MINI COMPILER

Project submitted to the

SRM University – AP, Andhra Pradesh

for the partial fulfillment of the requirements to award the degree of

Bachelor of Technology/Master of Technology

In

Computer Science and Engineering

School of Engineering and Sciences

Submitted by

Abdul Basheer Shaik AP21110010203

Gogineni Sai Rohit AP21110010219

Chaitanya Sai Nutakki AP21110010253

Ritesh AP21110010260

Under the Guidance of

(Dr.Arnab Mitra)

SRM University–AP

Neerukonda, Mangalagiri, Guntur

Andhra Pradesh – 522 240

[November 2023]

Table of Contents

Certificate

Date: 20-Nov-23

This is to certify that the work present in this Project entitled "**MINI COMPILER**" has been carried out by **Abdul Basheer Shaik AP21110010203 , Gogineni Sai Rohit AP21110010219 , Chaitanya Sai Nutakki AP21110010253 , Ritesh AP21110010260** under my/our supervision. The work is genuine, original, and suitable for submission to the SRM University – AP for the award of Bachelor of Technology/Master of Technology in **School of Engineering and Sciences**.

Supervisor

(Signature)

Prof. / Dr. Arnab Mitra

Designation,

Affiliation.

Co-supervisor

(Signature)

Prof. / Dr. Arnab Mitra

Designation,

Affiliation.

Acknowledgments

I would like to express my sincere gratitude to the Dean, the Vice Chancellor, and the Head of the Organization for giving me the chance to work on this project. Their constant support and encouragement have inspired me to strive for excellence.

I am grateful to Dr. Arnab Mitra, my Faculty Mentor at SRM University, Amaravati, for his ongoing support, critical insight, and intellectual mentorship. I cannot thank him enough for being a consistent source of inspiration and help throughout this academic adventure. His excellent aid has played a big influence in enhancing my research process and has been crucial in the effective completion of this task.

I also want to express my gratitude to my family, friends, and anybody else who guided and assisted me in my studies. Their encouragement has inspired me to overcome challenges and achieve my research goals.

Lastly, I would like to express my sincere thanks to the whole SRM University academic community for creating a welcoming environment that aided in researching this project. I am appreciative of the chance to collaborate with SRM University, Amaravati's esteemed and encouraging academic community. Thank you all for your help and advice in creating this research project.

Abstract

This project presents the design and implementation of a mini compiler for a subset of the C language. The compiler encompasses fundamental aspects of lexical analysis, syntax analysis, and code generation. Lexical analysis involves breaking down the input code into meaningful tokens, such as keywords, identifiers, and operators. Syntax analysis, on the other hand, checks the structure and organization

Compilers are essential tools for translating high-level programming languages into low-level machine code that can be executed by computers. Designing a compiler involves a complex set of tasks, including lexical analysis, syntax analysis, semantic analysis, intermediate code generation, and code optimization. This mini-compiler design project aims to provide hands-on experience in building a compiler for a subset of a programming language.

The project focuses on implementing the front-end phases of a compiler, including lexical analysis and syntax analysis. Lexical analysis involves breaking down the source code into a stream of tokens, while syntax analysis involves checking the grammatical structure of the code. The project utilizes the tools Lex and Yacc to implement these phases.

The mini compiler is designed to handle a subset of the C programming language, including basic arithmetic expressions, assignment statements, and if-else statements. The compiler generates intermediate code in the form of three-address code, which is a low-level representation of the program that is easier for the code optimizer to handle.

Overall, the mini-compiler design project provides a valuable learning experience in implementing the fundamental concepts of compiler design. It allows students to gain hands-on experience with the tools and techniques used in building compilers, and it provides a deeper understanding of the compilation process as a whole.

of the code to ensure it adheres to the rules of the language. Finally, code generation transforms the parsed code into equivalent machine-level instructions that the computer can execute.

The mini compiler is implemented using lex and yacc, tools specifically designed for lexical and syntax analysis. Lex generates a lexical analyzer that identifies tokens from the input stream, while yacc constructs a parser that checks the syntactic structure of the code. The compiler also incorporates a symbol table to manage variable declarations and scope, ensuring that variables are used correctly within the program.

The mini compiler's capabilities extend to handling basic arithmetic expressions, assignments, and control flow statements like if-else and while loops. It produces assembly code as its output, which can be further processed into machine code for execution.

