Honors Thesis

Max Short

# A Compiler for a Toy Language in a Web Browser

Reader: Dr. K.M. George

Second Reader: Dr. Christopher Crick

## Abstract

Many websites exist today that allow users to compile code. Most of these websites require a network call to a remote server in order to compile this code. This is theoretically not necessary and is inefficient. EMCA Script (common name “JavaScript”), continues to grow as a programming language capable of building full scale applications.  The goal of this thesis is to demonstrate that JavaScript can be used to implement language translators on the browser platform.

This thesis presents a compiler for a small, Haskell-inspired toy language that can compute mathematical expressions. It documents a proof-of-concept computer programming of writing a compiler in JavaScript that can be run in a web browser.

## Table Of Contents

## Contents

## 1. Introduction

Functional languages are growing in popularity and importance for at least two reasons. First, over the last several years, large scale web applications that run on vast server architectures have become increasingly prominent. Programmers do not typically write these applications using functional languages. However, core functional concepts such as map-reduce are essential to these applications as they are too large scale to maintain consistent state [1].

The second reason that functional languages are increasingly important is the desire to prove code correct. As software continues to become more complex and expand into life-critical areas, it is important to be able to have mathematical assurances that program logic is correct. While it is technically possible to prove imperative code correct, it is not practical to do so. The structure of pure functional languages that are also statically typed makes them much easier to prove correct as the functional style is equivalent to mathematical implication and the strict types is equivalent to lambda calculus [2].

Another trend in today's computing environment is the use of full-fledged applications that exist only in a web browser. Web browsers have evolved greatly from their mid-1990s capabilities which were limited to displaying pages and perhaps some light JavaScript. Today, users can check their email, make a spreadsheet, and accomplish many other tasks that used to be previously restricted to desktop applications [3].

As web browsers continue to become more complex, it is logical that they should be able to compile code. There are already websites that compile code in a variety of languages [4] [5]. However, most of them require communication with a remote server to accomplish this compilation. This communication requirement could be impractical for at least two reasons. First, the programmer might have a poor or non-existent internet connection. Second, the programmer might be working on proprietary code and so for security reasons cannot release their code to a third party.

This project is a proof-of-concept that demonstrates a functional language that can be compiled in a web browser using pure JavaScript. The user types their program into an HTML text area and receives a mathematical result from a JavaScript alert. If there is a syntax error or something else goes wrong, the user will be notified via a JavaScript alert that is displayed instead of the success message. The high-level structure of the compiler is shown in Figure 1. The sections that follow will detail the lexical analysis, parsing, compilation, and execution phases of the program. Future work such as implementations of function calls and types will also be discussed.

