Project Report

Lisp Compiler

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Prateek Nayak	**PES1201800054**
Suhas R	**PES1201800186**
Vishnu Erapalli	**PES1201801297**

Under the guidance of

Madhura V
Professor
PES University, Bengaluru

TABLE OF CONTENTS

Chapter 1

Introduction

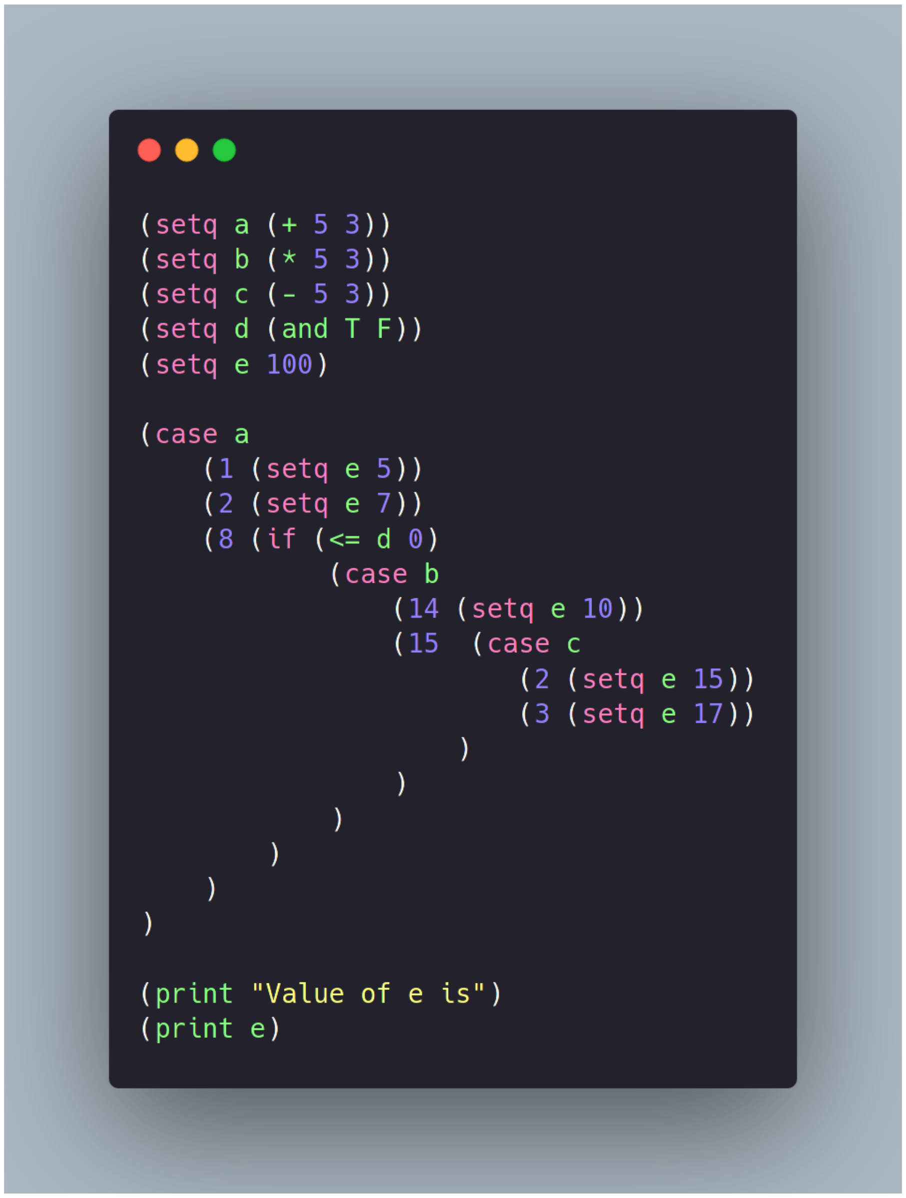
Mini-LISP is a compiler that extends on a subclass of common-LISP that covers integer arithmetic, an experimental string handling, if clause, if-else clause and switch clause. The authors of the module have taken liberty to handle strings in an experimental way that strays away from the common-LISP standard to give a better experience for the users.

As a part of the project, the authors have implemented:

- Lexical Analysis
- Parsing
- Abstract Syntax Tree creation
- Syntax Directed Translation to 3 Address Code
- Intermediate code optimization
- Superoptimization using pre-computation
- Assembly code generation to NASM targeting x86_64 machines running GNU/Linux

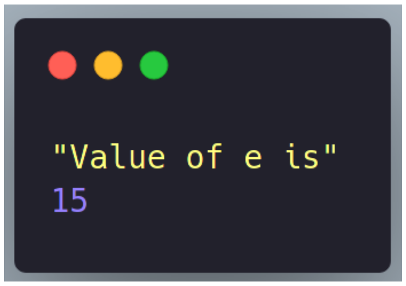
To generate relocatable binary, we are using GCC to convert object file generated by NASM assembler into a relocatable binary that can be run on an x86_64 machine running GNU/Linux based operating system.

For sake of testing all constructs at once, we use the following file as the test input:



```
(setq a (+ 5 3))
(setq b (* 5 3))
(setq c (- 5 3))
(setq d (and T F))
(setq e 100)

(case a
    (1 (setq e 5))
    (2 (setq e 7))
    (8 (if (<= d 0)
            (case b
                (14 (setq e 10))
                (15  (case c
                        (2 (setq e 15))
                        (3 (setq e 17))
                    )
                )
            )
        )
    )
)

(print "Value of e is")
(print e)
```

The output of the run being the following:



```
"Value of e is"
15
```

Chapter 2

Architecture of Language

As a part of the undertaking, the authors of the project have implemented:

- setq statement
- print statement
- Integer storage
- String storage*
- if clause
- if-else clause
- switch clause
- relational operation
- Integer arithmetic operator

* marks the experimental feature that strays away from the common-LISP standard. This was done as a part of undertaking to show a different train of thought and based on the request of the guiding professor.

Strings and integers can be stored in unique variables. These variables can be used for arithmetic operation. Arithmetic operations on string aren't defined and any operation of string with an integer will default to returning the integer. Operations dealing with only strings will be ignored. The variables storing string can be used for printing and as a condition for two-way branching statements where an empty string is considered as a false value.

As a part of optimization effort, we use precomputation to compute values and branch direction at the compile time. Although conversion to unoptimized Intermediate code is not a single pass operation, the optimization of intermediate code is done in a single pass thanks to strictly forward banching.

Due to design restriction, switch nesting of only 100 levels is supported. Anything larger will cause a stack overflow and will eventually lead to undefined behavior.

The compiler is built using:

- POSIX Lex
- POSIX Yacc
- GCC
- NASM

Chapter 3

Literature Survey

The following papers were referred when undertaking this project:

- T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the gnu C compiler. In Proceedings of the ACM SIGPLAN’92 Conference on Programming Language Design and Implementation, volume 27, pages 341–352, San Francisco, CA, June 1992.

