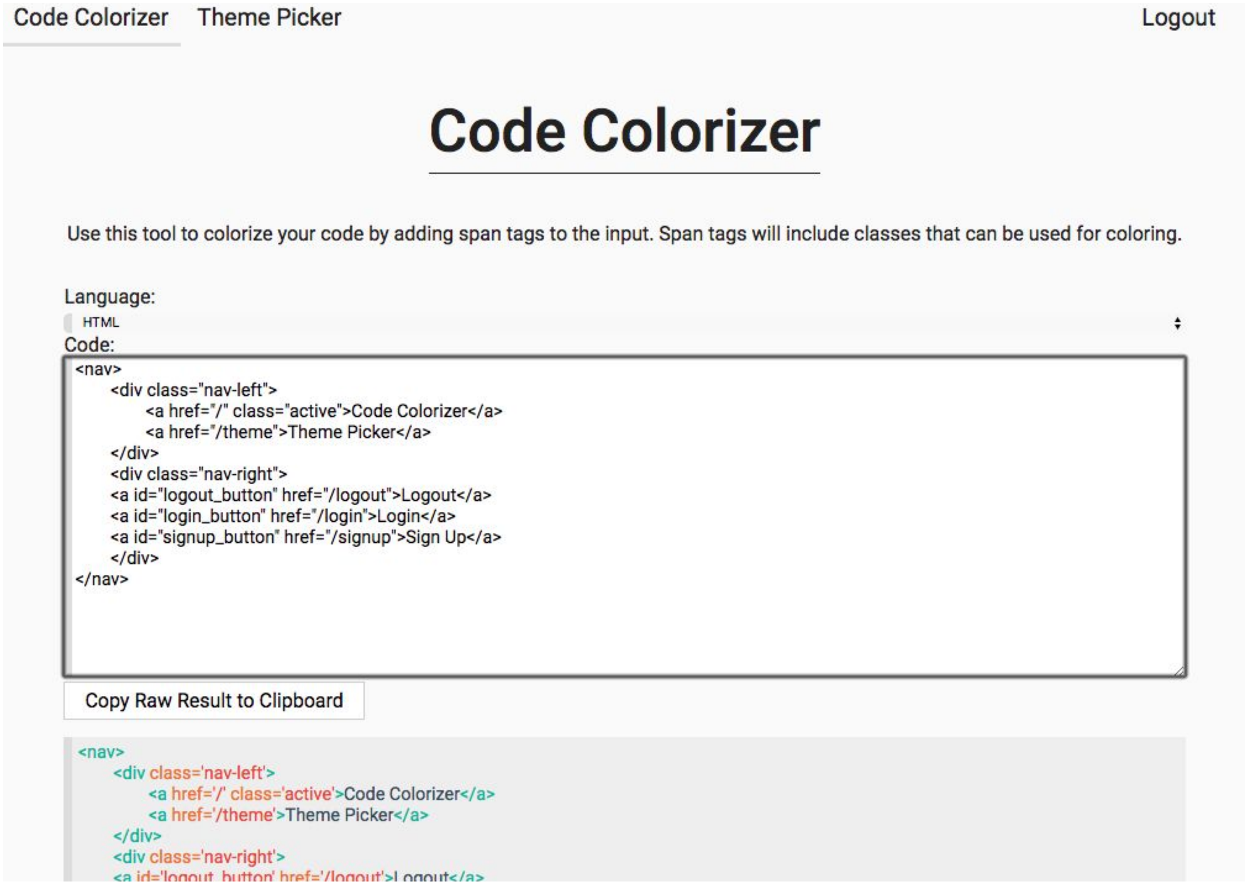
CODE COLORIZER

S E 319 - Portfolio 3



Code Colorizer

Use this tool to colorize your code by adding span tags to the input. Span tags will include classes that can be used for coloring.

Language:
HTML

Code:

```
<nav>
	<div class="nav-left">
		<a href="/" class="active">Code Colorizer</a>
		<a href="/theme">Theme Picker</a>
	</div>
	<div class="nav-right">
	<a id="logout_button" href="/logout">Logout</a>
	<a id="login_button" href="/login">Login</a>
	<a id="signup_button" href="/signup">Sign Up</a>
	</div>
</nav>
```

```
<nav>
	<div class='nav-left'>
		<a href='/' class='active'>Code Colorizer</a>
		<a href='/theme'>Theme Picker</a>
	</div>
	<div class='nav-right'>
	<a id='logout_button' href='/logout'>Logout</a>
```

Zach Newton, Annie Steenson

12.12.2016
Iowa State University

OVERVIEW

We created a Node.js application that does syntax highlighting using PEG.js, for text parsing we learned in class, Express.JS for HTTP request routing, basic HTML for DOM structure, CSS for aesthetics and the actual text highlighting, JavaScript for a dynamic User Interface, and Firebase for database authentication.

WHAT WE MADE

We made Code Colorizer and Theme Picker with the goal of making documentation easier to write as well as easier to read. To accomplish this, we used PEG.js, NodeJS, and JScolor alongside the standard web development tools that are HTML, CSS, and JavaScript. Ironically, we did not write a documentation page for the site, but that is partly because there is no download of the Parsers, yet. We decided to use PEG.js because it allowed for the generation of JavaScript parsers more easily than ANTLR.

WHY WE MADE IT

When we were thinking of the web development world and the lessons learned at Iowa State University in both Com S 309 and 319, we analysed the needs of today's developers. Upon evaluating the current state of code documentation for projects in industry, it appeared that only the largest of projects and products had decent, legible documentation. It was then that we arrived at the conclusion that we must create an easy to use code documentation application to better enable developers to write what is an absolute essential piece of maintainable code..

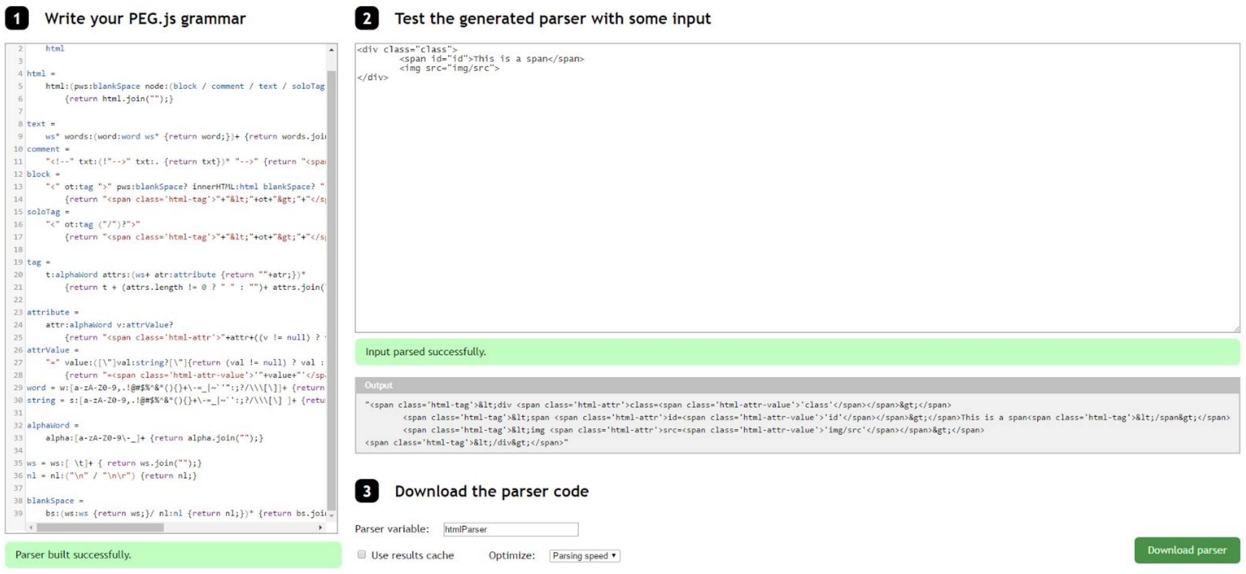
This code colorizing application is needed in industry because documentation is really not a lot of fun to write, but then when you look back on said documentation, if it doesn't look good, there is less incentive to write documentation next time. We thought that by giving developers the tools to make their documentation look good but maintain the ability for each documentation theme to be unique, we could increase incentive to document projects by giving developers the possibility to be proud of their documentation.