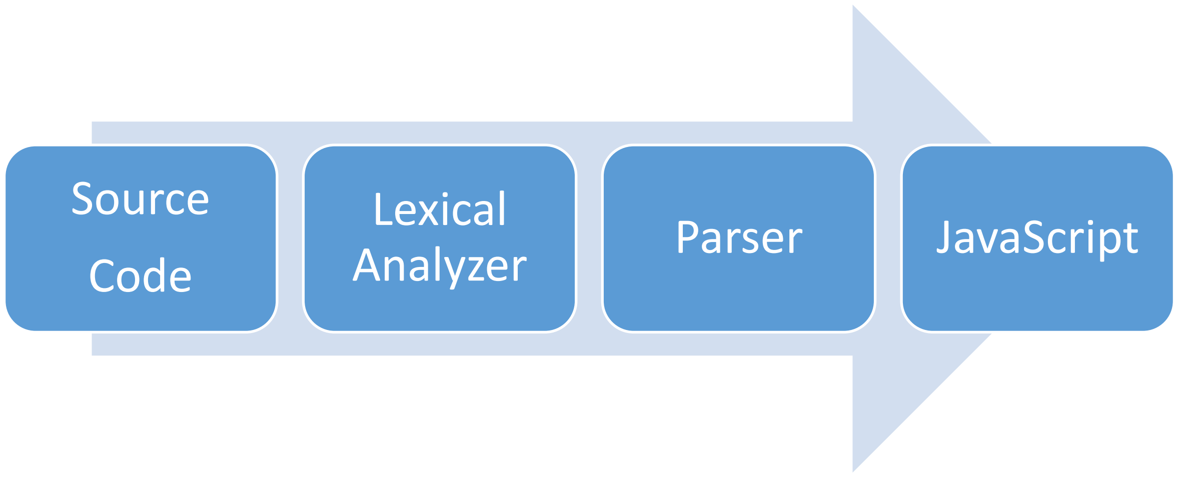

*Figure 1:Phases of the compiler.*

## 2. Lexical Analysis

The lexical analysis program (the “lexer”) is used to transform the program from a string to a series of tokens. This project uses a lexer generator. The lexer generator is based on the popular lexer generator lex. However, the lexer generator for this program is written in JavaScript and is part of the Jison library. It works in much the same way the traditional Lex does. The lexer generator takes a lex specification listing a translation of regular expressions to actions and compiles this specification  into an executable program that simulates a finite state machine (in this case, JavaScript instead of the traditional lexer’s C). The resulting program recognizes regular expressions and perform corresponding actions such as entering tokens and attributes in the symbol table. Figure 2 shows the schematic view of the relationships between the lex specification, the lex, and the lexical analyzer. Table I lists all the tokens, their meanings and regular expressions defining the tokens.

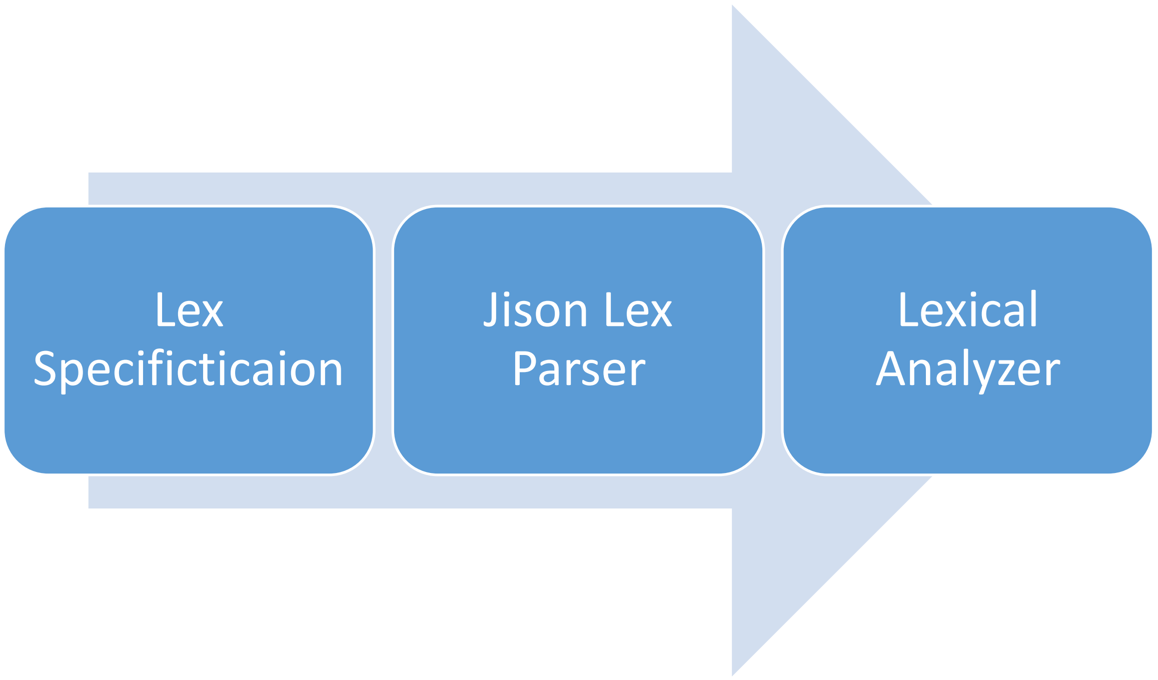

*Figure 2: The lexical analyzer is generated from the Jison Lex Specification.*

*Table 1: Lex specification.*

| Token Name | Regular Expression | Example | Explanation |
|---|---|---|---|
| Let keyword | Let | let | Beginning of a declaration of a let statements that declares one or more variable values to be used in evaluating an expression |
| In Keyword | in | in | Signals the end of a series of declarations for an expression that immediately follows this token |
| if keyword | If | if | Signals the start of an if-then-else statement. |
| then | then | then | Signals the end of the conditional portion of the if-then-else statement and the beginning of the expression to be executed if the conditional was true |
| else | else | else | Signals the end of the first expression to be executed if the condition of an if-then-else statement was true and the beginning of the expression to be executed if the if-then-else statement was false |
| Integer numeric literal | \d+ | 42 | Similar to the Int type in Haskell |
| Whitespace | \s+ | <tab> | Used to ignore all whitespace |

