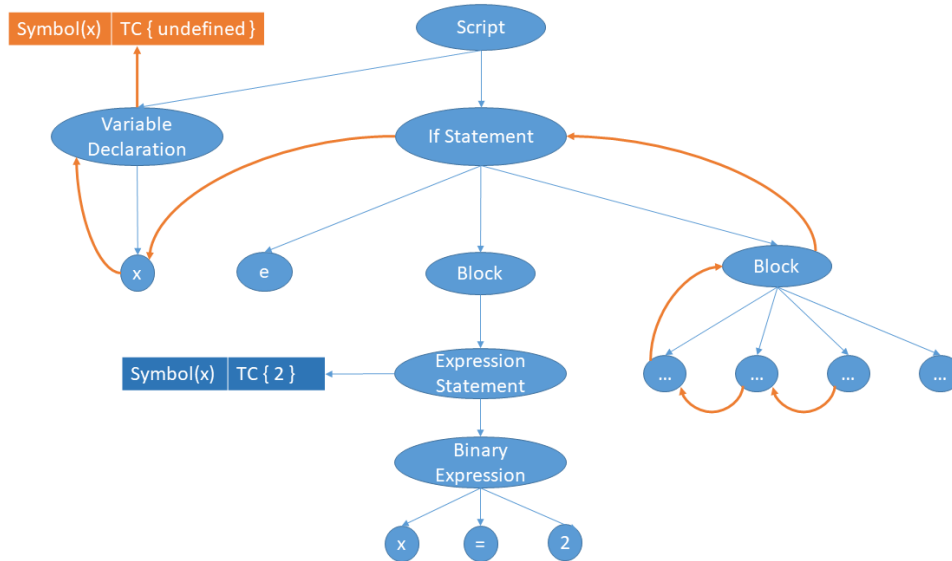
Also, we need to do a last modification to the algorithm to work properly. When we search for an active binder starting inside a then-else statement and we reach its parent if statement, we need to bypass all the other then-else statements and continue the search out of the top-level if statement. In the previous example if the developer typed `x` inside the else statement, we want to tell them that the type of `x` is number and not string.

```
function findActiveInParent(node, symbol, startNode, stopNode) {

    if(node === stopNode) { return; }

    if(node && isIfStatement(node)) {
        return findActiveOutOfIfStatement(
            [ node.expression ],
            symbol,
            startNode
        );
    }

    return findActive(node, symbol, startNode, stopNode);
};
```



**Figure 20** - Search active binders of  $x$  starting inside a then/else statement

## 9.2 Switch Statement

Switch statements are handled in a similar way to if statements. When we search for the active type binders of a symbol and a switch statement is met, we collect all the active binders found in its possible paths. But in some occasions we are not able to provide so accurate results as in if statements. The problem here is that switch cases might fall-through and a simple break detection in its case clauses won't do the trick, because the break statements could be inside if statements and the things get much more complicated.

Thus, we simply assume that there is no fall-through in the switch-cases. We search for active binders in each case clause individually and we always include the active binders found prior to the Switch Statement. In this way the results may not be so accurate, but the resulting type set is always a superset of the actual type set and we are not missing any possible type.

```
let x = 2;

switch(e) {
  case e1:
    x = true;
```

```

    default:
        x = '';
        break;
}

```

After the execution of the previous program, the type of `x` would always be number, because the `e1` case clause falls-through. Our system will not detect this but it can tell that the type of `x` could be undefined, or boolean, or number.

Since we assume that there is no fall-through, when we search for active binders of a symbol starting inside a case clause and we reach the case clause, which means that in this path there is no assignments to that symbol, we just continue the search out of the Switch Statement. Of course, the way we choose to handle this has an important drawback which may give misleading results in switch-cases that they indeed fall-through. In the following program, the type of `x` inside the default clause would be boolean but our system is not able to detect that.

```

let x = 2;

switch(e) {
    case e1:
        x = true;
    default:
        // Search for Active Binders of 'x' starting here
}

```

### 9.3 For Statement

When we search for the active binders of a symbol and a for statement is met, we collect the active binders inside its statement and we also include the active binders in its increment node and the active binders in its condition and all the previous statements. After the execution of the previous program, the type of `x` is just number or string.

```

let x = 2;

for(;; e; ) {
    x = "hello";
}

