Technical Lab Report

Compiler Design

Ambaliya Shankheshkumar M
Roll No.: CS22B005

November 19, 2024

Abstract

This report provides an in-depth analysis of the design and implementation of a compiler for a simplified programming language. The compiler is developed using Lex for lexical analysis and Yacc/Bison for syntax and semantic analysis. The report not only details the implementation but also explores theoretical concepts such as the principles of compiler design, lexical and syntax analysis, and symbol table management. A comparison of benefits and limitations of the tools used is included, highlighting their significance in modern compiler design.

Introduction

A compiler is a software tool that translates high-level source code into machine code, enabling programs to run on specific hardware architectures. This lab project involved building a simple yet functional compiler to process a custom programming language. The compiler comprises:

- A lexer for tokenization using Lex.
- A parser for syntax analysis using Yacc/Bison.
- A symbol table for semantic analysis and scoping.

The report provides a theoretical foundation for compiler design, discusses implementation details, and compares tools and methodologies.

Theoretical Background

Compiler Design Principles

A compiler operates in several phases, each performing a specific function. The phases include:

- **Lexical Analysis:** Converts source code into tokens.
- **Syntax Analysis:** Parses tokens to check grammatical structure.

- **Semantic Analysis:** Ensures meaning and consistency.
- **Optimization:** Improves performance without changing functionality.
- **Code Generation:** Produces machine code or intermediate representation.

Lexical Analysis

Lexical analysis identifies tokens such as keywords, operators, and identifiers. The Lex tool automates this process using regular expressions. Tokens are essential for subsequent syntax analysis.

Syntax Analysis

Syntax analysis constructs a parse tree based on the grammar of the language. Context-free grammars are often used, as they are powerful enough to describe most programming constructs.

Symbol Table

A symbol table is a data structure used to store information about program symbols (e.g., variables, functions). It supports operations such as insertion, lookup, and scope management.

Design and Implementation

Lexical Analysis (Lex)

Lex simplifies the process of tokenization. In the 'prob.l' file, patterns such as keywords and operators are defined using regular expressions.

Listing 1: Key Rules in Lex File

```
"int"       { return INTNUM; }
"while"     { return WHILE; }
"if"        { return IF; }
"+"         { return ADD; }
"-"         { return SUB; }
"="         { return EQUAL; }
```

Lex automatically generates a finite automaton for recognizing these patterns, making tokenization efficient.

Syntax Analysis (Yacc/Bison)

The 'prob.y' file contains grammar rules and associated semantic actions. These actions update the symbol table and check for errors.

Listing 2: Key Grammar Rules in Yacc File

```
stmt : IF '(' expr ')' stmt
     | WHILE '(' expr ')' stmt
```

```
    | expr ';' ;

expr : expr ADD expr
    | expr SUB expr
    | INTNUM ;
```

Yacc/Bison provides automatic conflict resolution mechanisms, such as handling shift-reduce conflicts.

Symbol Table Management

The symbol table implementation uses structures to store information about symbols. This enables efficient management of variable declarations, scoping, and array sizes.

Listing 3: Symbol Table Structure

```
typedef struct {
    char name[50];
    char type[20];
    int size;
    int offset;
    int isArray;
    int arraySize;
} Symbol;
```

Comparison of Tools and Benefits

Feature	**Lex**	**Yacc/Bison**
Purpose	Tokenizes input	Parses token stream
Strength	Efficient for regular expressions	Handles context-free grammars
Automation	Automatically generates token recognizer	Automatically constructs parse tree
Limitation	Cannot process nested structures	Limited to context-free grammars

Lex and Yacc are complementary tools, working together to automate key compiler tasks.

Code Walkthrough

Lex File ('prob.l')

Defines patterns for tokens such as operators, keywords, and delimiters.

Yacc File ('prob.y')

Specifies grammar rules and integrates semantic actions.

Sample Input Program (‘inp1‘)

Illustrates a program processed by the compiler.

Listing 4: Sample Input Program

```
{
    int a, b, c;
    while (a < 10) {
        if (b > 5) {
            c = a + b;
        } else {
            b--;
        }
    }
}
```

Limitations

The current implementation has several limitations:

- Limited error handling capabilities.
- No intermediate code generation or optimization.
- Basic support for type checking and scoping.

Code Workflow and Explanation

The compiler implemented in this lab project is structured into several key stages, each responsible for a specific task in the compilation process. The workflow of the code is as follows:

Lex file content

Inside the prob.l file i declared the all the tokens that i am going to use inside my prob.y file for the design of CFG and workflow of it and also converting some token to specific type in union such that it doesn't create type conflicts in the CGF.

Listing 5: some examples from prob.l file

```
{Equal}          { return EQ; }
"+="        { return AEQ; }
"-="        { return SEQ; }
"*="        { return MEQ; }
"=="        { return DoubleEQ; }
"/="        { return DEQ; }
"!="        { return NotEQ; }
"<"         { return LT; }
"<="        { return LTE; }
">"         { return GT; }
```

```
">="            { return GTE; }
"&&"            { return AND; }
"||"            { return OR; }
"!"             { return NEQ; }
[0-9]+\.[0-9]+ { yylval.numf = atof(yytext); return FLOATY; }
[0-9]+          { yylval.num = atoi(yytext); return INT; }
[a-zA-Z][a-zA-Z0-9]* { strcpy(yylval.address, yytext); return ID; }
[ \t\n]+        { }
```

Declaration of methods and Header files

In the start of of the prob.y file first of all i called some header files and declared some function and general variables which i am going to use.

