Fall 12-4-2023

Developing & Marketing a JavaScript Support Extension for the srcML Infrastructure

Andrew Blachly
aeblach@bgsu.edu

Developing & Marketing a JavaScript Extension for the srcML Infrastructure

By: Andrew Blachly

Faculty Advisors: Dr. Micheal Decker & Dr. Douglas Ewing

Honors Advisor: Christine Shaal

Table of Contents

CHAPTER 1 Introduction

Ever since its inception, the internet and the software technologies that utilize it have become increasingly significant to our modernized civilization. With this increased level of significance has come an increase in the complexity of software and in the difficulty of maintaining and developing internet-based software [1]. There is a need for tools that assist software developers in performing crucial analysis, maintenance, and updates on large internet software systems. One such development tool that is currently in the process of expanding its ecosystem into the realm of Internet software development is the srcML infrastructure.

The srcML infrastructure is an open-source software analysis and development tool developed by the srcML team, a group of software engineering and computer science researchers. This tool is used to transform source code into an XML format called srcML which enables software developers to utilize existing XML technologies to perform software analysis. The srcML infrastructure currently supports a variety of programming languages but the srcML team is working to expand the languages it supports by developing a new source code to XML parser generator using grammar files and a parser generator [2]. A grammar file is a file that contains all known syntactic elements of a programming language. This includes elements such as structured operators and expressions that form the rules for how that language is written [3]. Using these grammar files, the parser generator in development will be able to produce parsers that can translate any given language to XML. The first goal of the srcML team is to create a series of grammar files that will both add support for those languages but also to develop the parser generator itself. One of the most requested languages to add to the srcML Infrastructure is JavaScript. JavaScript is one of the primary languages used for the internet, and as such, it is also one of the most popular languages in the world. Approximately 63.61% of software developers around the globe have had professional experience with the JavaScript programming language [4].

Another goal of the srcML team is to grow the srcML community of users and developers. As JavaScript has a large community and is so highly requested, adding JavaScript is expected to lead to growth in the srcML community. To meet this goal of community growth via adding JavaScript, a marketing plan needs to be developed to outline the steps needed, the actions that need to be implemented, and the measurements that need to be taken to judge the plan's success. In this project, I investigate and develop both the marketing plan and the JavaScript grammar for the srcML Infrastructure.

In particular, I answer the following research questions:

- RQ1: How can the syntactic grammar of JavaScript be defined by a grammar file that is conducive to XML representation.
- RQ2: What is the best approach for marketing an existing piece of open-source research software as a professional development tool.
- RQ3: What is the value of the srcML infrastructure provides to the target markets and how does it compare to the value of similar competitors.

The remainder of this paper is structured as follows. In CHAPTER 2, we discuss work related to the project. CHAPTER 3 details the methodologies used to develop the JavaScript grammar file, the accompanying testing files, and the marketing plan. In CHAPTER 4, we cover the results of the project, the difficulties incurred, and how the development progressed. CHAPTER 5 discusses the implications of this project for future work, including the development of the srcML parser generator, additional grammar files for other programming languages, and the continued evolution of the srcML team's marketing strategy.

CHAPTER 2 Literature Review

In this chapter, we present the work and literature related to this project. In Section 2.1, we provide related work pertaining to the srcML infrastructure and its competitors. In Section 2.2, we describe related work relevant to the development of JavaScript grammar. Finally, in Section 2.3, we provide related work important to the development of the marketing plan, including contemporary software marketing materials and demographic statistics.

2.1. srcML

The srcML infrastructure is designed to perform lightweight partial parsing. This means that while performing fact extraction on the source code, some low-level such as non-syntactic elements cannot be derived directly [5]. However, this lightweight parsing approach also allows for robust parsing, enabling the srcML infrastructure to handle unprocessed, incomplete, or uncompilable code [6]. The partial processing done by the srcML infrastructure also preserves the documentary structural elements of the source code. This fixes the problem where important information held in comments, whitespace, and other documentation in source code is removed; a problem that has prevented the adoption of similar tools [7]. The srcML infrastructure is also extremely fast to convert to and from, beating many compilers at over 25KLOC/sec [8]. The study in [9] demonstrates the srcML infrastructure's capacity to automate manual tasks both efficiently and accurately, completing 2060 changes in under 11 minutes with 2 mistakes compared to the 304 mistakes made manually by developers in significantly more time. The srcML infrastructure is versatile and can be utilized alongside many other technologies, including XML technologies like XPath and Microsoft LINQ [6]. In addition to existing XML technologies, the srcML team has developed several software programs that build upon the existing srcML infrastructure. One such program is srcPtr, an application designed to perform more comprehensive pointer analysis directly on the abstract syntax tree by utilizing the srcML XML format to directly access the source code [10]. Another complimentary program is srcDiff, a tool that performs robust version differencing by leveraging the srcML XML format to access abstract syntactic information, allowing for complex structural differences in the source code to be read [11]. My work will complement the research done into the benefits and drawbacks of using the srcML infrastructure by leveraging this research to provide a compelling value proposition to market the software effectively.

2.2. JavaScript and Grammars

The syntactical elements of the JavaScript are documented by ECMA International's standard specification for the programming language [12]. ECMA International is the current owner and maintainer of ECMAScript, the language that is conventionally referred to as JavaScript. My work will build off of the ECMA specification by translating the syntax and grammar of JavaScript into a grammar file formatted for the srcML infrastructure's future parser generator.

Grammar files of the JavaScript programming language have been built for other existing parser generators. ANTLR [13] has created a JavaScript grammar file for its parser generator. My work will differentiate itself from these other JavaScript grammar files because the srcML infrastructure's focus on documentary structure preservation and the software's laissez-faire parsing approach necessitates a different format. This new JavaScript grammar file will be a part of the larger parser generator project and may change format to accommodate changes to the parser generator design.

The Software Development Laboratory (SDML) has begun to create several other grammar files that will be used in conjunction with the srcML infrastructure's future parser generator. These include grammar files that represent the syntax for the programming languages C++, C#, Java, Python, and Swift. My work will extend the progress made by these grammar files by utilizing the same tag-based format and elements to capture the unique syntax of the JavaScript programming language.

2.3. Marketing and Competitors

It is common for software companies and products to have marketing plans developed for them. In [14], Berestova creates a marketing plan for the case company EcoLeaks, a new software startup company that intends to offer a free to use mobile application that allows those who register to measure water quality and pH levels [14]. My work will differentiate itself from Berestova's because the srcML team's status as a research grant-funded development team with years of development and professional reputation requires a different approach to developing a marketing strategy than that used to develop a marketing strategy for a new startup company.

Tree-sitter is the main competitor to the srcML infrastructure. Like srcML, Tree-sitter is an open-source parsing tool that works with a variety of languages, including JavaScript [15]. Unlike the srcML infrastructure, Tree-sitter does not perform lossless transformations of source code but can operate at comparable speeds. My work will analyze and discuss Tree-sitter as a competitor to the srcML infrastructure in the market of JavaScript parsing tools.

CHAPTER 3 Background

This chapter provides background information on srcML, programming language syntax, and grammars. More specifically, Section 3.1 discusses grammar and syntax in programming languages, while Section 3.2 discusses srcML and the srcML infrastructure in further detail as it pertains to this project.

3.1. Grammar and syntax

In programming languages, syntax refers to the rules that are used to define how a program language is correctly written and structured. There are multiple levels of syntax. The lowest level is lexical grammar, where individual characters form word-like structures called tokens. The middle level is grammar, the level that determines how tokens are correctly combined to create phrase-like structures. As an example, “if” is a token in JavaScript and can be correctly used in a phrase like “if (x = true) {return x}”. The highest level is the context level, the level that checks for correctness in the context of the source code. For example, checking if a variable’s value is the correct type for a function would be done at this level. For this project, only the first two levels of syntax will be considered because srcML performs lightweight parsing, meaning that the context is not considered during translation.

There are multiple types of elements included in the grammar of programming languages, but there are several overarching types, including statements, expressions, operators, declarations, functions, and classes. Fig.1 shows an example of JavaScript code that uses all seven of these types

```
class manipulator {
    isFunctional = true
}
class yManipulator extends manipulator {
    modifyY (y){
        if (y > 0){
            return y - 1
        }
        else{
            return y + 10
        }
    }
}
var y = 0
const yManipulatorInstance = new yManipulator
for (let x = 0; x<10; x++){
   y = yManipulatorInstance.modifyY(y)
}
```

Fig.1. Code example of overarching types of grammar elements

of grammar elements. Statement elements are elements that are used to specify an action that needs to be performed. In Fig.1, examples of statement elements include the return element (an element that stops a functions execution and returns a specified value) used on lines 7 and 10, and the if element (an element that executes code if a specified condition is met) used on line 6. A special type of statement elements are loops, statement elements that execute multiple times given a certain condition is met. This includes the for-loop (a loop that executes a specified number of times) on line 16 that executes exactly ten times. Expression elements are elements that represent a value. There are many types of expressions used in Fig.1, including arithmetic expressions like *y + 10* from line 10, literals (elements that represent fixed values or objects) like `true` used on line 2, and variables (elements that referrer to a value stored in memory) like the variable `y` declared on line 14.

Operator elements are characters that are used in expressions to produce a result. Some operators are arithmetic operators that compute values like *+*, some are comparative operators that compare values like `>,` and others are assignment operators that assign a value to a variable like *=*. These operators are used in Fig.1 to compute *y + 10* on line 10, to determine if y is less than 0 on line 6, and to assign the value of `y` to the value returned by `yManipulatorInstance.modifyY(y)`on line 17. Declaration elements are statement elements that define identifiers for elements in the code, like variables, functions, and classes. Examples of declarations are on lines 2, 14, and 15 where variables are declared, on lines 1 and 4 where classes are declared, and on line 5 where the `modifyY()` function is declared.