```

```

function findActiveInStatement(node, symbol, startNode) {
    switch(node.kind) {
        case Block:
            return findActiveInBlock(node, symbol, startNode);
        // ...
        case ForStatement:
            return findActiveInForStatement(node, symbol, startNode);
    }
}

function findActiveInForStatement(node, symbol, startNode) {
    const binders = findActiveInStatement(
        node.statement, symbol, startNode
    );
    binders.push(findActiveOutOfForStatement(node, symbol, startNode));
    return binders;
}

function findActiveOutOfForStatement(node, symbol, startNode) {
    const binders = [];
    binders.push(
        findActiveInIncrement(node.increment, symbol, startNode)
    );
    binders.push(
        findActiveInForCondition(node.condition, symbol, startNode)
    );
    return binders;
}

```

Also we need to consider the case in which the search started inside a for statement. In order to achieve this, we need to modify `findActiveInParent` to search out of the for statement when the top-level node of the statement is reached.

## 9.4 While Statement

Handling while statements is similar to how we handle for statements. In fact it is simpler since while statements do not have an increment node. Also JavaScript supports three different kinds of for loops, the classic for loop, and the for/in and for/of loops to iterate objects and arrays respectively. All these kinds of loops are handled exactly the same way.

## 10. Editing Automations

### 10.1 Code Completion

Code completion or content assist is helping the users to write code faster, by suggesting them possible completions as they type. It also helps programmers to remember API names that they would have to search in documentations instead. In Visual Studio Code when a character is typed, the suggestions will pop up in a list. If you continue typing characters, that list is filtered to only include suggestions containing your typed characters.

Depending on the language, it may be needed to trigger the code completion in different cases. When a Language Server or an extension with programmatic language features is created, a list of trigger characters can be defined to trigger the code completion (such as the dot ‘.’ character in many programming languages.).

The first step in almost all editing automations is to find the AST node that corresponds to a specific position. For instance, when the developer hovers over a variable we want to know the corresponding AST node of the text under their pointer. The following algorithm was used for this purpose. Also, sometimes it is useful to implement an extra version of this algorithm that searches for nodes of a specific type (e.g. CallExpression).

```
function findInnermostNode(ast, offset) {
  return ast.forEachChild(function (node) {
    if (node.start <= offset && node.end >= offset) {
      const innermostNode = findInnermostNode(node);
      return innermostNode || node;
    }
  });
}
```

To support code completion, a handler for the LSP Completion event was implemented. When the user types a character, the AST node under the cursor is found. If the type of the node is Identifier, all the visible symbols from that node are collected. The list of the visible symbols is filtered to match the typed text and it is displayed by the client.

Moreover, a handler for the LSP Completion Resolve event was implemented. Every time the user navigates between completion items, a request is sent to the server to resolve additional information for the given completion item. The active binders for that symbol are found and their carriers are evaluated to be able to provide extra information about its possible types.

```
function onCompletion({ document, position, triggerCharacter }) {

  const ast = getAst(document);
  const node = findInnermostNode(ast, position);

  if(!node || !isIdentifier(node) || isDeclarationName(node)) {
    return;
  }

  if(isNameOfPropertyAccessExpression(node)) {
    return computePropertyCompletions(node.parent);
  } else {
    return computeIdentifierCompletions(node);
  }
}

function computeIdentifierCompletions(node) {
  const completions = [];
  const visibleSymbols = findVisibleSymbols(node);
  for(const s of visibleSymbols) {
    const binders = findActiveTypeBinders(node, s);
    const typeInfo = binders.flatMap(b => evaluate(b.carrier));
    completions.push(computeCompletion(s, typeInfo));
  }
  return completions;
}
```



```

65     rectangle.area = () => {
66         const area = width * height;
67         return area;
68     };
69
70     return rect;
71 }
72
73 function create
74     const box =
75     return box;
76 }
77
78 const shape = c
79
80
81 const ShapeType = {
82     Point: 0,
83     Line: 1,

```

Figure 21 – Code Completion

```

13
14 function createDog(name, age, breed) {
15     return {
16         name,
17         age,
18         breed,
19         bark: () => 'Woof!'
20     }
21 }
22
23
24 const goodBoy = createDog('Scooby', 2, 'Great Dane');
25
26 goodBoy.
27
28
29

