BACKGROUND REPORT: SugaryJS

Allowing Developers to Write their Own Syntactic Sugars for JavaScript

Chen, Susan
o1t0b@ugrad.cs.ubc.ca

Gzik, Gregory
k8o0b@ugrad.cs.ubc.ca

Lin, Hannah
g5m0b@ugrad.cs.ubc.ca

Pan, Haozhe
n6e2b@ugrad.cs.ubc.ca

1. INTRODUCTION/OVERVIEW

Open source APIs are changing how developers write production code, as developers no longer need to "reinvent the wheel" when working on their projects. These programs provide developers more flexibility, agility, and speed. However, open source APIs are limited in their scope by the syntax of the language itself. For example, async/await in JavaScript (JS) was limited by how the async library required the use of function calls found in prior JavaScript versions rather than the natural async/await syntax introduced in the more recent ES6 version of JS. Async/await is a feature that allows asynchronous functions to be structured in a way similar to synchronous functions. If open source components had the option to extend the language's syntax to create their own syntactic sugars, developers would no longer need to wait for the language to achieve compatibility on a syntactic level to receive the full range of benefits. We propose the creation of a mechanism to allow JS developers to extend the JS EBNF, so that they can use their own syntactic sugars in their JS files. We call this mechanism SugaryJS.

2. BACKGROUND AND SIGNIFICANCE

Syntactic sugar, which is the syntax of a programming language that makes code easier to read and understand by humans without changing its semantic meaning, has numerous benefits. It increases the human readability of code and makes development easier by making code easier to debug without changing how the program behaves [1].

However, syntactic sugar is not without drawbacks. Although it can simplify code for the developers, it can complicate how the language is to be used and understood [1]. For example, C programmers sometimes use `a[i]` instead of `*(a+i)` to access arrays. For a beginner, the sugared form of array access does not foster a understanding of how memory in C works. With syntactic sugar, developers may be able to use the language faster, but understanding the underlying code requires additional learning time. Syntactic sugar also extends the length and complexity of the language documentation, which requires greater effort to maintain throughout the language or API lifecycle [1].

Most programming languages today use syntactic sugar in some form, but these syntactic forms are may not be shared between languages. As a result, developers grow familiar with the sugars in the languages they typically program in. However, open source APIs that can be used to accelerate development are not guaranteed to be in written in the languages with which developers are most comfortable. In these situations, lack of familiar syntactic sugars can become an obstacle to learning and using some APIs. SugaryJS can be used to port syntactic sugars from other languages to JavaScript, reducing this learning curve.

JavaScript is the most widely used client-side scripting language and is supported by all web browsers [2]. It allows for creation of dynamic web pages across different platforms and devices and seamless integration with HTML and CSS [3]. The importance and versatility of JavaScript is highlighted by the varying transpilers, libraries,

and frameworks that exist to extend the language and make it easier for developers to accomplish specific tasks [3]. SugaryJS gives its users the ability to enhance this fundamental language to suit their requirements by overcoming the syntactic limitations that may be faced by developers.

3. POTENTIAL PROJECT

3.1 Features

SugaryJS gives developers the ability to define their own syntactic sugar by creating an extension to the JavaScript EBNF. By utilising SugaryJS, developers can use their preferred syntax from other languages while programming in JavaScript. Potential syntactic sugars users could add to JavaScript via SugaryJS include:

Sublist slice using ':' from Python:

```
1. var a = [1, 1, 2, 3, 5, 7, 13, 21,
34]
2. var b = a[3:7]; // [3, 5, 7, 13, 21]
3. var c = a[:4]; // [1, 1, 2, 3]
4. var d = a[5:]; // [7, 13, 21, 34]
```

Array repetition using '*' from Python:

```
1. var a = [1];
2. var b = a * 4;
// [1, 1, 1, 1]
```

The null coalescing operator using '??' from C#:

```
1. var a = 1;
2. var b = a ?? -1
// b = (a != null) ? a : -1
```

In addition to implementing the syntactic sugars from other languages, developers could use SugaryJS to extend the JavaScript EBNF to implement their own syntactic innovation. However, we provide these examples of porting syntactic sugar from other languages to make it easier to reason about what SugaryJS can do.