NEW AND COMPLEX

The new and complex elements of our project include the use of JScolor for theme styling, PEG.js to generate JavaScript parsers, and NodeJS to run the server side.

CODE COLORIZER

On the front-end, the most complex feature is the code parsing. In order to accomplish front-end parsing without the need for a server call, which would most likely slow things down for the user's experience, Parser Generator for Javascript (PEG.js) was used. There are a multitude of ways that PEG can be used, from NodeJS to client side JS, but we used the online version at https://pegjs.org/online in order to write grammars with instantaneous parser generating and subsequent input testing. The error output, both for the parser generation and parsing, was extremely easy to use in comparison to ANTLR which was an added benefit of using the online version.



1 Write your PEG.js grammar

```
    html

html =
    html:(pws:blankSpace node:(block / comment / text / soloTag
        {return html.join("");}

text =
    ws* words:(word:word ws* {return word;})+ {return words.joi
comment =
    "<!--" txt:(!"-->" txt:. {return txt})* "-->" {return "<spa
block =
    "<" ot:tag ">" pws:blankSpace? innerHTML:html blankSpace? "
        {return "<span class='html-tag'>"+"&lt;"+ot+"&gt;"+"</s
soloTag =
    "<" ot:tag ("/")?">"
        {return "<span class='html-tag'>"+"&lt;"+ot+"&gt;"+"</s

tag =
    t:alphaWord attrs:(ws+ atr:attribute {return ""+atr;})*
        {return t + (attrs.length != 0 ? " " : "")+ attrs.join(

attribute =
    attr:alphaWord v:attrValue?
        {return "<span class='html-attr'>"+attr+((v != null) ?
attrValue =
    "=" value:([\"]val:string?[\"]{return (val != null) ? val :
        {return "=<span class='html-attr-value'>'"+value+"'</sp
word = w:[a-zA-Z0-9,.!@#$%^&*(){}+\-=_|~`'":;?/\\\[\]]+ {return
string = s:[a-zA-Z0-9,.!@#$%^&*(){}+\-=_|~`':;?/\\\[\] ]+ {retu

alphaWord =
    alpha:[a-zA-Z0-9\-_]+ {return alpha.join("");}

ws = ws:[ \t]+ { return ws.join("");}
nl = nl:("\n" / "\n\r") {return nl;}

blankSpace =
    bs:(ws:ws {return ws;}/ nl:nl {return nl;})* {return bs.joi
```

Parser built successfully.

2 Test the generated parser with some input

```
<div class="class">
        <span id="id">This is a span</span>
        <img src="img/src">
</div>
```

Input parsed successfully.

Output

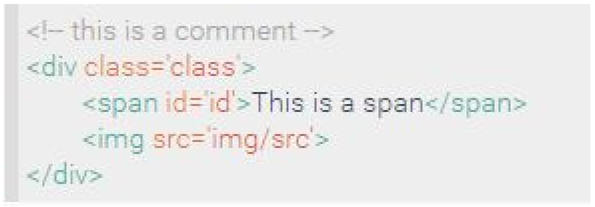
```
"<span class='html-tag'>&lt;div <span class='html-attr'>class=<span class='html-attr-value'>'class'</span></span>&gt;</span>
        <span class='html-tag'>&lt;span <span class='html-attr'>id=<span class='html-attr-value'>'id'</span></span>&gt;</span>This is a span<span class='html-tag'>&lt;/span&gt;</span>
        <span class='html-tag'>&lt;img <span class='html-attr'>src=<span class='html-attr-value'>'img/src'</span></span>&gt;</span>
<span class='html-tag'>&lt;/div&gt;</span>"
```

3 Download the parser code

Parser variable: htmlParser

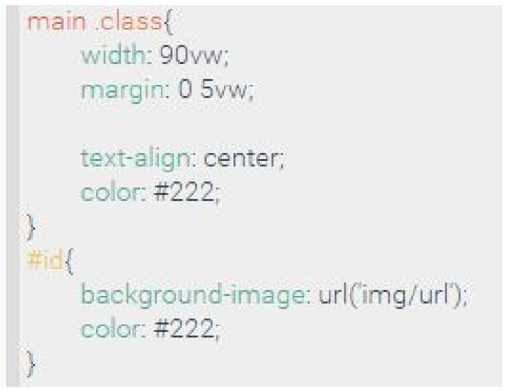
Use results cache   Optimize: Parsing speed

After spending a lot of time figuring out the ins and outs, do's and don't's of PEG.js, we got the HTML Parser up and running.

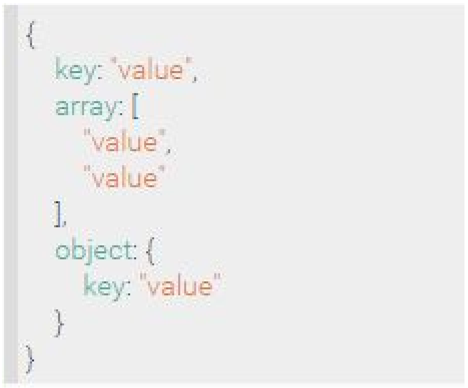


```
<!-- this is a comment -->
<div class='class'>
        <span id='id'>This is a span</span>
        <img src='img/src'>
</div>
```

From there, it was a relatively straightforward matter to make the CSS and JSON parsers. There were a lot of little hang ups in the process due to nuances in the PEG.js syntax that kept appearing, but for the most part it went pretty smoothly.

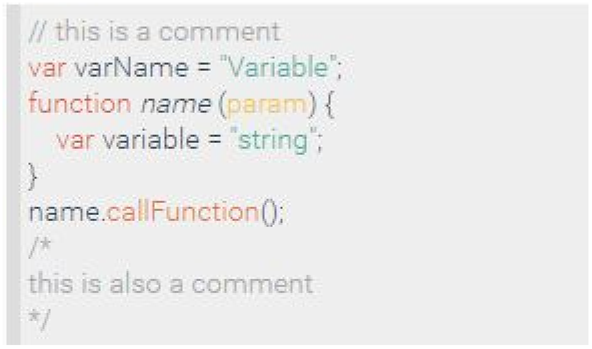


```
main .class{
      width: 90vw;
      margin: 0 5vw;

      text-align: center;
      color: #222;
}
#id{
      background-image: url('img/url');
      color: #222;
}
```



```
{
   key: "value",
   array: [
      "value",
      "value"
   ],
   object: {
      key: "value"
   }
}
```

Then came trying to parse JavaScript. At first it seemed simple enough; we defined the rules for variable declarations, function calls, function definitions, different data types (array, object literal, string, number, etc.), and comments. However, after those first rules were finished, we pasted some of our JavaScript into the parser to see how it handled. It did not. We soon discovered that there are a lot of little parts of programming languages that are really hard to account for, and often when trying to account for something, something else broke. In the end, we reverted back to the simplified JavaScript parser that doesn’t properly handle “if-else if-else” blocks, “try-catch” blocks, “switch-case” statements, and so on.