- J. F. Reiser, "Compiling three-address code for C programs," in The Bell System Technical Journal

Chapter 4

Context Free Grammar

The grammar used for parsing high level LISP syntax to Intermediate code is:

```
S               -> PROGRAM;
PROGRAM                 -> STMT STMTS
                | STMT;
STMTS           -> STMT STMTS
                | STMT;
STMT            -> EXP
                | SET_STMT
                | PRINT_STMT
                | IF_ELSE
                | CASE;
PRINT_STMT              -> '(' T_print EXP ')';
EXP             -> BOOL
                | NUM
                | STR
                | VARIABLE
                | NUM_OP
                | LOGICAL_OP;
NUM_OP                  -> PLUS
                | MINUS
                | MULTIPLY
                | DIVIDE
                | MODULES;
PLUS            -> '(' '+' EXP EXP ')';
MINUS           -> '(' '-' EXP EXP ')' ;
MULTIPLY        -> '(' '*' EXP EXP ')';
DIVIDE          -> '(' '/' EXP EXP ')';
MODULES         -> '(' T_mod EXP EXP ')' ;
LOGICAL_OP              -> AND_OP
                | OR_OP
                | NOT_OP
                | GREATER
                | SMALLER
                | EQUAL
                | GREATER_EQUAL
                | SMALLER_EQUAL;
```

```
AND_OP         -> '(' T_and EXP EXP ')' ;
OR_OP          -> '(' T_or EXP EXP ')' ;
NOT_OP         -> '(' T_not EXP ')';
GREATER        -> '(' '>' EXP EXP ')';
SMALLER        -> '(' '<' EXP EXP ')';
EQUAL          -> '(' '=' EXP EXP ')';
GREATER_EQUAL -> '(' T_geq EXP EXP ')' ;
SMALLER_EQUAL -> '(' T_leq EXP EXP ')';
SET_STMT                -> '(' T_setq VARIABLE EXP ')';

IF_ELSE                 -> '(' T_if EXP STMT STMT ')'
               | '(' T_if EXP STMT ')' ;
CASE           -> '(' T_case VARIABLE DIFFCASES ')';
DIFFCASES               -> DIFFCASE DIFFCASES
               | DIFFCASE;
DIFFCASE                -> '(' NUM STMTS ')' ;
VARIABLE                -> T_id;
NUM            -> T_number;
BOOL           -> T_bool_val;
STR            -> T_str;
```

Chapter 5

DESIGN STRATEGY

The design strategy for the individual components are as follows:

- ***Symbol Table*** - Symbol Table entries are created when a value is assigned to the variable for the first time. The type is inferred from the rvalue. If the rvalue is a constant, the type of constant is recorded. If it is another variable, the metadata of its type is copied over to the new variable

- ***Intermediate Code Generation*** - The Abstract Syntax Tree created using context free grammar mentioned above is translated using a Syntax Directed Translation scheme that targets the supported features of the language.

- ***Intermediate Code Optimization*** - The strategy used to optimize code is constant folding, constant propagation, branch computation, and dead code elimination.

- ***Error Handling*** - Errors aren't silently corrected as this promotes bad programming practices. The compiler reports the errors at compile time as line numbers and expects users to correct them.

Chapter 6

Implementation Details

The design strategy for the individual components are as follows:

- ***Symbol Table*** - Symbol Table is an array of structure that stores the identifier, type, line number and scope. Identifiers are repeated if the identifier is reassigned with new data.

- ***Intermediate Code Generation*** - The Abstract Syntax Tree created using context free grammar mentioned above is translated using a Syntax Directed Translation scheme that targets the supported features of the language. This is done by traversing the AST and writing relevant intermediate 3 address code to the IC file.

- ***Intermediate Code Optimization*** - Intermediate Code generated have strictly forward branches and hence precomputation is used for constant folding, constant propagation, and branch computation. When encountering a print a temporary variable is allocated with the value to be printed and written to the optimized IC file. This leads to dead code elimination.

- ***Error Handling*** - Errors aren't silently corrected as this promotes bad programming practices. The compiler reports the errors at compile time as line numbers and expects users to correct them.

Chapter 7

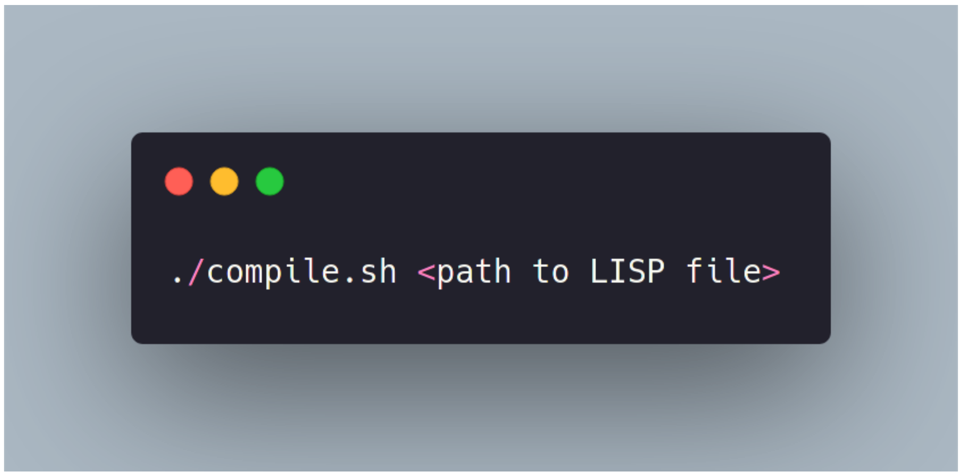
Results

Due to the small selective subset of common-LISP, our compiler is able to generate a program that uses a single register (to store addresses of data to be printed) and function calls to print making it the most effective design to reduce register usage.

The program only supports signed integer computation but can be extended to IEE754 Floating Point numbers. The intermediate code after optimization however will be the same format as a result of strictly forward branching and precomputation.

An interesting study would be to accept user input which would disable precomputation due to the dynamic nature of user input.

Run instructions

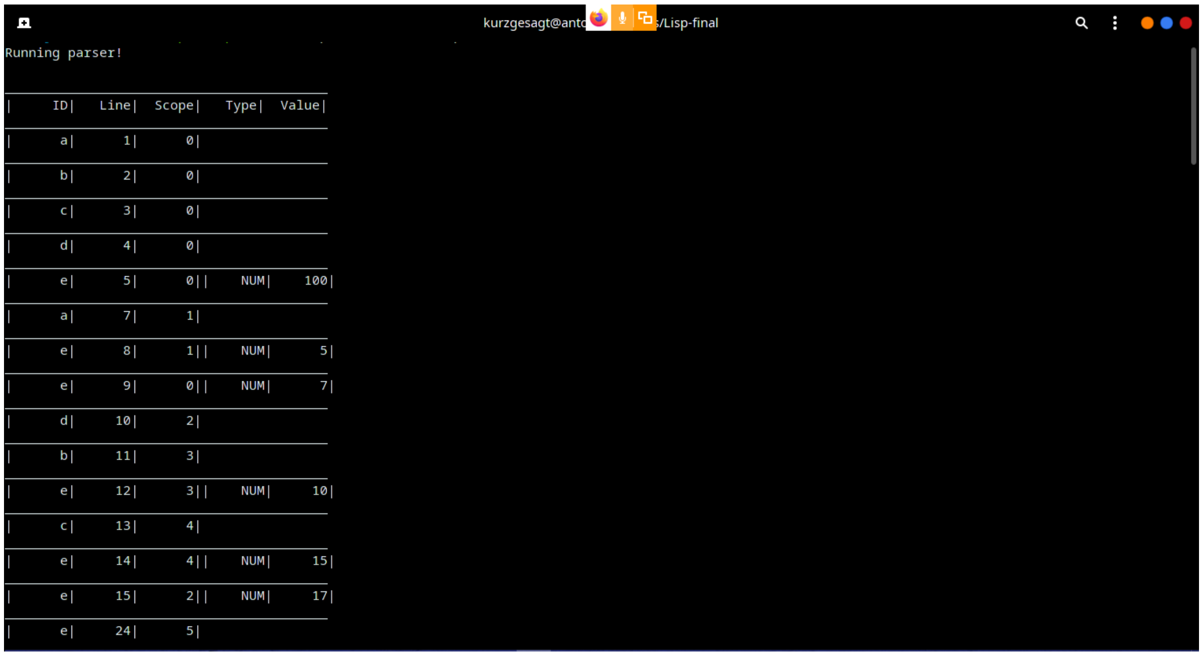


```
./compile.sh <path to LISP file>
```

