

***Report on***

**“Mini Compiler for JavaScript”**

*Submitted in partial fulfillment of the requirements for* ***Sem VI***

***Compiler Design Laboratory***

**Bachelor of Technology**

**in**

**Computer Science & Engineering**

***Submitted by:***

|  |  |
| --- | --- |
| **B Shashank**  **Bhavani Shankar**  **Chandresh L**  **D G Sudheer** | **01FB16ECS086**  **01FB16ECS089**  **01FB16ECS095**  **01FB16ECS101** |

*Under the guidance of*

|  |
| --- |
| **V Madhura**  **and**  **C O Prakash**  PES University, Bengaluru |

**January – May 2019**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

FACULTY OF ENGINEERING

**PES UNIVERSITY**

(Established under Karnataka Act No. 16 of 2013)

100ft Ring Road, Bengaluru – 560 085, Karnataka, India

**TABLE OF CONTENTS**

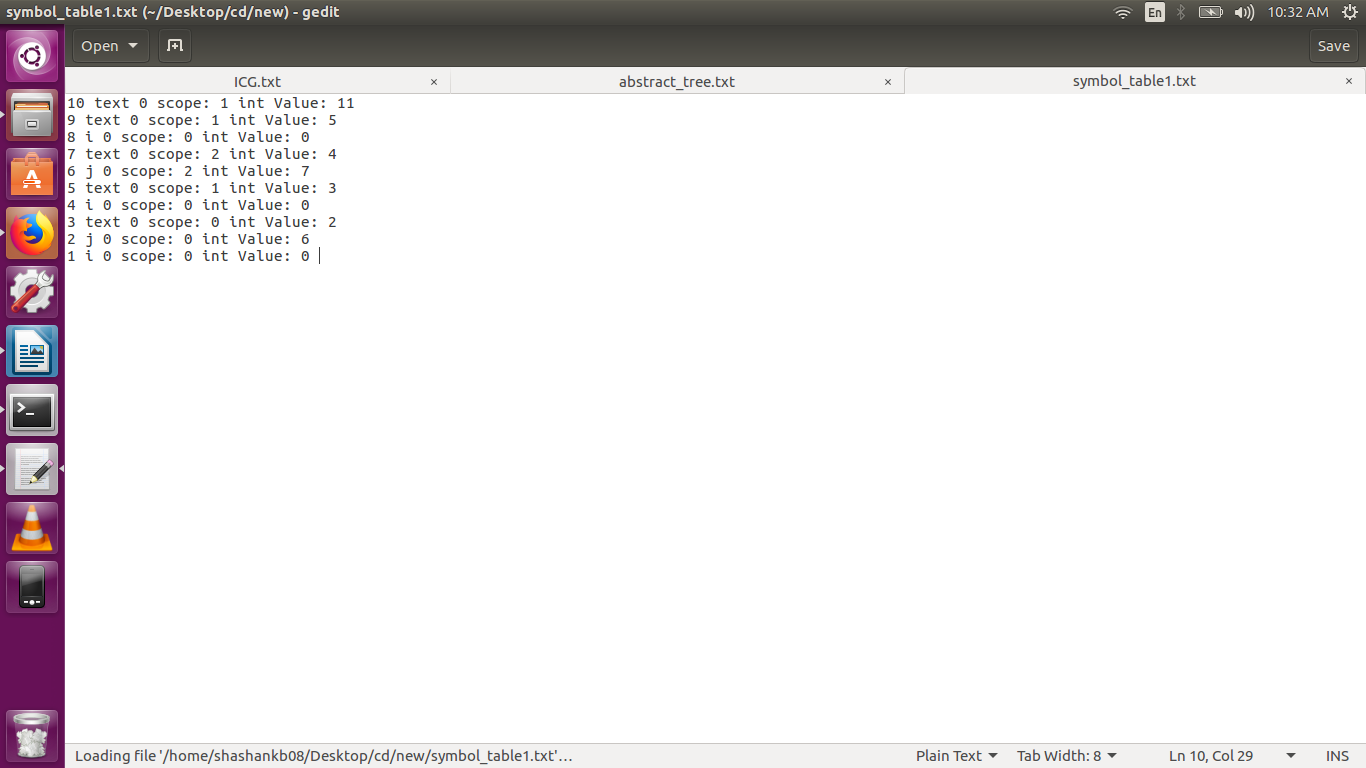
|  |  |  |
| --- | --- | --- |
| **Chapter No.** | **Title** | **Page No.** |
|  | **INTRODUCTION** | **01** |
|  | **ARCHITECTURE OF LANGUAGE:** | **02** |
|  | **LITERATURE SURVEY** | **03** |
|  | **CONTEXT FREE GRAMMAR** |  |
|  | **DESIGN STRATEGY**   * **SYMBOL TABLE CREATION** * **ABSTRACT SYNTAX TREE** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING *-* strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).** |  |
|  | **IMPLEMENTATION DETAILS (TOOL AND DATA STRUCTURES USED in order to implement the following):**   * **SYMBOL TABLE CREATION** * **ABSTRACT SYNTAX TREE (internal representation)** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING *-* strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).** * **Provide instructions on how to build and run your computer** |  |
|  | **RESULTS AND possible shortcomings of your Mini-Compiler** |  |
|  | **SNAPSHOTS** |  |
|  | **CONCLUSIONS** |  |
|  | **FURTHER ENHANCEMENTS** |  |
| **REFERENCES/BIBLIOGRAPHY** | |  |

