C compiler written purely in JavaScript

A COURSE PROJECT REPORT

By

Saurabh Pandey (RA2011003010207)

Naimish Pandey(RA2011003010147)

Devansh Pareek(RA2011003010184)

Under the guidance of

Dr. Nagadevi S

In partial fulfilment for the Course

Of

18CSC304J - Compiler Design

In CTECH

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chenpalpattu District

APRIL, 2023

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this mini project report " **A C compiler written purely in JavaScript**" is the bonafide work of Saurabh Pandey (RA2011003010207), Naimish Pandey (RA201100310147),Devansh Pareek (RA2011003010184), who carried out the project work under my supervision.

SIGNATURE

Dr. Nagadevi S

Assistant Professor

CTECH

SRM Institute of Science and Technology

CONTENTS

1. ABSTRACT

This project aims to develop a C compiler using the JavaScript programming language. The C compiler will perform key stages of the compilation process, including lexical analysis, parsing, semantic analysis, code generation, and optimization. JavaScript is an “interpreted” language, so it cuts down on the time needed for compilation in other languages like Java. The project will provide an overview of the C programming language and JavaScript as a programming language, discuss the design and architecture of the C compiler in JavaScript, and showcase a live demonstration of the compiler. We aim to showcase the functionality of building a C compiler using JavaScript, highlight the different stages of the compilation process.

2. INTRODUCTION

Designing a C compiler in Javascript involves several stages, including lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and code generation. Javascript's string manipulation, data structure handling, and system programming capabilities can be used to implement these stages. The first step is lexical analysis, breaking down C code into tokens using Javascript's string manipulation capabilities. Syntax analysis involves analyzing the code's structure and generating an abstract syntax tree using Javascript's flexible syntax and tree data structures. Semantic analysis checks for semantic errors using Javascript's dynamic typing and library support. Intermediate code generation uses Javascript's flexibility to generate abstract representations of the source code. Code optimization involves techniques likeconstant folding and code motion to optimize performance, using Javascript's built-in data structures and algorithms. Code generation translates the intermediate code into machine-specific instructions using Javascript's low-level system programming capabilities. Extensive testing is required using Javascript's testing framework and debugging tools to ensure the compiler generates correct and efficient machine code. In conclusion, designing a C compiler in Javascript is a challenging task that requires a solid understanding of compiler theory, programming language concepts, and Javascript programming skills, and can be achieved by leveraging Javascript's powerful capabilities for string manipulation, data structure handling, and system programming.

3. OBJECTIVE

The objective of this mini project is to build a C compiler in JavaScript , which can effectively translate C source code into target machine code or another form of executable code. The project will involve implementing a lexer/scanner that can tokenize C source code, a parser that can construct an abstract syntax tree (AST) representing the syntax and structure of the C code, and a symbol table to manage information about identifiers declared in the code. Semantic analysis routines will be implemented to perform static checks on the AST, ensuring adherence to the C language's semantics. Code generation routines will be developed to translate the AST into target machine code or other executable code. Basic support for C language features such as data types, expressions, statements, and control flow structures will be implemented. Robust error handling mechanisms will be incorporated to provide meaningful error messages and diagnostics for users. Testing and validation strategies, including unit testing, integration testing, and regression testing, will be implemented to ensure the correctness, performance, and reliability of the compiler. Optional optimization techniques may also be implemented to improve the generated code's performance, size, or other characteristics.

.

4. REQUIREMENTS

The hardware and software requirements for designing a C compiler in Javascript would typically include:

Hardware Requirements:

Computer system with adequate processing power and memory to handle the compilation process efficiently.
Storage space for storing the source code, intermediate code, and generated machine code.
Input/Output devices such as a keyboard, mouse, and display for interacting with the compiler and viewing output.

Software Requirements:

Javascript programming language: Javascript is the primary programminglanguage used for implementing the compiler.
Development environment: An integrated development environment (IDE) such as Atom, Visual Studio Code, or NetBeans for coding, debugging, and testing the compiler.
Lexical analyzer and parser tools: Javascript libraries such as jQuery or react can be used for implementing the lexical analysis and syntax analysis stages of the compiler.
Additional libraries: Depending on the specific requirements of the compiler, additional Javascript libraries for handling regular expressions, string manipulation, data structures, and system programming may beneeded.
Operating system: A compatible operating system such as Windows, macOS, or Linux that supports Javascript and necessary development tools.

5. ARCHITECTURE AND DESIGN

The architecture and design of a C compiler in Javascript would involve several key components and modules that work together to analyze,transform, and generate machine code from C source code. Here's ahigh-level overview of the potential architecture and design for the project:

Lexical Analysis: This module would handle the scanning and tokenization of the C source code using regular expressions or lexer tools in Javascript. It would identify keywords, operators, literals, identifiers, and other language elements and generate a stream of tokens as output.

Syntax Analysis: This module would implement the parsing of the token stream generated by the lexical analysis stage to construct an abstract syntax tree (AST) representation of the C source code. It would verify the syntax rules and enforce the grammar of the C language using a top-down or bottom-up parsing technique, such as LL or LR parsing, in Javascript.

Semantic Analysis: This module would perform semantic analysis on the AST to check for semantic errors, such as type checking, scope checking, and symbol table management. It would ensure that the C source code adheres to the language's semantics and generate appropriate error messages for any detected issues.

Intermediate Code Generation: This module would generate an intermediate representation (IR) of the C source code from the AST. The IR is an abstract representation of the code that is independent of the target machine architecture, and it can be used for further optimizations.

Code Optimization: This module would perform various optimizations on the generated IR to improve the performance and efficiency of the resulting machine code. This may include techniques such as constant folding, common subexpression elimination, and loop optimizations, implemented using Javascript's data structures and algorithms.

Code Generation: This module would generate the final machine code for the target architecture from the optimized IR. It would translate the IR into machine-specific instructions, such as assembly code or machine code, using Javascript's low-level system programming capabilities.

Testing and Debugging: This module would include comprehensive testing of the compiler using Javascript's testing frameworks, as well as ebugging tools for identifying and fixing issues in the compiler's implementation.

User Interface: This module may involve designing a user-friendly interface for interacting with the compiler, including input of C source code, displaying compiler output, and handling compiler options or settings.

Error Handling: This module would handle error reporting and recovery mechanisms in case of syntax errors, semantic errors, or other issues encountered during the compilation process.

Documentation: This module would involve documenting the design, implementation, and usage of the C compiler in Javascript, including instructions for building, running, and testing the compiler, as well as any relevant documentation for users and developers.

Overall, the architecture and design of a C compiler in Javascript would require careful planning, implementation, and testing of the differentstages and components to ensure a reliable, efficient, and accurate compiler that can translate C source code into machine code.