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

void yyerror(char*);
int yylex();
extern FILE* yyin;

#define IntSize 4
#define FloatSize 4
#define CharSize 1

typedef struct {
    char name[50];
    char type[20];
    int size;
    int offset;
    int isArray;
    int arraySize;
} Symbol;

typedef struct {
    Symbol symbols[100];
    int symbolCount;
    int Curroffset;
} Scope;

void enter();
void Mexit();
int SymAdd(char* name, char* type, int isArray, int arraySize);
bool Sfind(char* name);
int getType(char* type);
void printAllScopeSymbols();
char str[1000];
char* LineLabels();
char* BLable();

Scope scopeStack[100];
int scopeStackTop = -1;
Symbol symbolList[1000];  // Store all symbols
int symbolListCount = 0;  // Count of symbols in the list
int Curroffset = 0;  // this globally to 0 before declaring symbols

int t = 0;
int lab = 100;
int lineNo = 100;
char str1[10];
bool flag = false;
%}
```

Token Declarations in YACC

Tokens are defined in the LEX file (`prob.l`) and declared in the YACC file (`prob.y`) to establish their presence and usage. These tokens typically represent the building blocks of the grammar, such as keywords, operators, and symbols.

Listing 6: Token Declarations

```
%token IF ELSE ADD SUB MUL DIV SemiCol EQ AEQ SEQ MEQ DEQ NotEQ DoubleE
       LT LTE GT GTE NEQ AND OR INC DEC LPAREN RPAREN LCURL RCURL COLOI
       WHILE LEFT_SQ RIGHT_SQ INTNUM FLOAT CHAR COMMA
```

Operator Precedence and Associativity

To resolve shift-reduce conflicts, especially in the case of expressions, operator precedence and associativity are defined in the YACC file. Associativity can be specified as:

Listing 7: Operator Precedence and Associativity

```
%left     ADD SUB        // Left-associative addition and subtraction
%left     MUL DIV        // Left-associative multiplication and division
%right    EQ AEQ SEQ     // Right-associative equality operators
%nonassoc LT LTE GT GTE // Non-associative relational operators
```

Union Declaration for Parse Types

The `union` defines the data type(s) associated with tokens and non-terminals. This allows storing values for different tokens and grammar rules in a YACC file. The following `union` declaration is used to define the various data types:

Listing 8: Union Declaration for Parse Types

```
%union {
    char lexeme[100];     // Store lexemes (e.g., identifiers, keywords,
        char address[200];    // Store memory addresses or references
        char* label;          // Store labels (e.g., for code generation)
        int num;              // Store integer values
        float numf;           // Store floating point values
    }
```

Assigning Parse Types to Non-Terminals

Each non-terminal in the grammar is associated with a parse type that corresponds to one of the data types in the `union`. This ensures type consistency during parsing and helps maintain the correct structure throughout the grammar rules.

Listing 9: Assigning Parse Types to Non-Terminals

```
    %type <address> StmList
    %type <address> IfStm
    %type <address> ElseStmt
    %type <address> ComRelExp
    %type <address> Relexp
    %type <address> Statement
    %type <address> Term
    %type <address> Factor
    %type <address> PostfixOperation
    %type <address> PrefixOperation
    %type <address> SignVal
    %type <address> Val
    %type <label> Label
    %type <address> Type
    %type <num> ArraySize
    %type <address> DeclareList
    %type <address> Declare
```

Grammar Rules Explanation

In the following grammar rules, we define several non-terminals and their corresponding actions. The actions primarily consist of printing assembly-like instructions and adding symbols to the symbol table. We also handle error cases where the syntax doesn't follow the expected pattern.

1. StmList: Statement List

The `StmList` represents a list of statements in the program. The non-terminal is used to handle various statements such as `if`, `while`, and general expressions.

Listing 10: StmList Rule

```
StmList:
    Statement SemiCol StmList { printf("\n"); }
    | IfStm StmList { printf("\n"); }
    | WHILE LPAREN ComRelExp RPAREN LCURL Label Label Label {
        if(!flag) printf("%d %s: if %s goto %s\n", lineNo++, $6, $3, $8);
        if(!flag) printf("%d goto %s\n", lineNo++, $7);
        if(!flag) printf("%d %s:\n", lineNo++, $8);
    } StmList RCURL {
        if(!flag) printf("%d goto %s\n", lineNo++, $6);
    } { if(!flag) printf("%d %s:\n", lineNo++, $7); } StmList { printf("\n
    | LCURL { enter(); } Declare StmList RCURL StmList { Mexit(); }
    | {};
```

Explanation: - The `StmList` rule handles multiple types of statements: - `Statement SemiCol StmList`: A general statement followed by a semicolon and then another statement list. - `IfStm StmList`: An `if` statement followed by another statement list. - `WHILE LPAREN ComRelExp RPAREN LCURL Label Label Label StmList RCURL`: This handles a `while` loop with conditional checks, labels for jump instructions, and a statement list inside curly braces. It also prints out assembly-like instructions for jumps. - `LCURL Declare StmList RCURL StmList`: Handles a block of code within curly braces, entering a new scope, handling declarations, and then exiting the scope. - Empty case ({}): This represents an empty statement list.

2. Declare: Declarations

The `Declare` rule defines variable declarations, including arrays, and handles both basic types and array declarations.

Listing 11: Declare Rule