```

Figure 22 – Code Completion on objects

## 10.2 Parameter Help

Parameter Help, as its name suggests, helps users to fill the arguments in a call. It is usually triggered by some characters as the user types. These trigger characters are language dependent, since they must reflect the syntax of the call. As an example, in JavaScript and in most of the programming languages, the trigger characters are the left parenthesis ‘(’, which indicates that a call has begun, and the comma ‘,’, which indicates that another argument is going to be passed in the call. When the parameter help is triggered, a popover appears that typically displays the signature of the called function, along with extra information such as the documentation comments above its declaration. Like Code Completion, Parameter Help is essential to speed up the development process. It is nearly impossible for a programmer to remember the order

and all the parameters of every function in any API. Parameter Help saves all that time that they would need to search for these information in documentations. In Visual Studio Code, the Parameter Help popover appears when one of the predefined trigger characters is typed. Users can also trigger Parameter Help explicitly with some macros. Additionally, the current argument that the programmer is passing, is underlined in the function signature inside the Parameter Help popover.

In many languages, like C++ or C#, function overloading is supported. Function overloading allows multiple functions to have the same name but different parameters. In such cases, Parameter Help allows the programmers to select which implementation they mean, and the signature in the popover is adjusted to match the selected implementation. In Visual Studio Code, you can navigate between different implementations by pressing the Up or Down arrow while the Parameter Help popover is active.

To support Parameter Help, a handler for the LSP Signature Help event was implemented. When the user types a left parenthesis, or a comma, this event is fired. First of all, based on the offset of the cursor, we search for the inner-most call node that encloses the current offset. Then, we evaluate the Type Carrier of the called expression, and the resulting types are filtered to not contain any non-callable type. Afterwards, the signature of each possible called function is computed and returned to the client. In order to provide more information about each parameter, the induced types of the parameters are used in the signature. Moreover, we compute and show the plausible return types of each callee. Along with the signatures, the serial number of the current argument that the user is filling is computed and returned. Client collects these information and displays the Parameter Help popover.

Moreover, we use Parameter Help to collect information about the called function and solve ambiguities. When the called expression could be more than one function, and the user navigates between the passible functions of the call, he actually can select and lock which of those functions is actually called. This information is saved and updated during the changes of the document and is used to also determine the plausible types of other symbols, e.g. when the result of the function is stored in a variable, to compute its plausible types.

```

1  let foo;
2
3  if(e) {
4      /**
5       * @returns The sum of the arguments.
6       */
7      foo = function(a, b) {
8          return a + b;
9      }
10 } else {
11     foo = function(a, b) {
12         foo(a: number || string, b: number || string):
13         number || string
14     }
15 }
16
17 /**
18  * @returns The sum of the arguments.
19  */
20 foo()

```

Figure 23 – Parameter Help (1/2)

```

1  let foo;
2
3  if(e) {
4      /** Returns the sum of the arguments */
5      foo = function(a, b) {
6          return a + b;
7      }
8  } else {
9      foo = function(a, b) {
10         return a - b;
11     }
12 }
13
14
15 foo(a: number, b: number): number
16
17 foo()
18

```

Figure 24 – Parameter Help (2/2)

## 10.3 Quick Information

This automation allows the user to hover the mouse over various text items to view details about them. They can hover over the name of a variable and if the name is recognized, a quick information tooltip will be displayed.

To support quick information, a handler for the LSP Completion event was implemented. When the user hovers over a text item, the Completion event is fired. First of all, the focused AST node is found. Then, based on the type of the node we provide different information. In this section, we present the computation of quick information for Identifier nodes. Since we have the node under the cursor, and we know that it is an Identifier, we resolve its Symbol. As we described previously, symbols store their binders, so we are showing all the Type Information for that symbol throughout the program. Additionally, we search for its active Type Binders and we mark the appropriate Type Information, to let the developer know which of them are plausible on that specific position.