Function elements are sets of statements that are executed to perform a function. These elements often receive an input and return an output. Function elements are defined using declarations and are executed using calls that are like typical statement elements. In Fig.1, the `modifyY()` function is declared on line 5 and called on line 17. It takes a variable `y` as input and outputs two different modifications to the value of `y`. When it is called, it is passed the variable `y` on line 17 as input. The final element type relevant to this project is the class element, an element that represents a blueprint for an object within the code with its own member elements. Classes are defined using declarations and instances of classes can be created in JavaScript using the `new` keyword. Classes can inherit the properties of other classes using a hierarchical inheritance structure. Fig.1 has two separate classes, `manipulator` and `yManipulator`. The variable `isFunctional` is a member variable of the `manipulator` class and the function `modifyY()` is a member of the `yManipulator` class. However, the `extends` keyword used in the declaration of `yManipulator` (line 4) indicates that the class inherits from `manipulator`, meaning that the `yManipulator` class also has the variable `isFunctional` as a member variable. An instance of the `yManipulator` class called `yManipulatorInstance` is created on line 15 using the `new` keyword. There are many other elements, but most fall under one of these overarching types.

3.2. srcML Background

The backbone of the srcML Infrastructure is an XML format (i.e., srcML format) that uses XML tags to represent the abstract syntax elements of source code (i.e., the structure of code). This is similar to how one might break down an English sentence into parts (e.g., subject, verb, and complement). These tags are embedded in the source code such that they surround the source code with an opening and closing tag. Syntax elements are often encased in multiple sets of tags that encompass not only a single syntax element, but also the larger syntactic context that it is a part of.

TABLE 1. AN EXAMPLE OF SRCML TRANSLATION FROM SOURCE CODE

Source Code
`int A;`
srcML Translation
`<decl_stmt><decl><type><name>int</name></type> <name>A</name></decl>;</decl_stmt>`

To illustrate the srcML format, Table 1 contains an example of translating C++ source code into srcML format. The source code in Table 1 is C++ code that declares a variable named *A* of type *int*. Each element in the source code example is surrounded by the *decl_stmt* and *decl* tags, representing how the entire code is a part of the larger declaration statement and declaration structures. The exception is the semicolon that marks the end of the statement, which is not included in the declaration syntax but is punctuation marking the end of the declaration statement. The *int* type is marked by the *type* tag, which marks that it serves as a type in the source code, and the *name* tag, which displays how the name of the type is *int*. The *A* variable being defined is wrapped in the *name* tag. This example includes only four of the dozens of unique srcML tags documented by the srcML team [16].

Archives are files that combine multiple source code files into a single file, which is more convenient to manage than a large number of files [8]. srcML uses a similar type of file called a srcML archive, which combines the translations of multiple different srcML files into one larger XML file [8]. This srcML archive file consists of many independent *unit* elements that encase the srcML translation of a specific set of source code. These separate *unit* elements are then wrapped by a larger *unit* element to show that they are all part of the same srcML archive. Fig. 2 is an example of a srcML archive. The separate srcML translations of the *break* elements are wrapped in *unit* elements to distinguish themselves from each other and the entire archive is wrapped in a larger *unit* element. The srcML archive is used as the default output of the srcML infrastructure when multiple files are used as input simultaneously.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.srcML.org/srcML/src" language="Python" url="operator">

<unit language="JavaScript">
<break>break</break>
</unit>

<unit language="JavaScript">
<break>break <name>label</name></break>
</unit>

</unit>
```

Fig. 2. Example of srcML archive

CHAPTER 4 Methodology

This chapter outlines the methodology and approaches used to conduct the development of the JavaScript grammar file, the accompanying testing files, and the marketing plan for the JavaScript support extension for the srcML infrastructure. Section 4.1 discusses the methods used to plan and conduct the creation of the JavaScript grammar file. Section 4.2 discusses the same but for the creation of the files used to test the grammar. Section 4.3 discusses the approach used to produce the marketing plan.

4.1. JavaScript Grammar Methodologies

RQ1 of this project is related to the methodologies used to create the JavaScript grammar file. This section answers this guiding question by detailing the approach used to create the grammar file so that it is robust, comprehensive, and accurate. The JavaScript grammar file is an XML file designed to cover every grammar element within the JavaScript programming language. This includes every operator, statement, expression, and declaration used in JavaScript, as well as every correct way of writing those elements in source code. This was done by identifying grammar elements of JavaScript and encasing those elements in tags matching those elements. One example is the return statement element:

```
<return> return <expr/> </return>
```

The return statement in this example has a simple pattern where the *`return`* keyword is followed by any expression. To show this, the entire statement is surrounded by the return tags and the expression is represented by an expr tag. Many grammar elements have several correct grammatical patterns, so every pattern is included in the grammar file. For example, an array in JavaScript can have any number of expressions inside, so the array element was given the following three patterns to represent this:

```
<array> [                    ]</array>
<array> [ <expr/>            ]</array>
<array> [ <expr/> , <expr/> ]</array>
```

The parser generator being developed has been designed to recognize any pattern that grammar file contains. By representing the array with zero, one, and many expressions in it, the grammar parser will be able to recognize the grammar patterns for an array with zero to many expressions, meaning that any number of expressions would be correctly translated.

The development of the JavaScript grammar file took an iterative approach. Each week of development began with researching Mozilla's MDN documentation on JavaScript [17] to identify the different grammar patterns of syntactic elements in the language. MDN was used due to its succinct and thorough documentation of the JavaScript grammar. Once these were identified, the element and its grammar patterns were added to the grammar file. The existing documentation created by the srcML team was referenced to ensure consistency with previously used markup tags

[16]. Later in development, these resources were supplemented with the official ECMAScript language specification [12] to ensure that niche grammar patterns were accounted for and that previously implemented patterns followed the official documentation. While development progressed, I received feedback about the changes through two weekly meetings. During the first meeting, I met with my software engineering project advisor and usually another researcher from the srcML team to discuss the work performed during the iteration. These meetings were used to gauge the direction of the iteration's development, make initial corrections, and propose questions that would be introduced to the rest of the srcML team later in the week. Using this feedback, development continued until the progress of the week was introduced to the larger srcML team via a video conference meeting. During this time, the expert researchers on the srcML team would debate the implementations of the JavaScript syntax elements in relation to the srcML XML format and would suggest modifications and improvements. The srcML team was full of experts on the srcML infrastructure and grammar parsing, including the principal creators of srcML, Dr. Jonathan I. Maletic and Dr. Michael L. Collard, as well as my software engineering honors project advisor Dr. Michael J. Decker. I leveraged the expertise of the srcML team to further improve the grammar file. These suggestions and the ones from the previous meeting were then used to guide the development of the grammar file in the next week. This iterative feedback approach to development continued until the JavaScript grammar file was complete.

4.2. Testing File Methodologies

In addition to the grammar file, several files are also created to enable testing of the parser which will be produced from the parser generator. There are two types of testing files developed for the grammar file. The first type are JavaScript files that each demonstrate a specific element of the grammar, such as the operators or while-loops. These files demonstrated every correct way to structure the specific element as defined by the language standards, ensuring that all possibilities are accounted for. These files often contained multiple related elements that can appear together, such as the try-and-catch elements, to save space. The second type of testing files are srcML files. Each of these is the correct markup (as specified by the grammar I developed) of one of the JavaScript test files. Together, these specify the input and correct output that should be produced by the parser generated by the parser generator.

The development of the testing files for the JavaScript grammar file took place near the end of the grammar file's production. This was done to ensure that the design of the grammar file was mostly solidified before testing so that the testing files didn't need to be updated frequently to match the weekly changes made to the grammar file. The same iterative process used to develop the grammar file was used to develop the testing files. The process consisted of researching MDN documentation and ECMAScript specifications, referencing the grammar file, developing the source code and translation files, and getting feedback from experienced srcML team members.

4.3. Marketing Plan Methodologies

The marketing plan for the JavaScript support extension for the srcML infrastructure was, as the name suggests, a marketing plan that detailed the many different marketing tactics and strategies designed to properly introduce the JavaScript support extension into the market. The marketing plan used the approach and format for creating marketing plans outlined by Alexander

Chernev [18]. This approach highlighted the importance of defining the target market as well as what the company wanted to accomplish with the market offering before any other areas of the market plan were defined. These first two areas help to give the marketing plan focus and inform the decisions made in the other areas of the market plan.

Table 2. The sections outlined by Chernev's format for marketing plans and brief explanations.Table 2 lists the seven distinct sections outlined by Chernev's format for marketing plans in the order that they should appear in the marketing plan. The first of these sections is the executive summary, which provides a comprehensive summarization of the other sections. The next section, the situation overview, summarizes the context in which the marketing plan is being developed and what is relevant to the marketing plan, including information about the market, product, company, and competition. The goals section comes third and states the objectives of the srcML team that this marketing plan will attempt to achieve. The strategy section describes both the target market and demographic as well as the proposed value that the JavaScript support extension will provide to them. This is followed by the tactics section, which describes the actions taken to implement the strategy. Then, the implementation section outlines how the marketing strategy and tactics are introduced to the market. The control section is last and details how the success of the marketing plan should be monitored, measured, evaluated, and improved in future iterations. The marketing plan developed from this project implemented these sections heavily while modifying some to better fit the digital, opensource market offering.

TABLE 2. THE SECTIONS OUTLINED BY CHERNEV'S FORMAT FOR MARKETING PLANS AND BRIEF EXPLANATIONS.

Section	**Explanation**
Executive Summary	Summary of entire marketing plan
Situation Overview	Provides context about the current environment
Goals	Defines what the company seeks to achieve
Strategy	Defines target markets, target demographics, and value propositions
Tactics	Outlines the actions that will be taken
Implementations	Outlines how the strategy and tactics will be introduced to the market
Control	Defines how the success of the marketing plan will be measured.

Chernev's work [18] also provided useful tools for developing a marketing plan beyond structure and approach. The book stressed the importance of being clear, succinct, and actionable in the marketing plan, and while actionability was hindered by the ambiguous release window for the extension, the marketing plan was developed with these three core tenants. The book also contained example marketing plans that were used as a reference for the type of language and information used throughout the different sections of the marketing plan.

The marketing plan for the JavaScript support extension of the srcML infrastructure began development after a majority of the testing files had been developed. This was done so that the marketing plan could be developed with more finalized information about the product offering. This is commonly referred to as being product oriented, developing a product first and then developing a marketing plan around its most exemplary traits [19]. This contradicts the modern market orientated approach where the market factors of a product are considered before the product begins development [20]. Product orientation was chosen due to the srcML team's status as a research team who has already created a software offering but wishes to spread awareness of it.

The process of developing the marketing plan for the JavaScript support extension followed a similar iterative process to that used during the development of the grammar file and its associated test files. The iterative process taken to develop the marketing plan was as follows. Research and development were performed for the marketing plan based on the next element that needed development. Once a week, I meet with my marketing project advisor to discuss the development of the iteration and to get feedback focused on improving the plan. As an accomplished professor of marketing, Dr. Douglas Ewing's skill and expertise with developing marketing plans was leveraged to provide indispensable feedback on the plan. This feedback was then used to guide development during the next iteration. As the development process progressed, my software engineering project advisor was also consulted for feedback since, as a member of the srcML team, he could provide important insight into the goals of the research team. This process continued until the marketing plan was complete.

CHAPTER 5 Results

This chapter discusses the results and outcomes of the project and is split into two separate sections. Section 5.1 discusses the results of producing the JavaScript grammar. Section 5.2 discusses the results of creating the files that will be used to test the grammar file. Section 5.3 details the results of producing the marketing plan for the JavaScript support extension of the srcML infrastructure.

5.1. Results of JavaScript Grammar File Development

The JavaScript grammar file required approximately four and a half months of development to complete. Development began in May 2023 and continued through the middle of September. The produced JavaScript grammar file is a 470-line XML file that accounts for each syntactical element present in base JavaScript. The file includes all correct grammar patterns for every operator, statement, expression, and declaration in JavaScript, as well as extensive comments detailing most of the syntactical elements included. The complete grammar file can be found in APPENDIX I.

There were no significant difficulties preventing the development of the grammar file. However, there were several times where the implementation of certain element's grammar patterns was heavily debated. As stated before, the grammar file was extensively scrutinized by both me and the srcML team during our weekly meetings. Many design changes often occurred as a direct result of this scrutiny, one of which being the implementation of declaration specifiers. In JavaScript, when you declare a variable, you need to declare its scope using either `const`, `let`, or `var`. One of these specifiers must be included for declarations, but only one of the three. For an extended period, there were three different syntax patterns for declaration statements in the grammar file to handle the three options.

Fig. 3 contains the original implementation of JavaScript declarations in the grammar file. This implementation allowed for declarations to be used with any of the three specifiers, with an initialization, or with both. This implementation was revisited once it was noticed that multiple declarations could be performed on the same line and that they would need to use the same specifier. In addition, the `export` keyword necessitated the change of the declaration patterns.