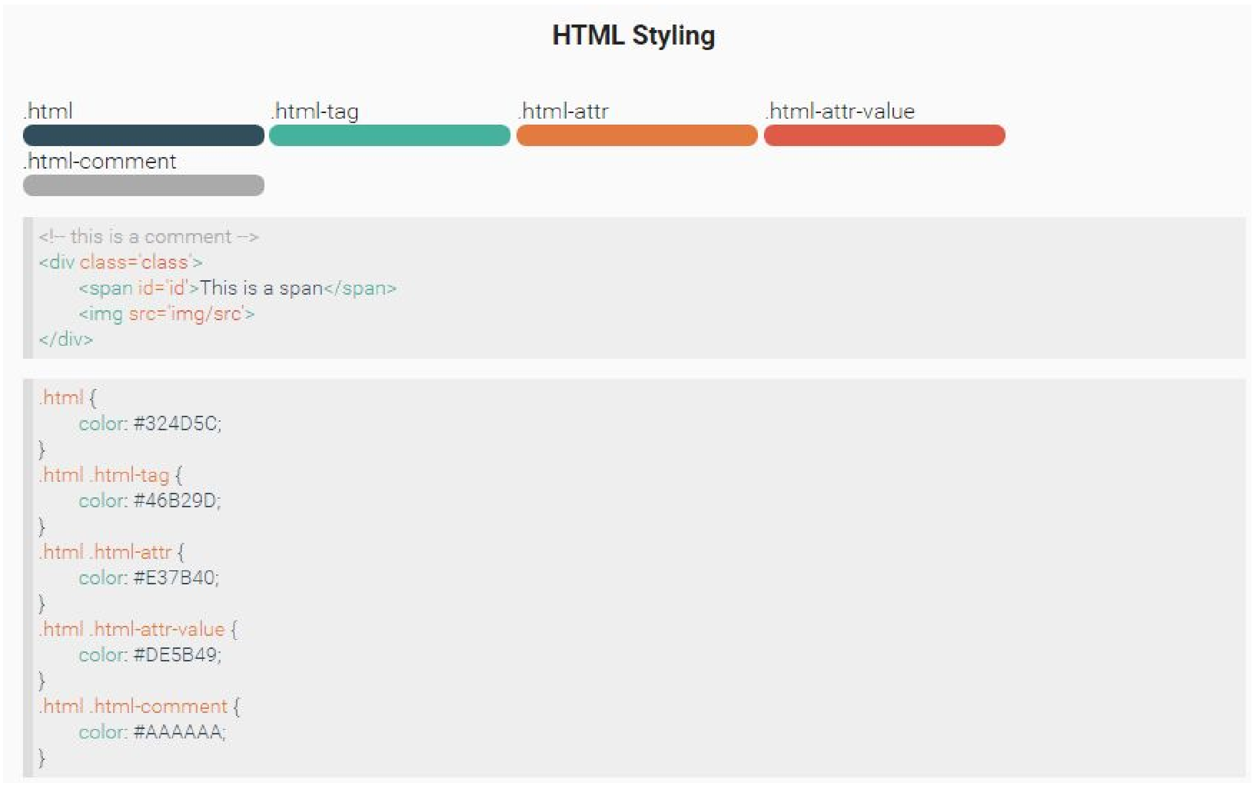
```
// this is a comment
var varName = "Variable";
function name (param) {
   var variable = "string";
}
name.callFunction();
/*
this is also a comment
*/
```

This major roadblock completely derailed our initial hope to have a full code colorizer for both JavaScript and PHP, hence the disabled PHP parsing option on the site. We hope to continue this endeavor in the future as it could prove to be a very useful tool.

Despite the initial plan not coming to full fruition (yet), we also made the parsers usable which is key when developing for the web. Doing so was surprisingly straightforward using the online version of PEG.js. We were able to plug the grammar files into the online version of PEG.js, and download the fully functional, standalone JavaScript file for each. From there, it was a simple matter of updating the DOM with the parsed code on input from the user, with the parser being used depending on the value of the drop down select.

THEME PICKER

The next fun step in making a fully-fledged documentation beautifier was giving the user the ability to quickly and easily customize the color scheme(s) for said documentation. To accomplish this task, we utilized our previously made parser, along with JScolor (http://jscolor.com/) to make a responsive and visually pleasing color picking interface.



```
<!-- this is a comment -->
<div class='class'>
      <span id='id'>This is a span</span>
      <img src='img/src'>
</div>
```

```
.html {
      color: #324D5C;
}
.html .html-tag {
      color: #46B29D;
}
.html .html-attr {
      color: #E37B40;
}
.html .html-attr-value {
      color: #DE5B49;
}
.html .html-comment {
      color: #AAAAAA;
}
```

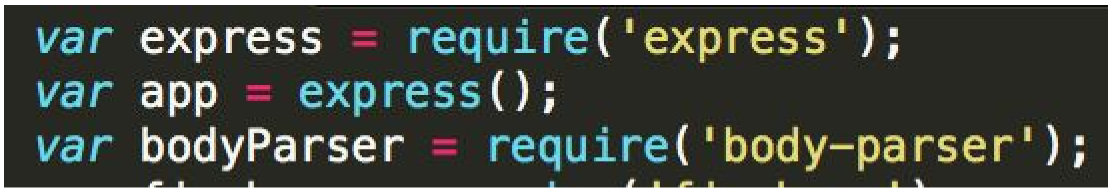
The ability for JScolor form inputs to return values in HEX codes was a key factor in the choice to use it for the forms, because it allowed us to plug those hex values directly into the styles for that block. In order to give the best visualization of the changes being made, we used each language's corresponding parser on it's respective sample block, then used the CSS Parser on the “scoped” style block for each section to output a visualization of the CSS that would be copied to the clipboard with real-time changing color codes.

NODE JS

Neither of us have worked with Node.JS before. We wanted to experiment using JavaScript on the server-side. Node.js uses event-driven, non-blocking I/O model that making data-intensive real-time applications lightweight and efficient. Node.js is open source and there are countless modules for just about anything you could think of.

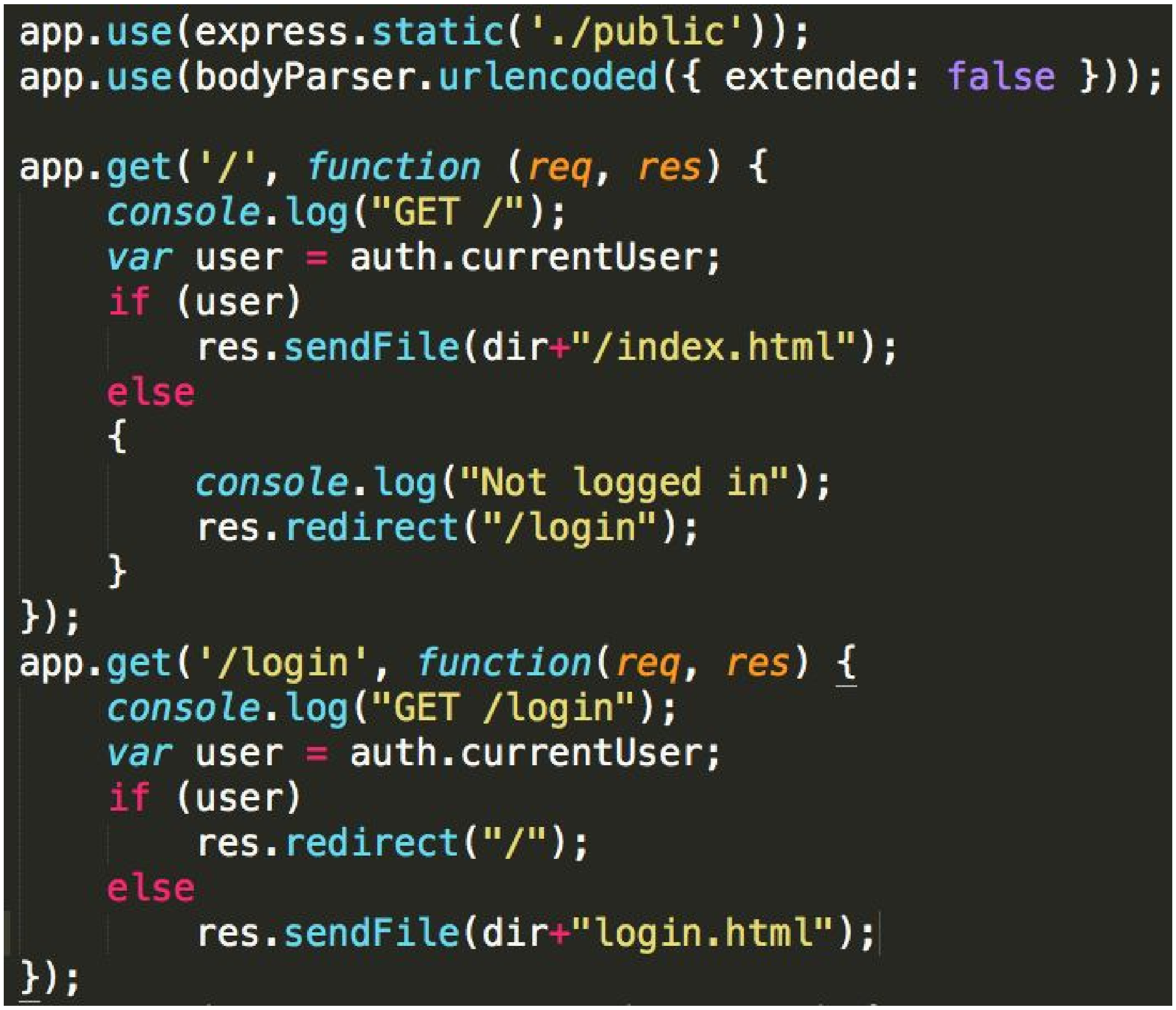
We began by initializing a Node.js project with the command: “npm init” Npm stands for node package manager. With npm, we could install any node module through the command line.