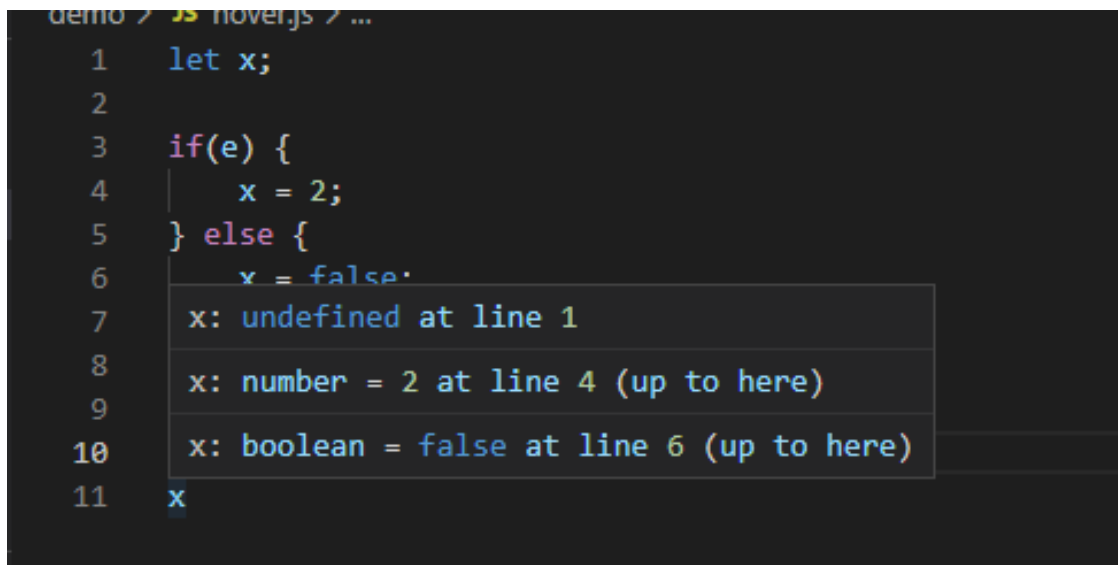
```
function onHover({ document, position }) {
  const ast = getAst(document);
  const node = findInnermostNode(ast, position);
  if(!node) { return noInfo; }
  switch(node.kind) {
    // ...
    case ts.SyntaxKind.Identifier:
      return createIdentifierInfo(ast, node);
    // ...
    default: return noInfo;
  }
}

function createIdentifierInfo(ast, node) {
  const contents = [];
  const symbol = getIdentifierSymbol(node);
  if(!symbol || !symbol.binders.length) {
    return createAnyInfo(node);
  }
  for(const b of symbol.binders) {
    contents.push(createQuickInfo(b));
  }
  return { contents };
}
```

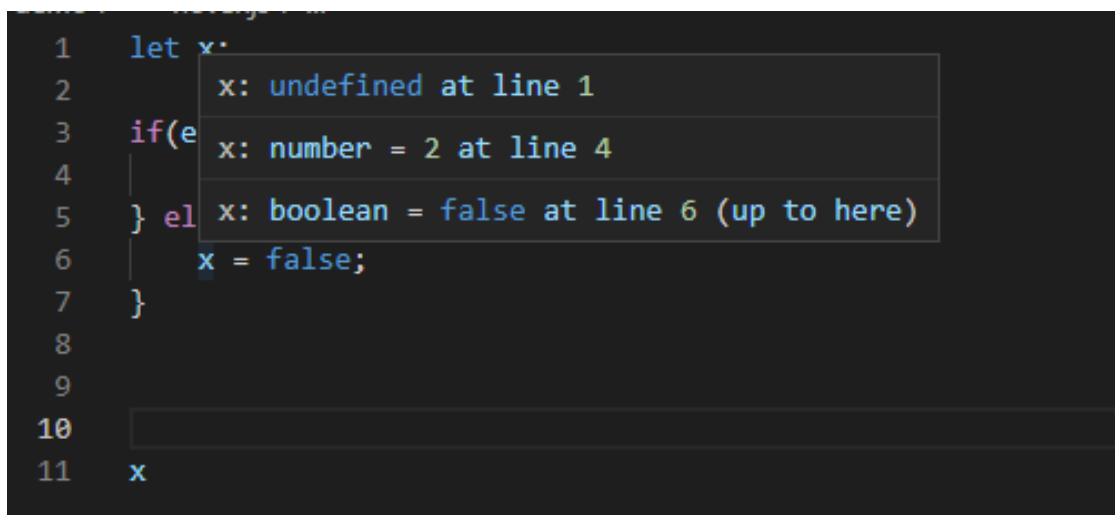
```

function getIdentifierSymbol(node) {
  if(isNameOfPropertyAccessExpression(node)) {
    const propertyName = node.getText();
    const properties = getPropertySymbols(node.parent);
    return properties.find(p => p.name === propertyName);
  } else {
    return lookUp(node, node.text);
  }
}