6. IMPLEMENTATION

codeGenerator.js

```
var stack = [];
var allFuncs = [];
var volatileRegs = ['rcx','rdx','r8','r9']; // this is for Win64-FastCall Calling Convention
var strLiteralSection = '';
var currentFunc = '';
// List of additional settings coming in the future:
// setting the mnemonic: AT&T or Intel
// var mnemonic = 'AT&T';
// setting the syntax: GCC-OSX, GCC-Linux, MASM, NASM, etc.
// var syntax = 'GCC-OSX'
// setting the calling convention: Win64-FastCall, Linux-FastCall, CDECL, etc.
// var calling_convention = 'Win64-FastCall';
// setting the mode: 16, 32, 64.
// var mode = 64.

function initGenerate(TheBigAST) {
  var globalItems = TheBigAST[0];
  var functionBox = TheBigAST[1];
  allFuncs = findAllFuncs(functionBox[0].body);
  //strLiteralSection = generateStrLiteralSection();
  var funcsAsm = findFunctionNames(functionBox[0].body);
  var headers = includeHeaders(globalItems);
  var dataSection = generateDataSection(globalItems);
  var textHeader = generateTextHeader(TheBigAST[1]);
  var compiled = textHeader + funcsAsm + dataSection;
  console.log(compiled);
  return 0;
}
```

```
function findAllFuncs(funcDefs) {
  var current = 0;
  allFuncs = [];
  while (current < funcDefs.length) {
    allFuncs.push(funcDefs[current].name);
    current++;
  }
  return allFuncs;
}

function generateFunctionAssembly(functionBody, functionArgs) {
  var current = 0;
  var functionAssembly = "";
  var ifParts = [];
  if (functionBody.length !== 1 || functionArgs.length !== 0) {
    functionAssembly += initStack();
  }
  if (functionArgs.length !== 0) {
    var regIndex = 0
    for (var i = 0; i < functionArgs.length; i++) {
      if (functionArgs[i].type === 'int') {
        functionAssembly += '\tpush %' + volatileRegs[regIndex] + '\n';
        stack.push({
          type: 'LocalVariable',
          name: functionArgs[i].name,
          value: volatileRegs[regIndex],
          variableType: 'int'
        });
        regIndex++;
      }
    }
  }
  while (current < functionBody.length) {
    var part = functionBody[current];
```

```
    if (part.type === 'Statement') {
        var partValue = part.value;
        functionAssembly += checkForStatements(partValue);
    } else if (part.type === 'if') {
      functionAssembly += checkForIfs(part);
    } else if (part.type === 'Call') {
      functionAssembly += generateCall(part);
    }

    /** finally, if we are at the end of the block and there is absolutely nothing to parse
    **  we add stack clearing code
    **/
    if (current !== 0 && current === (functionBody.length - 1)) {
      clearStack();
      restoreRBP();
    }
    //clearStack();
    current++;
  }
  // for now we only support the functions that return 1 value.
  // hence, we clear the stack only in the end of a function;
  //clearStack();
  return functionAssembly;
}

function generateStrLiteralSection() {
  var strLiteralSection = '\t.section\t__TEXT,__cstring,cstring_literals\n';
  return strLiteralSection;
}

function initStack() {
  var prologue = '\tpush %rbp\n';
  prologue += '\tmov %rsp,%rbp\n';
  saveRBP();
```

```
  return prologue;
}

function saveRBP() {
  stack.push({
    type: 'SavedRBP',
    name: '',
    value: 'rbp',
    variableType: ''
  });
}

function restoreRBP() {
    stack.pop();
}

function addRestoreRBPAsm() {
   return '\tpop %rbp\n';
}

function findFunctionNames(functionPack) {
  var funcsAsm = '';
  for (var i = 0; i < functionPack.length; i++) {
    currentFunc = functionPack[i].name;
    if (functionPack[i].name === 'main') {
      funcsAsm += '\t.globl            main\n\n';
      funcsAsm += 'main:\n';
    } else {
      funcsAsm += '\t.globl            _' + functionPack[i].name +'\n\n';
      funcsAsm += '_' + functionPack[i].name + ':\n';
    }

    funcsAsm += generateFunctionAssembly(functionPack[i].body, functionPack[i].args);
  }
```

```
  return funcsAsm;
}

function generateTextHeader() {
  var header = '\t.text\n';
  return header;
}
function generateDataSection(globalItems) {
  var globalVariables = findGlobalVariables(globalItems);
  var dataHeader = generateDataHeader();
  var dataBody = generateDataBody(globalVariables);
  var dataSection = dataHeader + dataBody;
  return dataSection;
}

function generateDataHeader() {
    var header = '\t.data\n';
    return header;
}
function generateDataBody(globalVariables) {
  var dataSection = '';
  for (var i = 0; i < globalVariables.length; i++) {
    dataSection += '\t.globl\t_' + globalVariables[i].name + '\n';
    dataSection += '_' + globalVariables[i].name + ':\n';
    if (globalVariables[i].type === 'int') {
      dataSection += '\t.long\t' + globalVariables[i].value + '\n\n';
    }
  }
  return dataSection;
}
function findGlobalVariables(globalItems) {
  var globalVariables = [];
  var current = 0;
  while (current < globalItems.length) {
```

```
    if (globalItems[current].type === 'GlobalStatements') {
      break;
    }
    current++;
  }
  var globalStatementsBody = globalItems[current].body;
  for (var i = 0; i < globalStatementsBody.length; i++) {
    if (globalStatementsBody[i].type === 'Statement') {
      var current = 0;
      var parts = globalStatementsBody[i].value;
      while (current < parts.length - 1) {
        if (parts[current].type === 'Word' && keywords.indexOf(parts[current].value) === -1 &&
parts[current + 1].type === 'Equal') {
          globalVariables.push ({
            type: parts[current - 1].value,
            name: parts[current].value,
            value: parts[current + 2].value
          });
        }
        current++;
      }
    }
  }
  return globalVariables;
}

function reverseOffset(offset) {
  var reverseOffset = Math.abs(((stack.length -1) - offset));
  return reverseOffset;
}

function findOnTheStack(value){
  for (var i = 0; i < stack.length; i++) {
    if(stack[i].name === value){
```

```
      return i;
    }
  }
  return -1;
}

function generateReturn(returnValue) {
  var retAsm = '';
    if (returnValue.type === 'NumberLiteral') {
       if (returnValue.value === '0') {
         retAsm += '\txor %rax,%rax\n';
       } else {
         retAsm += '\tmov $' + returnValue.value + ',%rax\n';
       }
    }
    else if (returnValue.type === 'Word') {
      if (!isAKeyword(returnValue.value)) {
       stackEntry = findOnTheStack(returnValue.value);
       if (stackEntry !== -1) {
         stackOffset = reverseOffset(stackEntry);
         switch (stack[stackEntry].variableType) {
           case 'int': {
             if ( stackOffset !== 0) {
               retAsm += '\tmov ' + stackOffset * 8 + '(%rsp), %rax\n';
             }
             else {
               retAsm += '\tmov (%rsp), %rax\n';
             }
             break;
           }
           default:
             break;
         }
       }
```

```
      }
    }
    retAsm += addClearStackAsm();
    retAsm += addRestoreRBPAsm();
    retAsm += '\tret\n\n';
    return retAsm;
}

function addClearStackAsm() {
  var counter = 0;
  for (var i = 0; i < stack.length; i++) {
    if (stack[i].type === 'LocalVariable') {
      counter++;
    }
  }
  return '\tadd $' + (counter * 8).toString() + ',%rsp\n';
}

function clearStack() {
  var entries = [];
  for (var i = 0; i < stack.length; i++) {
    if (stack[i].type === 'LocalVariable') {
      entries.push(i);
    }
  }
  while (entries.length !== 0) {
    stack.splice(entries[entries.length - 1], 1);
    entries.pop();
  }
  return;
}

function generateIncByOne(offset){
    var incAssembly = '';
```

```
    if (offset === 0) {
      incAssembly = '\tincl (%rsp)\n';
    } else {
      incAssembly = '\tincl ' + (offset * 8).toString() + '(%rsp)\n';
    }
    return incAssembly;
}

function generateIfClause(offset, cmpValue, name) {
  var ifClause = '';
  if (offset === 0) {