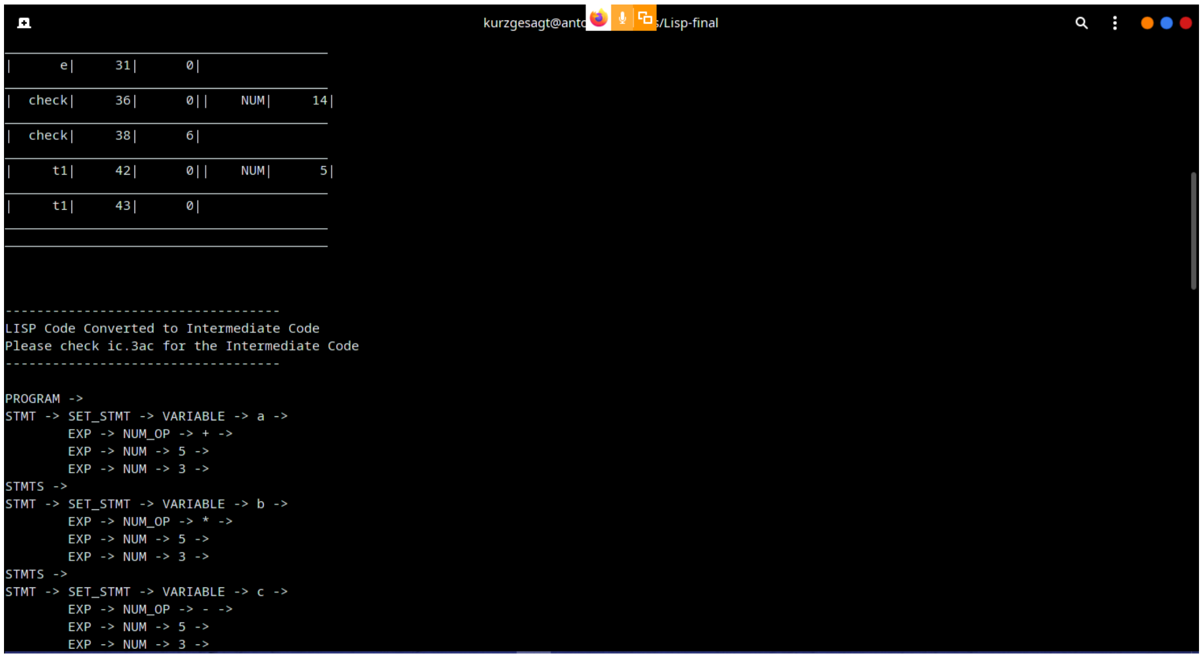
Chapter 8

Snapshots

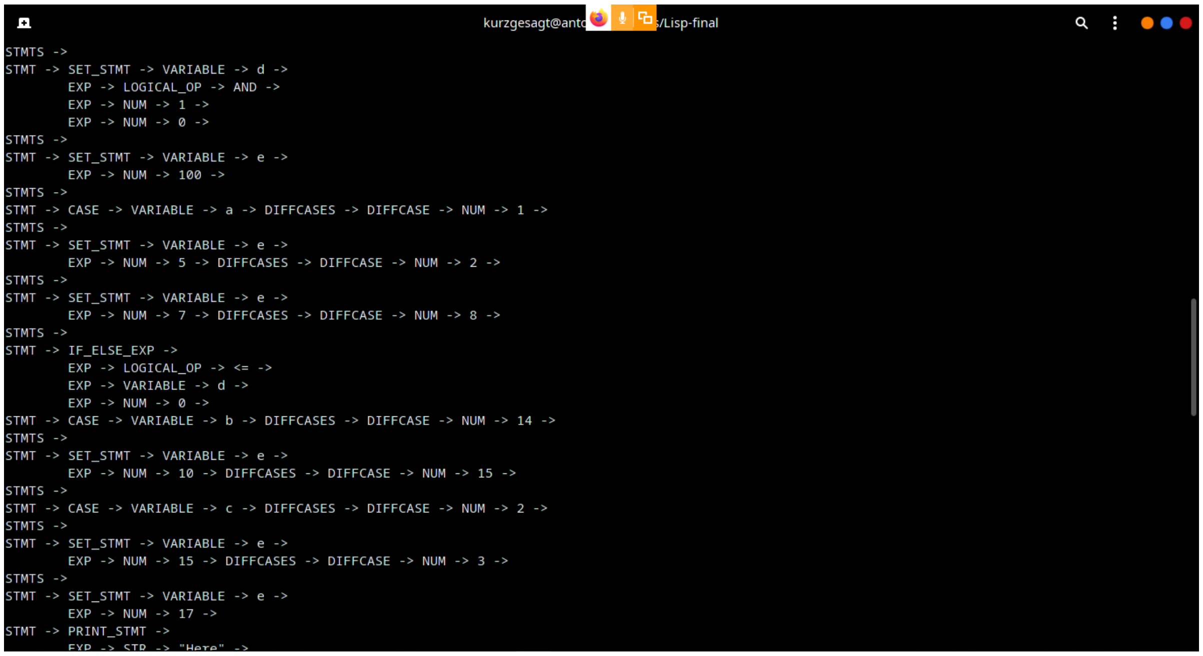
The following are snapshots generated by running the compiler script



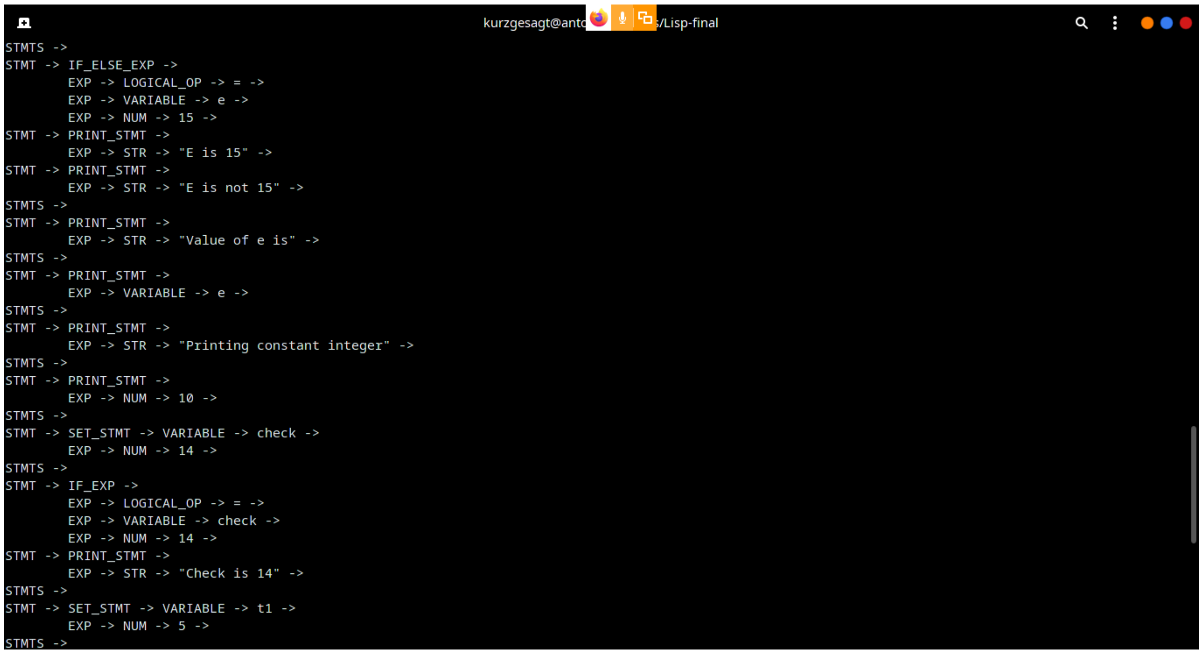
```
Running parser!

___________________________________________
|      ID|    Line|  Scope|   Type|  Value|
___________________________________________
|       a|       1|      0|
___________________________________________
|       b|       2|      0|
___________________________________________
|       c|       3|      0|
___________________________________________
|       d|       4|      0|
___________________________________________
|       e|       5|      0||    NUM|    100|
___________________________________________
|       a|       7|      1|
___________________________________________
|       e|       8|      1||    NUM|      5|
___________________________________________
|       e|       9|      0||    NUM|      7|
___________________________________________
|       d|      10|      2|
___________________________________________
|       b|      11|      3|
___________________________________________
|       e|      12|      3||    NUM|     10|
___________________________________________
|       c|      13|      4|
___________________________________________
|       e|      14|      4||    NUM|     15|
___________________________________________
|       e|      15|      2||    NUM|     17|
___________________________________________
|       e|      24|      5|
```