Through the development of this mini compiler, we gain valuable insights into the compilation process and the challenges involved in designing a language translator. The project serves as a stepping stone towards understanding the complexities of real-world compilers and their role in software development.

1. Introduction

Compilers are essential tools for translating high-level programming languages into low-level machine code that can be executed by computers. Designing a compiler involves a complex set of tasks, including lexical analysis, syntax analysis, semantic analysis, intermediate code generation, and code optimization. This mini-compiler design project aims to provide hands-on experience in building a compiler for a subset of a programming language.

The project focuses on implementing the front-end phases of a compiler, including lexical analysis and syntax analysis. Lexical analysis involves breaking down the source code into a stream of tokens, while syntax analysis involves checking the grammatical structure of the code. The project utilizes the tools of Lex and Yacc to implement these phases.

The mini compiler is designed to handle a subset of the C programming language, including basic arithmetic expressions, assignment statements, and if-else statements. The compiler generates intermediate code in the form of three-address code, which is a low-level representation of the program that is easier for the code optimizer to handle.

Overall, the mini-compiler design project provides a valuable learning experience in implementing the fundamental concepts of compiler design. It allows students to gain hands-on experience with the tools and techniques used in building compilers, and it provides a deeper understanding of the compilation process as a whole.

Sample Input:

```
a=10
    b=9
    c= a+b+100
    e=10
    f=8
    d=e*f
    if(a>=b):
        a=a+b
        g=e*f*100
    u=10
    j=99
```

Sample Output:

```
MOV RO, #10
MOV R1, #9
MOV R2, #119
MOV R3, #8
MOV R4, #80

10:
MOV R5, #0
BNEZ R5, 11
MOV R6, #19
MOV R7, #8000

11:
MOV R8, #99
ST b, R1
ST c, R2
ST e, RO
ST f, R3
ST d, R4
ST a, R6
ST g, R7
ST u, RO
ST j, R8
```

2. Methodology:

A. Lexical Analysis (Scanner):

- **Purpose:** Breaks down the source code into tokens or lexemes.
- The lexical analyzer (or scanner) reads the source code character by character, identifying meaningful units (tokens) like keywords, identifiers, operators, etc. It discards irrelevant characters such as whitespaces and comments. Tokens are typically defined using regular expressions or patterns.

B. Syntax Analysis (Parser):

- **Purpose:** Checks the syntax of the program and creates a parse tree or syntax tree.
- The parser processes the tokens generated by the lexical analyzer, ensuring they adhere to the language's syntax rules, as defined by a grammar (usually specified in Backus-Naur Form or BNF). It creates a hierarchical representation of the code's structure, such as an Abstract Syntax Tree (AST) or Parse Tree.

C. Semantic Analysis:

- **Purpose:** Performs semantic checks and ensures correct meaning within the program.
- This phase goes beyond syntax and checks for meaningfulness and context in the code. It includes tasks like type checking (ensuring compatible data types are used), scope analysis (validating variable scopes), and building a symbol table (mapping identifiers to their declarations).

D.Code Optimization:

- **Purpose**: Improves the efficiency of the generated code.
- This phase applies various techniques to refine the intermediate code, enhancing its performance and reducing its size. Common optimization techniques include dead code elimination, common subexpression elimination, and loop unrolling.

D. Code Generation:

- **Purpose:** Translates the validated source code into target machine code or assembly.
- The code generator takes the intermediate representation (such as AST or Intermediate Code) produced in the previous stages and converts it into

machine-specific code. This involves mapping high-level constructs to low-level instructions, applying optimizations to improve efficiency, and generating the final executable code.

Each stage plays a critical role in transforming high-level source code into machine-executable instructions, ensuring correctness, meaning, and efficiency at various levels of abstraction. The phases work sequentially, with the output of one stage serving as the input to the next, culminating in the generation of executable code.