| | | | |
|---|---|---|---|
| End of File | <<EOF>> | N/A | Used to notify the lexer when the end of the user program has been reached |
| Variable identifier | [a-zA-Z][a-zA-z0-9_']* | answer' | Names a variable. Similar to Haskell variable names, legal names start with an upper or lower case letter that can optionally be followed by a series of letters, numbers, underscores, or apostrophes. |
| Plus sign | + | + | Denotes integer addition |
| Minus sign | - | - | Denotes integer subtraction |
| Multiplication Sign | * | * | Denotes integer multiplication |
| Division Sign | / | / | Denotes integer division |
| Left Parenthesis | ( | ( | Denotes the start of an expression enclosed in parenthesis |
| Right Parenthesis | ) | ) | Denotes the end of an expression enclosed in parenthesis |
| Equality conditional check | == | == | Denotes a boolean check that will return true if and only if the two arguments are equal |
| Greater than conditional check | > | > | Denotes a boolean check that will return true if and only if the first argument is greater than the second argument |
| Less than conditional check | < | < | Denotes a boolean check that will return true if and only if the first argument is less than the second argument |

The tokens generated by the lexer will be input to the parser.

## 3. The Parser

The parser is one of the most important components of this compiler. It parses the token stream supplied by the lexical analyzer, checks for syntax validity, and produces intermediate representation in the form of an Abstract Syntax Tree(AST). Along with the Lexer mentioned in the previous section, this project also uses Jison to generate a parser. The parser generation of Jison is similar to the traditional parser generating program Bison in that it takes a file denoting a grammar and transforms it into an executable parser. In Jison's case, this code is JavaScript unlike Bison's C. Bison-style parsers also allow custom functions in the grammar file to assist in construction of the AST. Jison allows JavaScript functions instead of Bison's C functions. Bison-style parsers (included Jison) are Look Ahead, Left-Right (LALR) parsers [6]. Table 2 describes the language designed in this project. Appendix A shows this grammar (as well as the lex specification in Jison format).

### 3.1 Grammar

*Table 2: Jison Grammar*

| Productions | Semantics |
|---|---|

| | |
|---|---|
| PRGRM ⇒ | The start symbol. Denotes an entire program |
| LET_STMT EOF | Denotes a program that has a let statement before the expression to be evaluated |
| \|EXPR EOF | Denotes a program that does not have a let statement before the expression to be evaluated. |
| LET_STMT⇒ "let" EQ_STMT_GRP "in" EXPR | Denotes a series of variable value declarations followed by an expression |
| EQ_STMT_GRP ⇒ | One or more assignment statements |
| EQ_STMT EQ_STMT_GRP | Multiple assignment statements |
| \|EQ_STMT | A single assignment statement |
| EQ_STMT ⇒ ID "=" EXPR | An assignment of an expression value to a variable |
| EXPR ⇒ | A mathematical construct that contains one or more of operations, numeric literals, and variables |
| '(' EXPR ')' | An expression may be contained in parenthesis to influence evaluation order |
| \|EXPR '+' EXPR | Adds the values of two expressions |
| \|EXPR '-' EXPR | Subtracts the first expression from the second expression |
| \|EXPR '*' EXPR | Multiplies the first expression times the second expression |
| \|EXPR '/' EXPR | Divides the first expression by the second expression |
| \|NUMERIC LITERAL | An atomic level of expression –a number |
| \|ID | The other atomic level of expression – a variable |
| \|"if" CONDL_EXPR "then" EXPR "else" EXPR | An expression that is equivalent to one of the two sub-expressions. It is equivalent to the first if and only if the conditional expression evaluates to true. |
| CONDL_EXPR ⇒ | Evaluates a boolean condition and returns either true or false |
| EXPR "==" EXPR | Returns true if and only if the two expressions are numerically equivalent |
| \|EXPR "<" EXPR | Returns true if and only if the first expression is less than the second expression |
| \|EXPR ">" EXPR | Returns true if and only if the first expression is greater than the second expression |

## 3.2 Building the Abstract Syntax Tree (AST)

An Abstract Syntax Tree (AST) is a data structure that represents an expression in a programming language [7]. It is generally used as an intermediate form between source and target language (in this case, the Haskell-like toy language and JavaScript). This project generates multiple ASTs – one for each expression. Examples of expressions with their own ASTs include assignments in let statements and the main expression. Like other ASTs, this project's ASTs has leaves consisting of literal values or variables and internal nodes consisting of functions that combine these values.