```
<decl>               <name/> <init/> </decl>
<decl> <specifier/> <name/>          </decl>
<decl> <specifier/> <name/> <init/> </decl>

        <specifier> const </specifier>
        <specifier> let   </specifier>
        <specifier> var   </specifier>
```

Fig. 3. JavaScript declaration original srcML implementation.

Fig. 4 displays the second implementation of the JavaScript declarations in the grammar file. This implementation adds a `decl_stmt` tag around the previous `decl` to allow for multiple declarations in the same element, and it also adds the `export` keyword as an optional specifier to the new `decl_stmt`. The declarations themselves are like their previous implementation, particularly in how the declaration specifiers are still optional in the declarations. However, the option of using an expression or a name was added to account for situations where entire arrays can be declared using the same grammar as individual variables. The implementation of the JavaScript declaration shifted many more times throughout the project, flipping back and forth between this implementation and another more streamlined implementation that defined a new `decl_specifier` metatag used in the `decl_stmt` element.

```
<decl_stmt>                                         <decl/> </decl_stmt>
<decl_stmt>                                <decl/>, <decl/> </decl_stmt>
<decl_stmt> <specifier> export </specifier>         <decl/> </decl_stmt>
<decl_stmt> <specifier> export </specifier> <decl/>, <decl/> </decl_stmt>

               <decl>              <name/> <init/> </decl>
               <decl> <specifier/> <name/>         </decl>
               <decl> <specifier/> <name/> <init/> </decl>
               <decl>              <expr/> <init/> </decl>
               <decl> <specifier/> <expr/>         </decl>
               <decl> <specifier/> <expr/> <init/> </decl>

                     <specifier> const </specifier>
                     <specifier> let   </specifier>
                     <specifier> var   </specifier>
```

Fig. 4. Second srcML implementation of JavaScript declaration

Fig. 5 shows the streamlined implementation of the JavaScript declarations in the grammar file. This implementation moves the declaration specifier from the declarations to the `decl_stmt`, simplifying the declarations by making declarations that are on the same line uniform. In the previous implementation, the specifier would only be in the first declaration and not the ones following it. This implementation ensures that no declarations have the specifier in them, making all declarations on the same line more uniform. This implementation also simplifies the `decl_stmt` pattern by removing the redundant line where the `export` specifier and multiple declarations are used, a pattern that can be inferred to be possible without direct specification. The streamlined implementation was chosen due to its simplicity, despite the worry of some srcML team members that the addition of metatags excessively will hinder comprehension. This example exemplifies the development of both the JavaScript grammar file and the testing files. Every syntactical element captured by this project experienced this type of scrutiny and changed multiple times before reaching its current state.

```
<decl_stmt>                                 <m:decl_specifier/> <decl/>          </decl_stmt>
<decl_stmt>                                 <m:decl_specifier/> <decl/>, <decl/> </decl_stmt>
<decl_stmt> <specifier> export </specifier> <m:decl_specifier/> <decl/>          </decl_stmt>

                       <decl> <name/> <init> = <expr/> </init> </decl>
                       <decl> <name/>                          </decl>
                       <decl> <expr/> <init> = <expr/> </init> </decl>
                       <decl> <expr/>                          </decl>

           <m:decl_specifier> <specifier> const   </specifier> </m:decl_specifier>
           <m:decl_specifier> <specifier> let     </specifier> </m:decl_specifier>
           <m:decl_specifier> <specifier> var     </specifier> </m:decl_specifier>
```

Fig. 5. Streamlined srcML implementation of JavaScript declaration with metatag

Another syntactical element that spurred a great deal of discussion was the export statement which is used to export non-declaration elements. In JavaScript, the export statement's have unique grammar patterns that were complicated to implement and sparked much discourse during srcML team meetings. One example of the complexity of the export statement is the *`default`* keyword. The *`default`* keyword can be used once per file to denote that the expression that comes after is the default expression for input and output. The syntax of *`export default`* allows for any expression to be set as the default export. Fig. 6 shows examples of *`export default`* being used with several distinct types of expressions. The first expression is a reference to a variable and the second expression is an arithmetic expression. Despite the JavaScript specifications labeling *`default`* as a keyword, the srcML team and I decided to label it as a specifier to *`export`*. There was much debate about this, but the conclusion was that since the default keyword isn't used anywhere else like this in the grammar file, it should be treated the same as a specifier that modifies export or an expression that can be used inside of the block used for import.

```
export default variable
export default 1 + 1
```

Fig. 6. Export default can be used with any expression.

Fig. 7 displays the implementation of *`export default`* in the grammar file. The unique *`default`* specifier is used here, and an undefined expression tag is used to denote that any type of expression can be used alongside *`export default.`* Defining *`default`* as a specifier in this context rather than as a keyword is a good example of the frequent occurrence where the grammar patterns of a syntactical element were used to deduce how it needed to be implemented.