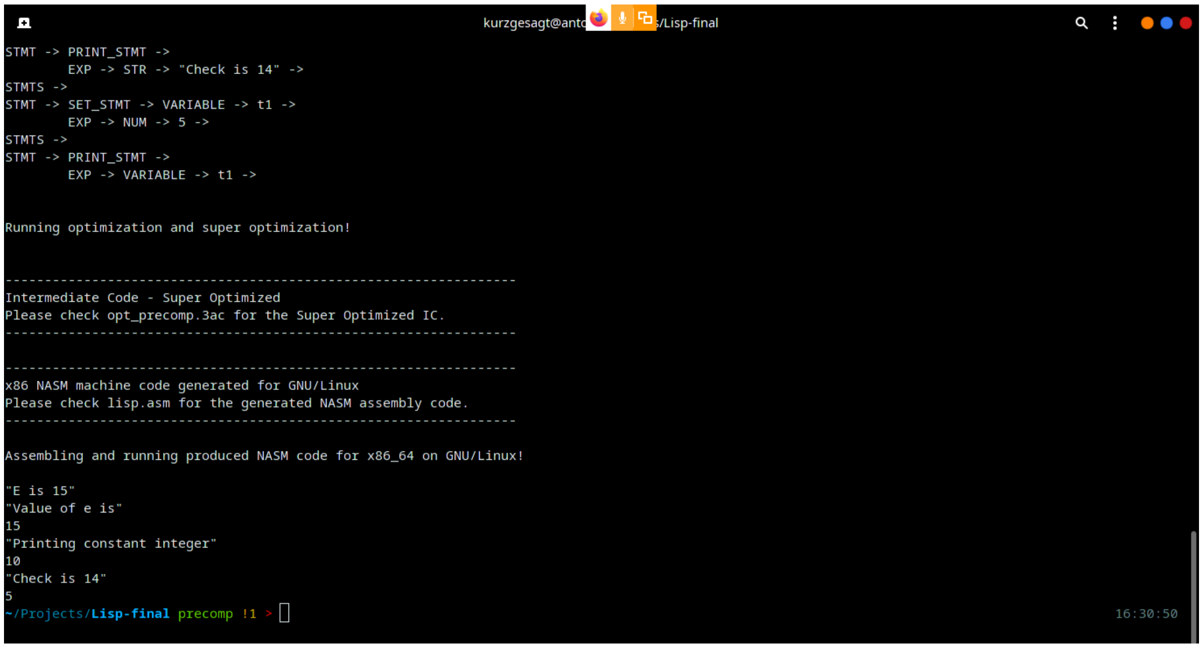
```
___________________________________________
|       e|      31|      0|
___________________________________________
|   check|      36|      0||    NUM|     14|
___________________________________________
|   check|      38|      6|
___________________________________________
|      t1|      42|      0||    NUM|      5|
___________________________________________
|      t1|      43|      0|
___________________________________________
___________________________________________



-----------------------------------
LISP Code Converted to Intermediate Code
Please check ic.3ac for the Intermediate Code
-----------------------------------

PROGRAM ->
STMT -> SET_STMT -> VARIABLE -> a ->
        EXP -> NUM_OP -> + ->
        EXP -> NUM -> 5 ->
        EXP -> NUM -> 3 ->
STMTS ->
STMT -> SET_STMT -> VARIABLE -> b ->
        EXP -> NUM_OP -> * ->
        EXP -> NUM -> 5 ->
        EXP -> NUM -> 3 ->
STMTS ->
STMT -> SET_STMT -> VARIABLE -> c ->
        EXP -> NUM_OP -> - ->
        EXP -> NUM -> 5 ->
        EXP -> NUM -> 3 ->
```



```
STMTS ->
STMT -> SET_STMT -> VARIABLE -> d ->
        EXP -> LOGICAL_OP -> AND ->
        EXP -> NUM -> 1 ->
        EXP -> NUM -> 0 ->
STMTS ->
STMT -> SET_STMT -> VARIABLE -> e ->
        EXP -> NUM -> 100 ->
STMTS ->
STMT -> CASE -> VARIABLE -> a -> DIFFCASES -> DIFFCASE -> NUM -> 1 ->
STMTS ->
STMT -> SET_STMT -> VARIABLE -> e ->
        EXP -> NUM -> 5 -> DIFFCASES -> DIFFCASE -> NUM -> 2 ->
STMTS ->
STMT -> SET_STMT -> VARIABLE -> e ->
        EXP -> NUM -> 7 -> DIFFCASES -> DIFFCASE -> NUM -> 8 ->
STMTS ->
STMT -> IF_ELSE_EXP ->
        EXP -> LOGICAL_OP -> <= ->
        EXP -> VARIABLE -> d ->
        EXP -> NUM -> 0 ->
STMT -> CASE -> VARIABLE -> b -> DIFFCASES -> DIFFCASE -> NUM -> 14 ->
STMTS ->
STMT -> SET_STMT -> VARIABLE -> e ->
        EXP -> NUM -> 10 -> DIFFCASES -> DIFFCASE -> NUM -> 15 ->
STMTS ->
STMT -> CASE -> VARIABLE -> c -> DIFFCASES -> DIFFCASE -> NUM -> 2 ->
STMTS ->
STMT -> SET_STMT -> VARIABLE -> e ->
        EXP -> NUM -> 15 -> DIFFCASES -> DIFFCASE -> NUM -> 3 ->
STMTS ->
STMT -> SET_STMT -> VARIABLE -> e ->
        EXP -> NUM -> 17 ->
STMT -> PRINT_STMT ->
        EXP -> STR -> "Here" ->
```

