# Pookie-Lang Compiler

Session: 2021 – 2025

**Submitted by:**

Muhammad Abubakar Siddique Farooqi 2021-CS-171

**Submitted to:**

Mr. Laeeq uz Zaman Khan Niazi

Department of Computer Science

**University of Engineering and Technology**

**Lahore Pakistan**

# Acknowledgments

I would like to begin by expressing my profound gratitude to Allah, the Almighty, for His endless support, guidance, and blessings throughout my educational journey. His grace has been my source of strength, inspiration, and resilience, enabling me to overcome challenges and achieve success, particularly in the Compiler Construction course.

I would also like to extend my deepest thanks to my esteemed teacher, Mr. Laeeq uz Zaman Khan Niazi, for his unwavering dedication, guidance, and support. His invaluable contributions have played a pivotal role in the successful development of my project, enriching my learning experience and equipping me for future endeavors. I am truly grateful for his mentorship.

# Contents

**Acknowledgments** **i**

**List of Figures** **iv**

**1 Introduction** **1**
- 1.1 Why Pookie-Lang . . . 1

**2 Lexical Analyzer Phase** **2**
- 2.1 Token Struct . . . 2
- 2.2 Explanation of Lexer Code . . . 3
  - 2.2.1 Explanation . . . 3

**3 Parser Phase** **4**
- 3.1 ASTNode Struct . . . 4
- 3.2 Explanation of Parser Code . . . 5
  - 3.2.1 Explanation . . . 5

**4 Three Address Code Generation Phase** **7**
- 4.1 TAC Struct . . . 7
- 4.2 Explanation of TAC generator Code . . . 8
  - 4.2.1 Explanation . . . 8

**5 Assembly Code Generation Phase** **9**
- 5.1 RegisterInfo Struct . . . 9
- 5.2 DataSegment Struct . . . 9
- 5.3 Explanation of Assembly Generator Code . . . 10
  - 5.3.1 Explanation . . . 10

**6 Assembly Language Program Compilation and Execution** **12**
- 6.1 Source Code Preparation . . . 12
- 6.2 Generating Object and Listing file . . . 12
- 6.3 Linking and Generating Executable file . . . 13
- 6.4 Flow Diagram . . . 14

**7 Extra Features** **15**
- 7.1 All Relational Operators . . . 15

7.2 Abstract Syntax Tree (AST) . . . . . . . . . . . . . . . . . . . . . . . . . 15
7.3 Input and Output Statements . . . . . . . . . . . . . . . . . . . . . . . 15
7.4 Efficient Register Allocation with Limited Resources and System Stack usage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
7.5 Loop Constructs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
7.6 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
7.7 Variable Scope . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
7.8 Efficient Use of Symbol Table . . . . . . . . . . . . . . . . . . . . . . 16
7.9 Integration with Irvine32 Library . . . . . . . . . . . . . . . . . . . . 16
7.10 Listing File . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
7.11 Object file and Executable file . . . . . . . . . . . . . . . . . . . . . 16
7.12 Pookie-Lang Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

# List of Figures

2.1 CRC card for Lexer. . . . . . . . . . . . . . . . . . . . . . . . . . 3

3.1 CRC card for Parser. . . . . . . . . . . . . . . . . . . . . . . . . . 5

4.1 CRC card for TACgenerator. . . . . . . . . . . . . . . . . . . . . . . 8

5.1 CRC card for Assembly. . . . . . . . . . . . . . . . . . . . . . . . . 10

6.1 Irvine32.lib linking. . . . . . . . . . . . . . . . . . . . . . . . . . 13
6.2 Compier Flow Diagram . . . . . . . . . . . . . . . . . . . . . . . . . 14

# Chapter 1

# Introduction

Welcome to **PookieLang**, the sassiest, quirkiest programming language you'll ever encounter! Inspired by the hilarious simplicity of BhaiLang, PookieLang takes the drama, sarcasm, and fun to the next level. It's a language where code meets chaos, variables become superheroes, and loops come with their own comedy skits.

