The following are the two main files for my PL4 submission. Though you'd notice more files on my student account if you take a look, these two are the minimal set for my assignment to work properly. The other files just add effects (colors, layout, etc) but don't have effect on the functionality of my PL4.

index.html

```
<!DOCTYPE html>
<html>
  <head>
      <title>Joe Pea - CSc at CSUS - PL Homework 4</title>
      <meta charset="utf-8" />
      <script src=" http://code.jquery.com/jquery-1.9.1.js"></script>
      <link href='http://fonts.googleapis.com/css?family=Josefin+Sans:400,700'
rel='stylesheet' type='text/css' />
      <meta http-equiv="Expires" content="Tue, 01 Jan 1995 12:12:12 GMT" />
      <meta http-equiv="Pragma" content="no-cache" />

      <link href='./x.css' rel='stylesheet' type='text/css' />

  </head>
  <body>
      <div id="page">
      <div id="joePea" class="lighterText rotater" style="font-size: 0.3em;">
            <span id="name"><a href="../password/">&lt;&lt; Go Back</a></span>
      </div>
      <div id="more" class="lighterText">
            <h2><span>PL Assignment 4 - Recursive Descent Parser</span></h2>

            <div class="in">
                  <div id="log"></div>
                  <form>
                  <input type="text" name="string" id="string" /><label
id="usd">$    (Don't worry about the dollar sign, it's here for you
already)</label>
                  </form>
                  <p id="inputTip">
                  □ <span>Enter a string of characters here to see if it's valid (keep
reading for details).</span>
                  </p>
            </div>

            <p>
                  This is a recursive descent recognizer that tests whether or not an
input string
                  is described by the BNF grammar below (in english: this tool
                  tests whether a sequence of characters match the format that
                  we've specified by the ruleset below). In this case, if the string of
characters
                  form a proper arithmetic expression (with the exception that only the
numbers 0 through 3, the addition, subtraction, multiplication and division operators, and
properly matched parentheses can be used, while no variables are allowed), the string
passes,
```

```
                otherwise it fails. For example, the string
                <span class="inlinecode">2+(3-1)*2</span> passes while the string
                <span class="inlinecode">2-*6(3</span> obviously does not.
            </p>

<pre class="in">
EXP   ::= EXP  + TERM   | EXP - TERM       | TERM
TERM   ::= TERM * FACTOR | TERM / FACTOR | FACTOR
FACTOR ::= ( EXP ) | DIGIT
DIGIT  ::= 0 | 1 | 2 | 3
</pre>

      </div><!-- #more -->
      </div><!-- #page -->


      <script src="./x.js"></script>
      <script src="./recursive_descent.js"></script>
  </body>
</html>
```

recursive_descent.js

```
var parensOpen = 0; // to keep track of parentheses.

/*
 *lexiScanner is a class with methods.
 */
var lexiScanner = function() {
      var string = "";
      var charPtr = 0; // points to the initial character at first.
      var advanced = false;

      this.setString = function(inputString) {
      string = inputString;
      charPtr = 0; // back to the beginning.
      parensOpen = 0;
      // count open/close parens
      if (this.token() == "(") { parensOpen++; } else if (this.token() == ")") {
parensOpen--; }
            /*console.log("PARENS: "+parensOpen);*/
            /*console.log("charPtr: "+charPtr);*/
      };

      this.firstToken = function() {
      // get first character from string.
      return string.charAt(0);
      };
      this.token = function() {
      return string.charAt(charPtr);
      };
```

```
	this.tokenAt = function(i) {
	return string.charAt(i);
	};
	this.getNextTerminal = function() {
	return string.charAt(charPtr < string.length-1 ? ++charPtr : charPtr);
	};
	this.getPreviousTerminal = function() {
	return string.charAt(charPtr > 0 ? --charPtr : charPtr);
	};
	this.index = function() {
	return charPtr;
	};
	this.advancePtr = function() {
	if (charPtr < string.length-1) {
		charPtr++;
		advanced = true;
		// count open/close parens
		if (this.token() == "(") { parensOpen++; } else if (this.token() == ")") {
parensOpen--; }
			/*console.log("PARENS: "+parensOpen);*/
			/*console.log("charPtr: "+charPtr);*/
	}
	else {
		advanced = false;
	}
	return charPtr;
	};
	this.rewindPtr = function() {
	if (charPtr > 0) {
		charPtr--;
		advanced = false;
	}
	else {
		advanced = false;
	}
	return charPtr;
	};
	this.hasAdvanced = function() {
	return advanced;
	};
	this.isLastChar = function() {
	if (charPtr == string.length-1) {
		return true;
	}
	return false;
	};

}
var scanner = new lexiScanner(); // to be used globally by all functions. Yeah, this is
bad, but it's good enough for now. :)

function exp() {
	if (!term()) {
```

```
	/*console.log("EXP error 1: TERM expected.");*/
	return false;
	}

	/*console.log("EXP WHILE 1: "+scanner.token());*/
	while (scanner.token() == "+" || scanner.token() == "-") {
	/*console.log("EXP WHILE 2: "+scanner.token());*/

	scanner.advancePtr();
		if (parensOpen < 0) {return false;}
		if (scanner.isLastChar() && parensOpen != 0) {return false;}

	if (!term()) {
		/*console.log("EXP error 2: TERM expected after '+' or '-'.");*/
		return false;
	}
	}

	return true;
}

function term() {
	if (!factor()) {
	/*console.log("TERM error 1: FACTOR expected.");*/
	return false;
	}
	/*console.log("TERM WHILE 1: "+scanner.token());*/

	while (scanner.token() == "*" || scanner.token() == "/") {

	scanner.advancePtr();
		if (parensOpen < 0) {return false;}
		if (scanner.isLastChar() && parensOpen != 0) {return false;}

	if (!factor()) {
		/*console.log("TERM error 2: FACTOR expected after '*' or '/'.");*/
		return false;
	}
	}

	return true;
}


function factor() {
	if (scanner.token() == "(") {

	scanner.advancePtr();
		if (parensOpen < 0) {return false;}
		if (scanner.isLastChar() && parensOpen != 0) {return false;} // there MUST be
something after (, or it's invalid.