```
<export> export <specifier> default </specifier> <expr/> </export>
```

Fig. 7. Default srcML implementation as a specifier to export

A notable contribution this project makes to srcML is the addition of the *`debugger`* markup tag. The *`debugger`* keyword in JavaScript simply turns on any debugging tools that the program may be using. However, this keyword is a feature unique to JavaScript, and since syntactically it is called just by itself and performs a task when executed, it necessitates being written in the grammar file as a statement.

5.2. Results of Test File Development

The testing files produced to evaluate the JavaScript grammar file required approximately a month and a half of development to complete. Development began in August and concluded in September. In total, there were forty testing files produced: twenty JavaScript source code files and twenty srcML files. 901 lines of code were written across the forty files.

The testing files were developed to test specific elements of the JavaScript grammar file by using the patterns described in the grammar file to exhaustively test every possible implementation of the element. Some elements only have one or two forms they can take, so there are very few implementations in their respective testing files. For example, the keyword `return` only has two ways that it can be used: with or without an expression.

Table 3 contains the testing files used to test the grammar file's implementation of the `return` keyword. In the source code (*return.js*) file, there are two separate examples of source code corresponding to the two patterns of the `return` statement. The first used the `return` keyword by itself, while the second uses the Boolean literal `false` as an expression. In the srcML archive file (*return.js.xml*), these two examples of source code have been manually translated to srcML. Like in the source code archive, the first example (line 5) is just the `return` keyword, but it is wrapped in the `return` element tag and two `unit` element tags, one to denote the single compilation unit (i.e., example) and another for the archive to wrap all examples. The second example (line 9) is translated like the first, but with the addition of `false`, which is wrapped by the `expr` element tag and the `literal`, Boolean type element tag. Some elements, including the `if` element which is evaluated using testing files shown in APENDIX B and APENDIX C, have far more syntax patterns and have more line of source code and srcML translation in their files.

TABLE 3. RETURN KEYWORD TEST FILES

return.js

```




return



return false


```

return.js.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.srcML.org/srcML/src" language="Python" url="operator">

<unit language="JavaScript">
<return>return</return>
</unit>

<unit language="JavaScript">
<return>return <expr><literal type="boolean">false</literal></expr></return>
</unit>
</unit>
```

5.3. Results of Marketing Plan Development

This question will discuss the end results of developing a marketing plan for the JavaScript support extension of the srcML infrastructure. This section will answer both RQ2 and RQ3. RQ2 will be answered by the discussion of tactics and implementations used to market the JavaScript support extension of the srcML infrastructure given its status as an opensource research project. RQ3 will be answered by the discussion of the target demographic, the value that the srcML infrastructure brings to that group, and how it compares to its primary competitor Tree-sitter.

The development of the marketing plan for the JavaScript support extension of the srcML infrastructure required two months of development to complete. Production began in early September and concluded development in late October. The marketing plan developed into a four-thousand-word document detailing the marketing strategy and tactics used to achieve the srcML team's goals for the JavaScript support extension. The marketing plan is included as APENDIX D.

The goals of the srcML team were split into a primary and secondary goal. The primary goal was to use the JavaScript support extension to grow the adoption of srcML on software industry development teams by enabling the use of the srcML infrastructure with the popular JavaScript programming language. The secondary goal of the JavaScript support extension was to grow the number of collaborators contributing software and grammar files to the srcML project. These goals helped to define the two target demographics of the marketing plan to software developers or computer scientists who use JavaScript in industry development and software developers or computer scientists who would be interested in contributing to the srcML project. While this seems like a broad field, research into demographics of this target market revealed that most people who work with software are college educated Caucasian males that are between young adult and middle aged. Market research was done alongside the defining of the target market to form a better understanding of the greater context in which the JavaScript support extension will be marketed.

The defined target market and demographics then informed the development of the value proposition for the JavaScript support extension. The value provided to customers who would use the srcML infrastructure and the JavaScript support extension in industry development is primarily the preservation of source code during translation, something that the competitor Tree-Sitter is unable to accomplish. Other valuable aspects of using srcML included its low cost, its simplicity, the publicly available teaching tools, the robust suite of compatible tools, and public communication systems set up by the srcML team. The value for potential contributors was also defined and included the support of the srcML team, the notoriety of the srcML brand, the networking opportunities, and the potential for monetary compensation. The value of the JavaScript support extension was also defined as increasing the notoriety and adoption rate of the srcML infrastructure and exposing it into the JavaScript ecosystem.

The tactics used to promote the JavaScript support extension were impacted by the existing state of the srcML team. The srcML team is a research team funded by the Community Infrastructure for Research in Computer and Information Science and Engineering (CIRC) grant awarded by the National Science Foundation. Due to this, they have a strict marketing budget that cannot be redistributed to allow for an increase in funding for marketing purposes. This restricted the marketing tactics to the same activities that the srcML team was doing prior to the development of the JavaScript support extension, word-of-mouth marketing done through research papers, industry conference presentations, community workshops, and traffic to the website. Additional grass-roots tactics were chosen due to their low cost and utilization of existing resources, such as community updates delivered through existing community contact channels and documentation updates to resources held on the srcML website.

The designed implementation of these tactics was heavily influenced by the development of the parser generator that the srcML team is currently developing. Because the extension of the srcML infrastructure into JavaScript heavily depends on the completion of the future parser generator, the time frame in which the tactics for marketing the JavaScript support extension are implemented relies on the completion of the parser generator. The srcML team estimates that the parser generator will be finished within two years and the marketing for the extension will begin several months before the JavaScript support extension is released alongside the parser generator. The control over the effectiveness of the marketing plan was decided to be measured by many factors following the release of the JavaScript support extension, including the number of new srcML infrastructure users, monthly website traffic, the number of attendees at srcML conference events, and the number of new collaborators who join the srcML team per month.

CHAPTER 6 Conclusion and Future work

This project has produced three things: a JavaScript grammar file for the srcML infrastructure, forty files that will be used to test the grammar file, and a marketing plan for introducing the JavaScript support extension into the software market. All three were developed using an iterative process that leveraged the knowledge of contributing experts and research into existing documentation. For the JavaScript grammar file and its accompanying testing files, this iterative process involved researching JavaScript grammar and syntax documentation and discussing the implementation with the experts of the srcML team. The iterative process used to develop the marketing plan involved utilizing Chernev's framework for marketing plans and receiving critique from Dr. Ewing.

Future work for this project requires the development of the srcML parser generator to conclude. The grammar file and testing files have been developed for this future software so future work revolves around testing these products with the future technology. The grammar file and testing files will be updated and improved based on the results of this future testing. Additional future work involving the grammar file and testing files would be updating them to reflect changes made to the JavaScript language standards. Future work involved with the marketing plan also relies on the finishing of the srcML parser generator as the extension of support for JavaScript will be released to the public alongside the parser generator. Once the marketing plan has been followed, the market reaction will be monitored to judge the effectiveness of the marketing plan. The information gleaned from monitoring the public's reaction to the release of the JavaScript support extension will be used to further develop and improve the marketing plan.

References

[1] Lehman, M.M. (1996) Laws of Software Evolution Revisited. Proceedings of European Workshop on Software Process Technology (EWSPT'96), Nancy, 9-11 October 1996, 108-124.

[2] "Award # 2016465 - CCRI: ENS: Collaborative research: Enabling automated language support for the SRCML infrastructure," nsf.gov, https://www.nsf.gov/awardsearch/showAward?AWD_ID=2016465 (accessed Nov. 24, 2023).

[3] K. Slonneger and B. L. Kurtz, Formal Syntax and Semantics of Programming Languages a Laboratory Based Approach. Reading, Mass: Addison-Wesley Publishing Company, 1995.

[4] Vailshery, Lionel Sujay. "Most Used Languages among Software Developers Globally 2023." Statista, 19 July 2023, www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/.

[5] Collard, M. L., Kagdi, H. H., & Maletic, J. I. (2003). An XML-based lightweight C++ Fact Extractor. MHS2003. Proceedings of 2003 International Symposium on Micromechatronics and Human Science (IEEE Cat. No.03TH8717). https://doi.org/10.1109/wpc.2003.1199197

[6] Collard, M. L., Decker, M. J., & Maletic, J. I. (2011). Lightweight transformation and fact extraction with the srcML toolkit. 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation. https://doi.org/10.1109/scam.2011.19

[7] Augustine, V. (2012). Automating adaptive maintenance changes with SRCML and LINQ. Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. https://doi.org/10.1145/2393596.2393604

[8] Collard, M. L., Decker, M. J., & Maletic, J. I. (2013). srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. 2013 IEEE International Conference on Software Maintenance. https://doi.org/10.1109/icsm.2013.85

[9] Collard, M. L., Maletic, J. I., & Robinson, B. P. (2010). A lightweight transformational approach to support large scale adaptive changes. 2010 IEEE International Conference on Software Maintenance. https://doi.org/10.1109/icsm.2010.5609719

[10] V. Zyrianov, C. Newman, D. Guarnera, M. Collard and J. Maletic, "srcPtr: A Framework for Implementing Static Pointer Analysis Approaches," 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), Montreal, QC, Canada, 2019, pp. 144-147, doi: 10.1109/ICPC.2019.00031.

[11] Decker MJ, Collard ML, Volkert LG, Maletic JI (2020) srcdiff: A syntactic differencing approach to improve the understandability of deltas. J Softw Evol Process 32(4). https://doi.org/10.1002/smr.2226

[12] ECMA, E. (2022). 262: ECMAScript language specification. ECMA (European Association for Standardizing Information and Communication Systems), Pub-ECMA: Adr,.

[13] T. Parr, The Definitive ANTLR 4 Reference, 2nd ed. Raleigh, NC: Pragmatic Bookshelf, 2013.

[14] A. Berestova, "Marketing plan for a high-tech product," Theseus, https://urn.fi/URN:NBN:fi:amk-201505127623 (accessed Nov. 24, 2023).

[15] "Introduction," Tree-sitter, https://tree-sitter.github.io/tree-sitter/ (accessed Nov. 24, 2023).

[16] “SRCML markup documentation,” srcml.org, https://www.srcml.org/documentation.html (accessed Nov. 24, 2023).

[17] MozDevNet, “JavaScript,” MDN Web Docs, https://developer.mozilla.org/en-US/docs/Web/JavaScript (accessed Nov. 24, 2023).

[18] A. Chernev, The Marketing Plan Handbook, 6th ed. Chicago, IL: Cerebellum Press, 2020.