### 3.2.1 Constructing the nodes of the AST

Several functions are embedded directly in the Right Hand Side(RHS) of the Jison grammar to assist in the creation of the AST. The first of these is a constructor function for Node, which creates an object that represents a node in the AST. The Node constructor function takes three parameters: the first, the "value" of the node, is required. The second two, the left child node and the right child node, are optional. The value of the node could be one of several things (this is accomplished cleanly using JavaScript's dynamic typing system).

First, the value of a node could be a numeric literal. This implies that the node is a leaf of the AST and the left and right children should not be evaluated during the evaluation phase (described in a later section) as they should not exist.

Second, the node could be a variable name. In this case, left and right children should also not be evaluated in the evaluation phase but instead the variable value should be determined from the symbol table (described below).

Third, the node's value could be a two-parameter function. This implies that the left and right children of the node should be evaluated and then passed to the two-parameter function to ultimately determine the value of the node during the evaluation phase of the program.

The final potential type of a node value is itself a special kind of node called a conditional node. A conditional node is first constructed using a separate constructor. It implies that during the evaluation phase the left and right children of the *conditional node* should be evaluated and then used to determine whether the *node's* final value should be determined from the node's left child or right child. This is represented in the source syntax as an if-then-else statement.

### 3.2.2 Storing values in the symbol table

The procedures defined in the previous section are adequate to build the AST for the expression to be evaluated. However, the language of this project also necessitates the use of a symbol table. Complicating the implementation of the symbol table is this project's language's imitation of Haskell's lazy evaluation. In order to implement lazy evaluation efficiently, the symbol table stores two different kinds of values: numeric values and expressions. This way, an expression can be assigned to a variable without being evaluated.

The final symbol table is a JavaScript object mapping names to expressions or values. The symbol table must be built up over time as the AST is built. This is accomplished using the shallow merge function. Initially, when an expression or value is assigned to a variable, it is put into a new symbol table consisting of only that assignment.

## 4. The Compiler

Code Generation and running of the program occurs after the syntax tree is generated. The nodes of the abstract syntax tree are evaluated in a modified post-order traversal, using the function evalNode (see Appendix B). The method of evaluation depends on the type of node. See Figure 3 for a visual representation of this process.

The first type of node, numeric literals, are immediately returned.

The second type of node, variable names, are looked up in the symbol table (called "context" in the compiler source code). The variables may point to an expression that has not yet been evaluated due to the lazy evaluation semantics of the language. However, once an expression is evaluated, its value is stored back to the symbol table to avoid unnecessary re-computation of values. This does not make the language stateful as it has the same effect as evaluated the expression multiple times. This process is depicted in Figure 4.

The third type of node, a two parameter function, requires that both of its children be evaluated before the function is called. This behavior is the source of the general post order traversal. Examples of two parameter functions include addition, subtraction, multiplication and division. Once this function is evaluated, it is replaced in the tree by its literal result.

The final type of node is a conditional node. A conditional node consists of the usual left and right children plus a value with three properties: a conditional function, a conditional left argument and a conditional right argument. The conditional function is a binary function that compares two values. An example of a conditional function include an equality function (==) that returns true if and only if the left conditional argument is equal to the right conditional argument. The greater than function (>) returns true if and only if the left conditional argument is greater than the right conditional argument. The less than function (<) returns true if and only if the left conditional argument is greater than the right conditional argument. When a conditional function evaluates to true, its node's value becomes equal to the value of the node's left child. When a conditional function evaluates to false, its node's value becomes equal to the node's right child.