3.2 Usage

The SugaryJS package will be downloadable via JavaScript's package manager npm [4]. Developers will write their intended sugared JavaScript in an extended JS file (*.ejs). They must also define their own syntactic sugars by creating an EBNF extension in a JavaScript EBNF file (*.ebnf.js). This file will also contain the desugared JavaScript form for their syntax addition. SugaryJS will be invoked via the command line, requiring both the *.ejs filename and the *.ebnf.js filename as input. It will output a desugared JavaScript *.js file. SugaryJS will reject any *.ejs files that do not fit the specification defined in the *.ebnf.js file.

Consider the following example of how a user would add Python's sublist slice operator (':') as syntactic sugar to JavaScript. The implementation of this syntax takes advantage of JavaScript's existing array slice() function.

First, developers would write their intended JavaScript, including the syntactic sugar they wanted to use, in an extended JavaScript file (*.ejs). This file may have the following contents:

```
1. var a = [1, 1, 2, 3, 5, 7, 13, 21,
34]
2. var b = a[3:7]; // [3, 5, 7, 13, 21]
3. var c = a[:4]; // [1, 1, 2, 3]
4. var d = a[5:]; // [7, 13, 21, 34]
```

Then, they will need to define their syntactic sugar in a separate EBNF JavaScript file (*.ebnf.js). Each EBNF extension will be represented as a JavaScript object with two fields: syntax and semantics. The syntax field is a string which defines the syntax of the sugar via EBNF, which can be seen in lines 2-5 in the code snippet below. The semantics field is a function which takes each non-terminal of the syntax EBNF as an input and returns its desugared form as a string. This function essentially produces the desugared

JavaScript. This process requires that the developer is familiar with the JavaScript EBNF to be able to extend it properly. In our example, the contents of this file appear as follows:

```
1. export const subListOperator = {
2.   syntax: `
3.       <MemberExpression> ::=
<Expression>"["(<Expression>)?":"(<Expressi
on>)?"]"
5.   `,
6.   semantics: function(v, i, n) {
7.       if ((v.constructor !== Array)) {
8.           throw new TypeError("Cannot
use sublist operator on non-Arrays");
9.       }
10.
11.       if (i == null && n == null) {
12.           return `${v}.slice()`;
13.       }
14.       if (i == null) {
15.           return `${v}.slice(0, ${n})`;
16.       }
17.       if (n == null) {
18.           return `${v}.slice(${i})`;
19.       }
20.
21.       return `${v}.slice(${i}, ${n})`;
22.   }
23. }
```

Once the user is ready to transpile their JavaScript from its sugared *.ejs form to its desugared *.js form, they will invoke SugaryJS via command line like so:

```
> sugaryjs *.ebnf.js *.ejs
```

The following output *.js file will be generated. The filename of the output file will correspond to the filename of the *.ejs file provided as input. The following is the output SugaryJS would produce for the sublist slice example:

```
1. var a = [1, 1, 2, 3, 5, 7, 13, 21,
34];
2. var b = a.slice(3,7);
3. var c = a.slice(0,4);
4. var d = a.slice(5);
```

3.3 Implementation

SugaryJS will be built as a plugin for the Babel compiler, which is used by numerous JavaScript frameworks including React, Flow, and TypeScript to transpile their syntax into JavaScript [5]. Babel represents the JavaScript Abstract Syntax Tree (AST) using ESTree, a popular AST implementation described as the "lingua franca for tools that manipulate JavaScript code." [6] Given a valid AST, Babel automatically handles the transpiling of JavaScript code based on its input AST. Therefore, the job of SugaryJS is to parse the *.ejs file into an valid AST that Babel can accept as input, and let Babel handle the transpilation from that AST into JavaScript output code.