```
Declare:
    Type DeclareList SemiCol Declare{ }
    | {}
```

Explanation: - `Type DeclareList SemiCol Declare`: This rule handles declarations where variables of a certain `Type` are declared, followed by a list of declared variables and an optional next declaration. - Empty case ({}): This represents an empty declaration.

3.DeclareList: List of Declarations

The `DeclareList` non-terminal handles the declaration of individual variables, including the handling of array sizes and value assignments.

Listing 12: DeclareList Rule

```
DeclareList:
    ID { SymAdd($1, str1, 0, 0); }
    | ID LEFT_SQ ArraySize RIGHT_SQ { SymAdd($1, str1, 1, $3); }
    | ID { SymAdd($1, str1, 0, 0); } COMMA DeclareList
    | ID EQ Val { SymAdd($1, str1, 0, 0); }
    | ID EQ Val { SymAdd($1, str1, 0, 0); } COMMA DeclareList
```

Explanation: - `ID`: A basic variable declaration. - `ID LEFT`$_SQArraySizeRIGHT_SQ$: $Anarraydeclarationwithaspecifiedsize. -$`ID EQ Val` : $Avariableisinitializedwithavalue. - Comma - separatedlist : Thishandlesmultiplevariabledeclarations.$

4. Type: Data Types

The Type non-terminal handles the assignment of data types to variables (integer, float, or character).

Listing 13: Type Rule

```
Type:
    INTNUM { strcpy(str1, "int"); }
    | FLOAT { strcpy(str1, "float"); }
    | CHAR { strcpy(str1, "char"); }
```

Explanation: - Assigns a type (int, float, or char) to the variable being declared.

ArraySize: Array Size

The ArraySize non-terminal handles the size of arrays.

Listing 14: ArraySize Rule

```
ArraySize:
    INT { $$ = $1; }
```

Explanation: - This rule specifies the size of an array as an integer value.

5. IfStm: If Statement

The IfStm rule handles the parsing of if statements and generates corresponding assembly-like instructions for conditional jumps.

Listing 15: IfStm Rule

```
IfStm: IF LPAREN ComRelExp RPAREN Label Label LCURL {
        if(!flag) printf("%d␣if␣%s␣goto␣%s\n", lineNo++, $3, $5);
        if(!flag) printf("%d␣goto␣%s\n", lineNo++, $6);
        if(!flag) printf("%d␣%s:\n", lineNo++, $5);
    } StmList Label RCURL {
        if(!flag) printf("%d␣goto␣%s\n", lineNo++, $10);
        if(!flag) printf("%d␣%s:\n", lineNo++, $6);
    } ElseStmt { if(!flag) printf("%d␣%s:\n", lineNo++, $10); } StmList
    | IF error { yyerror("Missing␣'('␣in␣'if'␣statement."); }
    | IF LPAREN error RPAREN { yyerror("Invalid␣condition␣in␣'if'␣state
    | IF LPAREN ComRelExp error { yyerror("Missing␣')'␣in␣'if'␣statemen
    | IF LPAREN ComRelExp RPAREN Label Label error { yyerror("Missing␣b
```

Explanation: - Valid if structure: Handles the if statement with conditional expressions and jump instructions. - Error handling: Provides specific error messages for common syntax mistakes in if statements (e.g., missing parentheses, invalid conditions).

ElseStmt: Else Statement

The ElseStmt non-terminal handles the parsing of else clauses in if-else statements.

Listing 16: ElseStmt Rule

```
ElseStmt: ELSE LCURL StmList RCURL { }
        | ELSE IfStm { }
        | ELSE error { yyerror("Missing␣'{'␣in␣else␣statement."); }
        | ELSE LCURL StmList error { yyerror("Missing␣'}'␣in␣else␣state
        | {};
```

Explanation: - Else with block: Handles the else keyword followed by a block of statements. - Else with nested if: Handles else followed by an if statement. - Error handling: Provides error messages for missing braces in the else statement.

6. Label: Labels for Jumps

The Label rule handles the creation of labels for jump instructions in assembly-like code generation.

Listing 17: Label Rule

```
Label: { $$ = (char*)malloc(100 * sizeof(char)); $$ = BLable(); };
```

Explanation: - This rule allocates memory for a label and generates a new label using the BLable() function.

Non-Terminals Overview:
- StmList: A list of statements, including conditionals, loops, and blocks.
- Declare: Handles variable declarations.
- DeclareList: A list of variables to be declared, with support for arrays.
- Type: Specifies the data type (e.g., int, float, char).
- ArraySize: Specifies the size of an array.
- IfStm: Handles the if statement.
- ElseStmt: Handles the else clause of an if-else statement.
- Label: Generates labels for jump instructions in assembly-like output.

7. Statement: Assignment and Operations

The Statement rule handles variable assignments, including basic assignments and compound operations like addition, subtraction, multiplication, and division.

Listing 18: Statement Rule

```
Statement:
    ID EQ Statement {
        if(Sfind($1) == false) { printf("error:␣var␣'%s'␣is␣not␣declare
        if(!flag) printf("%d␣%s␣=␣%s\n", lineNo++, $1, $3);
    }
```

Explanation: - ID EQ Statement: This rule handles a basic assignment where a variable is assigned the value of another statement. It checks if the variable is declared in the scope and prints the corresponding assignment in assembly-like format.

8. ComRelExp: Logical Expressions

The ComRelExp rule handles logical operations like AND and OR between relational expressions.

Listing 19: ComRelExp Rule

```
ComRelExp:
    ComRelExp AND Relexp {
        strcpy($$, LineLabels());
        if(!flag) printf("%d␣%s␣=␣%s␣&&␣%s\n", lineNo++, $$, $1, $3);
    }
```

Explanation: - ComRelExp AND Relexp: This rule handles logical AND operations between two relational expressions and generates the corresponding assembly-like instruction.

9. Relexp: Relational Expressions

The Relexp rule handles relational operators (e.g., less than, greater than) between terms.

Listing 20: Relexp Rule