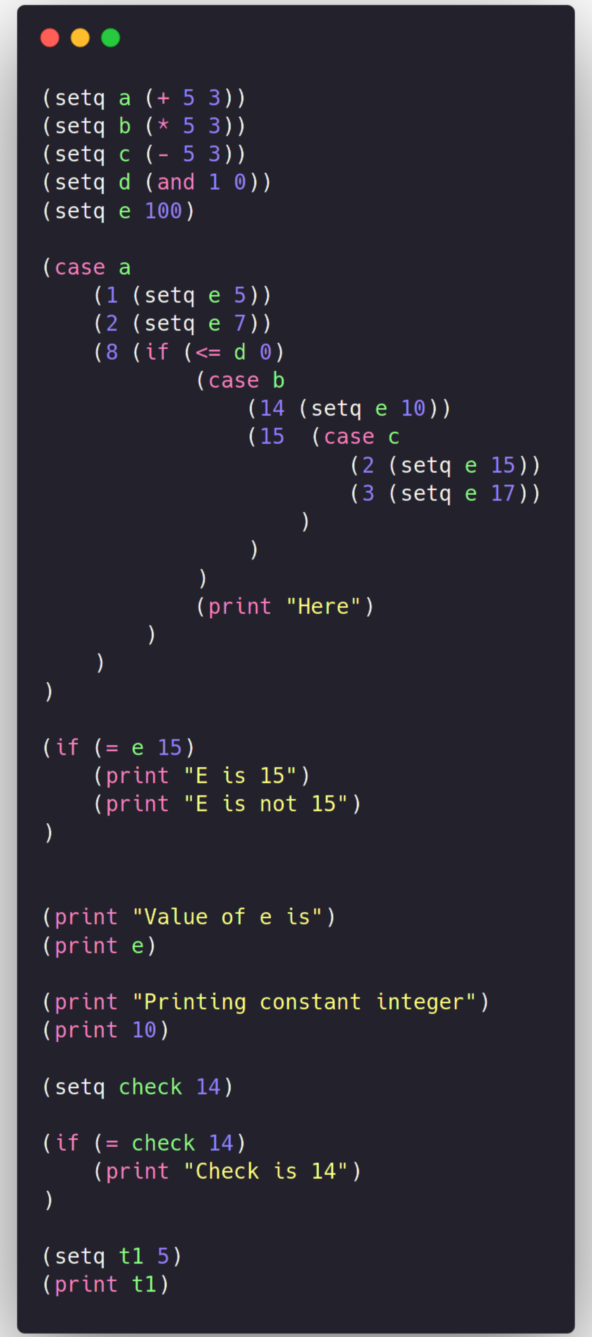


```
STMTS ->
STMT -> IF_ELSE_EXP ->
        EXP -> LOGICAL_OP -> = ->
        EXP -> VARIABLE -> e ->
        EXP -> NUM -> 15 ->
STMT -> PRINT_STMT ->
        EXP -> STR -> "E is 15" ->
STMT -> PRINT_STMT ->
        EXP -> STR -> "E is not 15" ->
STMTS ->
STMT -> PRINT_STMT ->
        EXP -> STR -> "Value of e is" ->
STMTS ->
STMT -> PRINT_STMT ->
        EXP -> VARIABLE -> e ->
STMTS ->
STMT -> PRINT_STMT ->
        EXP -> STR -> "Printing constant integer" ->
STMTS ->
STMT -> PRINT_STMT ->
        EXP -> NUM -> 10 ->
STMTS ->
STMT -> SET_STMT -> VARIABLE -> check ->
        EXP -> NUM -> 14 ->
STMTS ->
STMT -> IF_EXP ->
        EXP -> LOGICAL_OP -> = ->
        EXP -> VARIABLE -> check ->
        EXP -> NUM -> 14 ->
STMT -> PRINT_STMT ->
        EXP -> STR -> "Check is 14" ->
STMTS ->
STMT -> SET_STMT -> VARIABLE -> t1 ->
        EXP -> NUM -> 5 ->
STMTS ->
```


kurzgesagt@anto /Lisp-final

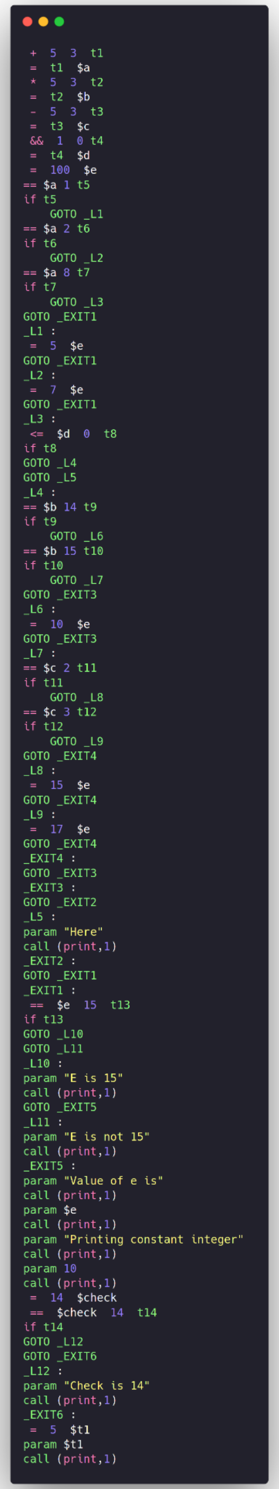
```
STMT -> PRINT_STMT ->
        EXP -> STR -> "Check is 14" ->
STMTS ->
STMT -> SET_STMT -> VARIABLE -> t1 ->
        EXP -> NUM -> 5 ->
STMTS ->
STMT -> PRINT_STMT ->
        EXP -> VARIABLE -> t1 ->


Running optimization and super optimization!


-----------------------------------------------------------------
Intermediate Code - Super Optimized
Please check opt_precomp.3ac for the Super Optimized IC.
-----------------------------------------------------------------

-----------------------------------------------------------------
x86 NASM machine code generated for GNU/Linux
Please check lisp.asm for the generated NASM assembly code.
-----------------------------------------------------------------

Assembling and running produced NASM code for x86_64 on GNU/Linux!

"E is 15"
"Value of e is"
15
"Printing constant integer"
10
"Check is 14"
5
~/Projects/Lisp-final precomp !1 >                                                  16:30:50
```

Following is the input file:



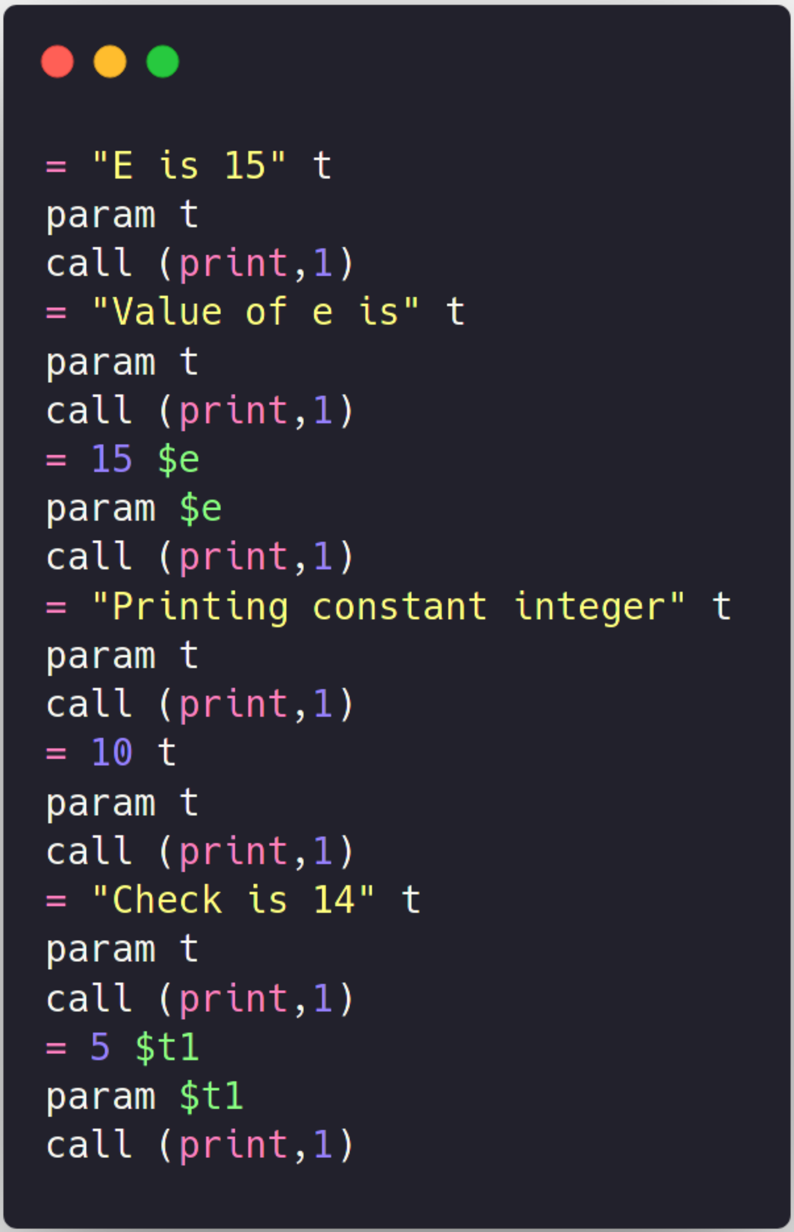
```
(setq a (+ 5 3))
(setq b (* 5 3))
(setq c (- 5 3))
(setq d (and 1 0))
(setq e 100)

(case a
    (1 (setq e 5))
    (2 (setq e 7))
    (8 (if (<= d 0)
            (case b
                (14 (setq e 10))
                (15  (case c
                        (2 (setq e 15))
                        (3 (setq e 17))
                    )
                )
            )
            (print "Here")
        )
    )
)

(if (= e 15)
    (print "E is 15")
    (print "E is not 15")
)


(print "Value of e is")
(print e)

(print "Printing constant integer")
(print 10)

(setq check 14)

(if (= check 14)
    (print "Check is 14")
)

(setq t1 5)
(print t1)
```