1. **Introduction -**

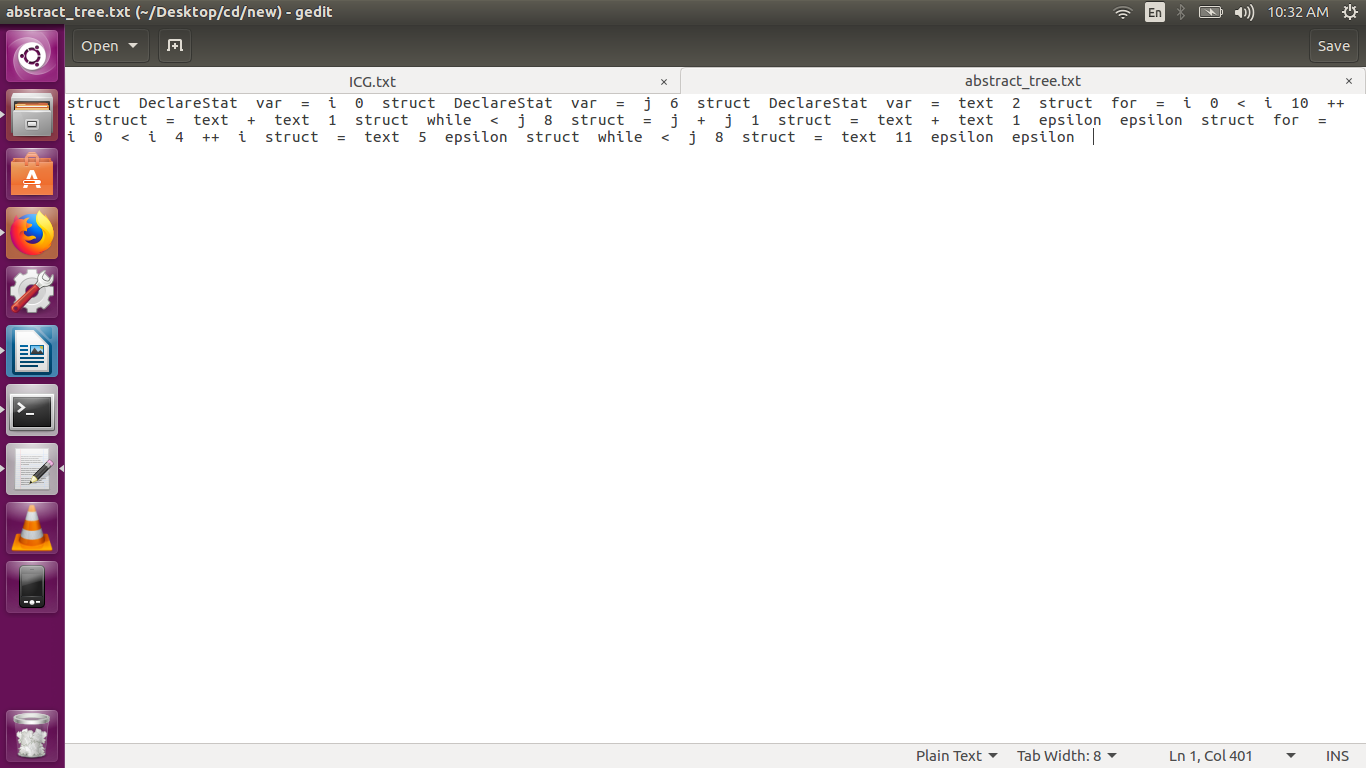
The following project demonstrates the construction and working of a compiler for javascript language using bison-flex .