New to Node, we started our learning by looking for a Node module for http request routing. Node comes with a default http package, but we wanted something more robust to create seamless routing. After googling, we came by Express.js. Express organizes your web application into an Model-View-Controller architecture. In order to extract the body of the HTTP request, we use Node module body-parser.



```
var express = require('express');
var app = express();
var bodyParser = require('body-parser');
```

Here is an example of a route configured using Express and body-parser.



```
app.use(express.static('./public'));
app.use(bodyParser.urlencoded({ extended: false }));

app.get('/', function (req, res) {
    console.log("GET /");
    var user = auth.currentUser;
    if (user)
        res.sendFile(dir+"/index.html");
    else
    {
        console.log("Not logged in");
        res.redirect("/login");
    }
});
app.get('/login', function(req, res) {
    console.log("GET /login");
    var user = auth.currentUser;
    if (user)
        res.redirect("/");
    else
        res.sendFile(dir+"login.html");
});
```

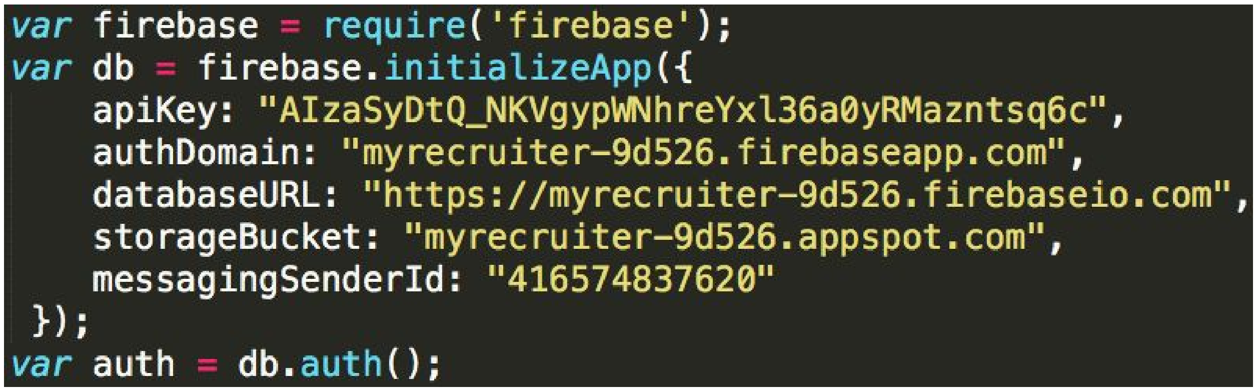
Using Express proved harder than the code looks. We didn't understand exactly how HTML files we're suppose to be served, then realized we had to add in the "app.use(express.static('./public/')" line. Trying to get the request body was extremely confusing and difficult, before we brought in body-parser.

FIREBASE

Only familiar with MySQL, we wanted to experiment with a different type of database. Google Firebase is a real-time cloud-hosted database. Data is stored as JSON and synchronized in real-time to every connected client. This is useful for cross-platform wide user-base application.

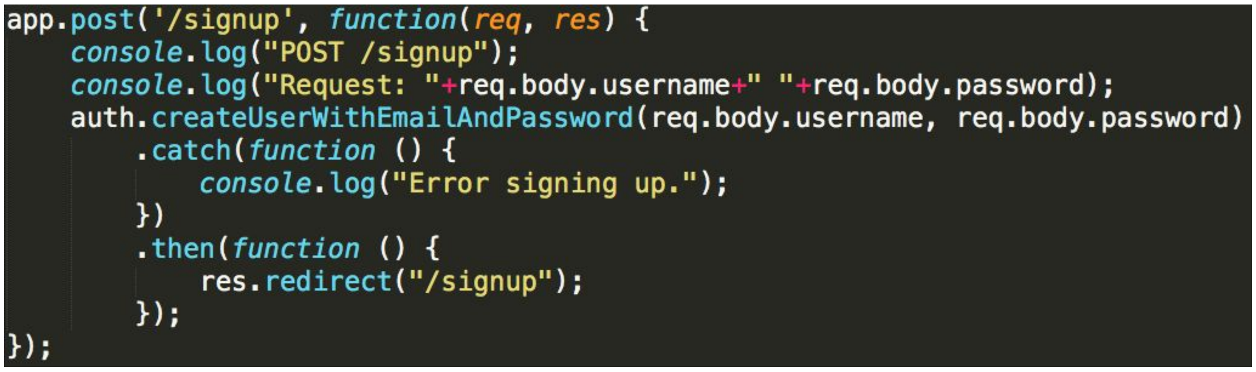
Although, our application isn't a wide use-base application, we still wanted to get our feet wet, so we used Firebase for authentication.

Bringing Firebase into the stack was relatively easy since Google provides a Firebase Node module. We had to register the application online in the Firebase console to receive an API key and other database configuration information.



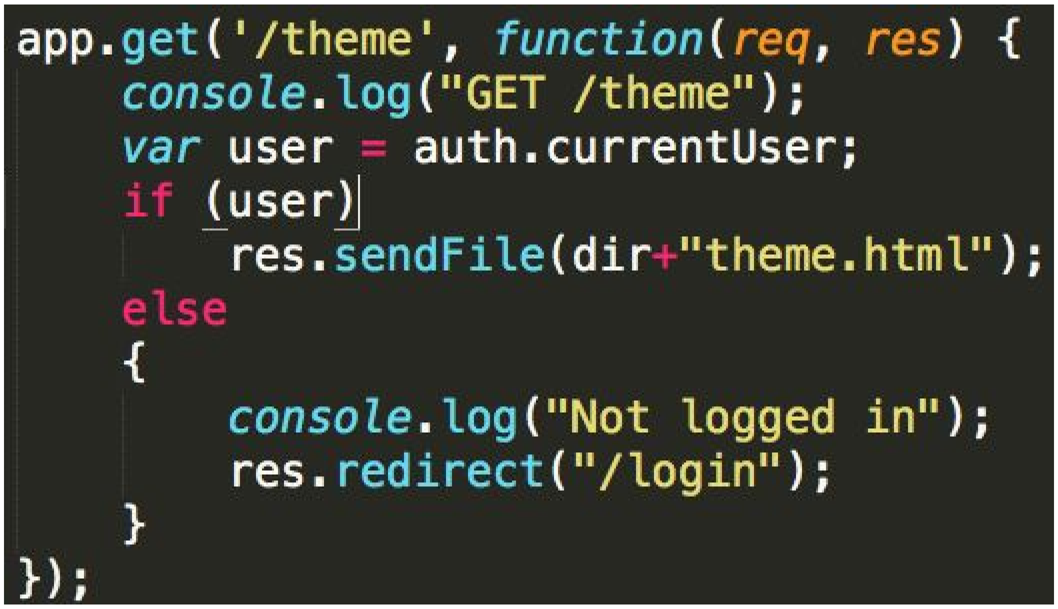
```
var firebase = require('firebase');
var db = firebase.initializeApp({
    apiKey: "AIzaSyDtQ_NKVgypWNhreYxl36a0yRMazntsq6c",
    authDomain: "myrecruiter-9d526.firebaseapp.com",
    databaseURL: "https://myrecruiter-9d526.firebaseio.com",
    storageBucket: "myrecruiter-9d526.appspot.com",
    messagingSenderId: "416574837620"
 });
var auth = db.auth();
```

After creating a new Express route, we implemented creating a user by using Firebase’s API functions. This function also logs them in.



```
app.post('/signup', function(req, res) {
    console.log("POST /signup");
    console.log("Request: "+req.body.username+" "+req.body.password);
    auth.createUserWithEmailAndPassword(req.body.username, req.body.password)
        .catch(function () {
            console.log("Error signing up.");
        })
        .then(function () {
            res.redirect("/signup");
        });
});
```

Each page checks if the current user is logged in, if the current user isn’t logged in, we redirect them to the login page.