[19} “Product orientation,” Monash Business School, https://www.monash.edu/business/marketing/marketing-dictionary/p/product-orientation (accessed Nov. 24, 2023).

[20] Kirca, A. H., Jayachandran, S., & Bearden, W. O. (2005). Market Orientation: A Meta-Analytic Review and Assessment of its Antecedents and Impact on Performance. Journal of Marketing, 69(2), 24-41. https://doi.org/10.1509/jmkg.69.2.24.60761

APPENDIX A:
srcML JavaScript Grammar File

```
<?xml version="1.0"?>
<unit xmlns="http://www.srcML.org/srcML/src" xmlns:m="http://www.srcML.org/srcML/meta" type="grammar">

<!--Operators-->
<!--Assignment-->
<m:operator class="modifier"> = </m:operator>
<m:operator class="arithmetic modifier"> *= </m:operator>
<m:operator class="arithmetic modifier"> -= </m:operator>
<m:operator class="arithmetic modifier"> += </m:operator>
<m:operator class="arithmetic modifier"> /= </m:operator>
<m:operator class="arithmetic modifier"> %= </m:operator>
<m:operator class="arithmetic modifier"> **= </m:operator>

<!--Shift Assignment-->
<m:operator class="arithmetic modifier"> &lt;&lt;= </m:operator>
<m:operator class="arithmetic modifier"> &gt;&gt;= </m:operator>
<m:operator class="arithmetic modifier"> &gt;&gt;&gt;= </m:operator>

<!--Bitwise Assignment-->
<m:operator class="arithmetic modifier"> &amp;= </m:operator>
<m:operator class="arithmetic modifier"> |= </m:operator>
<m:operator class="arithmetic modifier"> ^= </m:operator>

<!--Logical Assignment-->
<m:operator class="relational modifier"> &amp;&amp;= </m:operator>
<m:operator class="relational modifier"> ||= </m:operator>
<m:operator class="relational modifier"> ??= </m:operator>

<!--Comparison-->
<m:operator class="relational"> == </m:operator>
<m:operator class="relational"> === </m:operator>
<m:operator class="relational"> != </m:operator>
<m:operator class="relational"> !== </m:operator>
<m:operator class="relational"> &lt; </m:operator>
<m:operator class="relational"> &gt; </m:operator>
<m:operator class="relational"> &lt;= </m:operator>
<m:operator class="relational"> &gt;= </m:operator>

<!--Arithmetic-->
<m:operator class="arithmetic"> + </m:operator>
<m:operator class="arithmetic"> - </m:operator>
<m:operator class="arithmetic"> * </m:operator>
<m:operator class="arithmetic"> / </m:operator>
<m:operator class="arithmetic"> % </m:operator>
<m:operator class="arithmetic"> ** </m:operator>
<m:operator class="arithmetic modifier"> ++ </m:operator>
<m:operator class="arithmetic modifier"> -- </m:operator>

<!--Bitwise-->
<m:operator class="arithmetic"> | </m:operator>
<m:operator class="arithmetic"> ^ </m:operator>
<m:operator class="arithmetic"> ~ </m:operator>
<m:operator class="arithmetic"> &lt;&lt; </m:operator>
<m:operator class="arithmetic"> &gt;&gt; </m:operator>
<m:operator class="arithmetic"> &gt;&gt;&gt; </m:operator>

<!--Logical-->
<m:operator class="relational"> &amp;&amp; </m:operator>
<m:operator class="relational"> || </m:operator>
<m:operator class="relational"> ! </m:operator>

<!--Nullish Coalescing-->
<m:operator class="relational"> ?? </m:operator>

<!--Acessing Operator-->
<m:operator class="access"> . </m:operator>

```

```
<!--Optional Chaining Operator-->
<m:operator class="access"> ?. </m:operator>


<m:operator class="relational"> in </m:operator>
<m:operator> delete </m:operator>

<m:operator> void </m:operator>
<m:operator> new </m:operator>
<m:operator> instanceof </m:operator>
<m:operator> await </m:operator>

<!-- Statements -->
<m:statement> <function/>   </m:statement>
<m:statement> <block/>      </m:statement>
<m:statement> <break/>      </m:statement>
<m:statement> <class/>      </m:statement>
<m:statement> <debugger/>   </m:statement>
<m:statement> <do/>         </m:statement>
<m:statement> <export/>     </m:statement>
<m:statement> <decl_stmt/>  </m:statement>
<m:statement> <empty_stmt/> </m:statement>
<m:statement> <expr_stmt/>  </m:statement>
<m:statement> <for/>        </m:statement>
<m:statement> <if/>         </m:statement>
<m:statement> <else/>       </m:statement>
<m:statement> <import/>     </m:statement>
<m:statement> <label/>      </m:statement>
<m:statement> <return/>     </m:statement>
<m:statement> <switch/>     </m:statement>
<m:statement> <case/>       </m:statement>
<m:statement> <throw/>      </m:statement>
<m:statement> <try/>        </m:statement>
<m:statement> <catch/>      </m:statement>
<m:statement> <finally/>    </m:statement>
<m:statement> <while/>      </m:statement>

<!-- Simple Statement Definitions -->
<!--<block>               {                 <m:stmts/>                      } </block> //TODO: Needed?-->
<block>                { <block_content> <m:stmts/>     </block_content> } </block>
<block type="pseudo">   <block_content> <m:statement/> </block_content>   </block>
<!-- //TODO: Special tag for expression block? block_expr, name_list, expr_list, block type="expr"? Otherwise, we need meta tag-->

<expr_stmt>  <expr/>                </expr_stmt>
<return>     return                 </return>
<return>     return <expr/>         </return>
<break>      break                  </break>
<break>      break <name/>          </break>
<continue>   continue               </continue>
<continue>   continue <name/>       </continue>
<debugger>   debugger               </debugger>
<throw>      throw <expr/>          </throw>
<label>      <name/>:               </label>
<alias>      <name/> as <name/>     </alias>
<!-- <alias> <expr/> as <name/> </alias> --> <!-- //TODO: Find a reason to use this -->

<empty_stmt> ; </empty_stmt>      <!-- How to do if ; is optional? How to not hit every semicolon or how to detect em-->


<!-- Declarations-->
<!-- //TODO: Add stuff for diffrent objects (array and list) -->
<decl_stmt>                                <m:decl_specifier/> <decl/> </decl_stmt>
<decl_stmt>                                <m:decl_specifier/> <decl/>, <decl/> </decl_stmt>
<decl_stmt> <specifier> export </specifier> <m:decl_specifier/> <decl/> </decl_stmt>

```

```
<decl> <name/> <init> = <expr/> </init> </decl>
<decl> <name/>                          </decl>

<decl> <expr/> <init> = <expr/> </init> </decl>
<decl> <expr/>                          </decl>

<!--Specifiers-->
<m:decl_specifier> <specifier> const   </specifier> </m:decl_specifier>
<m:decl_specifier> <specifier> let     </specifier> </m:decl_specifier>
<m:decl_specifier> <specifier> var     </specifier> </m:decl_specifier>

<specifier> export  </specifier>
<specifier> default </specifier>

<!-- Expressions -->
<expr> <call/>     </expr>
<expr> <literal/>  </expr>
<expr> <name/>     </expr>
<expr> <function/> </expr>
<expr> <lambda/>   </expr>
<expr> <yield/>    </expr>
<expr> <alias/>    </expr>
<expr> <block/>    </expr>
<expr> <array/>    </expr>
<expr> <name/> : <expr/> </expr>

<array> [                   ]</array>
<array> [ <expr/>           ]</array>
<array> [ <expr/> , <expr/> ]</array>

<literal type="boolean"> true          </literal>
<literal type="boolean"> false         </literal>
<literal type="null">    null          </literal>
<literal type="regex">   / <m:text/> / </literal> <!-- can have these flags after the final slash: d,g,i,m,s,u,v,y-->

<!-- call, argument list, argument -->
<call> <name/> <argument_list/> </call>

<argument_list> (                           ) </argument_list>
<argument_list> ( <argument/>               ) </argument_list>
<argument_list> ( <argument/> , <argument/> ) </argument_list>

<argument> <expr/> </argument>

<!-- Functions -->
<function> <name/> <parameter_list/> <block/> </function>
<function> function   <name/> <parameter_list/> <block/> </function>
<function> function           <parameter_list/> <block/> </function>
<function>                  <specifier> async  </specifier> <name/> <parameter_list/> <block/> </function>
<function>                  <specifier> export </specifier> <name/> <parameter_list/> <block/> </function>
<function>                  <specifier> export </specifier> <specifier> default </specifier>            <name/> <parameter_list/> <block/> </function>

<function type="generator"> * <name/> <parameter_list/> <block/> </function>
<function type="generator"> function * <name/> <parameter_list/> <block/> </function>
<function type="generator"> function *         <parameter_list/> <block/> </function>
<function type="generator"> <specifier> async  </specifier> * <name/> <parameter_list/> <block/> </function>
<function type="generator"> <specifier> export </specifier> * <name/> <parameter_list/> <block/> </function>
<function type="generator"> <specifier> export </specifier> <specifier> default </specifier>          * <name/> <parameter_list/> <block/> </function>
```

```

<lambda>                                    <parameter_list/> =&gt; <block/> </lambda>
<lambda>                       <specifier/> <parameter_list/> =&gt; <block/> </lambda>

<parameter_list> (                             ) </parameter_list>
<parameter_list> ( <parameter/>                ) </parameter_list>
<parameter_list> ( <parameter/> , <parameter/> ) </parameter_list>

<parameter> <decl/> </parameter>

<!-- Classes -->
<class>                           class <name/>                                  <block/> </class>
<class>                           class <name/> extends <super> <name/> </super> <block/> </class>
<class> <specifier/>              class <name/>                                  <block/> </class>
<class> <specifier/> <specifier/> class <name/>                                  <block/> </class>


<!-- Export and Import-->
<export> export <specifier> default </specifier> <expr/> </export>
<export> export <expr/> from <expr/> </export>
<export> export <block> { <expr/>             } </block>              </export>
<export> export <block> { <expr/>             } </block> from <expr/> </export>
<export> export <block> { <expr/>  , <expr/>  } </block>              </export>

<import> import <expr> <name/>                                              </expr> </import>
<import> import <expr/>                                                from <expr/> </import>
<import> import <expr> <name> defaultExport </name> </expr> , <expr/>  from <expr/> </import>
<import> import <block> { <expr/>             } </block> from <expr/> </import>
<import> import <block> { <expr/>  , <expr/>  } </block> from <expr/> </import>


<!-- If Statment -->
<if_stmt> <if/>                                                 </if_stmt>
<if_stmt> <if/> <if type="elseif"/>                             </if_stmt>
<if_stmt> <if/> <if type="elseif"/> <if type="elseif"/>         </if_stmt>
<if_stmt> <if/>                                         <else/> </if_stmt>

<if>               if      <condition> ( <expr/> ) </condition> <block/> </if>
<if type="elseif"> else if <condition> ( <expr/> ) </condition> <block/> </if>
<else>             else                                         <block/> </else>

<!--Loops-->
<do> do <block/> while <condition> ( <expr/> ) </condition> </do>
<while> while <condition> ( <expr/> ) </condition> <block/> </while>

<for> for                    <control> ( <init/> <condition>        ; </condition> <incr/> ) </control> <block/> </for>
<for> for                    <control> ( <init/> <condition> <expr/> ; </condition> <incr/> ) </control> <block/> </for>
<for> for                    <control> ( <init> <decl> <name/> <range> in <expr/> </range> </decl> </init> ) </control> <block/> </for>
<for> for                    <control> ( <init> <decl> <name/> <range> of <expr/> </range> </decl> </init> ) </control> <block/> </for>
<for type="await"> for await <control> ( <init> <decl> <name/> <range> of <expr/> </range> </decl> </init> ) </control> <block/> </for>

<init>                                ; </init>
<init> <specifier/> <decl/>           ; </init>
<init> <specifier/> <decl/> , <decl/> ; </init>
<init> <expr/>                        ; </init>

<incr>         </incr>
<incr> <expr/> </incr>