```
Relexp:
    Term LT Term {
        strcpy($$, LineLabels());
        if(!flag) printf("%d␣%s␣=␣%s␣<␣%s\n", lineNo++, $$, $1, $3);
    }
```

Explanation: - Term LT Term: This rule handles the less-than comparison between two terms and generates the corresponding assembly-like instruction for the comparison.

10. Term: Arithmetic Operations

The Term rule handles arithmetic operations, such as addition and subtraction, between terms and factors.

Listing 21: Term Rule

```
Term:
    Term ADD Factor {
        strcpy($$, LineLabels());
        if(!flag) printf("%d␣%s␣=␣%s␣+␣%s\n", lineNo++, $$, $1, $3);
    }
```

Explanation: - Term ADD Factor: This rule handles the addition of two factors and prints the corresponding assembly-like instruction.

11. Factor: Multiplicative Operations

The Factor rule handles multiplicative operations like multiplication and division.

Listing 22: Factor Rule

```
Factor:
    Factor MUL SignVal {
        strcpy($$, LineLabels());
        if(!flag) printf("%d␣%s␣=␣%s␣*␣%s\n", lineNo++, $$, $1, $3);
    }
```

Explanation: - Factor MUL SignVal: This rule handles multiplication between a factor and a value (SignVal) and generates the corresponding assembly-like instruction.

12. PostfixOperation: Postfix Increment/Decrement

The PostfixOperation rule handles postfix increment and decrement operations on variables.

Listing 23: PostfixOperation Rule

```
PostfixOperation:
    ID INC {
        if(Sfind($1) == false) { printf("error:␣var␣'%s'␣is␣not␣declare
        strcpy($$, LineLabels());
        if(!flag) printf("%d␣%s␣=␣%s\n", lineNo++, $$, $1);
        if(!flag) printf("%d␣%s␣=␣%s␣+␣1\n", lineNo++, $1, $$);
    }
```

Explanation: - ID INC: This rule handles the postfix increment operation on a variable and generates the corresponding instructions for incrementing the variable.

13. PrefixOperation: Prefix Increment/Decrement

The PrefixOperation rule handles prefix increment and decrement operations on variables.

Listing 24: PrefixOperation Rule

```
PrefixOperation:
    INC ID {
        if(Sfind($2) == false) { printf("error:␣var␣'%s'␣is␣not␣declare
        strcpy($$, LineLabels());
        if(!flag) printf("%d␣%s␣=␣%s\n", lineNo++, $$, $2);
        if(!flag) printf("%d␣%s␣=␣%s␣+␣1\n", lineNo++, $2, $$);
    }
```

Explanation: - INC ID: This rule handles the prefix increment operation on a variable and generates the corresponding instructions for incrementing the variable.

14. SignVal: Signed Values

The SignVal rule handles signed values, including positive and negative numbers.

Listing 25: SignVal Rule

```
SignVal:
    ADD Val { strcpy($$, "+"); strcat($$, $2); }
```

Explanation: - ADD Val: This rule handles signed values, where a value can be prefixed with a '+' sign, and it appends the value accordingly.

15. Val: Values

The Val rule handles various types of values like variables, integers, and floats.

Listing 26: Val Rule

```
Val:
    ID { strcpy($$, $1);
        if(Sfind($1) == false) { printf("error:␣var␣'%s'␣is␣not␣declare
    }
```

Explanation: - ID: This rule handles a variable value and checks if the variable is declared in the scope. If the variable is not declared, it prints an error message.

Function Definitions Explanation

1. LineLabels

The LineLabels function generates temporary variable labels, such as t1, t2, etc., for intermediate values.

Listing 27: LineLabels Function

```
char* LineLabels() {
    char* s = (char*)malloc(100);
    sprintf(s, "t%d", t++);
    return s;
}
```

Explanation: - Generates unique temporary labels (e.g., t1, t2) and returns them as strings.

2. BLable

The BLable function generates labels for branches (e.g., L1, L2, etc.).

Listing 28: BLable Function

```
char* BLable() {
    char* s = (char*)malloc(100);
    sprintf(s, "L%d", lab++);
    return s;
}
```

Explanation: - Generates unique labels for branches, typically used in conditional or loop statements (e.g., L1, L2).

3. yyerror

The yyerror function handles syntax errors during parsing by printing an error message.

Listing 29: yyerror Function

```
void yyerror(char* s) {
    if (strcmp(s, "parse␣error") != 0) {
        printf("\n---------------------------Syntax␣Error:␣%s----------
    }
}
```

Explanation: - Prints a syntax error message if the provided string is not "parse error".

4. getType

The getType function returns the size of a variable type (e.g., int, float, etc.).

Listing 30: getType Function

```
int getType(char* type) {
    if (strcmp(type, "int") == 0) return IntSize;
    if (strcmp(type, "float") == 0) return FloatSize;
    if (strcmp(type, "char") == 0) return CharSize;
    return 0;
}
```

Explanation: - Returns the size of the variable type based on a predefined size for int, float, and char.

5. enter

The enter function begins a new scope by incrementing the scope stack.

Listing 31: enter Function

```
void enter() {
    scopeStackTop++;
    scopeStack[scopeStackTop].symbolCount = 0;
    scopeStack[scopeStackTop].Curroffset = 0;
}
```

Explanation: - Enters a new scope and initializes counters for symbol count and current offset.

6. Mexit

The Mexit function exits the current scope by decrementing the scope stack.

Listing 32: Mexit Function