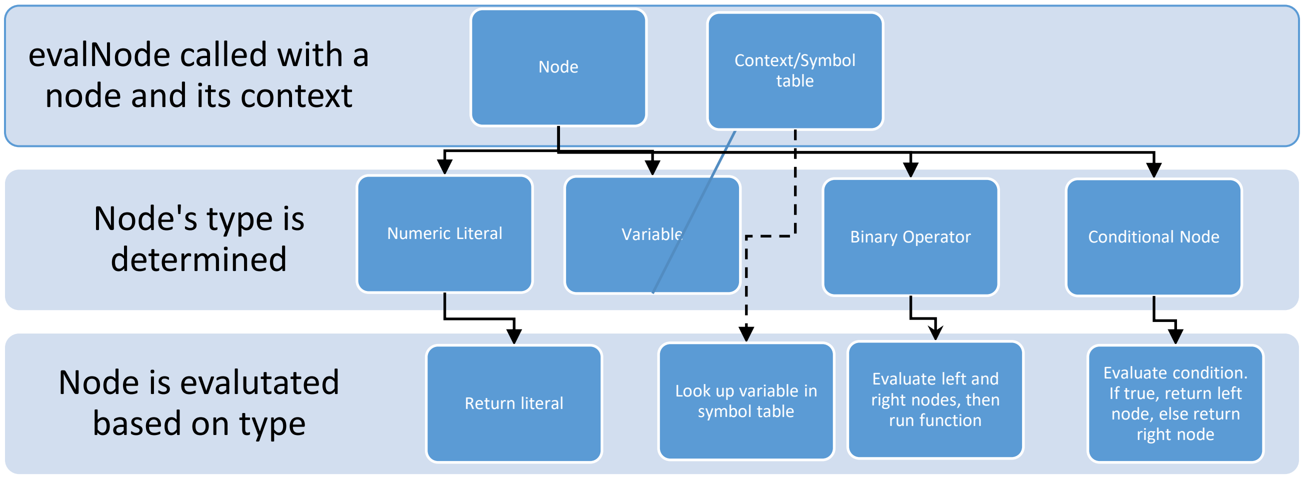
evalNode called with a node and its context
Node
Context/Symbol table
Node's type is determined
Numeric Literal
Variable
Binary Operator
Conditional Node
Node is evalutated based on type
Return literal
Look up variable in symbol table
Evaluate left and right nodes, then run function
Evaluate condition. If true, return left node, else return right node

*Figure 3: Process of evaluating a node in the AST*

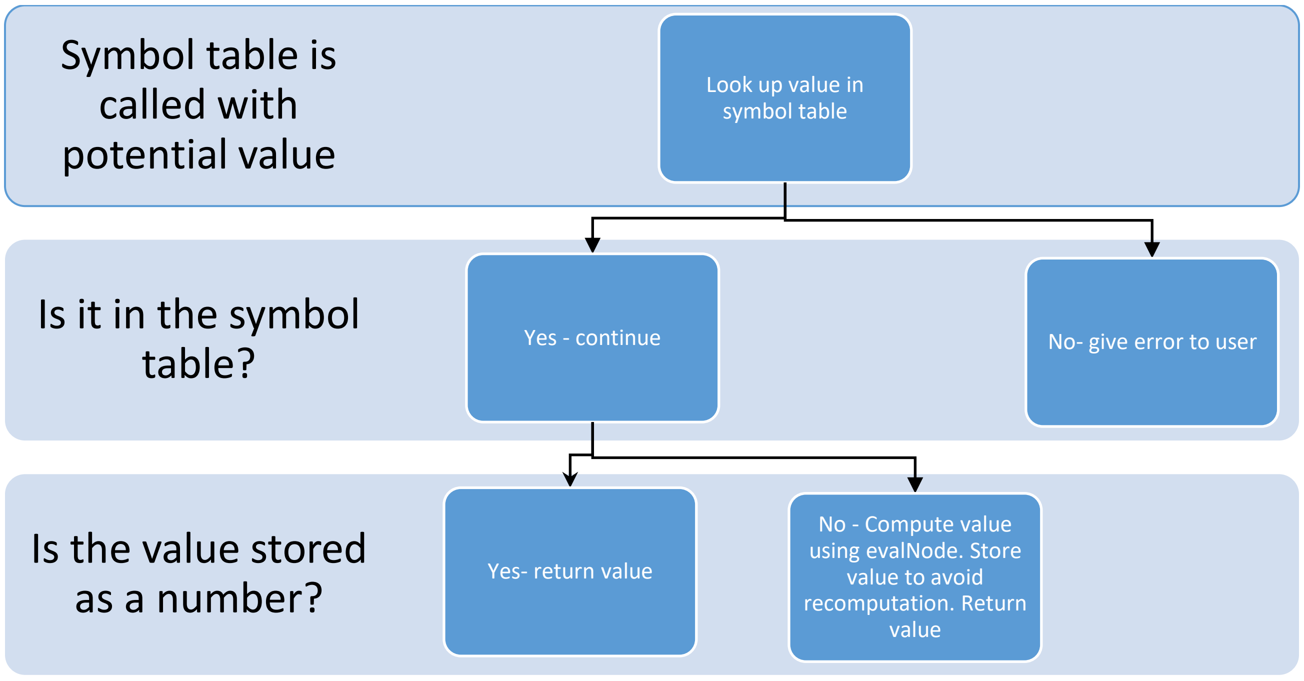
Symbol table is called with potential value
Look up value in symbol table
Is it in the symbol table?
Yes - continue
No- give error to user
Is the value stored as a number?
Yes- return value
No - Compute value using evalNode. Store value to avoid recomputation. Return value