    ifClause += '\tcmp $' + cmpValue  + ',(%rsp)\n';
  } else {
    ifClause += '\tcmp $' + cmpValue  + ',' + ((parseInt(offset))*8).toString() + '(%rsp)\n';
  }
  ifClause += '\tjne _if' + name + cmpValue + '_after\n';
  return ifClause;
}

function generateIfInside(ifInside, ifName, ifCmpValue) {
  var current = 0;
  var assembledifInside = '';
  var stacklen = stack.length;
  while (current < ifInside.length) {
    var part = ifInside[current];
    if (part.type === 'Statement') {
      var partValue = part.value;
      assembledifInside += checkForStatements(partValue);
    }
    current++;
  }
  if (stacklen < stack.length) {
    var counter = 0;
    while (stacklen < stack.length) {
```

```
      stack.pop();
      counter++;
      stacklen++;
    }
    assembledifInside += '\tadd $' + (counter * 8).toString() + ',%rsp\n';
  }
  assembledifInside += '\n';
  return assembledifInside;
}

function isAKeyword(word){
  if (keywords.indexOf(word) !== -1) {
    return true;
  }
  return false;
}

function checkForStatements(part) {
  var functionAssembly = '';
  var didaddEpilouge = 0;
  if (part[0].type === 'Word' && isAKeyword(part[0].value)) {
    if (part[0].value === 'return') {
      functionAssembly += generateReturn(part[1]);
      didaddEpilouge = 1;
    }

    if (part[0].value === 'int') {
      if (part.length === 5) {
        functionAssembly += generateVariableAssignment(part[0].value, part[1].value, part[3]);
      } else if (part.length > 5) {
        functionAssembly += generateVariableAssignmentWithAddition(part);
      }
    }
```

```
    if (part[0].value === 'char') {
      if (part.length === 6) {
        generateStringVariable(part);
      }
    }
  } else if (part[0].type === 'Word' && !isAKeyword(part[0].value)) {
      for (var i = 0; i < stack.length; i++) {
        if (stack[i].type === 'LocalVariable') {
          if (stack[i].name === part[0].value) {
            if (part[1].type === 'IncByOne') {
              functionAssembly += generateIncByOne(reverseOffset(i));
            }
          }
        }
      }
    }
  return functionAssembly;
}

function checkForIfs(part) {
  var functionAssembly = '';
  if (part.condition.length === 3) {
    var cond = part.condition;
    if (cond[1].type === 'ComparisonE') {
      if (cond[0].type === 'Word' && cond[2].type === 'NumberLiteral') {
        for (var i = 0; i < stack.length; i++) {
          if (stack[i].type === 'LocalVariable' && stack[i].name === cond[0].value) {
            functionAssembly += generateIfClause(reverseOffset(i), cond[2].value, cond[0].value);
            functionAssembly += generateIfInside(part.body, cond[0].value, cond[2].value);
            functionAssembly += '_if' + cond[0].value + cond[2].value + '_after:\n';
          }
        }
      }
    }
```

```
  }
  return functionAssembly;
}
function generateCall(part) {
  callAsm = '';
  if (allFuncs.indexOf(part.callee) !== -1) {
    var params = part.params;
    var current = 0;
    var regIndex = 0;
    while (current < params.length) {
      if (params[current].type === 'Delimiter') {
        current++;
        continue;
      } else if (params[current].type === 'NumberLiteral') {
        callAsm += '\tmov $' + params[current].value + ',%' + volatileRegs[regIndex] + '\n';
        regIndex++;
        current++;
        continue;
      }
      current++;
    }
    callAsm += '\tcall _' + part.callee + '\n';
  }
  return callAsm;
}

function generateStringVariable(part) {
  if (part[1].type === 'Word' && part[2].type === 'Arr') {
    for (var i = 0; i < stack.length; i++) {
      if (stack[i].type === 'LocalVariable') {
        if (stack[i].name === part[1].value) {
          throw new TypeError('Varibale already defined: ' + part[1].value);
        }
      }
```

```
    }
    var theCharArray = part[2].value
    if (theCharArray.length === 0) {
      if (part[4].type === 'StringLiteral') {
        strLiteralSection += '\nL_' + currentFunc + '_' + part[1].value + ':\n';
        strLiteralSection += '\t.ascii\t\"' + escapeThis(part[4].value) + '\"\n';
      }
    }
  }
}

function escapeThis(strLiteral) {
  var escaped = '';
  escaped = strLiteral.replace(/\n/g, '\\n');
  escaped = escaped.replace(/\r/g, '\\r');
  escaped = escaped.replace(/\t/g, '\\t');
  return escaped;
}

function generateVariableAssignment(varType, varName, varValue) {
  assignmentAsm = '';
  if (varValue.type === 'NumberLiteral') {
    if (varType === 'int') {
      assignmentAsm += '\tpush $' + varValue.value + '\n';
      stack.push({
        type: 'LocalVariable',
        name: varName,
        value: varValue.value,
        variableType: varType
      });
    }
  } else if (varValue.type === 'Word') {
    if (varType === 'int') {
      for (var i = 0; i < stack.length; i++) {
```

```
      if (stack[i].type === 'LocalVariable') {
        if (stack[i].name === varValue.value) {
          if (i !== stack.length) {
            assignmentAsm += '\tmov '+ (reverseOffset(i) * 8).toString() + '(%rsp)' + ',%rax\n';
          } else {
            assignmentAsm += '\tmov (%rsp),%rax\n';
          }
          assignmentAsm += '\tpush %rax\n';
          stack.push({
            type: 'LocalVariable',
            name: varName,
            value: stack[i].value,
            variableType: varType
          });
        }
      }
    }
  }
  return assignmentAsm;
}

function generateVariableAssignmentWithAddition(statement) {
  var current = 0;
  var counter = 0;
  var sum = 0;
  var statementAssembly = '';
  statementAssembly = '\txor %rax,%rax\n';
  while (current < statement.length) {
    if (statement[current].type === 'Equal') {
      var varName = statement[current - 1].value;
      var pmCounter = 0 ;
      if (statement[current + 1].type === 'Minus') {
        pmCounter = current + 3;
```

```
    } else {
      pmCounter = current + 2;
    }
    while (statement[pmCounter].type === 'Plus' || statement[pmCounter].type === 'Minus') {
      if (statement[pmCounter - 1].type === 'NumberLiteral' && statement[pmCounter + 1].type ===
'NumberLiteral') {
        if (counter === 0) {
          if (statement[current + 1].type === 'Minus') {
            sum -= parseInt(statement[pmCounter - 1].value);
            statementAssembly += '\tsub $' + statement[pmCounter - 1].value + ',%rax\n';
          } else {
            sum += parseInt(statement[pmCounter - 1].value);
            statementAssembly += '\tadd $' + statement[pmCounter - 1].value + ',%rax\n';
          }
        }
        if (statement[pmCounter].type === 'Plus') {
          sum += parseInt(statement[current + 1].value);
          statementAssembly += '\tadd $' + statement[pmCounter + 1].value + ',%rax\n';
        } else if (statement[pmCounter].type === 'Minus') {
          sum -= parseInt(statement[current + 1].value);
          statementAssembly += '\tsub $' + statement[pmCounter + 1].value + ',%rax\n';
        }
        counter++;
      }
      pmCounter += 2;
    }
  }
  current++;
 }
 if (counter !== 0) {
  statementAssembly += '\tpush %rax\n';
  stack.push({
    type: 'LocalVariable',
    name: statement[1].value,
```

```
      value: sum,
      variableType: "int"
    });
    return statementAssembly;
  }
  return 'Error!';
}

function includeHeaders(globalItems) {
  var includes = [];
  var current = 0;
  var globalStatements = globalItems[0];
  while (current < globalStatements.length) {
    if (globalStatements[current].type === 'Macro') {
      if (globalStatements[current].subtype === 'include') {
        console.log('In the near future, included flies from the C standard library will be added!');
        console.log('But we first need to fully compile every valid C syntax!');
      }
    }
    current++;
  }
  return includes;
}
```