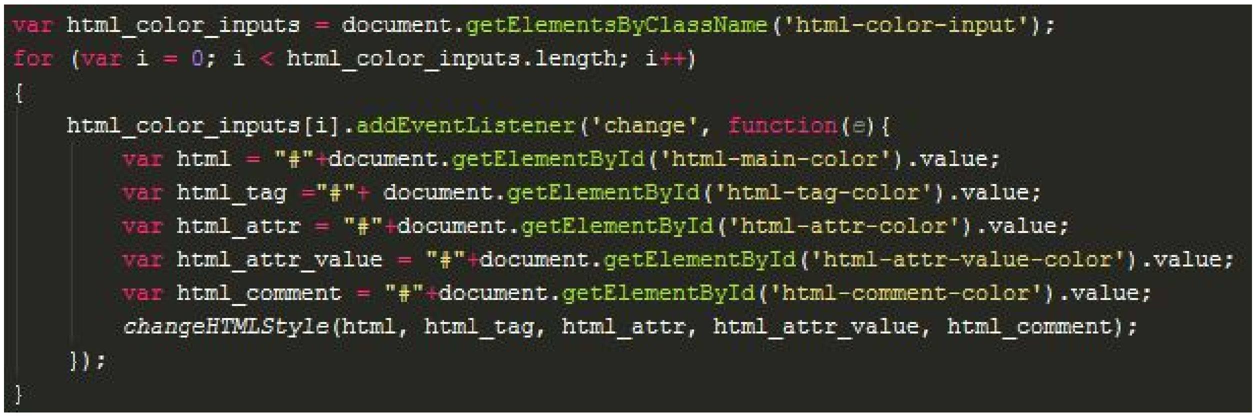


```
app.get('/theme', function(req, res) {
    console.log("GET /theme");
    var user = auth.currentUser;
    if (user)
        res.sendFile(dir+"theme.html");
    else
    {
        console.log("Not logged in");
        res.redirect("/login");
    }
});
```

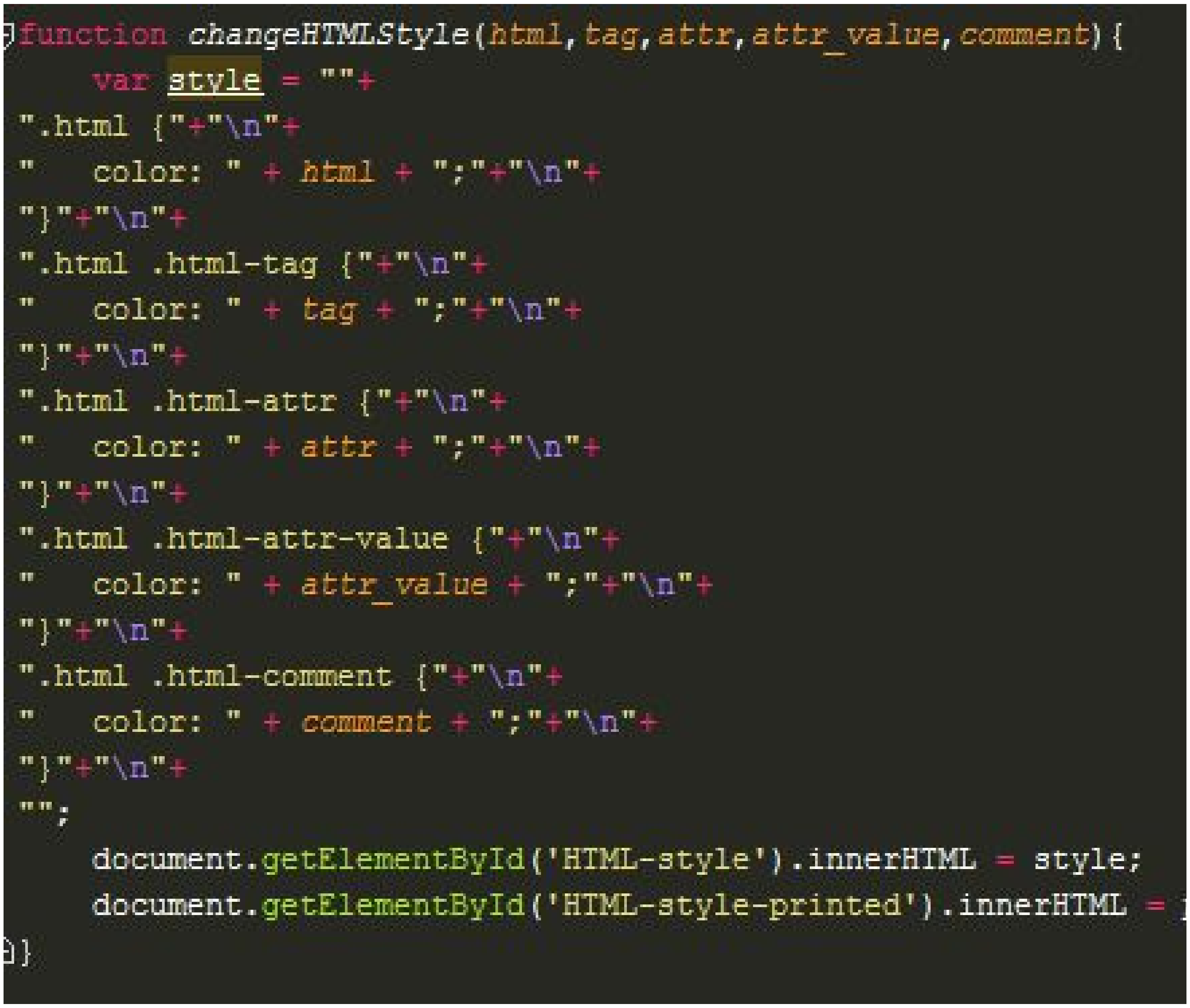
Firebase provided us a relatively easy to use database to integrate into our Node application. Firebase is also a lot more secure than using a simple MySQL server for authentication. Firebase provides encryption algorithms for storing passwords, while MySQL does not.

