

***Report on***

**“Mini compiler for LISP”**

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

***Compiler Design Laboratory***

**Bachelor of Technology**

**in**

**Computer Science & Engineering**

***Submitted by:***

|  |  |
| --- | --- |
| **Siddharth Kailasam**  **Arjun Rajesh**  **I Gautam** | **01FB16ECS153**  **01FB16ECS148**  **01FB16ECS140** |

*Under the guidance of*

|  |
| --- |
| **Preet Kanwal**  Assistant Professor  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 (Mini-Compiler is built for which language. Provide sample input and output of your project)** | **01** |
|  | **ARCHITECTURE OF LANGUAGE:**  **What all have you handled in terms of syntax and semantics for the chosen language.** | **02** |
|  | **LITERATURE SURVEY (if any paper referred or link used)** | **03** |
|  | **CONTEXT FREE GRAMMAR (which you used to implement your project)** |  |
|  | **DESIGN STRATEGY (used to implement the following)**  **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 (of different outputs)** |  |
|  | **CONCLUSIONS** |  |
|  | **FURTHER ENHANCEMENTS** |  |
| **REFERENCES/BIBLIOGRAPHY** | |  |

**Introduction**

This project is a mini-compiler implemented for the language LISP. It supports arithmetic operators, logical operators, assignment statements, relational operators, bit-wise operators and print statements. It also supports one iteration method (dotimes) and the decision methods (if-then and if-else), apart from nested loops and nested decision statements. It performs syntax evaluation, intermediate code generation, syntax tree generation, expression evaluation, symbol table evaluation and optimizes constant folding and propagation wherever possible.

The parse tree is printed in the form (parent child1 <child2> <child2> <child3> <child4>) where optional fields are included in angular brackets.

Sample input -

(dotimes (a (dotimes (b 10 11) (- 2 3)) 30 ) (\* 40 20) (+ 40 50))

(if (>= a 10) then (/ 3 4))

(\* a 20)

Output parse tree -

( STMTS ( dotimes ( dotimes 10 11 ( STMTS ( - 2 ( - 3 0 ))( - 2 ( - 3 0 )))) 30 ( STMTS ( \* 40 ( \* 20 1 ))( STMTS ( + 40 ( + 50 0 ))( + 40 ( + 50 0 )))))( STMTS ( if ( >= a 10 )( expn ( / 3 4 )))( STMTS ( \* a ( \* 20 1 ))( \* a ( \* 20 1 )))))

Intemediate code -

a = 0

b = 0

L2:

t0=b<10

ifFalse t0 goto next2

t1 = 2 - 3

t2 = t1

b=b+1

goto L2

next2:

L1:

t3=a<11

ifFalse t3 goto next1

t4 = 40 \* 20

t5 = t4

t6 = 40 + 50

t7 = t6

a=a+1

goto L1

next1:

t8 = a >= 10

ifFalse t8 goto L3

goto next3

L3:

t9 = 3 / 4

next3:

t10 = a \* 20

t11 = t10

Optimized code -

a = 0

b = 0

L2:

t0=b<10

ifFalse t0 goto next2

t1=-1

t2=-1

b=b+1

goto L2

next2:

L1:

t3=a<11

ifFalse t3 goto next1

t4=800

t5=800

t6=90

t7=90

a=a+1

goto L1

next1:

t8 = a >= 10

ifFalse t8 goto L3

goto next3

L3:

t9=0

next3:

t10 = a \* 20

t11 = t10

Input -

(print "hello world!")

(setq a 10)

(print a)

(setq a (\* a 3))

(print a)

Output -

hello world!

10

30

Input -

setq a 30

Output-

syntax error, unexpected T\_ASSIGN, expecting $end

**Architecture of the language**

This project validates the syntax of the given code by using a lexer implemented using lex and parser implemented using yacc. The parser contains the grammar and lexer contains the regular expressions to identify tokens such as keywords, identifiers, numbers, operators and so on.

The semantics of a language is evaluated in the parser and a helper C file. The parser constructs the syntax tree based on the grammar, while the helper C file contains the functions to generate intermediate code into a file and evaluate the symbol table, which is invoked from the parser.

Finally, a python file reads the input from the output 3 address code and outputs the optimized code.