```



**Figure 25** – Quick Information (Showing all plausible types after an if statement)



**Figure 26** – Quick Information (In Else Statement)

## 10.4 Goto Definition

This editing automation grants the user the ability to find the definition of a symbol quickly. In Visual Studio Code, the user can find a definition by clicking on a symbol while holding Ctrl, or by pressing F12 while hovering over a symbol, or by right-clicking to a symbol and selecting ‘Go to Definition’.

To support Goto Definition, a handler for the LSP definition event was implemented. When a user triggers this event, the first thing to do is to find the AST Node under their cursor. If the type of the node is any other than Identifier, it is ignored. Then, a look up to the AST is performed, to find a symbol named as the text of the Identifier node. Since symbols store their declaration AST nodes, it is trivial to get the file which the symbol was declared in and its text range inside that file. This information is sent to the client, which opens the specified file, scrolls to the specified text range and highlights it.

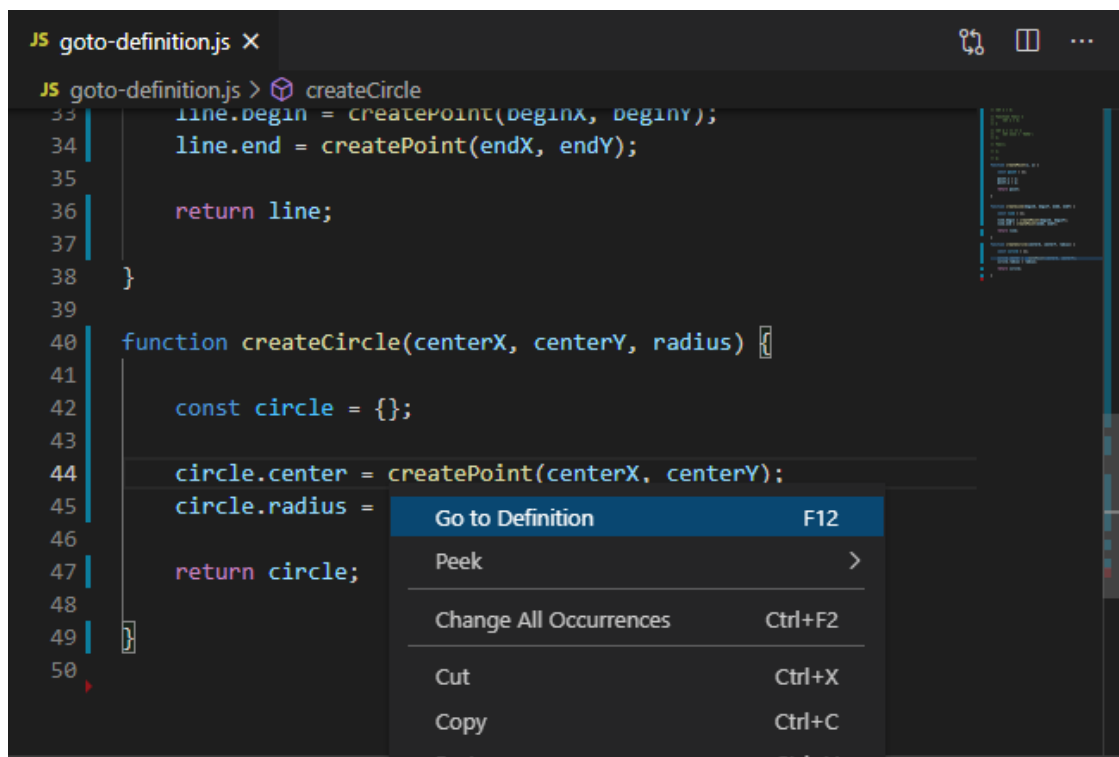


Figure 27 - Go to Definition (Before)