CODE SNIPPETS

Example of tying form to styles in Theme Picker

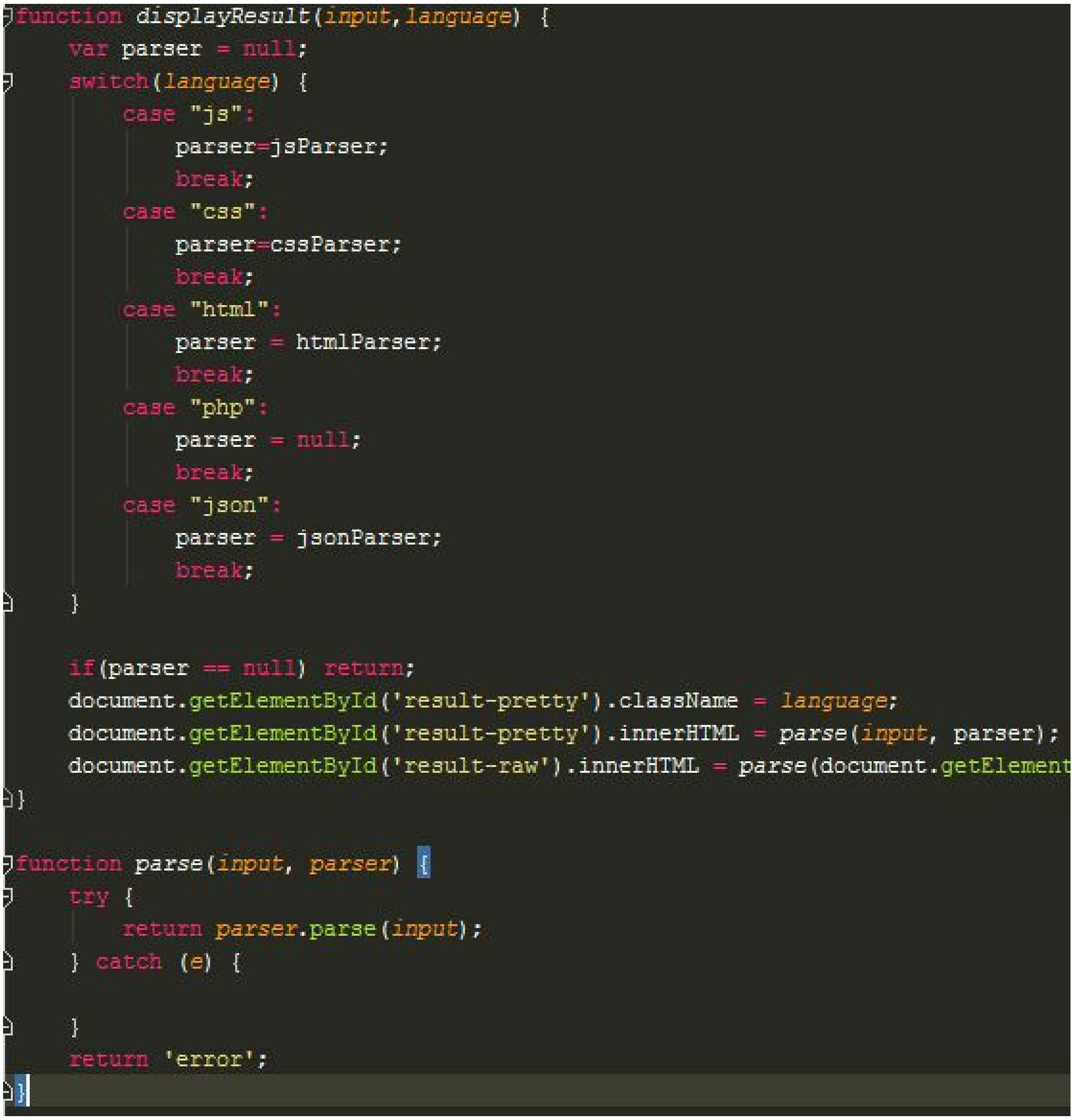


```
var html_color_inputs = document.getElementsByClassName('html-color-input');
for (var i = 0; i < html_color_inputs.length; i++)
{
    html_color_inputs[i].addEventListener('change', function(e){
        var html  = "#"+document.getElementById('html-main-color').value;
        var html_tag ="#"+ document.getElementById('html-tag-color').value;
        var html_attr = "#"+document.getElementById('html-attr-color').value;
        var html_attr_value = "#"+document.getElementById('html-attr-value-color').value;
        var html_comment = "#"+document.getElementById('html-comment-color').value;
        changeHTMLStyle(html, html_tag, html_attr, html_attr_value, html_comment);
    });
}
```



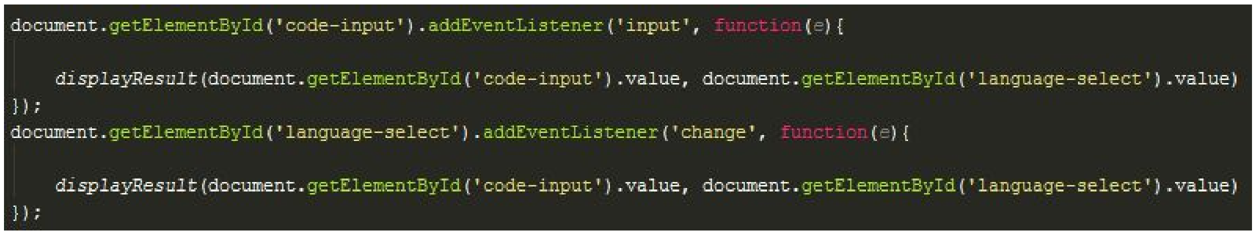
```
function changeHTMLStyle(html,tag,attr,attr_value,comment){
    var style = ""+
".html {"+"\n"+
"   color: " + html + ";"+"\n"+
"}"+"\n"+
".html .html-tag {"+"\n"+
"   color: " + tag + ";"+"\n"+
"}"+"\n"+
".html .html-attr {"+"\n"+
"   color: " + attr + ";"+"\n"+
"}"+"\n"+
".html .html-attr-value {"+"\n"+
"   color: " + attr_value + ";"+"\n"+
"}"+"\n"+
".html .html-comment {"+"\n"+
"   color: " + comment + ";"+"\n"+
"}"+"\n"+
"";
    document.getElementById('HTML-style').innerHTML = style;
    document.getElementById('HTML-style-printed').innerHTML =
}
```

Code for parsing text and displaying the pretty result along with the raw result



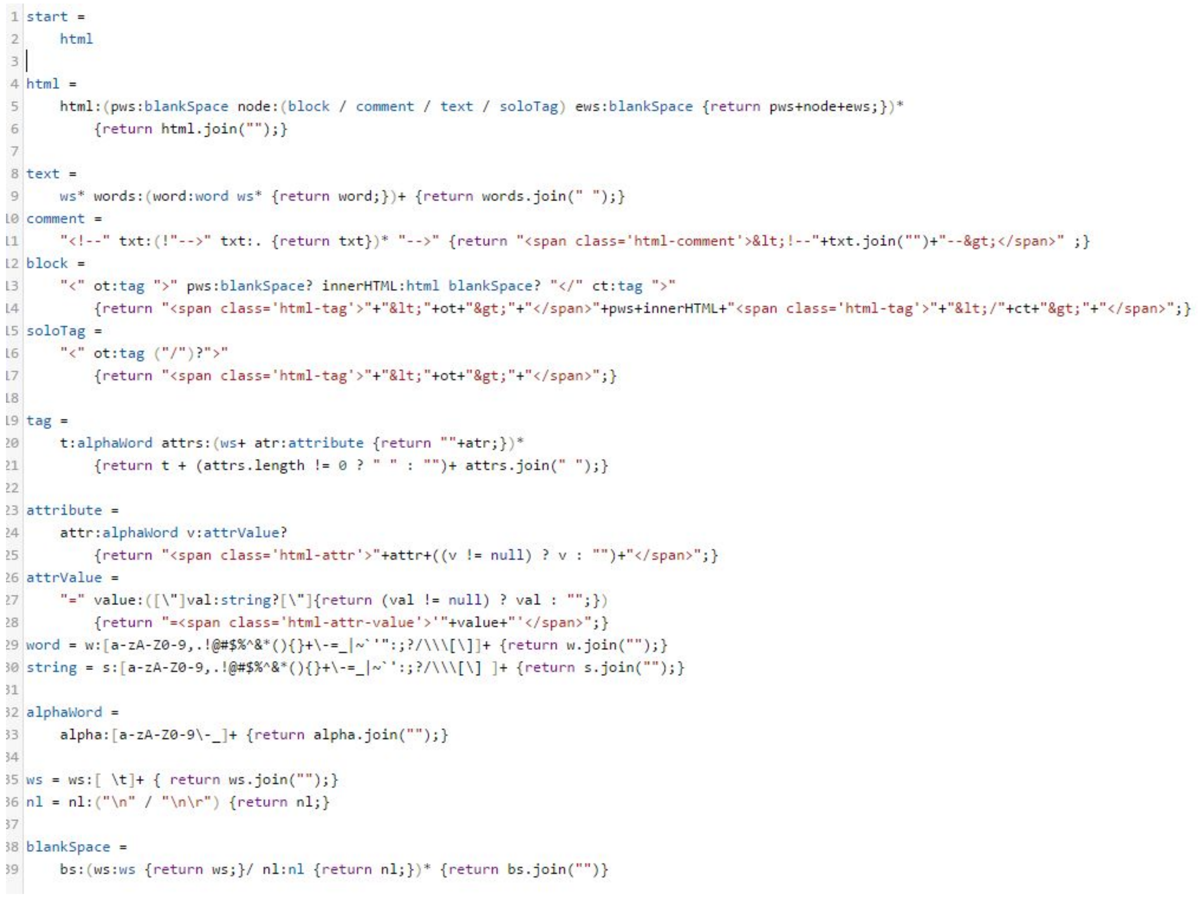
```
function displayResult(input,language) {
    var parser = null;
    switch(language) {
        case "js":
            parser=jsParser;
            break;
        case "css":
            parser=cssParser;
            break;
        case "html":
            parser = htmlParser;
            break;
        case "php":
            parser = null;
            break;
        case "json":
            parser = jsonParser;
            break;
    }

    if(parser == null) return;
    document.getElementById('result-pretty').className = language;
    document.getElementById('result-pretty').innerHTML = parse(input, parser);
    document.getElementById('result-raw').innerHTML = parse(document.getElement
}

function parse(input, parser) {
    try {
        return parser.parse(input);
    } catch (e) {

    }
    return 'error';
}
```