**Literature Survey**

<https://avinashsuryawanshi.files.wordpress.com/2016/10/9.pdf>

<http://dharmanath1244.blogspot.com/2014/11/yacc-program-to-evaluate-arithmetic.html>

<http://btechsource.blogspot.com/2011/11/yacc-program-symbol-table.html>

<http://www.professionalcipher.com/2017/07/intermediate-code-generation-using-lex-and-yacc-using-switch-case-and-control-flow.html>

**Context Free Grammar**

S -> STMTS

STMTS -> STMT STMTS | [epsilon]

STMT -> ( ST2 ) | F

ST2-> ASSIGN | LOOP |DES | LOG | REL | ARITH | BIT

ASSIGN -> ASSIGNEXP id STMT

ASSIGNEXP -> setq | defvar | let

ARITH -> + DP | \* DM | / DD | - DS

DP -> STMT DP | [epsilon]

DM -> STMT DM | [epsilon]

DS -> STMT DS | [epsilon]

DD -> STMT DD | [epsilon]

F -> id | num

REL -> >= DGE | <= DLE | /= DNE | > DG | < DL | = DE

DGE -> STMT STMT

DLE -> STMT STMT

DNE -> STMT DNE

DG -> STMT DG

DL -> STMT DL

DE -> STMT DE

LOG -> logand DBA | logior DBO| logxor DBX| lognor DBN | logeqv DBE

DBA -> STMT DD | [epsilon]

DBO -> STMT DS | [epsilon]

DBX -> STMT DD | [epsilon]

DBN -> STMT DS | [epsilon]

DBE -> STMT STMT

BIT -> and DLA | or DLO | not DLN

DLA -> STMT DD | [epsilon]

DLO -> STMT DS | [epsilon]

DLN -> STMT

LOOP -> dotimes ( id STMT STS ) STMTS

DES -> if CONDN EXPN

CONDN -> STMT

EXPN -> STMT | STMT STMT | then STMT | [epsilon]

STS -> STMT | [epsilon]

**Design Strategy**

**SYMBOL TABLE CREATION**

We used a 2D character pointer array to implement the symbol table.

**ABSTRACT SYNTAX TREE:**

We used a tree structure with each node having 4 children nodes.

**INTERMEDIATE CODE GENERATION:**

We used a stack structure along with the codegen() function to generate thee intermediate code from the stack.

**ERROR HANDLING:**

Catching errors whenever encountered by the parser.

**IMPLEMENTATION DETAILS**

**SYMBOL TABLE CREATION:**

We used a 2D array to construct our symbol table .

Our implementation of the symbol table has 4 columns and 100 rows.

Column 1: Name of the entity

Column 2:Is the identifier valid or not

Column 3 : Is the value valid or not

Column 4 : The value which is stored

Methods related to the symbol table:

int put\_in\_tabl();

This method returns the row number associated with the identifier or else if there is no entry related to that identifier it creates a new entry in table.

**ABSTRACT SYNTAX TREE:**

The abstract syntax tree had a tree structure where each node had the following representation.

typedef struct node

{

struct node \*left;

struct node \*m1;

struct node \*m2;

struct node \*right;

int eval;

char \*token;

}node;

We use the makenode() function to generate a node

node \*mknod(node \*left, node\*m1,node\*m2,node \*right, char \*token,int eval)

**INTERMEDIATE CODE GENERATION**

We use a stack structure to help implement the generation of the intermediate code.

Arithmetic Expression :

We push the desired expression in a stack structure and we use a function called codegen() which pops 2 operands at time and stores the expression in a temporary variable.The temporary variable is then pushed into the stack for further evaluation.

Loop and Descision Expressions:

We use a label generator to create the various labels ( condition.true , condition.false and statement.next).We use a table named labels to help support nested loops and decision.

A combination of on the fly code generation and SDT scheme was implemented to achieve this.

**ERROR HANDLING**

Divide By Zero Error :

For this the second operand for division was evaluated .If it was found to be zero the above error was raised .

Syntax Error :

The grammar rules were defined which when violated calls a function named yyerror() , which deletes the previously drawn parse trees and intermediate code before it raises the error.

Expression Evaluation in Loops and Descisions:

There was a flag variable which was set to 1 when called from a loop . In case setq() was present in the loop the flag was checked and variable was made undefined since it is not possible to perform expression evaluation in yacc with loops and decisions.

**INSTRUCTIONS TO RUN THE CODE**

chmod a+x compile.sh

./compile.sh

Contents of compile.sh

yacc -d -v parser.y && lex lexer.l && gcc -Wall dep.c y.tab.c lex.yy.c -ly -ll && ./a.out

python constantfold.py intermediate.tac > optimized.tac

**RESULTS AND POSSIBLE SHORTCOMINGS OF THE COMPILER**

**Results :**

A compiler for lisp was made which handles structures like decisions (if then, if else) and loops (dotimes).

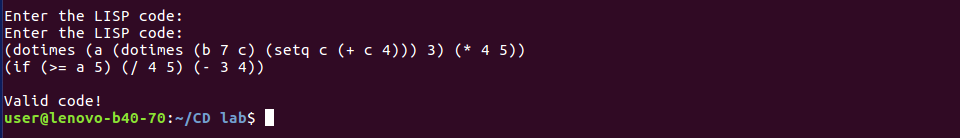
It generates a syntax tree, intermediate code and performs optimizations on the intermediate code.