Parser.js

```
// Now we start parsing. We define a function named parser which accepts our tokens array.
function parser(tokens) {

  var current = 0;

  // Inside it, we define another function called walk() which enables use to do some recursive
acrobatics
```

```
function walk() {
  var token = tokens[current];

  /* if the the current token type is equal, then we should check all different possibilities
  such as == and = */
  if (token.type === 'equal') {
    if (tokens[++current].type == 'equal') {
      ++current;
      return {
        type: 'ComparisonE',
        value: token.value + token.value
      };
    } else {
      return {
        type: 'Equal',
        value: token.value
      };
    }
  }

  if (token.type === 'star') {
    current++;
    return {
      type: 'Pointer',
      value: token.value
    };
  }

  if (token.type === 'hash') {
    current++;
    return {
      type: 'Macro',
      value: token.value
    };
```

```
}

// if the token type is 'not' (!), then we check for != too
if (token.type === 'not') {
   if (tokens[++current].type === 'equal') {
      ++current;
      return {
        type: 'ComaprisonN',
        value: token.value + "="
      };
   } else {
     return {
      type: 'Not',
      value: token.value
     };
   }
}

// yawwn, same...
if (token.type === 'plus') {
   if (tokens[++current].type === 'equal') {
      ++current;
      return {
        type: 'IncByNum',
        value: "+="
      };
   } else if (tokens[current].type === 'plus') {
     ++current;
     return {
      type: 'IncByOne',
      value: "++"
     };
   } else {
     return {
```

```
      type: 'Plus',
      value: "+"
     };
   }
}

// same but we remember the arrow sign =>
if (token.type === 'minus') {
 if(tokens[++current].type === 'minus') {
 current++;
   return {
    type: 'DecByOne',
    value: "--"
   };
 } else if (tokens[current].type === 'equal') {
   current++;
   return {
    type: 'DecByNum',
    value: "-="
   };
 } else if (tokens[current].type === 'greater') {
   current++;
   return {
    type: 'Arrow',
    value: "->"
   };
 } else {
   return {
    type: 'Minus',
    value: token.value
   };
 }
}
```

```
// if it's a name token, we chaning it to Word. Don't ask.. :D
if (token.type === 'name') {
   current++;
   return {
    type: 'Word',
    value: token.value
   };
}

if (token.type === 'question') {
   current++;
   return {
    type: 'Question',
    value: token.value
   };
}

if (token.type === 'less') {
 if(tokens[++current].type === 'equal') {
 current++;
   return {
    type: 'LessOrEqual',
    value: "<="
   };
 } else {
   return {
    type: 'Less',
    value: token.value
   };
 }
}

if (token.type === 'and') {
 if(tokens[++current].type === 'and') {
```

```
  current++;
    return {
      type: 'AndAnd',
      value: "&&"
    };
  } else {
    return {
      type: 'And',
      value: token.value
    };
  }
}

if (token.type === 'pipe') {
  if(tokens[++current].type === 'pipe') {
  current++;
    return {
      type: 'OrOr',
      value: "||"
    };
  } else {
    return {
      type: 'Pipe',
      value: token.value
    };
  }
}

if (token.type === 'greater') {
  if(tokens[++current].type === 'equal') {
  current++;
    return {
      type: 'GreaterOrEqual',
      value: ">="
```

```
    };
  } else {
    return {
      type: 'Greater',
      value: token.value
    };
  }
}

if (token.type === 'caret') {
  if(tokens[++current].type === 'equal') {
  current++;
    return {
      type: 'XorEqual',
      value: '^='
    };
  } else {
    return {
      type: 'Xor',
      value: token.value
    };
  }
}

if (token.type === 'comma') {
  current++;
  return {
    type: 'Delimiter',
    value: token.value
  };
}

if (token.type === 'colon') {
  current++;
```

```
  return {
    type: 'Colon',
    value: token.value
  };
}

if (token.type === 'backslash') {
  token = tokens[++current];
  if (token.type === 'name') {
    if (token.value === 't') {
      current++;
      return {
        type: 'Tab',
        value: /\t/
      };
    }

    if (token.value === 'n') {
      current++;
      return {
        type: 'Newline',
        value: /\n/
      };
    }

    if (token.value === 'r') {
      current++;
      return {
        type: 'CRet',
        value: /\r/
      };
    }

    if (token.value === 'b') {
```

```
  current++;
  return {
    type: 'Backspace',
    value: /\b/
  };
}

if (token.value === 'a') {
  current++;
  return {
    type: 'Alert',
    value: /\a/
  };
}

if (token.value === 'v') {
  current++;
  return {
    type: 'VTab',
    value: /\v/
  };
}

if (token.value === 'x') {
  current++;
  return {
    type: 'Hex',
    value: /\x/
  };
}

if (token.value === 'o') {
  current++;
  return {
```

```
      type: 'Oct',
      value: /\o/
     };
    }
   }

   if (token.type === 'question') {
    current++;
    return {
     type: 'QueMark',
     value: /\?/
    };
   }
  }

  /* here we perform some recursive acrobatics. If we encounter an opening bracket, we create a
  new node, call our walk fuction again and push whatever there is inside the bracket,
  inside a child node. When we reach the closing bracket, we stop and push the child node,
  in its parent node */
  if (token.type === 'bracket' &&
    token.value === '['
  ) {
   token = tokens[++current];

   var node = {
    type: 'Arr',
    params: []
   };

   while (
    (token.type !== 'bracket') ||
    (token.type === 'bracket' && token.value !== ']')
   ) {
```

```
    node.params.push(walk());
    token = tokens[current];
  }
  current++;
  return node;
}

// same story here. This time we call it a 'CodeDomain'.
if (token.type === 'curly' &&
    token.value === '{'
) {
  token = tokens[++current];

  var node = {
    type: 'CodeDomain',
    params: []
  };

  while (
    (token.type !== 'curly') ||
    (token.type === 'curly' && token.value !== '}')
  ) {

    node.params.push(walk());
    token = tokens[current];
  }
  current++;
  return node;
}

if (token.type === 'semi') {
  current++;
  return {
    type: 'Terminator',
```

```
    value: token.value
  };
 }

 if (token.type === 'dot') {
  current++;
  return {
    type: 'Dot',
    value: token.value
  };
 }

 if (token.type === 'number') {
  current++;
  return {
    type: 'NumberLiteral',
    value: token.value
  };
 }

 if (token.type === 'string') {
  current++;
  return {
    type: 'StringLiteral',
    value: token.value
  };
 }

 if (token.type === 'forwardslash') {
  current++;
  return {
    type: 'ForwardSlash',
    value: token.value
  };
```

```
}

// same as brackets and curly braces but for paranthesis, we call it 'CodeCave'
if (
 token.type === 'paren' &&
 token.value === '('
) {
 token = tokens[++current];
 let prevToken = tokens[current - 2];
 if (typeof(prevToken) != 'undefined' && prevToken.type === 'name') {
  var node = {
   type: 'CodeCave',
   name: prevToken.value,
   params: []
  };
 } else {
   var node = {
    type: 'CodeCave',
    params: []
   };
 }

 while (
  (token.type !== 'paren') ||
  (token.type === 'paren' && token.value !== ')')
 ) {
  node.params.push(walk());
  token = tokens[current];
 }

 current++;
 return node;
}
```

```
    //if we don't recognize the token, we throw an error.
    throw new TypeError(token.type);
  }

  // we declare this variable named AST, and start our walk() function to parse our tokens.
  let ast = {
    type: 'Program',
    body: [],
  };

  while (current < tokens.length) {
    ast.body.push(walk());
  }

  return ast;
}
```