Parsing Code on language selection and code input change



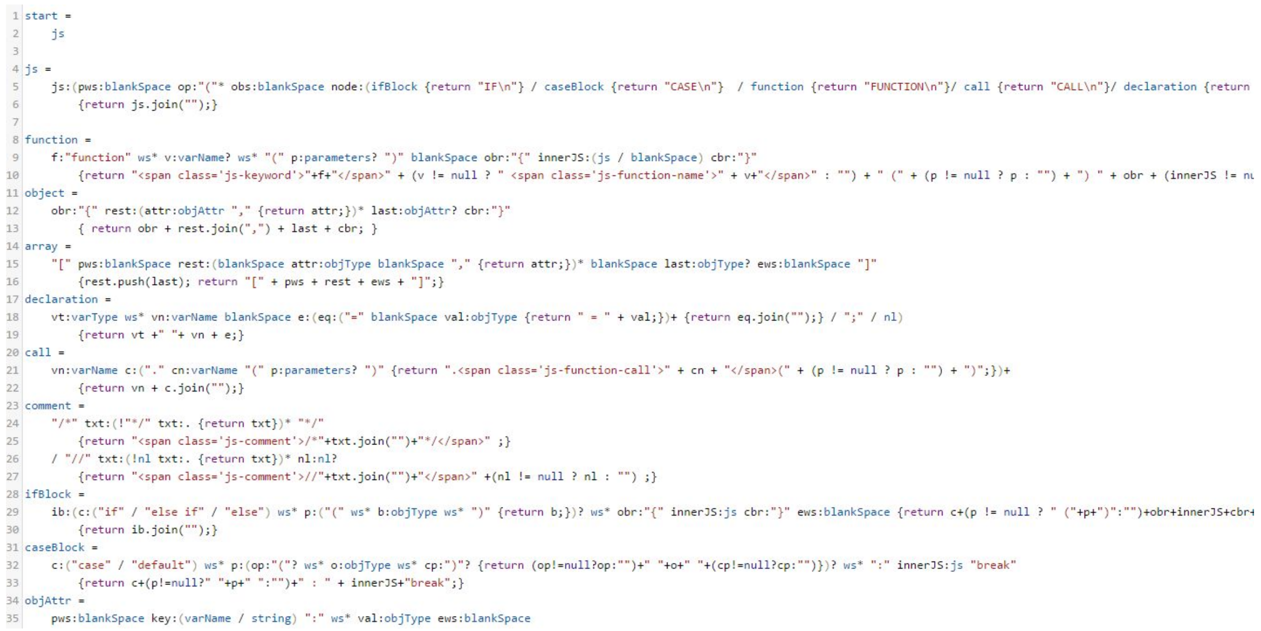
```
document.getElementById('code-input').addEventListener('input', function(e){

    displayResult(document.getElementById('code-input').value, document.getElementById('language-select').value)
});
document.getElementById('language-select').addEventListener('change', function(e){

    displayResult(document.getElementById('code-input').value, document.getElementById('language-select').value)
});
```

HTML Parser Grammar

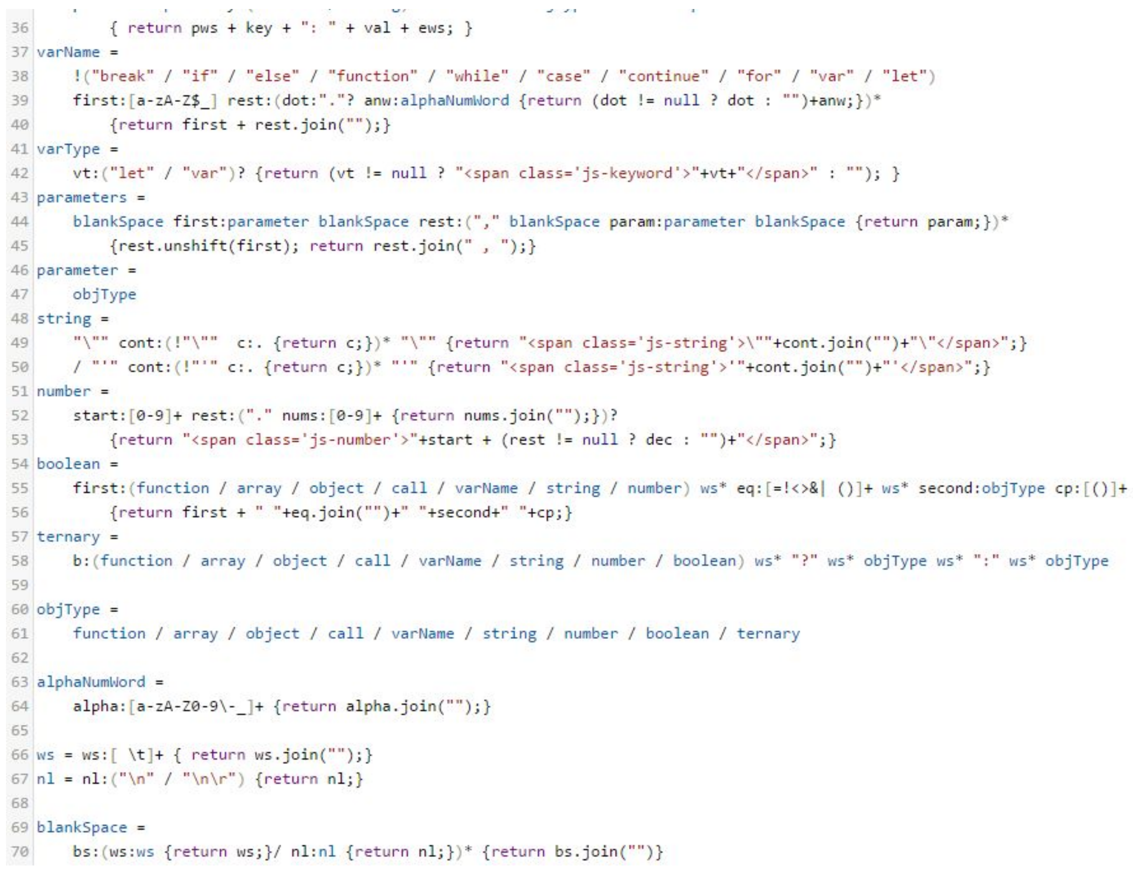


```
start =
    html

html =
    html:(pws:blankSpace node:(block / comment / text / soloTag) ews:blankSpace {return pws+node+ews;})*
        {return html.join("");}

text =
    ws* words:(word:word ws* {return word;})+ {return words.join(" ");}
comment =
    "<!--" txt:(!"-->" txt:. {return txt})* "-->" {return "<span class='html-comment'>&lt;!--"+txt.join("")+"--&gt;</span>" ;}
block =
    "<" ot:tag ">" pws:blankSpace? innerHTML:html blankSpace? "</" ct:tag ">"
        {return "<span class='html-tag'>"+"&lt;"+ot+"&gt;"+"</span>"+pws+innerHTML+"<span class='html-tag'>"+"&lt;/"+ct+"&gt;"+"</span>";}
soloTag =
    "<" ot:tag ("/")?">"
        {return "<span class='html-tag'>"+"&lt;"+ot+"&gt;"+"</span>";}

tag =
    t:alphaWord attrs:(ws+ atr:attribute {return ""+atr;})*
        {return t + (attrs.length != 0 ? " " : "")+ attrs.join(" ");}

attribute =
    attr:alphaWord v:attrValue?
        {return "<span class='html-attr'>"+attr+((v != null) ? v : "")+"</span>";}
attrValue =
    "=" value:([\"]val:string?[\"]{return (val != null) ? val : "";})
        {return "=<span class='html-attr-value'>'"+value+"'</span>";}
word = w:[a-zA-Z0-9,.!@#$%^&*(){}+\-=_|~`'":;?/\\\[\]]+ {return w.join("");}
string = s:[a-zA-Z0-9,.!@#$%^&*(){}+\-=_|~`':;?/\\\[\] ]+ {return s.join("");}

alphaWord =
    alpha:[a-zA-Z0-9\-_]+ {return alpha.join("");}

ws = ws:[ \t]+ { return ws.join("");}
nl = nl:("\n" / "\n\r") {return nl;}

blankSpace =
    bs:(ws:ws {return ws;}/ nl:nl {return nl;})* {return bs.join("")}
```