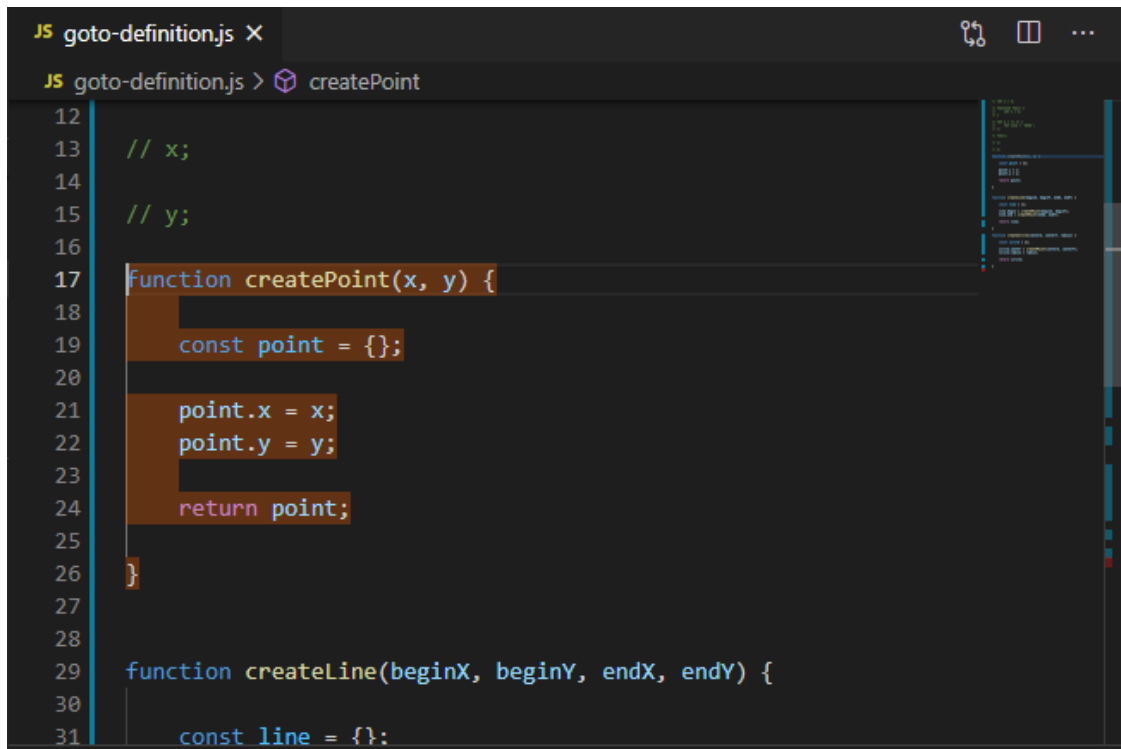
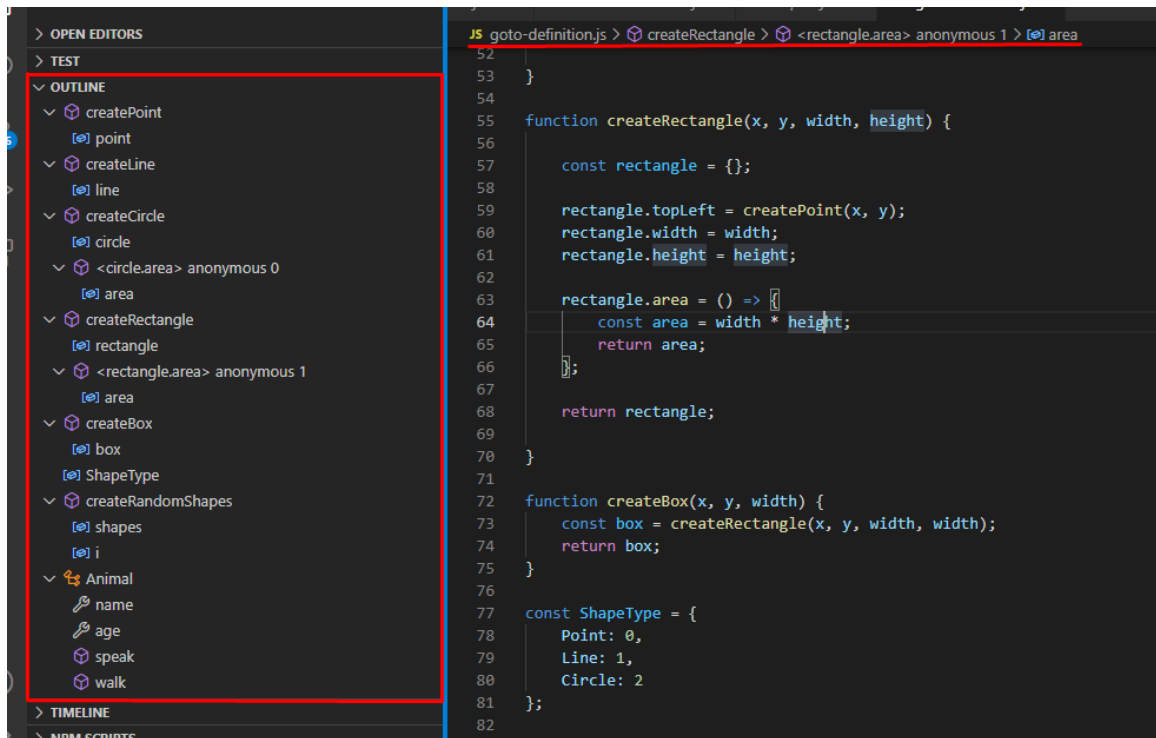