Processor.js

```
/* After the traversing and transforming the AST to a new AST, we pass the newAST to processor
 to... guess what? Yes! An even bigger and more organized AST! */

function processor(ast) {
  /* first we get our ast body and start finding the global stuff.
  Things like global variables, includes, structs are found here */
  var astBody = ast.body;
  // This variable globalItems will hold all of iur global stuff.
  var globalItems = [];
  // We start by looking for them using findGlobalStatements() function
  var globalStatements = findGlobalStatements(astBody);
  // Here we try to prerpocess out globalStatements to reorder our macro structure
  // This step is beutifies the macro structure in the final AST which means
  // easier to compile in the next step
  globalStatements = preprocessor(globalStatements);
```

```
// and we push it into our globalItems array
globalItems.push({
  type: 'GlobalStatements',
  body: globalStatements
});

// Then we define another top level node called fucntionPack.
// All the found functions will end up here.
var functionPack = [];

// We'll start by finding the functions and reorganize them using findFuncs() function
var foundFuncs = findFuncs(astBody);

// The we start our real task which is processing the body of each function
// We loop though each found function and call processBody() fucntion on them.
for (var i = 0; i < foundFuncs.length; i++) {
  // newBody will hold our organized body of our function
  var newBody = processBody(foundFuncs[i].body);

  // The we'll update our current function.body
  foundFuncs[i].body = newBody;

  //same goes for fucntion arguments :)
  var newFuncArgs = updateFunctionArguments(foundFuncs[i].args);
  foundFuncs[i].args = newFuncArgs;

}

// Then we push our found function with their updated bodies abd arguments into our
// top level functionPack array
functionPack.push({
  type: 'Functions',
  body: foundFuncs
});
```

```
  // At the end, we'll define our final abstract syntax tree structure and name it TheBigAST:)
  // and push our 2 top level arrays: globalItems and functionPack into it.
  // TheBigAST will then be ready to be compiled
  var TheBigAST = [];
  TheBigAST.push(globalItems,functionPack);

  return TheBigAST;
}

// This fucntion will get asBody and return global stuff like global variables, structs
// (and includes which is not yet added)
function findGlobalStatements(astBody) {

  // first we Clone the astBody to be able to fuck it up :D (Nah, seriously!)
  var astBodyClone = astBody;

  // We start by looping through our astBody and look for Terminator ( ; character)
  var current = 0;
  var globalStatements = [];

  while (current < astBodyClone.length) {
    //checking for macros
    if (astBodyClone[0].type === 'Macro') {
      var statement = [];
      if (astBodyClone[1].value === 'include') {
        var macro = [];
        var macroCounter = 0;
        if (astBodyClone[2].type === 'StringLiteral') {
          macro.push(astBodyClone[0]);
          macro.push(astBodyClone[1]);
          macro.push(astBodyClone[2]);
          for (var i = 0; i < 3; i++) {
            astBodyClone.shift();
          }
```

```
    } else {
      while ( astBodyClone[macroCounter].type !== 'Greater') {
        macro.push(astBodyClone[macroCounter]);
        macroCounter++;
      }
      macro.push(astBodyClone[macroCounter]);
      for (var i = 0; i < macroCounter + 1; i++) {
        astBodyClone.shift();
      }
    }
    globalStatements.push({
      type: 'Macro',
      value: macro
    });
    current = 0;
  }
}
//If we find a terminator (ehem)
if (astBodyClone[current].type === 'Terminator') {

  // we create a new array holding our statement
  var statement = [];

  for (var i = 0; i < current + 1; i++) {

    // if the first node is a 'struct', we treat it differently
    if (astBodyClone[i].value === 'struct') {

      // in case of an struct, we try to process its body recursively.
      var instruct = processBody(astBodyClone[i+2].expression.arguments);

      // then we push it in our globalStatements array
      globalStatements.push({
        type: 'struct',
```

```
          name: astBodyClone[current+1].value,
          body: instruct
        });
        i += 4;
      } else {

        // if not, we treat it like a normal statement, pushing each node into our
        // temporary statement array
        statement.push(astBodyClone[i]);
      }
    }

    // Then we delete the nodes we already went through
    for (var i = 0; i < current + 1; i++) {
      astBodyClone.shift();
    }

    // and push the found statement into globalStatements array
    if (statement.length !== 0) {
      globalStatements.push({
        type: 'Statement',
        value: statement
      });
    }
    current = 0;
   }
   current++;
  }

  //in the end, we return our globalStatements
  return globalStatements;
}

// this function re-finds our functions and reorganizes them
```

```
function findFuncs(astBody) {
  // found array will hold our found functions
  var found = [];

  // looping...
  for (var i = 0; i < astBody.length; i++) {

    //if the current node type is a function..
    if (astBody[i].type === 'Function') {

      // here we check if the node has a property of 'callee'
      // because a codeCave node can also be just some parenthesis for other purposes
      if (astBody[i].expression.hasOwnProperty('callee')) {

        // Then we do furthur varification and sanity checks to make sure this is a function definition
        if(astBody[i].expression.callee.type ==='Identifier') {
          if (astBody[i-1].type === 'Word' && astBody[i-2].type === 'Word') {
            if (astBody[i+1].type === 'Function'){
              if (astBody[i+1].expression.type === 'CodeDomain') {

                // If the current function is 'main'
                if (astBody[i].expression.callee.name ==='main') {

                  // we push it to found[] array but we'll name its type EntryPoint
                  found.push({
                    type: 'EntryPoint',
                    name: astBody[i].expression.callee.name,
                    returnType: astBody[i-2].value,
                    args: astBody[i].expression.arguments,
                    body: astBody[i+1].expression.arguments
                  });
                }

                // if not, we name its type 'FunctionDefinition'
```

```
            else {
              found.push({
                type: 'FunctionDefinition',
                name: astBody[i].expression.callee.name,
                returnType: astBody[i-2].value,
                args: astBody[i].expression.arguments,
                body: astBody[i+1].expression.arguments
              });
            }
          }
        }
      }
    }
  }
 }

 // In the end we return found[] array which holds our fucntions
 return found;
}

// This function processes the body of every codeDomain and it's the code of our processing stage
function processBody(inside) {
 // this variable holds our statements.
 // the name statements is not literal. It will group and hold whatever is inside a CodeDoamin
 var statements = [];

 // current variable is used to loop through the body
 var current = 0;

 var start = 0;

 while (current < inside.length) {
  var part = inside[current];
```

```
    // If the current node is a CodeCave (paranthesis) and after it there is a terminator (;)
    // and before it there is a word then this is a fucntion call.
    if (part.type === 'CodeCave' && inside[current + 1].type === 'Terminator' && inside[current - 1].type
=== 'Word') {
      // We push our function call inot our statement
      statements.push({
        type: 'Call',
        params: part.arguments,
        callee: part.callee.name
      });
      current++;
      continue;
    }
    // the rest of the code is self-explanatory...
    // next we look for () followd by {} which may be if, for, while,...
    if (part.type === 'CodeDomain' && inside[current - 1].type === 'CodeCave') {
      if (inside[current - 2].type === 'Word') {
        if (inside[current - 2].value === 'if') {
          if((current - 3) >= 0){
            if (inside[current - 3].type === 'Word') {
              if (inside[current - 3].value === 'else') {
                var inelseif = processBody(part.arguments);
                statements.push({
                  type: 'elseif',
                  condition: inside[current - 1].arguments,
                  body: inif
                });
                current++;
                continue;
              }
            } else {
              var inif = processBody(part.arguments);
              statements.push({
```

```
        type: 'if',
        condition: inside[current - 1].arguments,
        body: inif
      });
      current++;
      continue;
    }
  } else {
    var inif = processBody(part.arguments);
    statements.push({
      type: 'if',
      condition: inside[current - 1].arguments,
      body: inif
    });
    current++;
    continue;
  }
} else if (inside[current - 2].value === 'while') {
  var inwhile = processBody(part.arguments);
  statements.push({
    type: 'while',
    condition: inside[current - 1].arguments,
    body: inwhile
  });
  current++;
  continue;
} else if (inside[current - 2].value === 'for') {
  var infor = processBody(part.arguments);
  statements.push({
    type: 'for',
    condition: inside[current - 1].arguments,
    body: infor
  });
  current++;
```

```
  continue;
 } else if (inside[current - 2].value === 'switch') {
  var count = 0;
  var cases = [];
  var args = inside[current].arguments;
  args.reverse();
  var reverseCaseParts = [];
  while (count < args.length) {
   if (args[count].type !== 'Colon') {
      reverseCaseParts.push(args[count]);
   } else {
     var currentCaseType = args[count+1].type;
     var currentCaseValue = args[count+1].value;
     var currentStatementsGroup = reverseCaseParts.reverse();
     if (args[count+1].value === 'default') {
       count++;
     } else if (args[count+2].value === 'case') {
       count += 2;
     }
     reverseCaseParts = [];
     var caseStatements = processBody(currentStatementsGroup);
     cases.push({
       caseType: currentCaseType,
       caseValue: currentCaseValue,
       caseStatements: caseStatements
     });
   }
   count++;
  }
  statements.push({
   type: 'switch',
   condition: inside[current - 1].arguments,
   body: cases
  });
```

```
        current++;
        continue;
      }
    } else {
      throw new TypeError('Invalid Syntax!');
    }
  } else if (part.type === 'CodeDomain' && inside[current - 1].value === 'else') {
    var inelse = processBody(part.arguments);
    statements.push({
      type: 'else',
      body: inelse
    });
    current++;
    continue;
  } else if (part.type === 'CodeDomain' && inside[current - 1].value === 'do') {
    if (inside[current + 1].type === 'Word' && inside[current + 1].value === 'while') {
      if (inside[current + 2].type === 'CodeCave') {
        var indo = processBody(part.arguments);
        statements.push({
          type: 'do',
          condition: inside[current + 2].arguments,
          body: indo
        });
        current++;
        continue;
      }
    } else {
      throw new TypeError('Invalid Syntax!');
    }
  }

  // here we check for structs...
  else if (part.type === 'CodeDomain' && inside[current - 1].type === 'Word' && inside[current -
2] value === 'struct') {
```

```
    if (inside[current + 1].type === 'Terminator') {
      var instruct = processBody(part.arguments);
      statements.push({
        type: 'struct',
        name: inside[current - 1].value,
        body: instruct
      });
      current++;
      continue;
    }
  }


  if (part.type === 'Terminator') {
    var phrase = [];
    if (inside[current - 1].type === 'CodeCave' && inside[current - 2].value === 'while') {
      current++;
      continue
    }
    if (inside[current - 1].type === 'CodeDomain' && inside[current - 3].value === 'struct') {
      current++;
      continue
    }
    while (start <= current) {
      if (inside[start].type === 'Word') {
        if (inside[start].value === 'if' || inside[start].value === 'for' || inside[start].value === 'switch' ||
inside[start].value === 'while') {
          start += 3;
          continue;
        }
        if (inside[start].value === 'do') {
          start += 5;
          continue;
        }
```

```
        if (inside[start].value === 'else' && inside[start+1].type === 'CodeDomain') {
          start += 2;
          continue;
        }
        // since structs can have bunch of words right after their declaration
        // we loop until we find the terminator
        if (inside[start].value === 'struct') {
          while (inside[start].type !== 'Terminator') {
            start++;
          }
          if (inside[start].type === 'Terminator') {
            start++;
            break;
          }
        }
      }

      phrase.push({
        type: inside[start].type,
        value: inside[start].value
      });
      start++;
    }
    statements.push({
      type: 'Statement',
      value: phrase
    });
  }

  if (current === inside.length - 1) {
    if (part.type !== 'Terminator' && part.type !== 'CodeDomain') {
      throw new TypeError('Error in function definition: Function must return something\n or end with a
; \n or if this fucntion doesn\'t return anything\n you screwed up somewhere!');
      break;
```

```
      }
    }
    current++;
    continue;
  }
  return statements;
}

function updateFunctionArguments(cave) {
  var current = 0;
  var params = [];
  var last = 0;
  while (current < cave.length) {
    if (cave[current].type === 'Delimiter') {
      if ((current - last) === 2) {
        if (cave[current - 2].type === 'Word' && cave[current - 1].type === 'Word') {
          params.push({
            type: cave[current - 2].value,
            name: cave[current - 1].value
          });
          last += current;
        } else {
          throw new TypeError('Error in function definition: Invalid arguments!');
        }
      }
      current++;
      continue;
    }
    if (current === (cave.length - 1)) {
      if ((current - last) === 2) {
        if (cave[current - 1].type === 'Word' && cave[current].type === 'Word') {
          params.push({
            type: cave[current - 1].value
name: cave[current].value
```

```
          });
          last += current;
        } else {
          throw new TypeError('Error in function definition: Invalid arguments!');
        }
      }
      current++;
      continue;
    }
    current++;
  }
  return params;
}


function preprocessor(GlobalStatements) {
  var current = 0;
  while (current < GlobalStatements.length) {
    if (GlobalStatements[current].type === 'Macro') {
      var macro = GlobalStatements[current].value;
      if (macro[1].type === 'Word') {
        if (macro[1].value === 'include') {
          var counter = 0;
          var includedFile = "";
          while (counter < macro.length) {
            if (macro[counter].type === 'Less') {
              counter++;
              while (counter < macro.length && macro[counter].type !== 'Greater') {
                includedFile += macro[counter].value;
                counter++;
              }
            }
            if (macro[counter].type === 'StringLiteral') {
              includedFile += macro[counter].value;
```

```
          }
          counter++;
        }
        GlobalStatements[current] = {
          type: 'Macro',
          subtype: 'include',
          file: includedFile
        }
      }
      if (macro[1].value === 'define') {
        console.log('define macro not yet supported :(');
      }
      if (macro[1].value === 'ifndef') {
        console.log('define macro not yet supported :(');
      }
      if (macro[1].value === 'ifdef') {
        console.log('define macro not yet supported :(');
      }
      if (macro[1].value === 'pragma') {
        console.log('pragma macro not yet supported :(');
      }
    }
  }
  current++;
 }
 return GlobalStatements;
}
```