1. Context Free Grammar

P -> S

S -> Simple S | Compound S | epsilon

Simple -> Assignment | Conditional | Print | break | pass | continue

Assignment -> id op_assign E | id op_assign Conditional | id list_assign Arr
| id str_assign Str

E -> E op1 E | E op2 E | E op3 E | num | id | (E)

op_assign -> = | /= | *= | += | -=

op1 -> + | -

op2 -> / | *

op3 -> **

list_assign -> =

str_assign -> = | += | -=

Arr -> [list] | [list] mul | [list] add | mat

mat -> [listnum] | [liststr]

list -> listnum | liststr | Range

listnum -> num,listnum | epsilon | num

liststr -> Str,liststr | epsilon | Str

mul -> * integer

add -> + Arr

Range -> range (start , stop , step)

start -> integer | epsilon

stop -> integer

step -> integer | epsilon

Str -> string | string mul | string addstr

addstr -> + string

Compound -> if_else | while_loop

if_else -> if condition : LB IND else | if condition : LB IND | if condition : S | if condition : S else

else -> else : LB IND | else : S

while_loop -> while condition : LB IND | while condition : S

condition -> cond | (cond)

cond -> cond op_or cond1 | cond1

cond1 -> cond1 op_and cond2 | cond2

cond2 -> op_not cond2 | cond3

cond3 -> (cond) | relexp | bool

relexp -> relexp relop E | E | id | num

relop -> < | > | <= | >= | == | != | in | not in

bool -> True | False

op_or -> || | or

op_and -> && | and

op_not -> not | ~

IND -> indent S dedent

indent -> \t

dedent -> -\t

Print -> print (toprint) | print (toprint,sep) | print (toprint,sep,end) | print (toprint,end)

toprint -> X | X,toprint | epsilon

X -> Str | Arr | id | num

sep -> sep = Str

end -> end = Str

2. Architectural Design

1. **Generation of Symbol Table:**

- The construction of the symbol table relies on a linked list structure. This resultant table exhibits critical information including label, value, scope, line number, and type. Three fundamental functions facilitate the symbol table operation:
- Insert: Responsible for appending nodes onto the linked list.
- Display: Facilitates the visualization of the entire symbol table.
- Search: Enables the search for a specific label within the linked list.

2. **Parsed Syntax Tree:**

- The AST is realized through a structured entity encompassing three attributes - data, left pointer, and right pointer. Operations associated with the creation and display of this tree structure encompass:
- BuildTree: Creates a node conforming to this structure and incorporates it into the existing tree.
- printTree: Presents the abstract syntax tree through a pre-order traversal mechanism.

3. Intermediate Code Generation:

- Employing the stack data structure, the generation of intermediate code occurs through conditional function calls, pivotal in the code construction process.

4. Code Refinement:

- Utilization of a data structure, termed as quadruple, facilitates code optimization. This structure comprehensively stores details pertinent to assignment, label, and goto statements, thereby enhancing code efficiency.

5. Error Handling:

- Syntax Error: Detection of grammar-incompatible tokens triggers the use of yyerror() to delineate syntax errors, accompanied by the corresponding line number.
- Semantic Error: Identification of an identifier absent in the symbol table during an assignment statement is deemed a semantic error and is distinctly highlighted.
- Error Recovery: Implemented through Panic Mode Recovery, where mishandling of a variable declaration starting with a number disregards the number, treating the remainder as the variable name. Regex patterns dictate this recovery technique.

6.Target Code Generation:

- Optimized intermediate code is parsed line by line from a text file. Through an array of if-else loops, the target code is systematically constructed. A hypothetical target machine model with a restricted number of reusable registers (from R0 to R12) drives this generation process, adhering to a specific instruction set architecture.

```
Load/Store Operations:
 ST <loc>. R
 LD R. <loc>

2) Move Operations:
 MOV Rd. #<num>

3) Arithmetic Operations: <ADD/SUB/MUL/DIV> R. R₁. R₂

4) Compare Operations: CMP<cond> Ra R1 R2\

(<cond>: E for =-. NE for!-. G for >.L for <. GE for >= or LE for <=)

5) Logical Operations:
NOT R_{A} R
<AND/OR> R_{d} R1 R₂

6) Conditional Branch:
BNEZ R_{d} label

7) Unconditional Branch:
BR label
```

3. Implementation

A. ESTABLISHING SYMBOL TABLES

The structure declaration for the symbol table is depicted in the following snapshot:

```
struct symbtab
{
char label[20];
char type[20];
int value;
char scope[20];
int lineno;
struct symbtab *next;
};
```