*Figure 4: Looking up a value in the symbol table*

## 5. Future Work

This project was a proof-of-concept of a compiler for a functional language written in native JavaScript. There is much potential for future work, both in extending this project and in pursuing other avenues of compiling a functional language entirely in a web browser. There are several immediate extensions possible for this project. First, function calls would greatly enhance the power of the language. These were not implemented due to time constraints but in theory they would not force a large redesign of the program. The biggest challenge would be enforcing constraints on variables and maintaining proper context for parameters while still following lazy evaluation.  Another enhancement to the language would be the addition of a type system. This type system could work by adding types as an attribute nodes.

A more practical extension of this project would be to use an existing functional language such as Haskell and port that language to a web browser. This could be accomplished in different ways. First, a new compiler could be written from scratch in JavaScript. Alternatively, a transpiler (possibly created with the LLVM) could be used to transpile code from whatever language the compiler was written in to JavaScript.

## 6. Conclusion

The preceding paragraphs and the implementation of the compiler demonstrate that it is possible to make a simple compiler entirely in JavaScript, thanks in part to the tools provided by Jison. This project means that it is possible for compilers to follow word processors, spreadsheets, email clients, and many other types of software from the desktop to a web browser. Within this browser compiler, it is possible to use functional languages, which should be increasingly important over time.

## 7. Works Cited

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation,*, San Francisco, 2004.

[2] Haskell Wiki, "Curry-Howard-Lambek correspondence," February 2010. [Online]. Available: https://wiki.haskell.org/Curry-Howard-Lambek_correspondence.

[3] Google Inc., "Google Apps," [Online]. Available: apps.google.com.

[4] Sphere Research Labs, [Online]. Available: http://ideone.com/.

[5] Tutorials Point, "Compile and Execute Haskell Online," [Online]. Available: http://www.tutorialspoint.com/compile_haskell_online.php.

[6] Free Software Foundation, "Bison - GNU Project," 6 March 2014. [Online]. Available: https://www.gnu.org/software/bison/.

[7] A. V. Alfred , R. Sethi and J. D. Ullman, Compilers Principles, Techniques, and Tools, Menlo Park: Addison-Wesley, 1988.