**Shortcomings :**

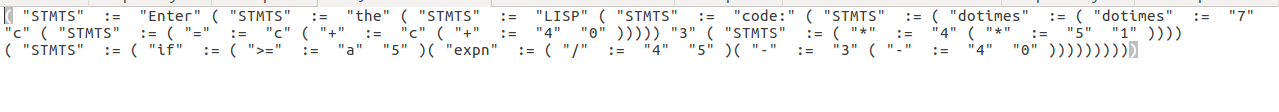
Expression Evaluation in Loops and Descisions was not handled.

**SNAPSHOTS:**

Code1



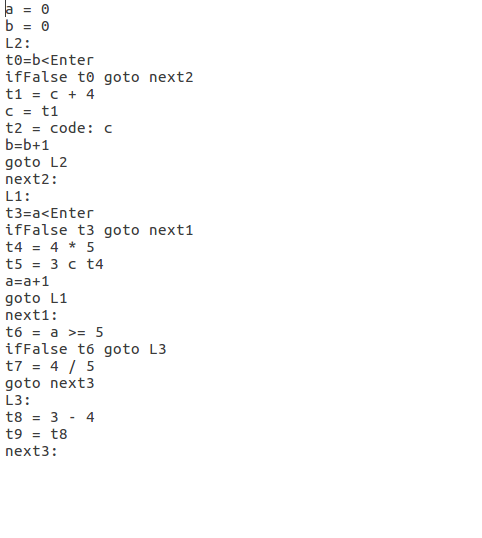
Generating the abstract sytax tree

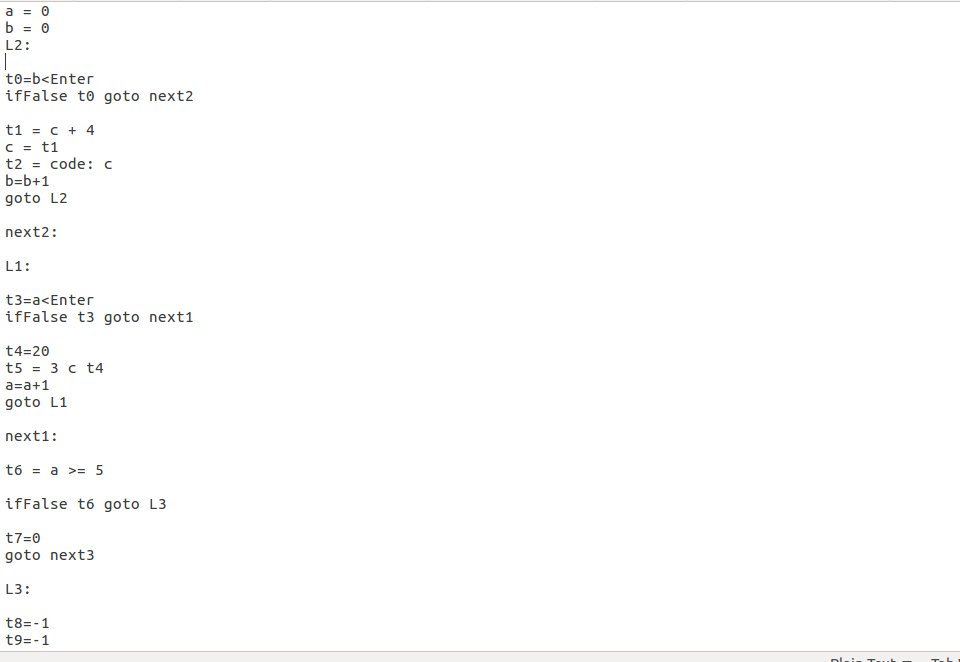


Code 2



ICG for code2





Optimized Code

**CONCLUSIONS**

We learnt the implementation of the various components of the compiler and the interactions between the various layers of the compiler.

**FURTHER ENHANCEMENTS :**

Expression Evaluation in Loops and Decisions was not handled.

More optimizations to the intermediate code could be done.

A more elegant representation of the syntax tree could be implemented.

**REFERENCES/BIBLIOGRAPHY**

<https://avinashsuryawanshi.files.wordpress.com/2016/10/9.pdf>

<http://dharmanath1244.blogspot.com/2014/11/yacc-program-to-evaluate-arithmetic.html>

<http://btechsource.blogspot.com/2011/11/yacc-program-symbol-table.html>

<http://www.professionalcipher.com/2017/07/intermediate-code-generation-using-lex-and-yacc-using-switch-case-and-control-flow.html>