These functions facilitate the creation of the symbol table:

```
void insert(char* l,char* t,int v,char* s,int ln); struct symbtab*
search(char lab[]); void display();
```

These excerpts are extracted from the proj1.y file within the Token and Symbol table directory

B. STRUCTURAL REPRESENTATION OF SYNTAX

The subsequent data structure serves to illustrate the abstract syntax tree:

```
typedef struct Abstract_syntax_tree
{
   char *name;
   struct Abstract_syntax_tree *left;
   struct Abstract_syntax_tree *right;
}node;
```

Below are the functions responsible for constructing and showcasing the syntax tree:

```
node* buildTree(char *, node *, node *); void printTree(node *);
```

These excerpts are extracted from the proj1.y file located within the Abstract Syntax Tree directory.

C. INTERMEDIATE CODE GENERATION

These arrays serve as stacks and are instrumental in generating the intermediate code:

```
char label[2]="L"; // labels
int l = 0; //count of labels (11,12,....)
char 1_[100] = {'\0'}; //labels
char st[100][10]; //stack used in icg generation
int top=0;
int i=0;//top of stack
char [100] = {'\0'}; //temp variables (t1, t2,...)
char temp[2]="t"
char ICG[10000]=""; //icg
char try1[5][10];
char try[5][10];
```

These functions facilitate the stack operation and generate the intermediate code as required under different conditions when invoked:

```
void push(char*);
void codegen(int val, char* aeval_);
void codegen_assign();
void codegen2();
void codegen3();
```

These images are sourced from the proj1.y file within the ICG directory.

D. CODE OPTIMISATION

Below is the declaration for the quadruple data structure.

```
typedef struct quadruples {
char *op;
char *arg1;
char *arg2;
char *res;
}quad;
```

The following functions are used to add to the quadruples table and display it onto the terminal:

```
void displayquad(); char addquad(char*,char*, char*, char*);
```

These excerpts are sourced from the proj1.y file within the Optimised_ICG folder.

E. TARGET CODE GENERATION

The primary dictionary governs the connection between each constant/identifier and its designated register. Additionally, there's a master list tracking identifiers requiring storage toward the program's conclusion. Register allocation involves two functions: 'getreg()' procures the subsequent available or unassigned register and utilizes 'fifo()' when all registers are occupied. 'fifo()' employs a 'First In First Out' mechanism to release a register, returning it to the 'getreg()' function. These functions are outlined below

```
def getreg():
for i in range(0,13):
if reg[i]==0: reg[i]=1 return 'R'+str(i)
register = fifo() return register
```

```
def fifo():
global fifo reg
global fifo_return_reg
for k,v in var.copy().items():
if(v == 'R'+str(fifo_reg)):
fifo_return_reg = v
var.pop(k)
if(k in store_seq): store_seq.remove(k) print("ST", k, ', ', v, sep='')
fifo_reg = int(fifo_return_reg[1:]) + 1
return fifo_return_reg
```

F. ERROR HANDLING

The subsequent capture illustrates the syntax error handling function:

```
tat verror(){
printf( error = 1;
SYNTAX ERROR at line number Nd -\n",yylineno-1
V=0; return 0;
```

The following snapshot shows semantic error handling functionality:

```
t_ptresearch($1);
if(t_ptr==NULL)
ERROR: variable undeclared- 51)
printf( error = 1;
```

These above snapshots are taken from proj1.y file in Symbol table folder.

The regex for panic mode recovery implemented in the lexer is as follows:

```
[0-9;!,@#]*/(({alpha}|"_")({alpha}|{digits}|"_")*)
```

The above snapshots are taken from proj.l file in the Symbol table folder.

4. Results

The mini-compiler developed within this project effectively processes Python's 'if-else' and 'while' constructs. Execution of the compiler occurs in distinct phases, allowing separate building and execution of code within different folders. The culmination of these phases manifests in a sequential output showcased on the terminal.

Initially, the display exhibits tokens, succeeded by a confirmation message of 'PARSE SUCCESSFUL'. Subsequently, the abstract syntax tree is presented, followed by the symbol table coupled with the intermediate code devoid of optimization. Lastly, the optimized symbol table and intermediate code are unveiled alongside the quadruples table. The ultimate output features the target code, adhering to the instruction set architecture characteristic of the hypothetical machine model introduced in this project.