Following is the intermediate code:



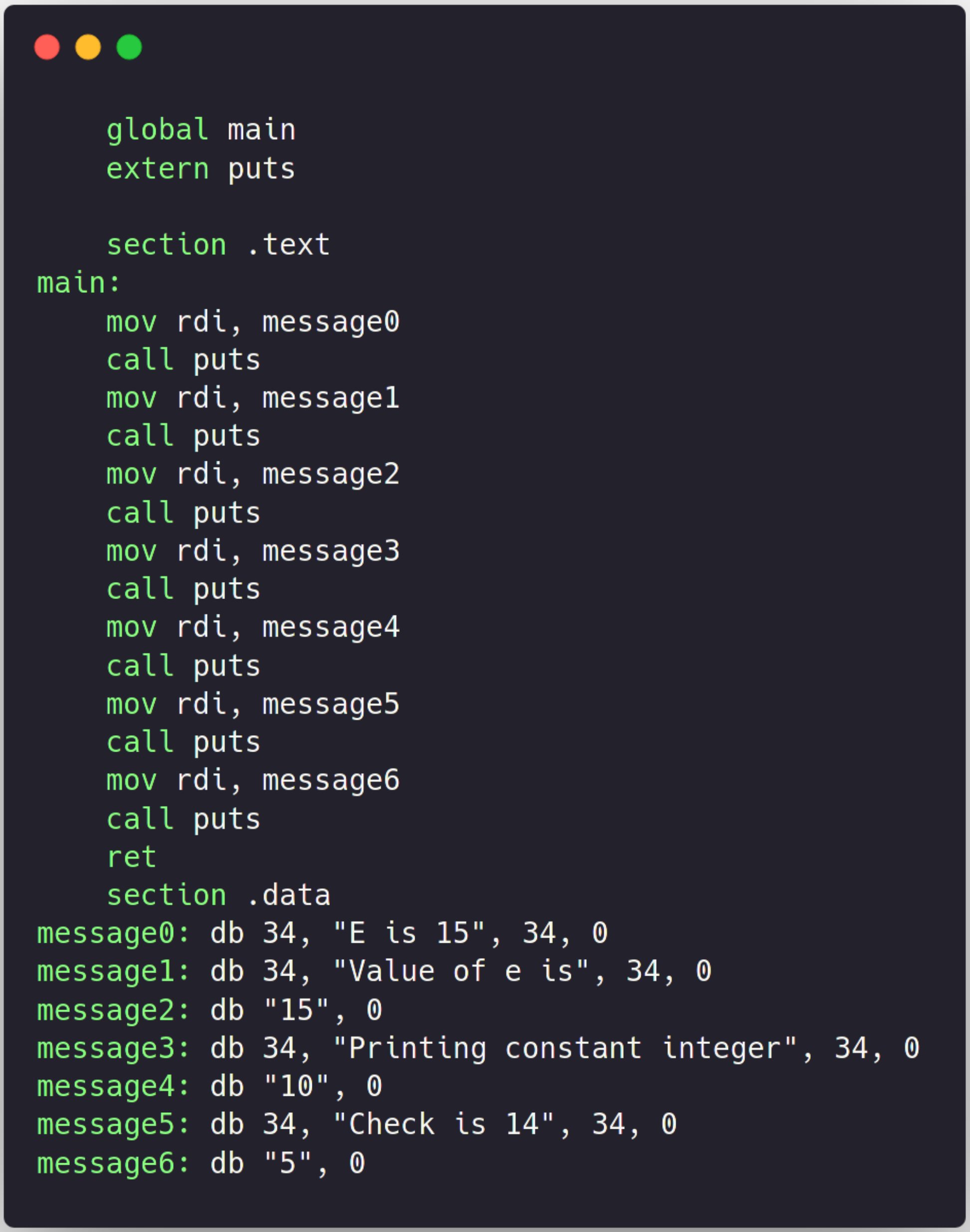
```
 +  5  3  t1
 =  t1  $a
 *  5  3  t2
 =  t2  $b
 -  5  3  t3
 =  t3  $c
 &&  1  0 t4
 =  t4  $d
 =  100  $e
== $a 1 t5
if t5
    GOTO _L1
== $a 2 t6
if t6
    GOTO _L2
== $a 8 t7
if t7
    GOTO _L3
GOTO _EXIT1
_L1 :
 =  5  $e
GOTO _EXIT1
_L2 :
 =  7  $e
GOTO _EXIT1
_L3 :
 <=  $d  0  t8
if t8
GOTO _L4
GOTO _L5
_L4 :
== $b 14 t9
if t9
    GOTO _L6
== $b 15 t10
if t10
    GOTO _L7
GOTO _EXIT3
_L6 :
 =  10  $e
GOTO _EXIT3
_L7 :
== $c 2 t11
if t11
    GOTO _L8
== $c 3 t12
if t12
    GOTO _L9
GOTO _EXIT4
_L8 :
 =  15  $e
GOTO _EXIT4
_L9 :
 =  17  $e
GOTO _EXIT4
_EXIT4 :
GOTO _EXIT3
_EXIT3 :
GOTO _EXIT2
_L5 :
param "Here"
call (print,1)
_EXIT2 :
GOTO _EXIT1
_EXIT1 :
 ==  $e  15  t13
if t13
GOTO _L10
GOTO _L11
_L10 :
param "E is 15"
call (print,1)
GOTO _EXIT5
_L11 :
param "E is not 15"
call (print,1)
_EXIT5 :
param "Value of e is"
call (print,1)
param $e
call (print,1)
param "Printing constant integer"
call (print,1)
param 10
call (print,1)
 =  14  $check
 ==  $check  14  t14
if t14
GOTO _L12
GOTO _EXIT6
_L12 :
param "Check is 14"
call (print,1)
_EXIT6 :
 =  5  $t1
param $t1
call (print,1)
```

Following is the optimized intermediate code:



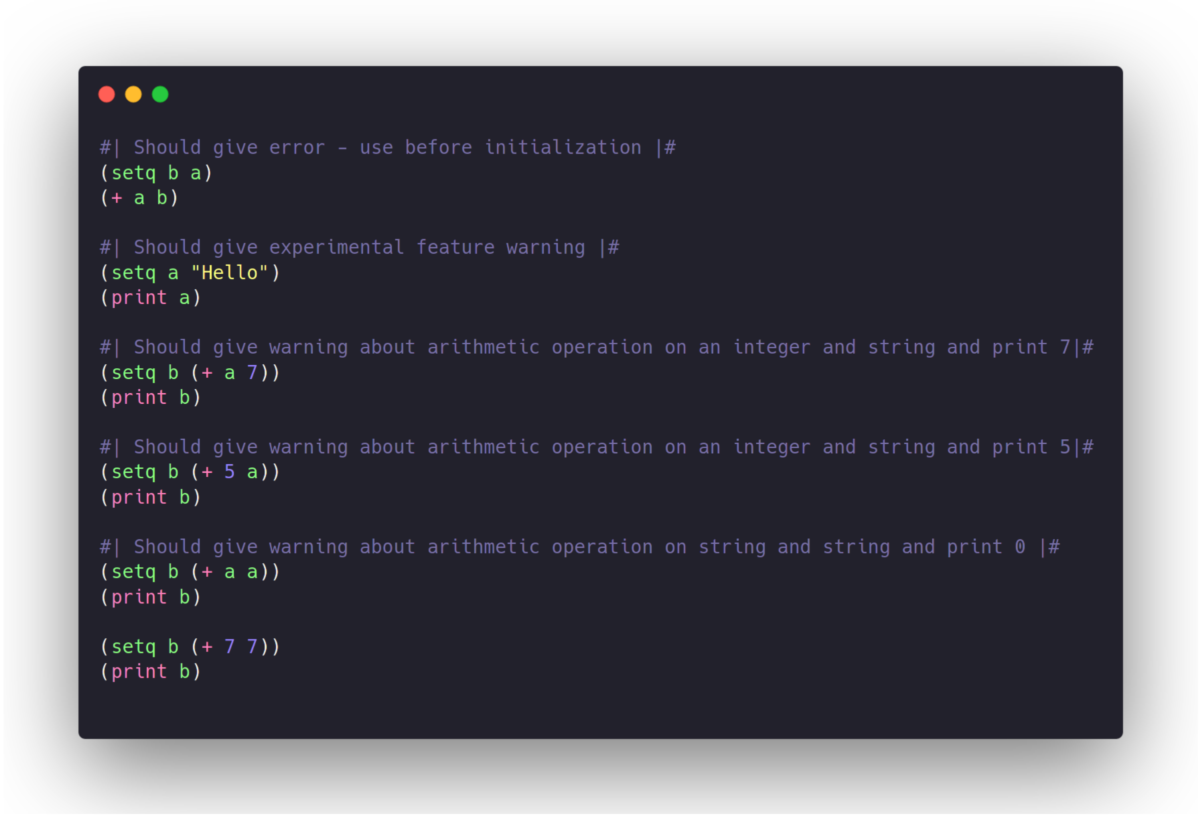
```
= "E is 15" t
param t
call (print,1)
= "Value of e is" t
param t
call (print,1)
= 15 $e
param $e
call (print,1)
= "Printing constant integer" t
param t
call (print,1)
= 10 t
param t
call (print,1)
= "Check is 14" t
param t
call (print,1)
= 5 $t1
param $t1
call (print,1)
```

Following is the NASM output generated:



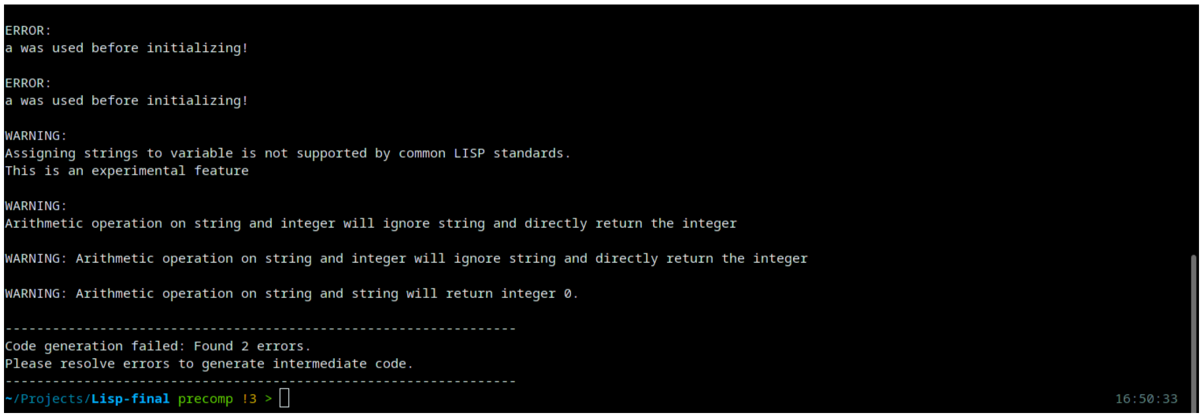
```
    global main
    extern puts

    section .text
main:
    mov rdi, message0
    call puts
    mov rdi, message1
    call puts
    mov rdi, message2
    call puts
    mov rdi, message3
    call puts
    mov rdi, message4
    call puts
    mov rdi, message5
    call puts
    mov rdi, message6
    call puts
    ret
    section .data
message0: db 34, "E is 15", 34, 0
message1: db 34, "Value of e is", 34, 0
message2: db "15", 0
message3: db 34, "Printing constant integer", 34, 0
message4: db "10", 0
message5: db 34, "Check is 14", 34, 0
message6: db "5", 0
```

File with errors:



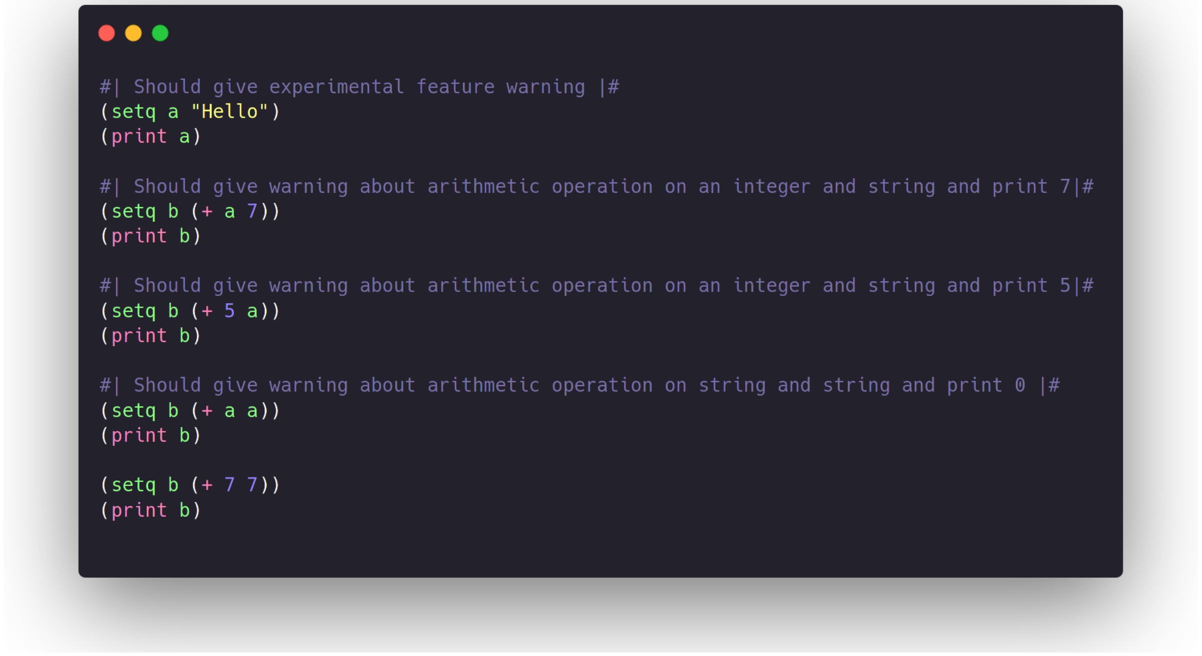
```
#| Should give error - use before initialization |#
(setq b a)
(+ a b)

#| Should give experimental feature warning |#
(setq a "Hello")
(print a)

#| Should give warning about arithmetic operation on an integer and string and print 7|#
(setq b (+ a 7))
(print b)

#| Should give warning about arithmetic operation on an integer and string and print 5|#
(setq b (+ 5 a))
(print b)

#| Should give warning about arithmetic operation on string and string and print 0 |#
(setq b (+ a a))
(print b)

(setq b (+ 7 7))
(print b)
```