```
void Mexit() {
    scopeStackTop--;
}
```

Explanation: - Exits the current scope by decrementing the scope stack pointer.

7. SymAdd

The SymAdd function adds a new symbol (variable) to the current scope and checks for redeclarations or type conflicts.

Listing 33: SymAdd Function

```
int SymAdd(char* name, char* type, int isArray, int arraySize) {}
```

Explanation: - Adds a symbol to the current scope, checking for redeclaration or type conflicts. Updates the symbol list for later use.

8. Sfind

The Sfind function searches for a symbol by name in the symbol list.

Listing 34: Sfind Function

```
bool Sfind(char* name) {
    for (int i = symbolListCount - 1; i >= 0; i--) {
        if (strcmp(symbolList[i].name, name) == 0) return true;
    }
    return false;
}
```

Explanation: - Searches through the symbol list in reverse order to check if a symbol has been declared.

9. printAllScopeSymbols

The printAllScopeSymbols function prints all symbols in the symbol list, showing their names, types, and sizes.

Listing 35: printAllScopeSymbols Function

```
void printAllScopeSymbols() {
    for (int i = 0; i < symbolListCount; i++) {
        Symbol* symbol = &symbolList[i];
        if (symbol->isArray) {
            printf("0x%04X␣%s␣%s␣array␣%d\n", symbol->offset, symbol->n
        } else {
            printf("0x%04X␣%s␣%s\n", symbol->offset, symbol->name, symb
        }
    }
}
```

Explanation: - Prints all the symbols in the symbol list, including their offset, name, type, and size. Handles array symbols separately.

Final Output

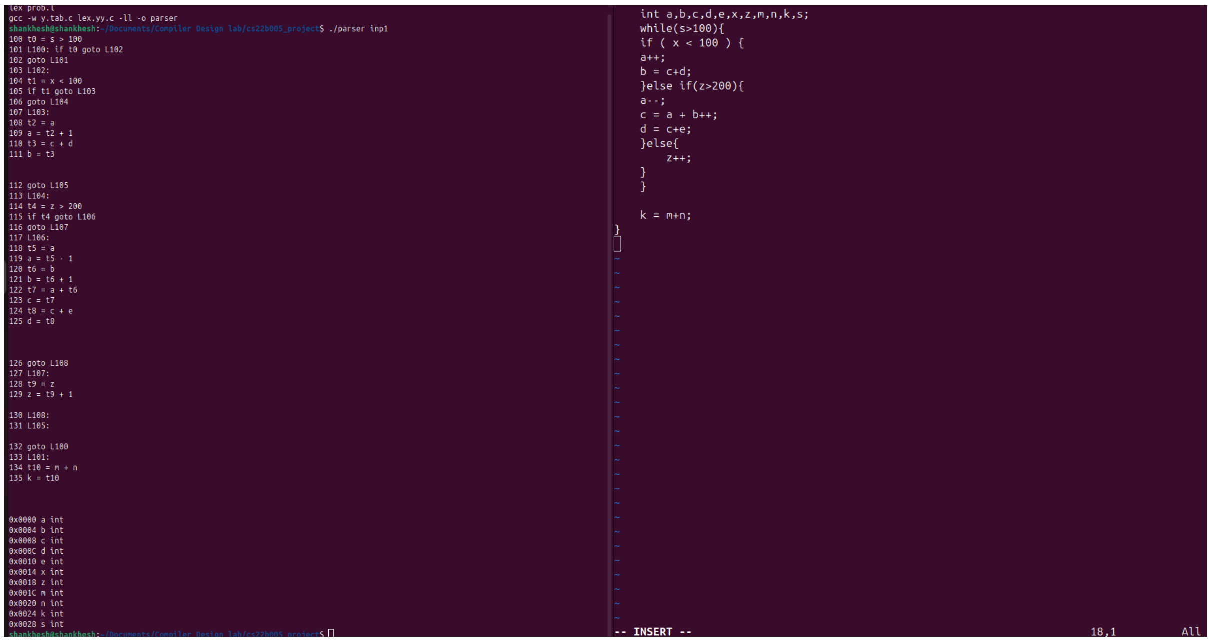
If the input program is valid, the compiler generates an abstract syntax tree and updates the symbol table, preparing the program for further stages such as intermediate code generation or machine code output.

Conclusion

This project demonstrates the design and implementation of a basic compiler. The use of Lex and Yacc simplifies the process of lexical and syntax analysis, making them invaluable tools in compiler design.

My Input files and Output according to it

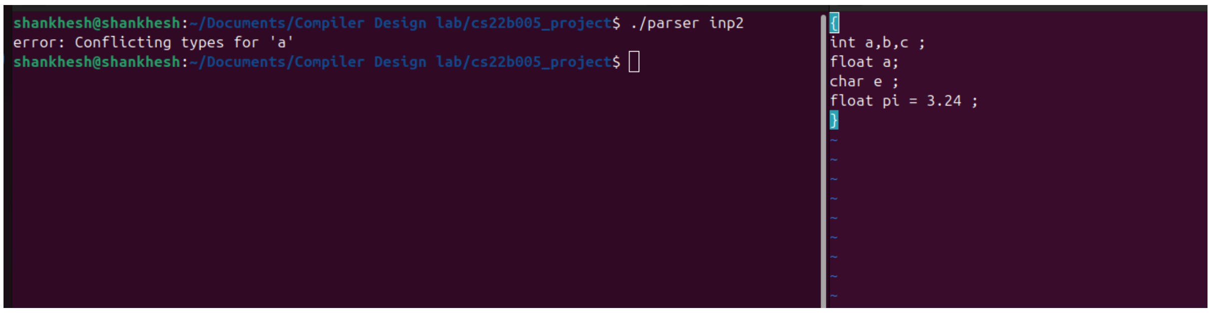
Below are my input files and outputs related to it. Each input file contains diffrent kind of test case which showa the features of my code and error handling inside my code.