## Appendix A: Jison file

```
%lex

%%
//Lex goes in order so keywords have to go first
"let" {return "let";}
"in" {return "in";}
"if" {return "if";}
"then" {return "then";}
"else" {return "else";}
\d+ {return "NUMERIC_LITERAL";}
\s+ /*Ignore whitespace*/
<<EOF>> {return "EOF";}
'+' {return "+";}
"-" {return "-";}
"*" {return "*";}
"/" {return "/";}
"(" {return "(";}
")" {return ")";}
[a-zA-Z][a-zA-z0-9_']* {return "ID";}
"==" {return "==";}
"=" {return "=";}
"<" {return "<";}
">" {return ">";}



. return "INVALID"

/lex

%left '*' '/'
%left '+' '-'
%nonassoc "=" "<" ">" "else"


%start PRGRM

%%
PRGRM: LET_STMT EOF {return $1; }
       | EXPR EOF {return {baseNode: $1, context:{}}; }
;

LET_STMT: "let" EQ_STMT_GRP "in" EXPR {$$ = {baseNode:$4, context:$2 };}
;

EQ_STMT_GRP: EQ_STMT EQ_STMT_GRP {$$ = shallowMerge($1, $2);}
                    | EQ_STMT {$$ = $1;}
;
```

```
EQ_STMT: ID "=" EXPR {sylList = {}; sylList[$1] = $3; $$ = sylList;}
;

/*EXPRS: EXPR EOF {return $1;}
;*/

EXPR: '(' EXPR ')' {$$= $2;}
       | EXPR '+' EXPR {$$= new Node(add, $1, $3);}
       | EXPR '-' EXPR {$$= new Node(sub, $1, $3);}
       | EXPR '*' EXPR {$$= new Node(mul, $1, $3);}
       | EXPR '/' EXPR {$$= new Node(div, $1, $3);}
       | NUMERIC_LITERAL {$$= new Node(+($1));}
       | ID {$$ = new Node($1)}
       | "if" CONDL_EXPR "then" EXPR "else" EXPR {$$ =  new Node($2, $4, $6);}
       ;

CONDL_EXPR: EXPR "==" EXPR {$$ = new CondlNode(eq, $1, $3);}
       |EXPR "<" EXPR {$$ = new CondlNode(lt, $1, $3);}
       |EXPR ">" EXPR {$$ = new CondlNode(gt, $1, $3);}
       ;
%%
function shallowMerge(x, y) {
              var xProps = Object.getOwnPropertyNames(x);
              var yProps = Object.getOwnPropertyNames(y);
              var expectedSize = xProps.length + yProps.length;
              var temp = Object.assign({}, x);
              var combined = Object.assign(temp, y);
              var combinedProps = Object.getOwnPropertyNames(combined);
              if (expectedSize > combinedProps.length) { //there was a duplicate so
set < sum of individual sizes
                     dup = xProps.find(function findADup(currentX) {
                            return y[currentX] !==undefined;
                     });
                     throw {message: "Duplicate identifier: " + dup}
              }
              return combined;
}

//value could be a literal value or an operation to combine l and r (which should be
literal values...
function Node(val, l, r) {
       if (val == null) {
              throw {message: "A Node must have a value"}
       }
       this.left = l;
       this.right = r;
       this.value = val;
}
function CondlNode(condF, condL, condR) {
       if (!condF) {
            throw {message: "No conditional function passed"}
        }
        if (!condL) {
            throw {message: "No left conditional passed"}
        }
        if (!condR) {
            throw {message: "No right conditional passed"}
        }
        this.condF = condF;
        this.condL = condL;
        this.condR = condR;
}

function add(x, y) {return x + y;}
function sub(x, y) {return x - y;}
function mul(x, y) {return x * y;}
function div(x, y) {return x / y;}

function gt(x, y) {return x > y;}
function lt(x, y) {return x < y;}
function eq(x, y) {return x == y;}
```

## Appendix B: Selected JavaScript Code

```
function evaluateNode(node, context) {
	if (node.value.condF != undefined) {//boolean special case
		node.value.condL.value = evaluateNode(node.value.condL, context);
		node.value.condR.value = evaluateNode(node.value.condR, context);
		var useLeft = node.value.condF(node.value.condL.value, 
node.value.condR.value);
		if (useLeft) {
			node.value = evaluateNode(node.left, context);
			return node.value;
		}
		else {
			node.value = evaluateNode(node.right, context);
			return node.value;
		}
	}

	if (!isNaN(node.value)) {
		return node.value;
	}
	//http://stackoverflow.com/questions/5999998/how-can-i-check-if-a-javascript-
variable-is-function-type
	else if (typeof node.value === "function" && 
Object.prototype.toString.call(node.value) == "[object Function]") {
		node.left = evaluateNode(node.left, context);
		node.right = evaluateNode(node.right, context);
		node.value = node.value(node.left, node.right);
		return node.value;
	}
	else if (typeof node.value === "string") {
		node.value = evaluateNode(context[node.value], context);
		return node.value;
	}
	else {
		console.log("Problem node value: ");
		console.log(node.value);
		throw {compileError:{message:"Unknown type for node value: " + 
node.value}};
	}

}
```