More concretely, SugaryJS will transpile *.ejs to *.js by the following four stage process:

1. **Define** - Interprets the EBNF from the syntax field from the EBNF extension object in the *.ebnf.js file to construct an AST. This AST represents both the JavaScript EBNF and the EBNF extension defined by the syntax object.
2. **Parse** - Parses all *.ejs files into the AST constructed in the define stage, rejecting any files whose syntax does not conform to the AST.
3. **Desugar** - Parses and interprets the semantics field from the EBNF extension object in the *.ebnf.js file into the JavaScript AST. All ASTs formed in the parse stage will be desugared into valid JavaScript ASTs using the interpreted semantics object.
4. **Output** - Inputs the translated ASTs from the translate stage into Babel to create the output *.js files.

3.4 The Final Result

In the 100% level implementation of SugaryJS, the four transpilation stages discussed in section 3.3 will be implemented as a Babel plugin. SugaryJS will be released to the open source community via JavaScript's package manager npm [4]. This release will contain multiple examples of syntactic sugars users could implement via SugaryJS, in addition to user documentation describing the step-by-step process for implementing these examples in SugaryJS.

Furthermore, SugaryJS will be able have multiple syntactic sugars defined in the input *.ebnf.js file and be able to report errors for syntactic sugars that have the same AST input from the *.ejs file but different AST output in the *.js file. This final result realizes the vision of giving external Web APIs the ability to define their own syntactic sugars for their libraries and giving developers the ability to use the sugars of multiple external web APIs in their projects.

A stub of the SugaryJS npm package has been set up and can be viewed at the following URL: https://www.npmjs.com/package/sugaryjs.

4. SIMILAR WORK

SugaryJS is not the first attempt to make an existing language easier to use. We have considered the following mechanisms that attempt to address syntactic shortcomings in their respective languages to learn lessons from their implementation. However, there is a key difference between some of these mechanisms and SugaryJS: SugaryJS only transpiles syntactic sugars to the existing JavaScript abstract syntax tree, but some of the examples discussed below implement sugared syntax and in certain cases, new semantics to their respective languages directly. More importantly, users can define their own syntax to suit their preferences.

4.1 Sweet.js

Sweet.js is a JavaScript extension that allows users to bring the hygienic macros of Rust and Scheme to JavaScript [11]. Using the "syntax" keyword, users can generate macro definitions by creating a new variable and binding it to a function definition so that it behaves like a compile-time function [11]. After the macro is defined, it can be invoked like so:

```
syntax hi = function (ctx) {
  return #`console.log('hello,
world!')`;
};
hi
```

Operators can also be defined with Sweet.js using the "operator" keyword. Unlike macros, precedence and associativity of the operator can be defined, such as left/right and prefix/postfix [11].

Similar to SugaryJS, Sweet.js allows users to specify custom syntax in place of more complex or unwieldy expressions in JavaScript. However, Sweet.js is limited by the fact that macro definitions only allow binding of functions to single variables at a time. In addition, infix operators are not supported and operator definitions cannot match arbitrary syntax [11]. SugaryJS aims to overcome this by parsing EBNF to construct an abstract syntax tree, so that new syntactic extensions are not limited by the use of a single operator.

4.2 JSX

JSX provides syntactic sugar that is used in libraries such as React. It is similar to XML and HTML in that it allows users to specify tag names, attributes, and children, and can be run in the browser by transpiling into JavaScript via Babel [7]. JSX expressions allow for embedding of valid JavaScript expressions through use of curly braces, {}, and combines markup and logic in singular components for powerful UI rendering. Babel then compiles JSX to React.createElement() calls, producing elements that represent objects in the

DOM [7]. The following two declarations are identical:

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);


const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

JSX provides syntactic sugar to take advantage of the coupling between rendering logic and business logic such as event handling, state, and display [7]. While JSX is primarily designed for React to build user interfaces, SugaryJS can extend JavaScript for any purpose. In fact, since JSX defines syntactic sugar on top of the existing JavaScript syntax, JSX could be implemented solely via SugaryJS.

4.3 CoffeeScript

CoffeeScript is a language that compiles into JavaScript and aims to expose the strengths of JavaScript by introducing simple syntactic sugars [8]. This language brings together the syntax of Ruby with the utility of JavaScript to make web development easier. Some recognizable features of CoffeeScript include the absence of parentheses, type declarations, and semicolons, giving the code an overall cleaner look.