test.html

```
<!DOCTYPE html>
<html>
<head></head>
<body>
<script src="tokenizer.js"></script>
```

```
<script src="parser.js"></script>
<script src="traverser.js"></script>
<script src="processor.js"></script>
<script src="verifier.js"></script>
<script src="codeGenerator.js"></script>
</body>
</html>
```

tokeniser.js

```
// We start by tokenizing our input by declaring a function named tokenizer()
function tokenizer(input) {
  // variable current will be our index counter
  var current = 0;
  // tokens will be holding all the tokens we found in our input
  var tokens = [];

  // some regex for later use
  var LETTERS = /[a-zA-Z]/;
  var NEWLINE = /\n/;
  var BACKSLASH = /\\/;
  var WHITESPACE = /\s/;
  var NUMBERS = /[0-9]/;

  // now we start looping through each character of our input
  while(current < input.length) {
    var char = input[current];

    /* From here on, we just compare our current character against all the characters
      thet we accept. If there is a match we add 1 to our current variable, push our
      character as a token to our tokens[] array and continue our loop */
    if (char === '=') {
      tokens.push({
        type: 'equal',
```

```
    value: '='
  });
  current++;
  continue;
}

if (char === '*') {
  tokens.push({
    type: 'star',
    value: '*'
  });
  current++;
  continue;
}

if (char === '#') {
  tokens.push({
    type: 'hash',
    value: '#'
  });
  current++;
  continue;
}

if (char === '!') {
  tokens.push({
    type: 'not',
    value: '!'
  });
  current++;
  continue;
}
```

```
if (char === '[' || char === ']') {
  tokens.push({
    type: 'bracket',
    value: char
  });
  current++;
  continue;
}

if (char === '-') {
  tokens.push({
    type: 'minus',
    value: '-'
  });
  current++;
  continue;
}

if (char === '+') {
  tokens.push({
    type: 'plus',
    value: '+'
  });
  current++;
  continue;
}

if (char === '/') {
  // 1) one-line comments
  if (input[++current] === '/') {
    while (current < input.length && !NEWLINE.test(input[current])) {
      current++;
    }
  }
```

```
  // 2) multilne comments
  else if (input[current] === '*') {
    current++;
    while (current < input.length) {
      if (input[current] === '*' && input[++current] === '/') {
        current++;
        break;
      }
      current++;
    }
  }
  // a single slash
  else {
    tokens.push({
      type: 'forwardslash',
      value: '/'
    });
  }
  continue;
}


if (BACKSLASH.test(char)) {
  tokens.push({
    type: 'backslash',
    value: '\\'
  });
  current++;
  continue;
}

if (char === '?') {
  tokens.push({
    type: 'question',
```

```
    value: '?'
  });
  current++;
  continue;
 }

 if (char === '<') {
  tokens.push({
    type: 'less',
    value: '<'
  });
  current++;
  continue;
 }

 if (char === '>') {
  tokens.push({
    type: 'greater',
    value: '>'
  });
  current++;
  continue;
 }

 if (char === '|') {
    tokens.push({
      type: 'pipe',
      value: '|'
    });
    current++;
    continue;
 }

 if (char === '&') {
```

```
  tokens.push({
    type: 'and',
    value: '&'
  });
  current++;
  continue;
 }

 if (char === '%') {
  tokens.push({
    type: 'percent',
    value: '%'
  });
  current++;
  continue;
 }

 if (char === '$') {
  tokens.push({
    type: 'dollar',
    value: '$'
  });
  current++;
  continue;
 }

 if (char === '@') {
  tokens.push({
    type: 'at',
    value: '@'
  });
  current++;
  continue;
 }
```

```
if (char === '^') {
  tokens.push({
    type: 'caret',
    value: '^'
  });
  current++;
  continue;
}

if (char === '~') {
  tokens.push({
    type: 'tilde',
    value: '~'
  });
  current++;
  continue;
}

if (char === '`') {
  tokens.push({
    type: 'grave',
    value: '`'
  });
  current++;
  continue;
}

if (char === '(' || char === ')') {
  tokens.push({
    type: 'paren',
    value: char
  });
  current++;
```

```
  continue;
}

if (char === ':') {
  tokens.push({
    type: 'colon',
    value: ':'
  });
  current++;
  continue;
}

if (char === '.') {
  tokens.push({
    type: 'dot',
    value: '.'
  });
  current++;
  continue;
}

if(char === ',') {
  tokens.push({
    type: 'comma',
    value: ','
  });
  current++;
  continue;
}

if (char === ';') {
  tokens.push({
    type: 'semi',
    value: ';'
```

```
  });
  current++;
  continue;
}

if (char === '{' || char === '}') {
  tokens.push({
    type: 'curly',
    value: char
  });
  current++;
  continue;
}

if(WHITESPACE.test(char) || NEWLINE.test(char)) {
  current++;
  continue;
}

/* If the character is a number, we need to check if the next character is also a number
in order to push them altogether as 1 number. i.e. if there is 762, we push "762" not "7","6","2" */
if(NUMBERS.test(char)) {
  var value = '';

  while(NUMBERS.test(char)) {
    value += char;
    char = input[++current];
  }
  tokens.push({
    type: 'number',
    value: value
  });
  continue;
}
```

```
/* while checking for LETTERS, we also check for NUMBERS and UNDERLINE
(i.e. imagine the input as s0m3_c00l_n4m3) or __my_joke_salary */
if(LETTERS.test(char) || char === '_') {
  var value = char;
  /* need to account for potential end-of-file :D */
  if (++current < input.length) {
    char = input[current];
    /* also need to remember to take care of the last character in the buffer */
    while((LETTERS.test(char) || NUMBERS.test(char) || char === '_') && (current+1 <= input.length)) {
      value += char;
      char = input[++current];
    }
  }
  tokens.push({
    type: 'name',
    value: value
  });
  continue;
}

/* if the character is a sigle quote or a double quote, we will treat it as a string.
Until we haven't found the next double quote or single quote, we continue looping.
When found, then we push the whole value as a string. */
if(char === '\'') {
  var value = '';
  char = input[++current];

  while(char !== '\''){
    value += char;
    char = input[++current];
  }
  char = input[++current];
  tokens.push({
```

```
      type: 'string',
      value: value
    });
    continue;
  }

  if(char === '"') {
    var value = '';
    char = input[++current];

    while(char !== '"'){
      value += char;
      char = input[++current];
    }
    char = input[++current];
    tokens.push({
      type: 'string',
      value: value
    });
    continue;
  }

    /*whatever else, we don't know jack! */
  throw new TypeError('Type Error! Unrecognized Character: ' + char);
 }
 return tokens;
}
```