	if (exp()) {
```

```
            if (scanner.token() != ")") {
                  /*console.log("FACTOR error 1: ')' expected after EXP");*/
                  return false;
            }

            scanner.advancePtr();
                  if (parensOpen < 0) {return false;}
                  if (scanner.isLastChar() && parensOpen != 0) {return false;}
      }
      else {
            /*console.log("FACTOR error 2: EXP expected after '('");*/
            return false;
      }
      }
      else {
      if (!digit()) {
            /*console.log("FACTOR error 3: DIGIT expected.");*/
            return false;
      }
      }

      return true;
}


function digit() {
      if (!(
      parseInt(scanner.token()) == 0 ||
      parseInt(scanner.token()) == 1 ||
      parseInt(scanner.token()) == 2 ||
      parseInt(scanner.token()) == 3
      )) {
      /*console.log("DIGIT error 1: '0', '1', '2', or '3' expected.");*/
      return false;
      }

      var digitIndex = scanner.index();
      /*console.log("Digit Index: "+digitIndex);*/

      scanner.advancePtr();
            if (parensOpen < 0) {return false;}
            if (scanner.isLastChar() && parensOpen != 0) {return false;}

      /*console.log("New Index: "+scanner.index());*/
      /*console.log("AFTER DIGIT: "+scanner.token());*/


      //a digit can be only be followed by an operator, closing parenthesis if open
parenthesis,
      //or END of string.  Return false otherwise.
      if (scanner.index() != digitIndex)
      {
            if (scanner.token() == "(") { // a ( following a DIGIT is automaticallly
```

```
failure.
                  /*console.log('FOO BAR BLAH');*/ return false;
            }
            /*console.log("ONE: "+scanner.token());*/
            if (parensOpen >= 0)
            {
                  /*console.log("TWO: "+scanner.token());*/
                  if (scanner.token() != ")"
                  && scanner.token() != "+"
                  && scanner.token() != "-"
                  && scanner.token() != "*"
                  && scanner.token() != "/"
                  ) {
                  /*console.log("THREE: "+scanner.token());*/
                  return false;
                  }
            }
      }
      // might not need this block at all since this is taken care of on each
advancePtr().
      else if (scanner.index() == digitIndex) { // if last character in string.
            /*console.log("SEVEN: "+scanner.token());*/
            if (parensOpen > 0) {
                  return false;
            }
      }


      return true;
}


function recognize(inputString) {
      scanner.setString(inputString);
      /*console.log(scanner.firstToken());*/

      if (exp()) {
      /*console.log("The input string is valid. Good job!");*/
      // show message on page.
      return true;
      }
      else {
      /*console.log("Sorry, invalid input. Try again.");*/
      // show message on page.
      return false;
      }
}

$(document).ready(function() {
      /*console.log("Enter a string to test: ");*/
      // RECEIVE INPUT into inputString... input changes on keyup of form element.
      var originalInputTip = $('#inputTip span').text();
      $('#string').on('keyup', function() {
```

```
	var _this = $(this);
	/*console.log("-----------------------------");*/
	if (_this.val() != "") {
		if (recognize(_this.val())) {
			$('#inputTip span').text("The input string is valid. Good job! ;)");
			//$('#inputTip span').css('color','white');
		}
		else {
			$('#inputTip span').text("Sorry, invalid input. :( Keep trying.");
			//$('#inputTip span').css('color','red');
		}
	}
	else {
		$('#inputTip span').text(originalInputTip);
		//$('#inputTip span').css('color','white');
	}
	});
});
```