Screenshot of execution:



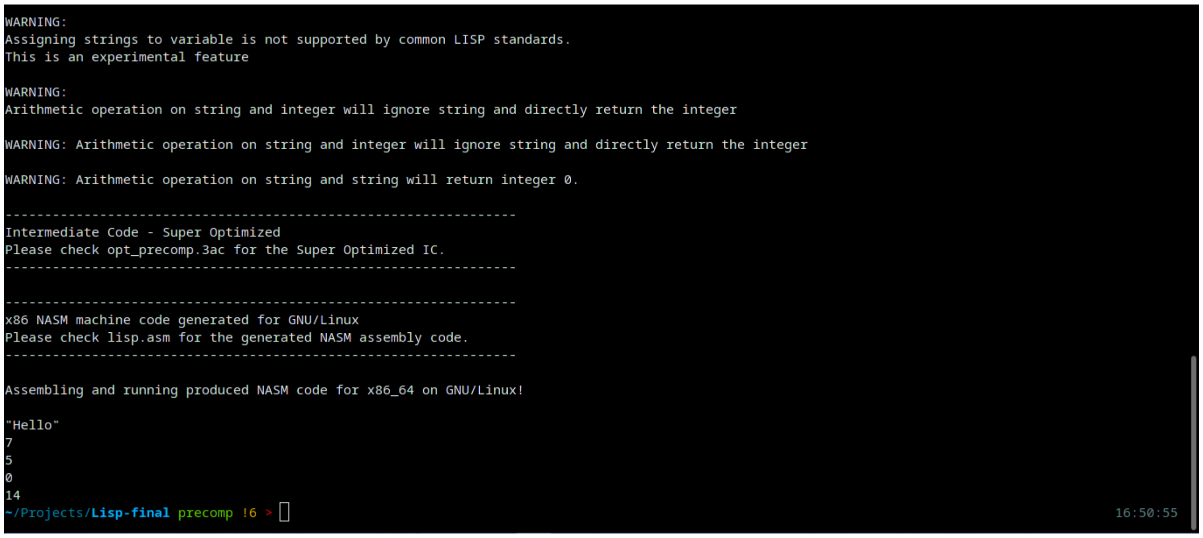
```
ERROR:
a was used before initializing!

ERROR:
a was used before initializing!

WARNING:
Assigning strings to variable is not supported by common LISP standards.
This is an experimental feature

WARNING:
Arithmetic operation on string and integer will ignore string and directly return the integer

WARNING: Arithmetic operation on string and integer will ignore string and directly return the integer

WARNING: Arithmetic operation on string and string will return integer 0.

------------------------------------------------------------------
Code generation failed: Found 2 errors.
Please resolve errors to generate intermediate code.
------------------------------------------------------------------
~/Projects/Lisp-final precomp !3 >                                                                 16:50:33
```

File with warnings:



```
#| Should give experimental feature warning |#
(setq a "Hello")
(print a)

#| Should give warning about arithmetic operation on an integer and string and print 7|#
(setq b (+ a 7))
(print b)

#| Should give warning about arithmetic operation on an integer and string and print 5|#
(setq b (+ 5 a))
(print b)

#| Should give warning about arithmetic operation on string and string and print 0 |#
(setq b (+ a a))
(print b)

(setq b (+ 7 7))
(print b)
```

Screenshot of execution:



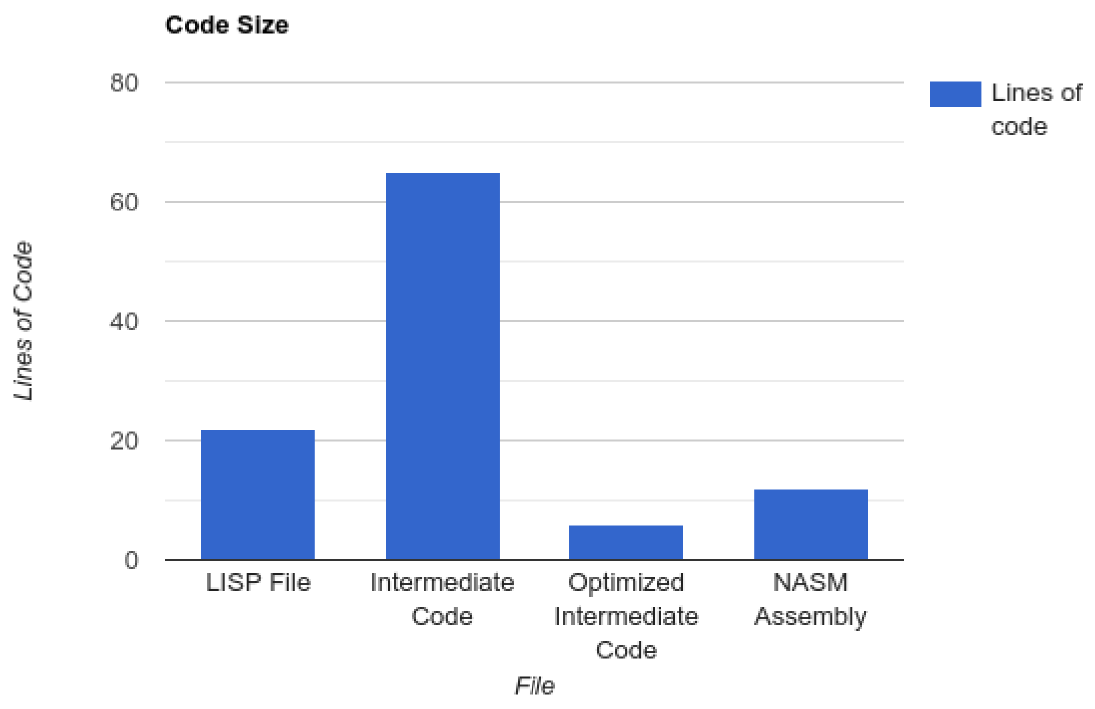
```
WARNING:
Assigning strings to variable is not supported by common LISP standards.
This is an experimental feature

WARNING:
Arithmetic operation on string and integer will ignore string and directly return the integer

WARNING: Arithmetic operation on string and integer will ignore string and directly return the integer

WARNING: Arithmetic operation on string and string will return integer 0.

-----------------------------------------------------------------
Intermediate Code - Super Optimized
Please check opt_precomp.3ac for the Super Optimized IC.
-----------------------------------------------------------------

-----------------------------------------------------------------
x86 NASM machine code generated for GNU/Linux
Please check lisp.asm for the generated NASM assembly code.
-----------------------------------------------------------------

Assembling and running produced NASM code for x86_64 on GNU/Linux!

"Hello"
7
5
0
14
~/Projects/Lisp-final precomp !6 >                                                        16:50:55
```

Chapter 9

Conclusion

We were able to successfully create a mini-LISP compiler for the given constructs that generate one of the smallest possible intermediate code footprint and NASM generation for the optimized intermediate code.

The following graphs depict SLoC for each file and how they change



The lines in the optimized intermediate code and NASM scale linearly with the number of print statements in the LISP file.

Chapter 9

Future Enhancements

The future enhancements that can be added to the project are:

- Add support for IEEE 754 Floating Point numbers
- Add support for more complex string operations
- Support for user input
- Add support for more standard functions
- Add support for while loop
- Add support for for loop
- Add support for arrays
- Add support for function and function calls

The features are listed based on ease of implementation and priority based on impact it'll have on user experience.

REFERENCES

The references used are as follows:

- [Article] IBM Developers: Write text parsers with yacc and lex
- [Book] LEX &YACC TUTORIALby Tom Niemann
- [Tutorial] Common LISP Tutorial
- [Blog] Compiler Optimization
- [Tutorial] NASM Tutorial
- [Presentation] 3 Address Code IR