Figure 28 - Go to Definition (After)

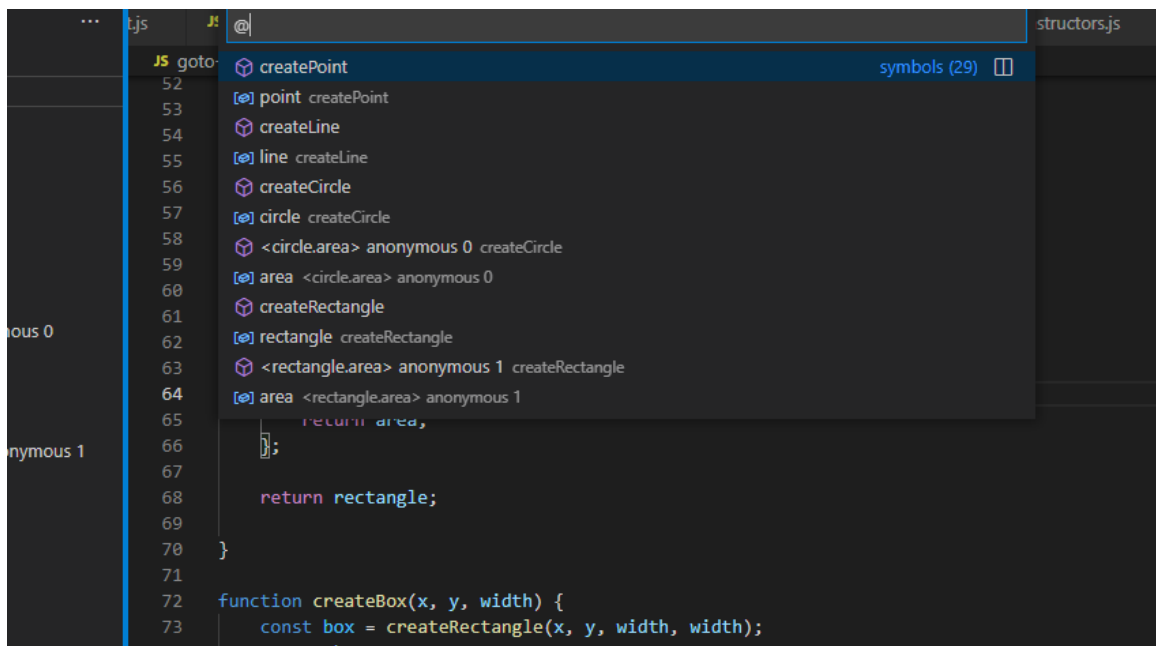
## 10.5 Document Symbols

This IntelliSense feature provides all the symbols in a document, usually in a hierarchy. The user can click on any symbol and navigate to its declaration. It is very useful when programmers want to navigate around a document and understand its structure. Visual Studio Code has a document outline tree view in the Explorer tab. Additionally, it has a code navigation bar (Breadcrumb), on the top of the editor, that displays the symbol path up to the cursor position. Finally, the 'Go to Symbol' command can be used to display a searchable list of all the symbols in a document. In any case, symbols are presented as buttons that navigate you to the symbol declaration when clicked.