Input File :

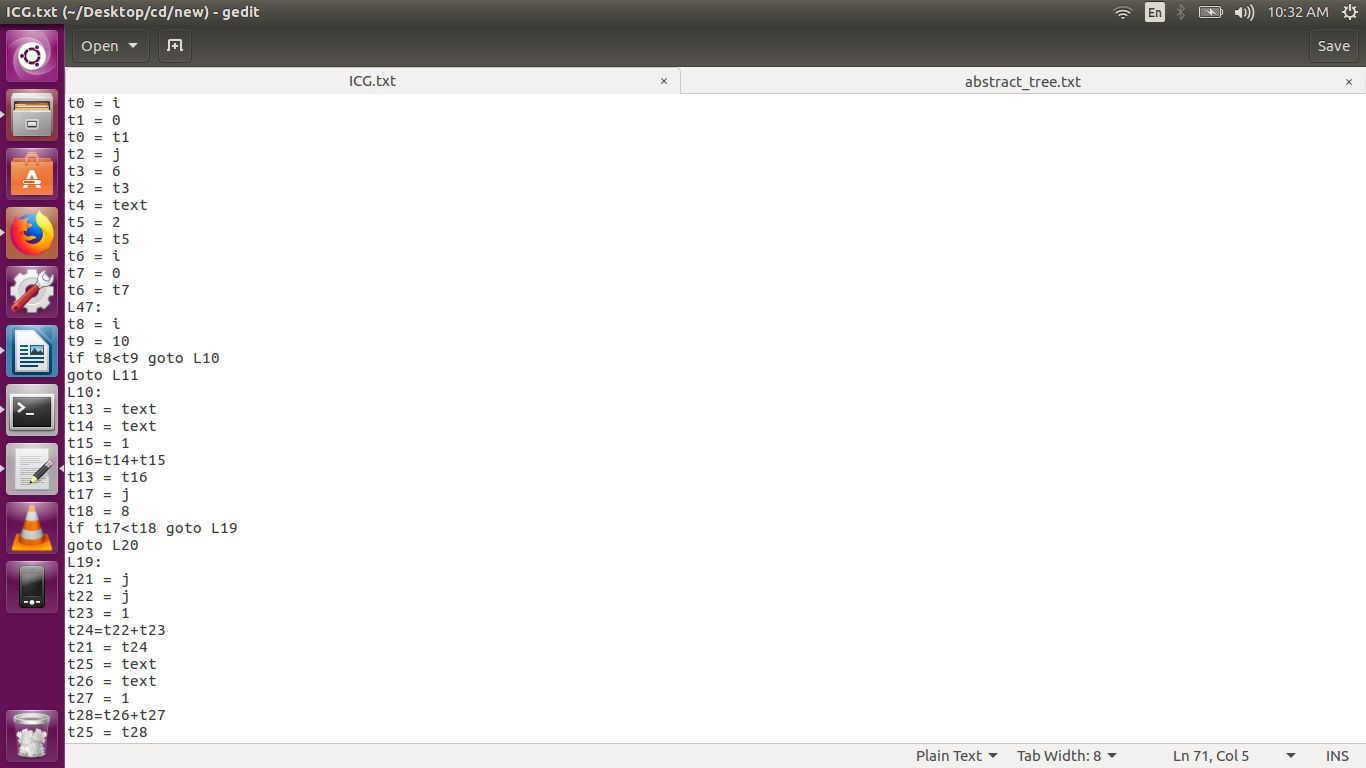
****

**Symbol table output -**

**Abstract Syntax Tree output -**

****

**Intermediate code generation output -**

****

**Architecture of language -**

**Semantics**

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

The syntax and semantics of the language were dealt with in the first phase of the project.

The project deals with the for and while loop constructs in the javascript language . Although there are variable versions of for loop in this project we will be dealing with the regular basic format of for loop .

It handles both the syntax and semantics of the constructs including variable declaration and assignment expressions .

All the syntax errors with respect to the given constructs are identified by the parser and also handles semantic errors as well .

**Literature Survey -**

<http://dinosaur.compilertools.net/bison/index.html>

<https://www.gnu.org/software/bison/manual/html_node/Simple-Error-Recovery.html>

<https://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm>

<https://stackoverflow.com/questions/9796608/error-handling-in-yacc>

**Context Free Grammar Used -**

%union

{

struct node \*ptr\_tree;

char \*str2;

int i;

float f;

};

%token T\_VAR T\_FOR T\_WHILE T\_BRK T\_CNT T\_CM T\_SCLN T\_TRUE T\_FALSE

%token <str2> T\_STRING T\_ID

%token <i> T\_INT

%token <f> T\_FLOAT

%token T\_OBR T\_CBR T\_UADD T\_USUB T\_SOBR T\_SCBR T\_FOBR T\_FCBR

%left T\_MUL T\_DIV T\_MOD T\_ADD T\_SUB

%right T\_EQ T\_MULASN T\_DIVASN T\_MODASN T\_ADDASN T\_SUBASN

%token T\_GT T\_GTE T\_LT T\_LTE T\_CMP T\_NEQ T\_SCMP T\_SNEQ T\_LAND T\_LOR T\_AND T\_OR T\_XOR

%token T\_NT T\_BTNT T\_RS T\_LS

%%

**P**:Statement

**Statement**: DeclareStat T\_SCLN Statement |

AssignExpression T\_SCLN Statement |

T\_FOR T\_OBR ForAssignExpression T\_SCLN CondExpression T\_SCLN UnaryExpression T\_CBR CompoundStatement Statement |