However, in cases of input errors, the token generation halts upon encountering an error, triggering the display of the corresponding error message.

This mini-compiler has several limitations:

- Absence of support for user-defined functions.
- Inability to handle library imports and calls to library functions.
- Exclusion of data types other than integers and floats, such as strings, lists, tuples, dictionaries, etc.
- Limited functionality as it lacks constructs beyond 'while' and 'if-else' statements within the compiler program.

5. Input/Output Screenshots

Sample Input:

```
a=10
    b=9
    c= a+b+100
    e=10
    f=8
    d=e*f
    if(a>=b):
        a=a+b
        g=e*f*100
    u=10
    j=99
```

Tokens and Symbol Table:

```
C:\Users\Ashish\Downloads\Python-Compiler-master\Python-Compiler-master\Project Code\1-Token_and_Symbol_Table>a.exe
ID equal int
ID equal int
ID equal ID plus ID plus int
ID equal int
ID equal int
ID equal ID mul ID
if special_start ID greaterthanequal ID special_end colon
indent ID equal ID plus ID
indent ID equal ID mul ID mul int

ID equal int
ID equal int
----------------------------PARSE SUCCESSFUL-------------------------

-----------------SYMBOL TABLE---------------
--------------------------------------------
LABEL   TYPE          VALUE   SCOPE   LINENO
a       IDENTIFIER    19      local   8
b       IDENTIFIER    9       global  2
c       IDENTIFIER    119     global  3
e       IDENTIFIER    10      global  4
f       IDENTIFIER    8       global  5
d       IDENTIFIER    80      global  6
g       IDENTIFIER    8000    local   9
u       IDENTIFIER    10      global  11
j       IDENTIFIER    99      global  12
```

Abstract Syntax Tree:

```
C:\Users\Ashish\Downloads\Python-Compiler-master\Python-Compiler-master\Project Code\2-Abstract_Syntax_Tree>a.exe

------------Abstract Syntax Tree--------------
( SEQ ( =  a  10 )( SEQ ( =  b  9 )( SEQ ( =  c ( + ( +  a  b ) 100 ))( SEQ ( =  e  10 )( SEQ ( =  f  8 )( SEQ ( =  d
( *  e  f ))( SEQ ( IF ( >=  a  b )( SEQ ( =  a ( +  a  b ))( SEQ ( =  g ( * ( *  e  f ) 100 )) NULL )))( SEQ ( =  u
10 )( SEQ ( =  j  99 ) NULL )))))))))
```