traverser.js

```
// We try to traverse each node to create a new AST

function traverser(ast, visitor) {
```

```
function traverseArray(array, parent) {
  array.forEach(child => {
    traverseNode(child, parent);
  });
}

function traverseNode(node, parent) {

  let methods = visitor[node.type];

  if (methods && methods.enter) {
    methods.enter(node, parent);
  }

  switch (node.type) {

    case 'Program':
      traverseArray(node.body, node);
      break;

    case 'CodeCave':
      traverseArray(node.params, node);
      break;

    case 'CodeDomain':
      traverseArray(node.params, node);

    case 'Arr':
    case 'NumberLiteral':
    case 'StringLiteral':
    case 'Word':
    case 'Delimiter':
```

```
case 'Terminator':
case 'Equal':
case 'Pointer':
case 'IncByOne':
case 'DecByOne':
case 'Arrow':
case 'Plus':
case 'Minus':
case 'IncByNum':
case 'DecByNum':
case 'ForwardSlash':
case 'ComparisonE':
case 'ComparisonN':
case 'Macro':
case 'Not':
case 'Colon':
case 'Less':
case 'Greater':
case 'LessOrEqual':
case 'GreaterOrEqual':
case 'Dot':
case 'XorEqual':
case 'OrOr':
case 'Pipe':
case 'AndAnd':
case 'And':
case 'Question':
case 'Hex':
case 'Tab':
case 'VTab':
case 'Oct':
case 'Newline':
case 'CRet':
case 'Alert':
```

```
    case 'Backspace':
    case 'QueMark':
      break;

    default:
      throw new TypeError(node.type);
   }

   if (methods && methods.exit) {
    methods.exit(node, parent);
   }
  }

  traverseNode(ast, null);
}

function transformer(ast) {

  let newAst = {
   type: 'Program',
   body: [],
  };

  ast._context = newAst.body;

  traverser(ast, {
   NumberLiteral: {
    enter(node, parent) {
     parent._context.push({
      type: 'NumberLiteral',
      value: node.value
     });
    },
   },
```

```
Word: {
  enter(node, parent) {
    parent._context.push({
      type: 'Word',
      value: node.value
    });
  },
},

IncByOne: {
  enter(node, parent) {
    parent._context.push({
      type: 'IncByOne',
      value: node.value
    });
  },
},

DecByOne: {
  enter(node, parent) {
    parent._context.push({
      type: 'DecByOne',
      value: node.value
    });
  },
},

Arrow: {
  enter(node, parent) {
    parent._context.push({
      type: 'Arrow',
      value: node.value
    });
```

```
  },
},

OrOr: {
  enter(node, parent) {
    parent._context.push({
      type: 'Or',
      value: node.value
    });
  },
},

Pipe: {
  enter(node, parent) {
    parent._context.push({
      type: 'Pipe',
      value: node.value
    });
  },
},

And: {
  enter(node, parent) {
    parent._context.push({
      type: 'And',
      value: node.value
    });
  },
},

AndAnd: {
  enter(node, parent) {
    parent._context.push({
      type: 'AndAnd',
```

```
      value: node.value
    });
  },
},

Tab: {
  enter(node, parent) {
    parent._context.push({
      type: 'Tab',
      value: node.value
    });
  },
},

VTab: {
  enter(node, parent) {
    parent._context.push({
      type: 'VTab',
      value: node.value
    });
  },
},

Hex: {
  enter(node, parent) {
    parent._context.push({
      type: 'Hex',
      value: node.value
    });
  },
},

Oct: {
  enter(node, parent) {
```

```
    parent._context.push({
      type: 'Oct',
      value: node.value
    });
  },
},

Newline: {
  enter(node, parent) {
    parent._context.push({
      type: 'Newline',
      value: node.value
    });
  },
},

CRet: {
  enter(node, parent) {
    parent._context.push({
      type: 'CRet',
      value: node.value
    });
  },
},

Alert: {
  enter(node, parent) {
    parent._context.push({
      type: 'Alert',
      value: node.value
    });
  },
},
```

```
Backspace: {
  enter(node, parent) {
    parent._context.push({
      type: 'Backspace',
      value: node.value
    });
  },
},

QueMark: {
  enter(node, parent) {
    parent._context.push({
      type: 'QueMark',
      value: node.value
    });
  },
},

Plus: {
  enter(node, parent) {
    parent._context.push({
      type: 'Plus',
      value: node.value
    });
  },
},

Minus: {
  enter(node, parent) {
    parent._context.push({
      type: 'Minus',
      value: node.value
    });
  },
```

```
},

IncByNum: {
  enter(node, parent) {
    parent._context.push({
      type: 'IncByNum',
      value: node.value
    });
  },
},

DecByNum: {
  enter(node, parent) {
    parent._context.push({
      type: 'DecByNum',
      value: node.value
    });
  },
},

ForwardSlash: {
  enter(node, parent) {
    parent._context.push({
      type: 'ForwardSlash',
      value: node.value
    });
  },
},

ComparisonE: {
  enter(node, parent) {
    parent._context.push({
      type: 'ComparisonE',
      value: node.value
```

```
    });
  },
},

ComparisonN: {
  enter(node, parent) {
    parent._context.push({
      type: 'ComparisonN',
      value: node.value
    });
  },
},

Macro: {
  enter(node, parent) {
    parent._context.push({
      type: 'Macro',
      value: node.value
    });
  },
},

Not: {
  enter(node, parent) {
    parent._context.push({
      type: 'Not',
      value: node.value
    });
  },
},

Colon: {
  enter(node, parent) {
    parent._context.push({
```

```
      type: 'Colon',
      value: node.value
    });
  },
},

Dot: {
  enter(node, parent) {
    parent._context.push({
      type: 'Dot',
      value: node.value
    });
  },
},

Question: {
  enter(node, parent) {
    parent._context.push({
      type: 'Question',
      value: node.value
    });
  },
},

Less: {
  enter(node, parent) {
    parent._context.push({
      type: 'Less',
      value: node.value
    });
  },
},

Greater: {
```

```
  enter(node, parent) {
    parent._context.push({
      type: 'Greater',
      value: node.value
    });
  },
},

GreaterOrEqual: {
  enter(node, parent) {
    parent._context.push({
      type: 'GreaterOrEqual',
      value: node.value
    });
  },
},

LessOrEqual: {
  enter(node, parent) {
    parent._context.push({
      type: 'LessOrEqual',
      value: node.value
    });
  },
},

XorEqual: {
  enter(node, parent) {
    parent._context.push({
      type: 'XorEqual',
      value: node.value
    });
  },
},
```

```
Arr: {
  enter(node, parent) {
    parent._context.push({
      type: 'Arr',
      value: node.params
    });
  },
},

Pointer: {
  enter(node, parent) {
    parent._context.push({
      type: 'Pointer',
      value: node.value
    });
  },
},

Equal: {
  enter(node, parent) {
    parent._context.push({
      type: 'Equal',
      value: node.value
    });
  },
},

Delimiter: {
  enter(node, parent) {
    parent._context.push({
      type: 'Delimiter',
      value: node.value
    });
```

```
  },
},

Terminator: {
  enter(node, parent) {
    parent._context.push({
      type: 'Terminator',
      value: node.value
    });
  },
},

StringLiteral: {
  enter(node, parent) {
    parent._context.push({
      type: 'StringLiteral',
      value: node.value
    });
  },
},

CodeCave: {
  enter(node, parent) {
    if (typeof(node.name) != 'undefined') {
      var expression = {
        type: 'CodeCave',
        callee: {
          type: 'Identifier',
          name: node.name
        },
        arguments: [],
      };
    } else {
      var expression = {
```

```
      type: 'CodeCave',
      arguments: [],
     };
    }

    node._context = expression.arguments;

    if (parent.type === 'Program') {

     expression = {
      type: 'Function',
      expression: expression
     };
    }

    parent._context.push(expression);
  },
},

CodeDomain: {
  enter(node, parent) {
    var expression = {
      type: 'CodeDomain',
      arguments: [],
    };

    node._context = expression.arguments;

    if (parent.type !== 'CodeDomain') {
      expression = {
        type: 'Function',
        expression: expression
      };
```

```
        }
        parent._context.push(expression);
      },
    }
  });
  return newAst;
}
```