```
lex prob.l
gcc -w y.tab.c lex.yy.c -ll -o parser
shankhesh@shankhesh:~/Documents/Compiler Design lab/cs22b005_project$ ./parser inp1
100 t0 = s > 100
101 L100: if t0 goto L102
102 goto L101
103 L102:
104 t1 = x < 100
105 if t1 goto L103
106 goto L104
107 L103:
108 t2 = a
109 a = t2 + 1
110 t3 = c + d
111 b = t3


112 goto L105
113 L104:
114 t4 = z > 200
115 if t4 goto L106
116 goto L107
117 L106:
118 t5 = a
119 a = t5 - 1
120 t6 = b
121 b = t6 + 1
122 t7 = a + t6
123 c = t7
124 t8 = c + e
125 d = t8



126 goto L108
127 L107:
128 t9 = z
129 z = t9 + 1

130 L108:
131 L105:

132 goto L100
133 L101:
134 t10 = m + n
135 k = t10



0x0000 a int
0x0004 b int
0x0008 c int
0x000C d int
0x0010 e int
0x0014 x int
0x0018 z int
0x001C m int
0x0020 n int
0x0024 k int
0x0028 s int
shankhesh@shankhesh:~/Documents/Compiler Design lab/cs22b005_project$
```

```
    int a,b,c,d,e,x,z,m,n,k,s;
    while(s>100){
    if ( x < 100 ) {
    a++;
    b = c+d;
    }else if(z>200){
    a--;
    c = a + b++;
    d = c+e;
    }else{
        z++;
    }
    }

    k = m+n;
}
-- INSERT --                                                    18,1          All
```

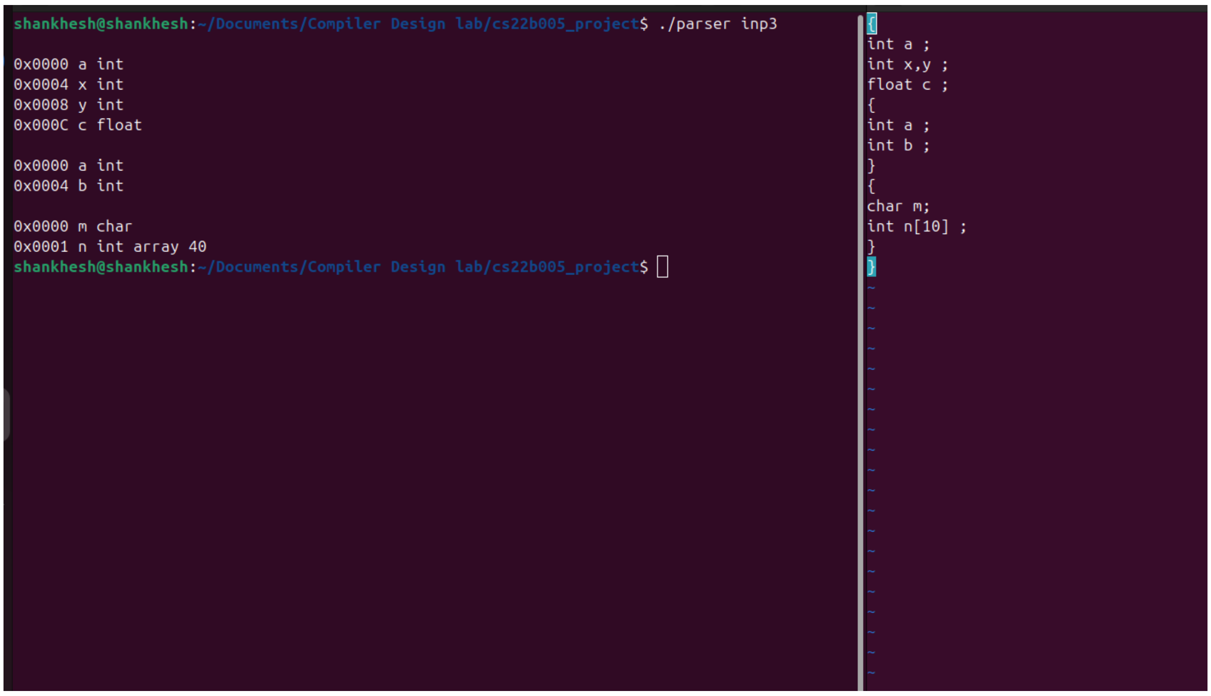
Figure 1: inp1 file and output related to it.