Attempted JS Parser Grammar



```
start =
    js

js =
    js:(pws:blankSpace op:"("* obs:blankSpace node:(ifBlock {return "IF\n"} / caseBlock {return "CASE\n"}  / function {return "FUNCTION\n"}/ call {return "CALL\n"}/ declaration {return
        {return js.join("");}

function =
    f:"function" ws* v:varName? ws* "(" p:parameters? ")" blankSpace obr:"{" innerJS:(js / blankSpace) cbr:"}"
        {return "<span class='js-keyword'>"+f+"</span>" + (v != null ? " <span class='js-function-name'>" + v+"</span>" : "") + " (" + (p != null ? p : "") + ") " + obr + (innerJS != nu
object =
    obr:"{" rest:(attr:objAttr "," {return attr;})* last:objAttr? cbr:"}"
        { return obr + rest.join(",") + last + cbr; }
array =
    "[" pws:blankSpace rest:(blankSpace attr:objType blankSpace "," {return attr;})* blankSpace last:objType? ews:blankSpace "]"
        {rest.push(last); return "[" + pws + rest + ews + "]";}
declaration =
    vt:varType ws* vn:varName blankSpace e:(eq:("=" blankSpace val:objType {return " = " + val;})+ {return eq.join("");} / ";" / nl)
        {return vt +" "+ vn + e;}
call =
    vn:varName c:("." cn:varName "(" p:parameters? ")" {return ".<span class='js-function-call'>" + cn + "</span>(" + (p != null ? p : "") + ")";})+
        {return vn + c.join("");}
comment =
    "/*" txt:(!"*/" txt:. {return txt})* "*/"
        {return "<span class='js-comment'>/*"+txt.join("")+"*/</span>" ;}
    / "//" txt:(!nl txt:. {return txt})* nl:nl?
        {return "<span class='js-comment'>//"+txt.join("")+"</span>" +(nl != null ? nl : "") ;}
ifBlock =
    ib:(c:("if" / "else if" / "else") ws* p:("(" ws* b:objType ws* ")" {return b;})? ws* obr:"{" innerJS:js cbr:"}" ews:blankSpace {return c+(p != null ? " ("+p+")":"")+obr+innerJS+cbr+
        {return ib.join("");}
caseBlock =
    c:("case" / "default") ws* p:(op:"("? ws* o:objType ws* cp:")"? {return (op!=null?op:"")+" "+o+" "+(cp!=null?cp:"")})? ws* ":" innerJS:js "break"
        {return c+(p!=null?" "+p+" ":"")+" : " + innerJS+"break";}
objAttr =
    pws:blankSpace key:(varName / string) ":" ws* val:objType ews:blankSpace
```



```
        { return pws + key + ": " + val + ews; }
varName =
    !("break" / "if" / "else" / "function" / "while" / "case" / "continue" / "for" / "var" / "let")
    first:[a-zA-Z$_] rest:(dot:"."? anw:alphaNumWord {return (dot != null ? dot : "")+anw;})*
        {return first + rest.join("");}
varType =
    vt:("let" / "var")? {return (vt != null ? "<span class='js-keyword'>"+vt+"</span>" : ""); }
parameters =
    blankSpace first:parameter blankSpace rest:("," blankSpace param:parameter blankSpace {return param;})*
        {rest.unshift(first); return rest.join(" , ");}
parameter =
    objType
string =
    "\"" cont:(!"\""  c:. {return c;})* "\"" {return "<span class='js-string'>\""+cont.join("")+"\"</span>";}
    / "'" cont:(!"'" c:. {return c;})* "'" {return "<span class='js-string'>'"+cont.join("")+"'</span>";}
number =
    start:[0-9]+ rest:("." nums:[0-9]+ {return nums.join("");})?
        {return "<span class='js-number'>"+start + (rest != null ? dec : "")+"</span>";}
boolean =
    first:(function / array / object / call / varName / string / number) ws* eq:[=!<>&| ()]+ ws* second:objType cp:[()]+
        {return first + " "+eq.join("")+" "+second+" "+cp;}
ternary =
    b:(function / array / object / call / varName / string / number / boolean) ws* "?" ws* objType ws* ":" ws* objType

objType =
    function / array / object / call / varName / string / number / boolean / ternary

alphaNumWord =
    alpha:[a-zA-Z0-9\-_]+ {return alpha.join("");}

ws = ws:[ \t]+ { return ws.join("");}
nl = nl:("\n" / "\n\r") {return nl;}

blankSpace =
    bs:(ws:ws {return ws;}/ nl:nl {return nl;})* {return bs.join("")}
```

BLOOM’S TAXONOMY

ANALYSIS

We split up the work into two **distinguished** parts: front-end and back-end. Front-end development involved Vanilla JavaScript, HTML, and extensive CSS. The back-end development involved setting up the Node.js Server, configuring Express.js routing, and integrating the Firebase database. Front-end is **different** from the back-end because the front-end is all the code rendered on the client-side and the back-end is all the code rendered on the server-side.

We **experimented** with using Node.js, PEG.js, Express.js, and Firebase. These were all new domains to us and proved less straight-forward than expected. We learned Node.js is **different** from other web frameworks, because it was the first framework to not use a typical scripting language on the server-side. PEG.js was a library we used to help our text parsing. Express.js was more seamless than Node.js’s default http module, because the library provides more functionality, such as handoffs and general http request handler functions. Firebase can be **compared** to MySQL, as both are databases.

EVALUATION

Support: We decided we liked using Node.js over other server-side frameworks because of Node.js wide variety of node modules. Because Node.js is in JavaScript, it’s a lot easier to bring in populate JavaScript libraries into the stack. **Judge:** Rather than using ANTLR, we used PEG.jg because PEG was easier to use for updating the client-side via JavaScript. ANTLR was harder to introduce into the stack. Express.js was better for routing because of it’s wide variety of functionality, rather than Node.js’s default http package. The default http package was very limited and didn’t support body-parser, making decoding HTTP requests a lot harder. **Evaluating** Firebase: Firebase provided a more secure authentication system than MySQL and already had a node module, while with MySQL, we would need to introduce a separate node module for a object relational mapping layer.

CREATION

We **assembled** a web application that parses through text and highlights the syntax of different languages. We **designed** the front to be user friendly and appealing to look at. We **formulated** a stack mainly based on JavaScript because of our Node.js server-side. We **wrote** many lines of JavaScript code in order to use PEG.js to recognize each language. We **created** an Express.js router to serve several HTTP requests.

CHECKLIST

1. To start the server (source files on lilannie@cs319.cs.iastate.edu/CodeColorizer)
    a. Run “npm install”
    b. Then “node server”
    c. Note: make sure you have an internet connection
    d. Note: These instructions are also included in the README.md
2. Go to https://cs319.cs.iastate.edu:3035
3. Create an account or Login
    a. Note: Creating an account will fail, if you:
        i. Do not use an actual email
        ii. Use a password < 6 characters long
        iii. Try to create an account that already exists
    b. You must be logged in to use the website’s main functionality
4. On the “Code Colorizer” page
    a. Use drop down to select a language (HTML, CSS, JSON, JS)
    b. Try parsing each
        i. Note: JavaScript - only simple things, such as variable declarations, functions, work
    c. Click the “Copy Raw Result to Clipboard” to copy the result as shown in the block labeled “Raw.”
        i. Using the HTML Parser, this code can be parsed repeatedly. No real reason to do so, but kinda neat.
5. On the “Theme Picker” page
    a. Try changing the colors of the parsed code by clicking on the color blocks
        i. Notice the real-time change to the colors in the sample code block
        ii. Notice the real-time change to the code in the CSS style block
    b. Click the “Copy <Language> Style to Clipboard” button for the language of your choice
    c. Use that CSS paired with the parsed code from the Code Colorizer page to

make super sweet looking documentation and tutorials!