```

```
<!--Switch statments-->
<switch> switch <condition/> <block/> </switch>
<!--<switch> switch <condition> ( <expr/> ) </condition>  <block/> </switch>-->
<case> case <expr/> : </case>
<default> default :  </default>

<!-- Try and Catch -->
<try> try <block/> <catch/>            </try>
<try> try <block/>          <finally/> </try>
<try> try <block/> <catch/> <finally/> </try>

<catch> catch                                                     <block/> </catch>
<catch> catch <parameter_list> ( <parameter/> ) </parameter_list> <block/> </catch>

<finally> finally <block/> </finally>

<with> with <init> ( <expr/> ) </init> <block/> </with>

</unit>
```

APPENDIX B:

If Statement Source Code Testing File

```




if(x == true){}



if(x == true){}
else{}



if(x == true){}
else if(y == true){}



if(x == true){}
else if(y == true){}
else{}



if(x == true){}
else if(y == true){}
else if(z == true){}



if(x == true){}
else if(y == true){}
else if(z == true){}
else{}


```

APPENDIX C

If Statement srcML Testing File

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.srcML.org/srcML/src" language="Python" url="operator">

<unit language="JavaScript">
<if_stmt><if>if<condition>(<expr><name>x</name> <operator>==</operator>
<name>true</name></expr>)</condition><block type="pseudo">{}</block></if></if_stmt>
</unit>

<unit language="JavaScript">
<if_stmt><if>if<condition>(<expr><name>x</name> <operator>==</operator>
<name>true</name></expr>)</condition><block type="pseudo">{}</block></if>
<else>else<block type="pseudo">{}</block></else></if_stmt>
</unit>

<unit language="JavaScript">
<if_stmt><if>if<condition>(<expr><name>x</name> <operator>==</operator>
<name>true</name></expr>)</condition><block type="pseudo">{}</block></if>
<if type="elseif">else if<condition>(<expr><name>y</name> <operator>==</operator>
<name>true</name></expr>)</condition><block type="pseudo">{}</block></if></if_stmt>
</unit>

<unit language="JavaScript">
<if_stmt><if>if<condition>(<expr><name>x</name> <operator>==</operator>
<name>true</name></expr>)</condition><block type="pseudo">{}</block></if>
<if type="elseif">else if<condition>(<expr><name>y</name> <operator>==</operator>
<name>true</name></expr>)</condition><block type="pseudo">{}</block></if>
<else>else<block type="pseudo">{}</block></else></if_stmt>
</unit>

<unit language="JavaScript">
<if_stmt><if>if<condition>(<expr><name>x</name> <operator>==</operator>
<name>true</name></expr>)</condition><block type="pseudo">{}</block></if>
<if type="elseif">else if<condition>(<expr><name>y</name> <operator>==</operator>
<name>true</name></expr>)</condition><block type="pseudo">{}</block></if>
<if type="elseif">else if<condition>(<expr><name>z</name> <operator>==</operator>
<name>true</name></expr>)</condition><block type="pseudo">{}</block></if></if_stmt>
</unit>