```
shankhesh@shankhesh:~/Documents/Compiler Design lab/cs22b005_project$ ./parser inp2
error: Conflicting types for 'a'
shankhesh@shankhesh:~/Documents/Compiler Design lab/cs22b005_project$
```

```
{
int a,b,c ;
float a;
char e ;
float pi = 3.24 ;
}
```

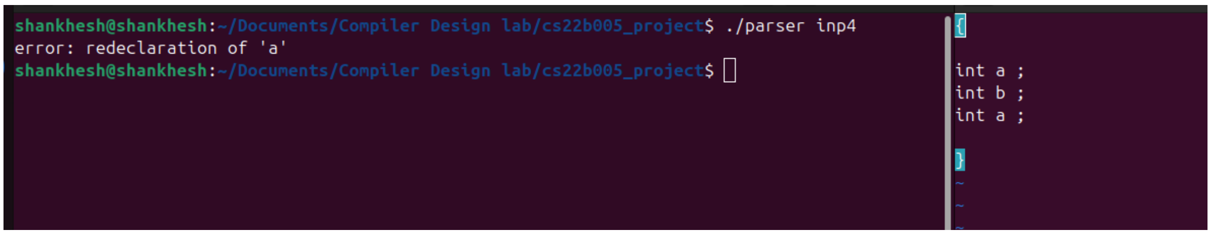
Figure 2: inp2 file and output related to it.



```
shankhesh@shankhesh:~/Documents/Compiler Design lab/cs22b005_project$ ./parser inp3

0x0000 a int
0x0004 x int
0x0008 y int
0x000C c float

0x0000 a int
0x0004 b int

0x0000 m char
0x0001 n int array 40
shankhesh@shankhesh:~/Documents/Compiler Design lab/cs22b005_project$
```

```
{
int a ;
int x,y ;
float c ;
{
int a ;
int b ;
}
{
char m;
int n[10] ;
}
}
```

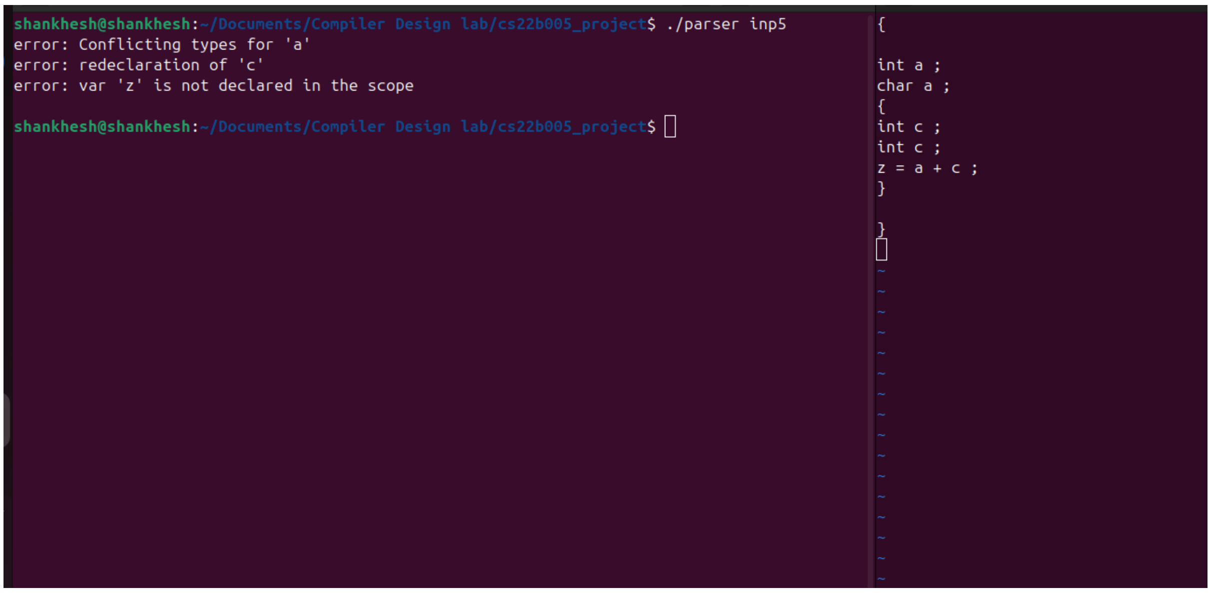
Figure 3: inp3 file and output related to it.



```
shankhesh@shankhesh:~/Documents/Compiler Design lab/cs22b005_project$ ./parser inp4
error: redeclaration of 'a'
shankhesh@shankhesh:~/Documents/Compiler Design lab/cs22b005_project$
```

```
{

int a ;
int b ;
int a ;

}
```

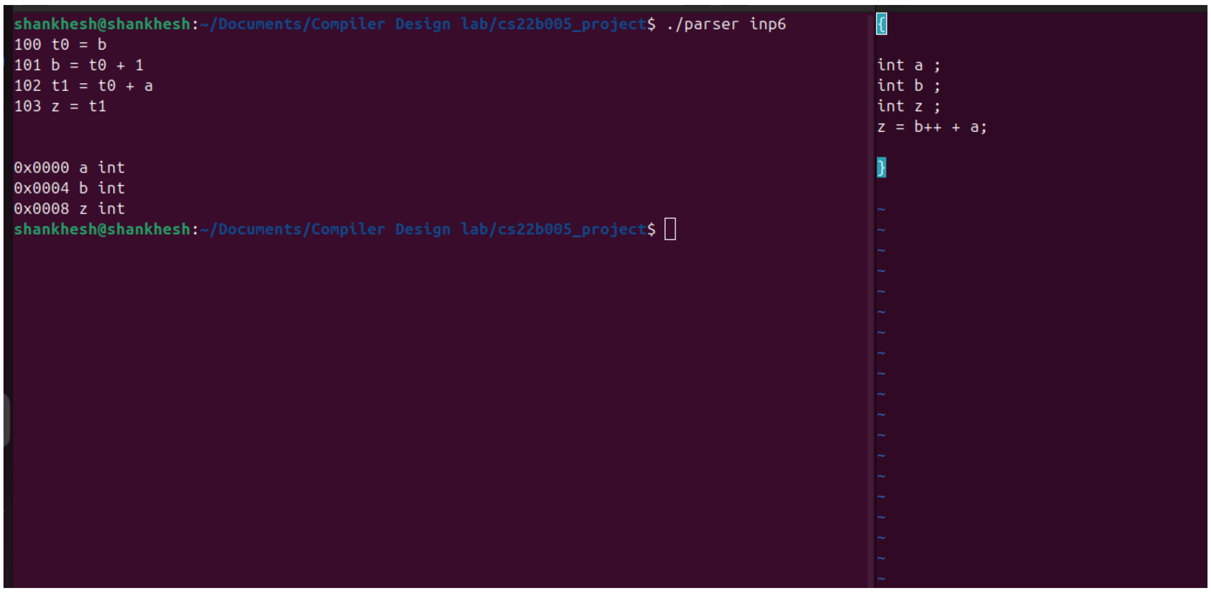
Figure 4: inp4 file and output related to it.