To support the features described above, a handler for the LSP Document Symbol event was implemented. This event is triggered when a document is changed. The symbols are stored in symbol tables on AST nodes. So, by iterating the nodes of the AST, the symbol tree is created and is passed to the client.



**Figure 29** – Code Outline and Breadcrumb



**Figure 30** - Go To Symbol Command



## 11. Conclusions and Future Work

In this thesis we focused on the most prominent of the identified requirements, while some of the areas remain open and require additional research work. Below, we briefly discuss key topics for future work.

First of all, there are features of the language that we are not handling yet. Some of these features are:

- Class Inheritance
- Prototypal Inheritance
- Destructuring Pattern
- Import / Export

Another topic for future work is the implementation of more source code editing automations. Our work covers the basic editing automations but there are still numerous automations like Rename or Find References that we have to consider for this work to be complete. Also, providing some common diagnostics and an automatic way to fix them is really helpful.

Another identified issue is the performance of the system. Our system is able to analyze some hundred lines of code without any noticeable delay, but more work is needed to be able to analyze large-scale systems that are consisted of thousands of lines of code. The most time consuming operation is the analysis. Thus, a big first step to be able to handle larger projects is to make the analysis incremental.

Finally, when we search for the active type binders of a symbol, we actually try to find the reverse control flow of the program on the AST. We believe that it would be saner to create an actual *Control Flow Graph* (CFG) [14] of the program during the analysis of the source code that will also store references to the previous nodes and search on the CFG all those information instead. That would reduce dramatically the complexity of our algorithms.

## References

- [1] Wikipedia, "Intelligent code completion," [Online]. Available: [https://en.wikipedia.org/wiki/Intelligent\\_code\\_completion](https://en.wikipedia.org/wiki/Intelligent_code_completion). [Accessed 25 March 2021].
- [2] Microsoft, "TypeScript Language," [Online]. Available: <https://www.typescriptlang.org/>. [Accessed 25 March 2021].
- [3] Facebook, "Flow," [Online]. Available: <https://flow.org/>. [Accessed 25 March 2021].
- [4] JSDoc, [Online]. Available: <https://jsdoc.app/>. [Accessed 25 March 2021].
- [5] Wikipedia, "Application Programming Interface," [Online]. Available: <https://en.wikipedia.org/wiki/API>. [Accessed 02 04 2021].
- [6] Wikipedia, "Documentation Generator," [Online]. Available: [https://en.wikipedia.org/wiki/Documentation\\_generator](https://en.wikipedia.org/wiki/Documentation_generator). [Accessed 02 04 2021].
- [7] Wallaby, "Quokka.js," [Online]. Available: <https://quokkajs.com/>. [Accessed 25 March 2021].

- [8] Microsoft, "Visual Studio Code," [Online]. Available: <https://code.visualstudio.com/>. [Accessed 25 March 2021].
- [9] Wikipedia, "Abstract Syntax Tree," [Online]. Available: [https://en.wikipedia.org/wiki/Abstract\\_syntax\\_tree](https://en.wikipedia.org/wiki/Abstract_syntax_tree). [Accessed 02 04 2021].
- [10] Microsoft, "Language Server Protocol," [Online]. Available: <https://microsoft.github.io/language-server-protocol/>. [Accessed 25 March 2021].
- [11] Wikipedia, "Syntax Analysis," [Online]. Available: <https://en.wikipedia.org/wiki/Parsing>. [Accessed 02 04 2021].
- [12] Wikipedia, "Context Sensitive Analysis," [Online]. Available: [https://en.wikipedia.org/wiki/Semantic\\_analysis\\_\(compilers\)](https://en.wikipedia.org/wiki/Semantic_analysis_(compilers)). [Accessed 02 04 2021].
- [13] C. GHEZZI and D. MANDRIOLI, "Incremental Parsing," *ACM Transactions on Programming Languages and Systems*, vol. I, pp. 58-70, January 1979.
- [14] Wikipedia, "Control Flow Graph," [Online]. Available: [https://en.wikipedia.org/wiki/Control-flow\\_graph](https://en.wikipedia.org/wiki/Control-flow_graph). [Accessed 02 04 2021].