<unit language="JavaScript">
<if_stmt><if>if<condition>(<expr><name>x</name> <operator>==</operator>
<name>true</name></expr>)</condition><block type="pseudo">{}</block></if>
<if type="elseif">else if<condition>(<expr><name>y</name> <operator>==</operator>
<name>true</name></expr>)</condition><block type="pseudo">{}</block></if>
<if type="elseif">else if<condition>(<expr><name>z</name> <operator>==</operator>
<name>true</name></expr>)</condition><block type="pseudo">{}</block></if>
<else>else<block type="pseudo">{}</block></else></if_stmt>
</unit>
</unit>
```

APPENDIX D

JavaScript Extension Marketing Plan

Marketing Plan for the srcML JavaScript Support Extension

1. Executive Summary

The srcML infrastructure is a source code transformation tool used for software analysis and development produced by the srcML team, an N.S.F. grant funded research team. The srcML team has developed an extension to the srcML infrastructure that will allow the software to be utilized with JavaScript source code.

The first of our primary goals is to spread the adoption of the srcML infrastructure among industry professionals by introducing it to the prevalent JavaScript development space. The second primary goal is to spread awareness of software and the srcML family of products among potential collaborators who we could bring into the srcML team to expand the srcML ecosystem.

Our first group of target customers are industry software engineers who work with JavaScript. This group potentially consists of 17.4 million people, although this target demographic is likely much smaller. The second group of target customers are English speaking software engineers and computer scientists who potentially have an interest in collaborating with the srcML team.

For software engineers and computer scientists, the srcML infrastructure provides lightweight, fast, and robust software analysis and development tool that preserves the critical non-syntactic documentary structures in source code. The srcML infrastructure has accessible documentation, educational resources, and consistent support updates and is free to use, simple to learn, designed for large systems, and compatible with a wide variety of XML technologies and srcML family software.

This plan outlines our key marketing activities central to implementing the JavaScript extension for the srcML infrastructure and meeting the goals of the srcML team.

2. Situation Overview

The srcML team is a software research team currently operating out of Kent State University. The team is comprised of a multitude of contributors working both remotely and on site with the team. The team is funded almost entirely the Community Infrastructure for Research in Computer and Information Science and Engineering grant awarded by the National Science Foundation.

The srcML team has developed many different software development tools, the most foundational of which being the srcML infrastructure, a software analysis and development tool that transforms source code into an XML format so that developers can utilize a suite of XML technologies to perform software analysis. All tools in the srcML family of software utilize the srcML infrastructure to perform helpful development tasks, such as the srcDiff tool utilizing the srcML XML format to perform source code differencing on multiple versions of a program. The srcML infrastructure and the srcML family of software are currently offered for download freely to promote their use in research and academia.

Recently, the srcML team has begun an initiative to extend the functionality of the srcML infrastructure to include more languages than those currently supported. One of the most anticipated additions to the languages supported by srcML is JavaScript. JavaScript is currently one of the most popular programming languages in the world. A 2023 developer survey conducted by Stack Overflow found that JavaScript was the most used programming language with 63.61% of developers having worked with it. A similar 2022 study conducted by SlashData found that approximately 17.4 million developers currently use JavaScript across the globe. Due to the immense popularity of JavaScript, there is a great incentive to make srcML JavaScript compatible.

The srcML team has developed a JavaScript extension that will allow for the srcML infrastructure to be compatible with source code written in JavaScript. This extension consists of a grammar file that denotes the markup of JavaScript in the srcML format and, when combined with the grammar parser that is currently in development, will allow for the srcML infrastructure and the srcML family of software to be used with JavaScript code.

The srcML infrastructure has been marketed largely through word of mouth. Until now, the srcML team has largely marketed the srcML infrastructure through its outreach initiatives. These initiatives primarily consist of attending prestigious conferences, discussing the software through conference presentations, or displaying how to utilize the srcML infrastructure through hands on workshops. Another way in which the srcML team has promoted the srcML infrastructure is through academic papers in prominent publications. In addition to these efforts, the srcML team has also developed a website for those who wish to learn more about the srcML infrastructure. The site contains extensive documentation, multiple tutorials, and an interactive web version of srcML, all of which aid new users in learning to utilize the software.

3. Goal

Our primary goal is to grow the adoption of srcML into industry software development teams, particularly those who work with the JavaScript language. Our secondary goal is to spread awareness of srcML and its family of software to potential collaborators who we could contract to continue to expand the srcML ecosystem. To achieve our primary goal, we have set the following objectives that we plan to meet within the year of release.

- *Customer Objectives*. Our key customer objectives are to spread awareness and promote adoption of srcML among the significant JavaScript development community. Our goal is to the monthly number of srcML downloads by 20% and increase use of the srcML website by 30% within the next year.
- *Collaborator Objectives.* We aim to create awareness of srcML among software engineering students and professionals and contract them as collaborators to create

additions to srcML similar to that of the JavaScript extention. We plan to bring on 8 more collaborators onto srcML to develop new language extensions by next year.

- *Internal Objectives*. Our primary internal objective is to mature the srcML family of software by extending the systems functionality into other languages. Our goal is to use the JavaScript extension as a template for future language extensions, inform the design of the grammar parser being developed, and create an opportunity for the srcML team in the industry marketplace.
- *Competitive Objectives*. We will strive to increase awareness of the benefits of srcML compared to our competitors. This effort will be focused on accentuating the strengths of srcML rather than maligning our competitors.

4. Strategy

4.1. Target Market

Customers

Our target market for the JavaScript extension is divided into two separate groups. The first group of customers are professional software developers working in industry who use JavaScript and need a robust, lightweight software analysis and development solution. There are potentially 17.4 million JavaScript users that could become new users of the srcML infrastructure due to the JavaScript extension, although the true number is likely smaller due to the niche application that this software fulfills. The expected demographic of this target group of customers are young adults or middle aged with a demographic tendency towards being college educated Caucasian males. This product is not geographically restricted, however, the dominate use of English by the srcML team and in the srcML infrastructure may limit our geographic reach.

The second group of customers are software developers or computer scientists who may be interested in collaborating on the srcML project. The demographics of this group overlaps significantly with those of the first. Those in the software developer and computer science fields are predominantly white Caucasian males. We will be targeting English speaking people within this group as the srcML team is comprised of English speakers. Bilingual or multicultural people will also be targeted, potentially opening the door for more potential collaborators than English speakers. The srcML team has experience working with remote contributors, so geographic location does not apply.

Company

The srcML team is a software research team led and founded by Dr. Maletic and Dr. Collard. The team is currently operating out of Kent State University and is comprised of a multitude of contributors working both remotely and on site with the team. The srcML team is currently funded by the Community Infrastructure for Research in Computer and Information Science and Engineering grant awarded by the National Science Foundation. This ongoing grant has been awarded for the goal of expanding the number of programming languages supported by the srcML infrastructure.

The driving goal behind the srcML team is to provide a robust research infrastructure to explore, analyze, and manipulate large scale software systems in a multitude of different programming languages. The main offering of the srcML team is the srcML infrastructure, a software analysis and development tool that transforms source code into an XML format so that developers can utilize a suite of XML technologies to perform software analysis. The team has also created a variety of software tools which utilize the srcML infrastructure. The strategic assets held by the srcML team consist of personnel who are experienced software engineers, a reputation for producing high quality tools an software, and recognition of the srcML brand.

Collaborators

Our current contributors are members of the srcML team working on the srcML family of software. This includes the contributors working on srcDiff, srcSlice, srcTl, srcType, srcUml, srcSax Event Dispatcher, srcPtr, and srcQl. These contributors are both remote and in person members of the team. The JavaScript extension is expected to enable the software produced by the contributors to be utilized on JavaScript source code because of how the srcML family of software is built from the srcML infrastructure.

Competitors

Other language transformation tools similar to the srcML infrastructure currently exist on the market. Of them the closest competitors are Tree-sitter and Txl. Both offer robust tools for performing analysis and transformations of source code comparable to the offerings of the srcML infrastructure. Both Tree-sitter and Txl currently support a larger selection of programming languages, but this JavaScript extension is the first step in mitigating that difference. Neither competitor provides preservation for documentation within source code while the srcML infrastructure does.

Context

Economic Context: Bear market, increasing inflation, increasing stock market volatility, and low GDP growth. The software development field has largely been unaffected by the wider economic situation. The industry has continued to grow exponentially, increasing total revenue and available positions drastically.

Regulatory Context: The srcML team has strict parameters in which it can operate outlined by the Community Infrastructure for Research in Computer and Information Science and Engineering grant awarded by the National Science Foundation. These obligations and restrictions of the grant, including strict allocations of funds, are outlined in detail by the organization. The srcML team also follows the Association for Computing Machinery's Code of Ethics and Professional Conduct, although this is voluntary and not enforced by a sanctioning body.

4.2. Customer Value Proposition

The JavaScript extension for the srcML infrastructure offers the following benefits to its target customers.

- *Preservation of source code*. Unlike most other language parsers, the srcML infrastructure performs its translation without losing important non-syntactic elements of the source code. This allows for meaningful documentary structures in source code such as comments, white space, formatting, and preprocessing information are preserved.
- *No Monetary Cost*. The srcML infrastructure costs nothing to download and use. Licenses can be used to include srcML in another product, but to utilize the srcML infrastructure costs no more than the time taken to learn it.
- *Simplicity and Learnability*. The srcML format is intentionally simple in the structure of its XML markup. This enables it to be easily understandable without extensive training. In addition to an easy to grasp structure, the srcML website contains extensive documentation that can answer any specific questions about the srcML format. The website also contains an interactive "playground" where customers can see how pieces of source code are translated into srcML format in an accessible way. Coupled with freely available tutorials, the srcML team provides extensive resources to help teach new users how to utilize the srcML infrastructure.
- *Fast while working with large systems*. Due to the srcML infrastructure being a lightweight parsing system, it provides an incredibly fast translation speed of approximately 3000 files per second. The srcML infrastructure has shown impressive results even with large programs such as the Linux Kernal, which was able to translate to srcML format in under 7 minutes.
- *XML tools and srcML family of software*. The srcML infrastructure is compatible with many commonly used XML tools such as XPath, XQuery, and XSchema. In addition, the srcML family of software further provides functionality compatible with the srcML infrastructure. srcDiff, srcSlice, and srcQl provide syntactic source code differencing, dataflow analysis, and syntax querying respectively.
- *Support and Maintenance*. The srcML team is continuously improving the srcML infrastructure to meet the needs of its users. The JavaScript extension is an example of a feature that was requested and implemented by the srcML team. Technical issues with srcML are also continuously being patched by the srcML team.

4.3. Collaborator Value Proposition

The JavaScript extension for the srcML infrastructure offers the following benefits to its current contributors.