```
shankhesh@shankhesh:~/Documents/Compiler Design lab/cs22b005_project$ ./parser inp5
error: Conflicting types for 'a'
error: redeclaration of 'c'
error: var 'z' is not declared in the scope

shankhesh@shankhesh:~/Documents/Compiler Design lab/cs22b005_project$
```

```
{

int a ;
char a ;
{
int c ;
int c ;
z = a + c ;
}

}
```

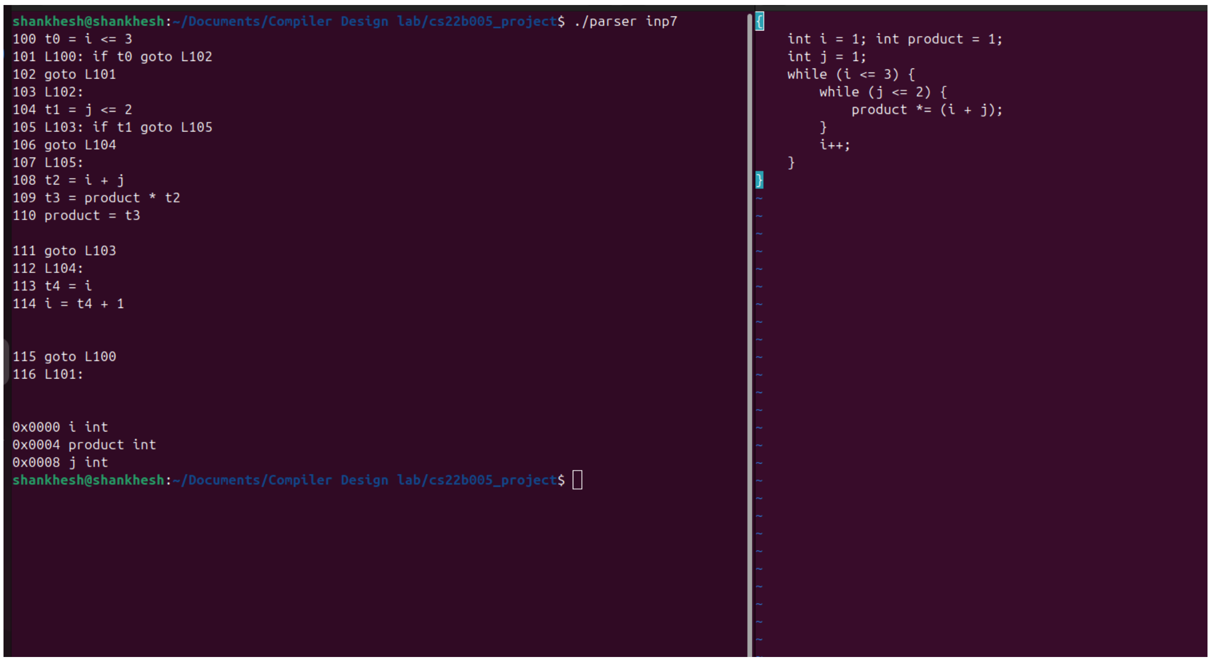
Figure 5: inp5 file and output related to it.



```
shankhesh@shankhesh:~/Documents/Compiler Design lab/cs22b005_project$ ./parser inp6
100 t0 = b
101 b = t0 + 1
102 t1 = t0 + a
103 z = t1


0x0000 a int
0x0004 b int
0x0008 z int
shankhesh@shankhesh:~/Documents/Compiler Design lab/cs22b005_project$
```

```
{

int a ;
int b ;
int z ;
z = b++ + a;

}
```

Figure 6: inp6 file and output related to it.



```
shankhesh@shankhesh:~/Documents/Compiler Design lab/cs22b005_project$ ./parser inp7
100 t0 = i <= 3
101 L100: if t0 goto L102
102 goto L101
103 L102:
104 t1 = j <= 2
105 L103: if t1 goto L105
106 goto L104
107 L105:
108 t2 = i + j
109 t3 = product * t2
110 product = t3

111 goto L103
112 L104:
113 t4 = i
114 i = t4 + 1


115 goto L100
116 L101:


0x0000 i int
0x0004 product int
0x0008 j int
shankhesh@shankhesh:~/Documents/Compiler Design lab/cs22b005_project$ 
```

```
{
    int i = 1; int product = 1;
    int j = 1;
    while (i <= 3) {
        while (j <= 2) {
            product *= (i + j);
        }
        i++;
    }
}
```

Figure 7: inp7 file and output related to it.