verifier.js

```
var keywords = ['auto','double','int',
'struct','break','else','long','switch',
'case','enum','register','typedef','char',
'extern','return', 'union','const','float',
'short','unsigned','continue','for','signed',
'void','default','goto','sizeof','volatile','do',
'if','static','while'];

var dataTypes = ['int','char','float','double','void'];

function verifier(foundFuncs) {
  var names = [];
  for (var i = 0; i < foundFuncs.length; i++) {
    names.push(foundFuncs[i].name);
  }

  if (!verifyFunctionNames(names)) {
    throw new TypeError('error in function definition: duplicate or illigal name!');
  }

  var current = 0;
  while (current < foundFuncs.length) {
    var func = foundFuncs[current];
```

```
    if (!verifyReturnType(func.returnType)) {
      throw new TypeError('returnType error in function definition for the function: ' + func.name);
      break;
    }

    if (!verifyFunctionArguments(func.args)) {
      throw new TypeError('Error in function definition: Invalid Arguments!');
      break;
    }

    if (!verifyFunctionBody(func.body)) {
      throw new TypeError('Error! Function Body (Statements) are screwed up!');
      break;
    }

    current++;
  }
  return 1;
}


function verifyFunctionBody(funcBody) {
  var current = 0;
  while (current < funcBody.length) {
    var part = funcBody[current];
    if (part.type === 'Statement') {
      var smallPart = part.value;
      if (smallPart[smallPart.length - 1].type !== 'Terminator') {
        return 0;
      }
      for (var i = 0; i < smallPart.length; i++) {
        if (smallPart[i].value === 'return') {
          if (i !== 0) {
```

```
            return 0;
          }
        }
      }
    }

    if (part.type === 'if') {
      var cond = part.condition;
      if (cond.length === 1) {
        if (cond[0].type === 'Word') {
          if (keywords.indexOf(cond[0].value) !== -1) {
            return 0;
          }
        } else if (cond[0].type === 'NumberLiteral') {
            current++;
            continue;
        }
      } else if (cond.length === 3) {
        if (cond[1].type === 'ComparisonE' || cond[1].type === 'ComparisonN' || cond[1].type ===
'Greater' || cond[1].type === 'GreaterOrEqual' || cond[1].type === 'Less' || cond[1].type ===
'LessOrEqual') {
          if (keywords.indexOf(cond[0].value) === -1 && keywords.indexOf(cond[1].value) === -1) {
            current++;
            continue;
          } else {
            return 0;
          }
        }
      }
      if(!verifyFunctionBody(part.body)){
        return 0;
      }
    }
```

```
    current++;

  }
  return 1;
}

function verifyFunctionNames(funcNames) {
  for (var i = 0; i < funcNames.length; i++) {
    if (keywords.indexOf(funcNames[i]) !== -1) {
      return 0;
    }
  }
  var sortedFuncNames = funcNames.slice().sort();
  var duplicates = [];
  for (var i = 0; i < sortedFuncNames.length - 1; i++) {
      if (sortedFuncNames[i + 1] == sortedFuncNames[i]) {
          duplicates.push(sortedFuncNames[i]);
      }
  }
  if (duplicates.length !== 0) {
    return 0;
  }
  return 1;
}

function verifyReturnType(returnType) {
  if (dataTypes.indexOf(returnType) > -1) {
    return 1;
  } else {
    return 0;
  }
}

function verifyFunctionArguments(funcArgs) {
```

```
  var current = 0;
  while (current < funcArgs.length) {

    var arg = funcArgs[current];
    if ((dataTypes.indexOf(arg.type) > -1) && (keywords.indexOf(arg.name) === -1)) {
      current++;
      continue;
    } else {
      return 0;
    }
  }
  return 1;
}
```

7. INPUT

```
        #include <stdio.h>


initGenerate(processor(transformer(parser(tokenizer("
//some comment here :D
int glob = 10;
int jack = 36;

void test_void(void){
   int a = 777;
   int b = a;
   b++;
   return;
}

int test_int_ret(int a, int b){
   a++;
   b++;
   char my_name[] = \"Arash\";
   return a;
}

int main(){
   char greet[] = \"hello\n\";
   int v = 1;
   int f = 8;

   v++;
   f++;

   int k = -4 - 3 + 5 - 7 - 8 + 2 - 32;

   int e = v;

   test(1,2);

   if(v == 2) {
      int y = 5;
      y++;
   }

   if(k == 35) {
```

```
        int p = 55 + 34;

    }

return 1;
}
"
)))))
```

8. OUTPUT

```
.text
.globl _test_void

_test_void:
push %rbp
mov %rsp,%rbp
push $666
mov 0(%rsp),%rax
push %rax
incl (%rsp)
add $16,%rsp
pop %rbp
ret

.globl _test_int_ret

_test_int_ret:
push %rbp
mov %rsp,%rbp
push %rcx
push %rdx
incl 8(%rsp)
incl (%rsp)
mov 8(%rsp), %rax
add $16,%rsp
pop %rbp
ret

.globl main

main:
push %rbp
mov %rsp,%rbp
push $1
push $8
incl 8(%rsp)
incl (%rsp)
xor %rax,%rax
sub $4,%rax
sub $3,%rax
add $5,%rax
sub $7,%rax
sub $8,%rax
add $2,%rax
sub $32,%rax
push %rax
mov 16(%rsp),%rax
push %rax
```

```
cmp $2,24(%rsp)
jne _ifv2_after
push $5
incl (%rsp)
add $8,%rsp

_ifv2_after:
cmp $35,8(%rsp)
jne _ifk35_after
xor %rax,%rax
add $55,%rax
add $34,%rax
push %rax
add $8,%rsp

_ifk35_after: mov $1,%rax
add $32,%rsp
pop %rbp
ret

.data
.globl _glob
_glob:
.long 10

.globl _jack
_jack:
.long 36
```

9. CONCLUSION

In conclusion, the design and implementation of a C compiler using JavaScript as a mini project is a complex and challenging endeavor that offers a unique opportunity to delve into the intricacies of compiler design and programming languages. Through the implementation of various compiler phases, including lexical analysis, parsing, semantic analysis, and code generation, one can gain practical experience in translating C source code into machine- specific instructions. This project allows for experimentation with different design choices, optimization techniques, and error handling strategies, providing valuable insights into real-world compiler implementation.

10. REFERENCES

GEEKS FOR GEEKS

YOUTUBE

STACK OVERFLOW

TUTORIALSPOINT

JAVAPIONT

WIKIPEDIA