- *Additional support and extension for contributions*. The software created for the srcML family of software are built using the srcML infrastructure. This results in improvements made to the srcML infrastructure being inherited by the software created by contributors. By expanding the functionality of the srcML infrastructure into the JavaScript ecosystem, the functionality of the software produced by contributors with be similarly expanded.

- *Notoriety for contributed software*. The JavaScript extension offers to broaden the userbase of software produced by contributors by allowing their software to be utilized by the large JavaScript ecosystem. This increase in the number of prospective users will provide increased notoriety for the software created by the contributors and for the contributors themselves.
- *Connections and Experience*. The software engineers who become contributors due to the JavaScript expansion will gain valuable experience in working in an environment comprised of both academic research and software development attributes. New contributors will also gain experience with remote team development, valuable experience to have due to the increase in remote employment caused by the Covid-19 pandemic. Many of the srcML team contributors are well know in the software development community, so collaborators will gain valuable connections by working with the team.
- *Monetary Compensation*. The srcML team has been allocated grant money to fund collaborators in producing similar extensions to that of the JavaScript extension. Collaborators can receive compensation for contributing language grammar files that will be used to extend the functionality of the srcML infrastructure.

4.4. Company Value Proposition

Strategic Value. The primary value provided by the JavaScript extension for the srcML team is increased notoriety and usership for the srcML brand of software. Introducing the srcML infrastructure to the massive JavaScript ecosystem will greatly broaden the prospective user base and allow for an increased user growth rate for the srcML brand. Potential to increase the number of prospective users by 15%. This increase in usership will likely cause a growth in both contracted and donated contributions to the srcML ecosystem.

5. Tactics

5.1. Product

The JavaScript extension is an addition to the srcML infrastructure that enables the software to be utilized on JavaScript source code. This software extension broadens the target audience of the srcML infrastructure to include the vast JavaScript community. The JavaScript extension will be distributed with future versions of the srcML infrastructure.

The JavaScript extension comprises of two main parts, a JavaScript grammar file and many test files. The JavaScript grammar file outlines all the possible syntax in the JavaScript programming language. This will be used by the future parser generator being developed for the srcML infrastructure to produce a JavaScript to srcML parser that can translate JavaScript source code to and from srcML XML format. The testing files that accompany the JavaScript grammar file are used to test the accuracy of the parser produced. There are 2 types of test files that come in pairs. The first are JavaScript files that contain several distinct syntax covered by the JavaScript grammar file. For example, one contains the different ways to write a for-loop and another contains all the JavaScript operators. The second type of test files are XML files that manually translate the JavaScript source code in the corresponding test file of the first type to the srcML XML format. Once the parser generator is complete, the test files of the first type will be translated by the produced JavaScript parser and the resulting translation will be compared with the manual translation to test for accuracy.

The srcML infrastructure is a software analysis and development tool that transforms source code into an XML markup called the srcML format so that developers can utilize a suite of XML technologies to perform software analysis. This tool is light weight, fast, and was designed to perform transformations on exceedingly large systems. The srcML infrastructure performs lossless transformation of source code, preserving important documentary structures like comments, white space, formatting, and preprocessor information. The srcML format is simple to understand but contains robust syntactic information that can be used to perform analysis on the transformed source code with existing XML based tools.

5.2. Service

We offer the following services to customers of the srcML infrastructure.

- *Extensive Educational Resources*. The srcML website hosts free resources to aid in learning how to use the srcML infrastructure. This includes extensive documentation detailing the srcML format and its language translations, the interactive playground customers can use to view source code translations in an intuitive format, and tutorials that guide customers through applications of the srcML infrastructure.
- *Communication Services*. We have created multiple avenues to contact the srcML team. Issue and error reporting has been made available through a public GitHub repository hosted by the srcML team (located at https://github.com/srcML/srcML/issues). An email service (srcmldev@gmail.com) has been created for team members to receive suggestions and questions directly from customers. A community discussion page is also hosted on Discord to foster wider discourse about srcML among its customers.

5.3. Brand

We use the srcML brand to identify our offerings. The srcML brand can be easily identified by the lowercase "src" naming convention used by the srcML infrastructure and the family of srcML software. Unique shared technologies also identify the srcML brand as all the brand's offerings heavily utilize the srcML format. The srcML team also has a unique logo, a white branch on a black or gray circle, to identify the work of the company. The srcML website features this logo and is styled after a command prompt terminal with light text on dark backgrounds, catering to the software engineers and computer scientists who are the target demographic of the team's software.

5.4. Price

The JavaScript extension, along with the srcML infrastructure itself, are free to use under a General Public License by anyone who wants to download the software. The only significant investment for customers is the time spent learning to utilize the software, a cost mitigated by the extensive documentation and instructional resources.

5.5. Incentives

Public Workshops. Often held at conferences, the srcML team will offer customers the opportunity to participate in a guided, hands-on demonstration of the srcML infrastructure. These educational workshops are held with a large audience and often feature an opportunity to ask the srcML team member questions after the demonstration.

Incentives for Collaborators. We offer collaborators several incentives to contribute to the srcML ecosystem. For collaborators who contribute to the ongoing effort to extend the srcML infrastructure into new programming languages by developing grammar files, grant funding may be given as compensation for your work. Collaborators may also be given the chance to travel with the srcML team to discuss srcML at a software development convention.

5.6. Communication

Customer Communication.

- *Confrence Presentations.* We plan to spread awareness of the JavaScript extension for the srcML infrastructure at major conventions the srcML team is planning to attend. This has proved effective at spreading awareness of previous major additions to the srcML ecosystem.
- *Community Updates.* We plan to update the community of current customers by directly updating users via the community discussion page and via emails that customers have been given. By making the current users of the srcML infrastructure aware of the JavaScript extension, we encourage them to pass information about the extension to new users who they believe would find it useful.

- *Discussion Online.* Online discussion boards such as Stack Overflow, AnswerHub, and Quora.com are used often by software developers and computer scientists to ask questions about specific issues and recommended solutions. By supplying questions about JavaScript source code analytics with answers involving the use of the srcML infrastructure, we spread awareness of srcML among the JavaScript community.

Collaborator Communication.

- *Message Directly.* We have contact information for all collaborators associated with the srcML team. Using this information, we will directly update the collaborators on the JavaScript extension to the srcML infrastructure.
- Weekly Team Meetings. Each week, online conference calls are made with the entire team to discuss the past week's progress and any major issues and updates that need to be shared with the team. Although not every collaborator is consistently in attendance, this weekly meeting will be used to inform the present collaborators about the JavaScript extension.

5.7. Distribution

The JavaScript extension and the srcML infrastructure are distributed through the srcML website. Executable versions of the software are available to download for Windows, macOS, and Linux systems. The source code of the software is also available to download from the srcML website. Access to the services we offer are also distributed through the srcML website, including access to the educational resources and links to the many communication services offered.

6. Implementation

6.1. Resource Development

The JavaScript extension of the srcML infrastructure requires further development of the srcML infrastructure to be completely implemented. This development consists of the creation of a grammar parser that will utilize the JavaScript grammar file created in the JavaScript extension to translate JavaScript source code to the srcML format and vice versa. The development of this technology is ongoing and slated to be completed within the next 2 years.

Monetary resources have already been allocated through the Community Infrastructure for Research in Computer and Information Science and Engineering grant. This money cannot be reallocated or increased unless the srcML sells a license to a private company allowing the use of the srcML infrastructure in their product, however, this likely won't occur.

Materials promoting the srcML infrastructure JavaScript extension will need to be developed. The srcML team has frequently created pins, stickers, and other similar items to promote the srcML infrastructure, so these same resources and methods can be used to create new promotional items for the JavaScript extension. Talks performed at conferences to promote the srcML infrastructure will be modified to advertise the JavaScript extension.

6.2. Market Offering Development

The JavaScript extension has already been developed. This development took place over the course of 5 months from May 2023 to September of that year. Testing files have been created to ensure the integrity of the extension. These tests will be conducted once the technology implementing the JavaScript extension has been completed. Further development of the JavaScript extension may be conducted based on the outcome of these tests.

6.3. Commercial Deployment

We plan to launch the JavaScript extension to the srcML infrastructure alongside the grammar parser for the software. Both offerings will be distributed in the same software package available for download on the srcML website. We also plan to begin distributing the promotional materials prior to the release of the JavaScript extension. The material promoting the JavaScript extension will be given out and the speeches promoting it will be delivered at conventions in the months leading up to the release of the extension to promote community awareness and engagement.

7. Control

7.1 Performance Evaluation

We plan to evaluate the success of the JavaScript extension largely by the response received from the srcML userbase. We plan to monitor the community communication services for feedback from the srcML infrastructure users so that we can continue to improve our software offerings. In addition, we plan to utilize the following metrics to monitor the performance of the JavaScript extension:

- New Users measured by the number of srcML infrastructure downloads per month.
- Website traffic measured by the number of visitors to the srcML website per month.
- Number of attendees at srcML events held at conferences.
- Number of collaborators brought into the srcML team per month.

This performance evaluation will be conducted by the srcML team outreach chair. This informal position is currently held by Dr. Collard but may change in the future. The responsibility for conducting performance evaluations may be delegated to another member of the srcML team.

7.2 Analysis of the Environment

We will be continuously monitoring for changes in the environment, including the following:

- Changes in customer needs and preferences.
- Changes in internal structure, resources, and specialties.
- Changes in industry trends and preferences.
- Changes in competing software.
- Changes in economic, business, social and cultural, technological, regulatory, and physical context in which the srcML team operates.

To ensure that the srcML team is aware of and prepared for these environmental changes, we plan to conduct community polls and surveys, examine environmental data as it is published, and continue to attend professional computer science and software development conferences.

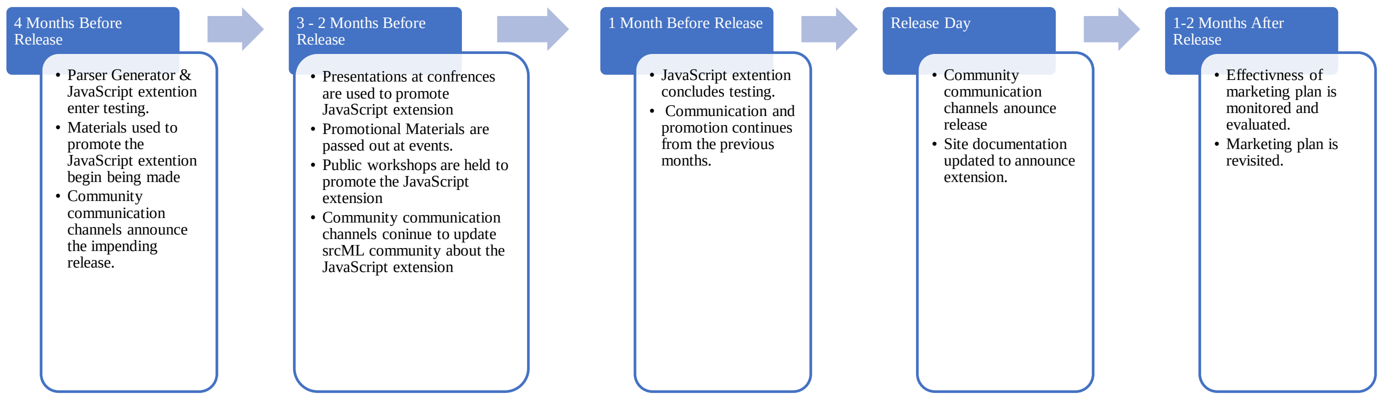
4 Months Before Release
• Parser Generator & JavaScript extention enter testing.
• Materials used to promote the JavaScript extention begin being made
• Community communication channels announce the impending release.
3 - 2 Months Before Release
• Presentations at confrences are used to promote JavaScript extension
• Promotional Materials are passed out at events.
• Public workshops are held to promote the JavaScript extension
• Community communication channels coninue to update srcML community about the JavaScript extension
1 Month Before Release
• JavaScript extention concludes testing.
• Communication and promotion continues from the previous months.
Release Day
• Community communication channels anounce release
• Site documentation updated to announce extension.
1-2 Months After Release
• Effectivness of marketing plan is monitored and evaluated.
• Marketing plan is revisited.

Figure: Tentative visual timeline of marketing plan implementation.