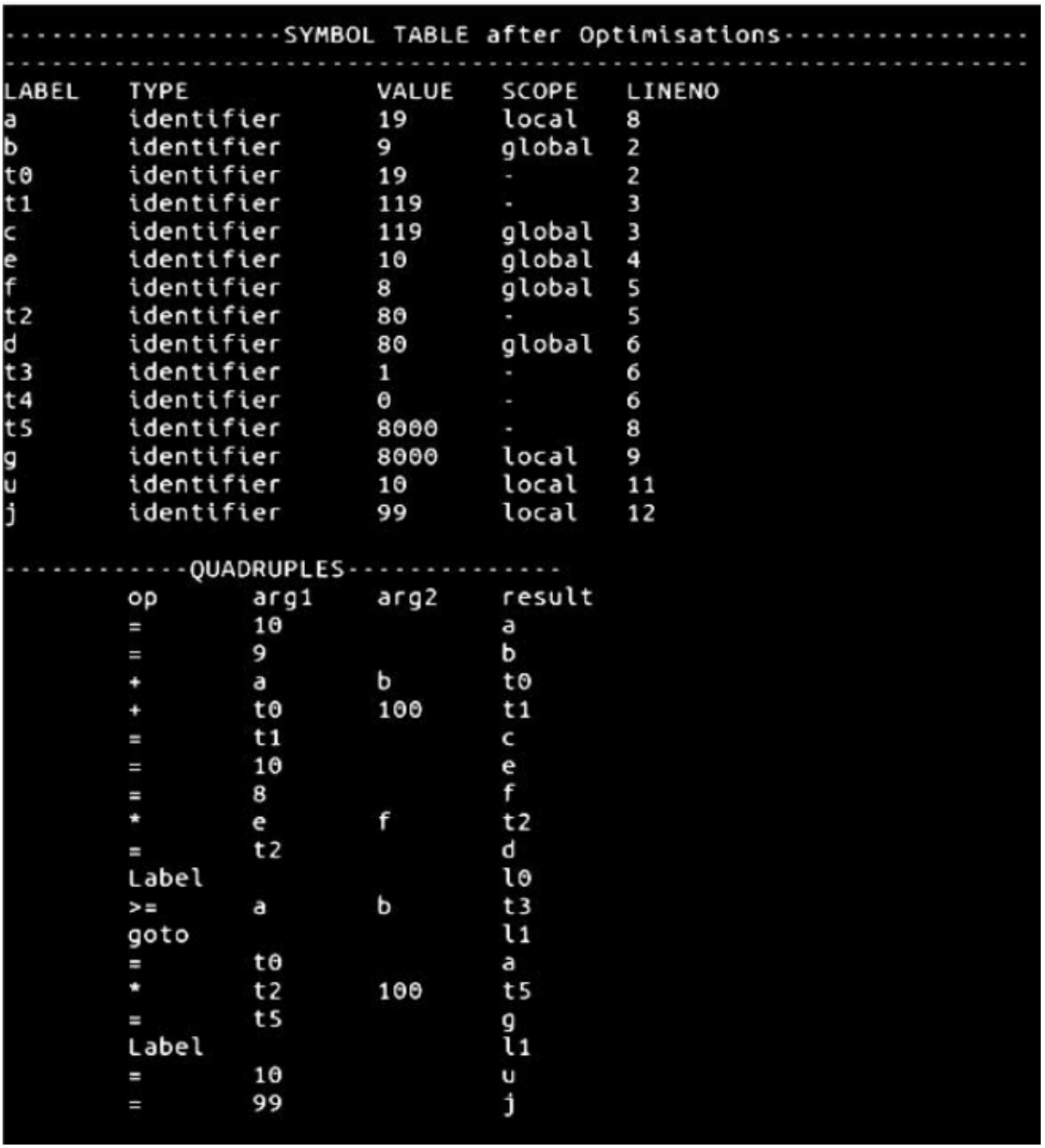
Symbol Table and Unoptimized Intermediate Code:

```
C:\Users\Ashish\Downloads\Python-Compiler-master\Python-Compiler-master\Project Code\3-Intermediate_Code_Generation>a.exe

-----------------SYMBOL TABLE before Optimisations---------------
---------------------------------------------------------------
LABEL   TYPE            VALUE   SCOPE   LINENO
a       identifier      9       local   8
b       identifier      9       global  2
t0      identifier      19      -       2
t1      identifier      119     -       3
c       identifier      119     global  3
e       identifier      10      global  4
f       identifier      8       global  5
t2      identifier      80      -       6
d       identifier      80      global  6
t3      identifier      0       -       6
t4      identifier      9       -       8
t5      identifier      80      -       8
t6      identifier      8000    -       9
g       identifier      8000    local   9
u       identifier      10      local   11
j       identifier      99      local   12

-------------ICG without optimisation-------------
a=10
b=9
t0=a+b
t1=t0+100
c=t1
e=10
f=8
t2=e*f
d=t2
l0 : t3=a>=b
if not t3 goto l1
t4=a+b
a=t4
t5=e*f
t6=t5*100
g=t6
l1 : u=10
j=99
```

Symbol Table, Quadruples Table and Optimised Intermediate Code:



```
-----------------SYMBOL TABLE after Optimisations---------------
------------------------------------------------------------------
LABEL    TYPE            VALUE    SCOPE    LINENO
a        identifier      19       local    8
b        identifier      9        global   2
t0       identifier      19       -        2
t1       identifier      119      -        3
c        identifier      119      global   3
e        identifier      10       global   4
f        identifier      8        global   5
t2       identifier      80       -        5
d        identifier      80       global   6
t3       identifier      1        -        6
t4       identifier      0        -        6
t5       identifier      8000     -        8
g        identifier      8000     local    9
u        identifier      10       local    11
j        identifier      99       local    12

------------QUADRUPLES--------------
         op      arg1    arg2    result
         =       10              a
         =       9               b
         +       a       b       t0
         +       t0      100     t1
         =       t1              c
         =       10              e
         =       8               f
         *       e       f       t2
         =       t2              d
         Label                   l0
         >=      a       b       t3
         goto                    l1
         =       t0              a
         *       t2      100     t5
         =       t5              g
         Label                   l1
         =       10              u
         =       99              j
```

```
ICG with optimisations(Packing temporaries & Constant Propagation)
a = 10
b = 9
t0 = 10 + 9
t1 = 19 + 100
c = 119
e = 10
f = 8
t2 = 10 * 8
d = 80
l0:
t3 = 10 >= 9
t4 = not 1
if 0 goto l1
a = 19
t5 = 80 * 100
g = 8000
l1:
u = 10
j = 99
```