The idea of simplifying syntax to increase readability is shared by CoffeeScript and SugaryJS. However, unlike SugaryJS, CoffeeScript provides replacement syntax to JavaScript rather than extending the existing syntax. This is both an advantage and a disadvantage. CoffeeScript limits the syntax it accepts from its users, making it easier to learn the syntax rules, but this may also limit the expressibility of the language. Since CoffeeScript's syntax replaces elements of JavaScript's syntax, CoffeeScript could not be solely written as an instance of SugaryJS.

4.4 C++

C++ was developed with the intention of making the existing language of C more efficient and elegant [9]. C is a popular low-level, procedural programming language known for its speed and portability. However, it lacks object-oriented programming and has weak type checking and data abstraction. The C++ programming language was developed for the purpose of adding these features to C [9]. Bjarne Stroustrup initially added features such as classes, inheritance, and strong type checking to C, creating a language he called C with Classes [9]. This language eventually became C++, which had even more features than C with Classes, including function overloading, references with the & symbol, and the const keyword. The modifications that C++ made to the C language gave developers the ability to program both at a high-level and low-level while maintaining the performance and speed of C [9].

Similar to C++, SugaryJS can make code look more refined. Adding syntactic sugar can also make the language easier to understand by users less familiar with Javascript. C++, however, in addition to providing notational support, added new behaviour that is not available in C. SugaryJS only allows for the addition of syntactic sugar to Javascript and is unable to change the semantics of the language.

5. LIMITATIONS

A major limitation to SugaryJS is that it only allows users to extend the syntax without overwriting the existing JavaScript EBNF. This means users cannot fundamentally change the original JavaScript syntax. SugaryJS can only append non-conflicting syntax that can be interpreted in terms of JavaScript's existing

semantics. In other words, SugaryJS cannot fix all of JavaScript's problems such as variable hoisting, lack of an integer type, and lack of proper implicit tail calls [10].

In addition, the use of SugaryJS requires developers to provide additional "code" in the form of EBNF extensions and to learn how to use the tool itself. Providing these EBNF extensions means that developers are required to familiarize themselves with the ESTree implementation of JavaScript's abstract syntax tree. However, we argue that this additional cost is offset by its ability to give developers the ability to use language syntax that is familiar to them while taking advantage of the power of open source APIs to accelerate development.

REFERENCES

[1] Kristopher Sandoval. 2016. Sweet API - Syntactic Sugar and You. Retrieved November 5, 2018 from https://nordicapis.com/syntactic-sugar-apis/

[2] TechArk Solutions. 2014. Importance Of JavaScript. Retrieved November 7, 2018 from https://gotechark.com/blog/importance-javascript

[3] Mindfire Solutions. 2017. How important is JavaScript for Modern Web Developers? Retrieved November 6, 2018 from https://medium.com/@mindfiresolutions.usa/how-important-is-javascript-for-modern-web-developers-2854309b9f52

[4] npm. 2018. What is npm? Retrieved November 5, 2018 from https://docs.npmjs.com/getting-started/what-is-npm

[5] Babel. 2018. Babel. Retrieved November 5, 2018 from https://babeljs.io/docs/en/

[6] ESTree. 2017. ESTree Specification. Retrieved November 6, 2018 from https://github.com/estree/estree

[7] Facebook Inc. 2018. Introducing JSX. Retrieved November 5, 2018 from https://reactjs.org/docs/introducing-jsx.html

[8] CoffeeScript. 2018. CoffeeScript. Retrieved November 5, 2018 from https://coffeescript.org/

[9] Bjarne Stroustrup. 2018. Bjarne Stroustrup's FAQ. Retrieved November 5, 2018 from http://www.stroustrup.com/bs_faq.html

[10] Richard Kenneth Eng. 2016. The 10 Things Wrong with JavaScript. Retrieved November 7, 2018 from https://medium.com/javascript-non-grata/the-top-10-things-wrong-with-javascript-58f440d6b3d8

[11] Sweet.js. 2017. Sweet.js - Hygienic Macros for JavaScript. Retrieved November 19, 2018 from https://www.sweetjs.org/