T\_WHILE T\_OBR WhileCondExpression T\_CBR CompoundStatement Statement | ;

**DeclareStat**: T\_VAR list

**list**: T\_ID |

AssignExpression | Array |

T\_ID |

AssignExpression T\_CM list | Array T\_CM list ;

**Array**: T\_SOBR list T\_SCBR | T\_SOBR T\_SCBR ;

**AssignExpression**: Assignment1 Assignment2 ;

**Assignment1**: T\_ID

**Assignment2**: AssignOp Expression | T\_ID AssignOp Array ;

**AssignOp**: T\_EQ | T\_MULASN | T\_DIVASN | T\_MODASN | T\_ADDASN | T\_SUBASN ;

**Expression**: Expression T\_ADD Exp\_mul |

Expression T\_SUB Exp\_mul |

Exp\_mul ;

**Exp\_mul**: Exp\_mul T\_MUL Exp\_end |

Exp\_mul T\_DIV Exp\_end |

Exp\_end ;

**Exp\_end**: T\_INT |

T\_FLOAT |

T\_STRING |

T\_ID |

T\_OBR Expression T\_SUB Expression T\_CBR ;

**ForAssignExpression**: DeclareStat | AssignExpression | ;

**WhileCondExpression**: CondExpression |

T\_TRUE |

T\_FALSE | T\_INT | T\_FLOAT ;

**CondExpression**: Expression CondOpt Expression ;

**CondOpt**: logicalOpt | RelOpt ;

**RelOpt**:

T\_LT | T\_GT | T\_GTE | T\_LTE | T\_CMP | T\_SCMP | T\_NEQ | T\_SNEQ ;

**logicalOpt**: T\_LAND | T\_LOR ;

**UnaryExpression**: UnaryOp T\_ID |

T\_ID UnaryOp | ;

**UnaryOp**: T\_UADD | T\_USUB ;

**CompoundStatement**: T\_FOBR CompStat2;

**CompStat2**: Statement T\_FCBR;

**Design Strategy -**

Symbol Table : The construction of the symbol tree was done using a linked list of structure(struct) . Each record in the symbol consists of the variable type, the value stored and its scope as well . A symbol table entry is updated if the variable value changes or is redeclared with a differnt value .

struct symbol{

char \*var\_name;

int entry;

int var\_type;

char data\_type[7];

union val u;

struct symbol \*next;

};

The struct used for the construction of symbol table

Abstract Syntax Tree : AST generation was done using a struct type where each of these struct type is treated as node of tree and has a link to branch to its children as well . The synatx tree design used in the prokect is not binary rather we could say it is similar to a B-tree and there is no restriction on the number of chldren a node can have .

typedef struct node{

int type;

char name[100];

int val\_type;

char val\_type\_name[100];

struct node \*child[10];

int count;

}node;

The struct used for the construction of AST

Intermediate Code Generation(ICG) : ICG generated is of **Three adress format** and is generated using on the fly mechanism .

Code Optimization – The optimization we are performing is removing the temporaries and removing assignments to a temporary variable.

Error Handling : The error handling method used is skipping the part of the input that is error prone . Type errors are handled in the abstract syntax tree .

**Implementation details -**

Symbol Table - As specified earlier the symbol table is constructed using a struct type which behaves like a linked list where a new entry is made by creating a new struct and linking it with the most prevous struct in the link .

And each time a variable is used or declared we reference the symbol table to check for its vlaue or for its existence in the symbol table . And the scope assigned is of a lower vlaue for a higher scope .

Abstract Syntax Tree – The tree is constructed as a tree of struct type nodes where each node defines a particular token or a leaf node representing the terminals . The tree is grown from left to right . The AST at the end is traversed in an inorder fashion .

Intermediate Code Generatioln - The Inermediate code is generated in three address format . The code so generated is not optimized and has to be optimized in the later part. The mechanism used here is on the fly go .

The code is generated appropriately for each grammar rule and is propogated upward by adding up the new code and printed all together near the start symbol .

**Results and Conclusions -**

The Compiler for Javascript which takes in the inputs of for loop and while loop is built usinng lex and yacc tools.

Intermediate code is generated through fly mechanism.

Abstract syntax tree is also built.

More code optimisation techniques could be used other than what is implemented.