Target Code:

```
MOV R0, #10
MOV R1, #9
MOV R2, #119
MOV R3, #8
MOV R4, #80
l0:
MOV R5, #0
BNEZ R5, l1
MOV R6, #19
MOV R7, #8000
l1:
MOV R8, #99
ST b, R1
ST c, R2
ST e, R0
ST f, R3
ST d, R4
ST a, R6
ST g, R7
ST u, R0
ST j, R8
```

TEST CASE 2 (Syntax Error):

Input:

```
a=10
    b=9
    c= a+b+100
    e=10
    f=8
    d=e*f
    if(a>=b):
        a=a+b
        g=e*f*100
    u=10
    j=99
```

Output:

```
C:\Users\Ashish\Downloads\Python-Compiler-master\Python-Compiler-master\Project Code\1-Token_and_Symbol_Table>a.exe
ID equal int
ID equal int
ID equal ID plus ID plus int
ID plus
-----------------SYNTAX ERROR : at line number 4 ------------------------
```

6. Conclusion

- Crafting a comprehensive compiler is a time-intensive and complex undertaking. In our endeavor, we've successfully developed a compact mini-compiler that carries out the following pivotal operations:

Python Compiler using Lex and Yacc:

- The mini-compiler designed for Python operates through lex and yacc files. It validates Python programs in line with a defined context-free grammar.

Token Generation with Regular Expressions:

- Employing meticulously crafted regular expressions, the compiler generates tokens essential for program interpretation.

Symbol Table Creation:

- Information pertaining to identifiers is systematically stored within a symbol table, facilitating efficient access and management.

Abstract Syntax Tree (AST) Generation:

- A robust abstract syntax tree is generated and presented through pre-order tree traversal, offering a structured visual representation of the code's hierarchical structure.

Intermediate Code Generation and Optimization:

- The compiler generates intermediate code while optimizing via Quadruples, employing techniques like constant propagation and temporary variable compression.

Target Code Conversion:

- The optimized intermediate code undergoes a transformation into target code utilizing a hypothetical machine model, ensuring compatibility with the model's architecture.

Error Handling and Recovery Mechanisms:

- Implemented error handling and recovery mechanisms adeptly manage erroneous inputs, ensuring smooth compiler functionality even in the presence of input anomalies.
- This mini-compiler, while compact, encompasses a substantial range of functionalities critical for code interpretation and validation. Despite its scaled-down nature, it proficiently executes essential tasks integral to the compilation process for Python programs.

7. Future Works

The compiler design can be improved in several ways:

- **Enhanced Efficiency:** By integrating advanced optimization techniques like loop unrolling, constant folding, and common subexpression elimination, the generated code's efficiency can be refined.

- **Usability Improvements:** Expanding the error handling system to provide more descriptive and user-friendly error messages will aid in debugging and user comprehension.

- **Wider Compatibility:** Ensuring the compiler works seamlessly across various platforms and architectures will broaden its usability.

- **Expanded Language Features:** Incorporating advanced language elements like exception handling, generics, and intricate control structures will enrich its capabilities.

- **Parallel Processing Support:** Adding support for parallel programming constructs such as threads or processes can enable concurrent task handling.

- **Documentation Enhancement:** Enhancing documentation with comprehensive explanations, examples, and usage scenarios will facilitate better adoption.

- **Integration with IDEs:** Integrating the compiler with popular Integrated Development Environments (IDEs) will offer a smoother development and debugging experience.

The mini-compiler could evolve into a full-fledged Python compiler with features like user-defined functions, library imports, support for various data types, and additional programming constructs. Further refinement of the program's output and exploration of alternative data structures or approaches could significantly enhance its overall efficiency and speed.