PookieLang is created for those who believe programming should be fun and expressive—where every line of code tells a story (or cracks a joke). Whether you're summoning your "Batman," spinning through "Pookie Ghumega," or laughing at the quirks of "Pookie Fas Gaya," this language will ensure coding never feels boring again.

## 1.1 Why Pookie-Lang

- **Hilarious Keywords:** Forget boring syntax; PookieLang has terms like "mera_batman" for variables and "aja_beta" for function calls.
- **Sarcasm in Code:** Every loop, function, and conditional statement oozes sarcasm and humor.
- **Superhero-Themed Syntax:** From "Pookie" to "Batman," every keyword adds personality to your code.
- **Inspired by BhaiLang:** While BhaiLang introduced us to programming in a Desi, fun way, PookieLang takes it up a notch with even more relatable slang, quirky Gen Z terms, and creative twists.

# Chapter 2

# Lexical Analyzer Phase

The lexical analyzer processes the source code as a string and scans it character by character, classifying sequences of characters that form valid tokens. Tokens represent various elements of the programming language, such as keywords, identifiers, operators, literals, and punctuation.

## 2.1 Token Struct

Each token is represented by a struct with the following properties.

```
struct Token {
    TokenType type;
    string value;
    size_t lineNo;
};
```

Listing 2.1: Token Struct.

## 2.2 Explanation of Lexer Code

The Lexer class is responsible for lexecial analysis. Following is detail explanation of lexer class.

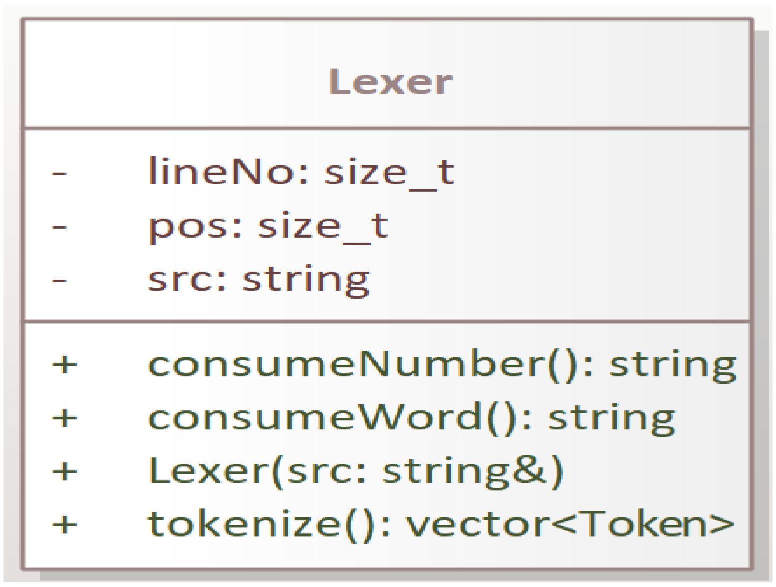


Figure 2.1: CRC card for Lexer.

### 2.2.1 Explanation

- The main function of the lexer that processes the source code and generates a vector of tokens.
- Reads the source code character by character and classifies sequences into meaningful tokens based on patterns and rules.
- Skips them when encountered (//).
- Skips them while tracking line numbers.
- Uses **consumeNumber()** to capture complete numeric literals.
- **consumeWord()** to process keywords and identifiers.
- Matches single-character and multi-character operators.
- Adds each identified token to the tokens vector.
- Appends an EOF token at the end to mark the end of the input.
- Tracks the type, value, and line number of each token.

# Chapter 3

# Parser Phase

The parser parses a sequence of tokens into an Abstract Syntax Tree (AST) while checking for syntax errors.

## 3.1 ASTNode Struct

Each AST node is represented by a struct with the following properties.

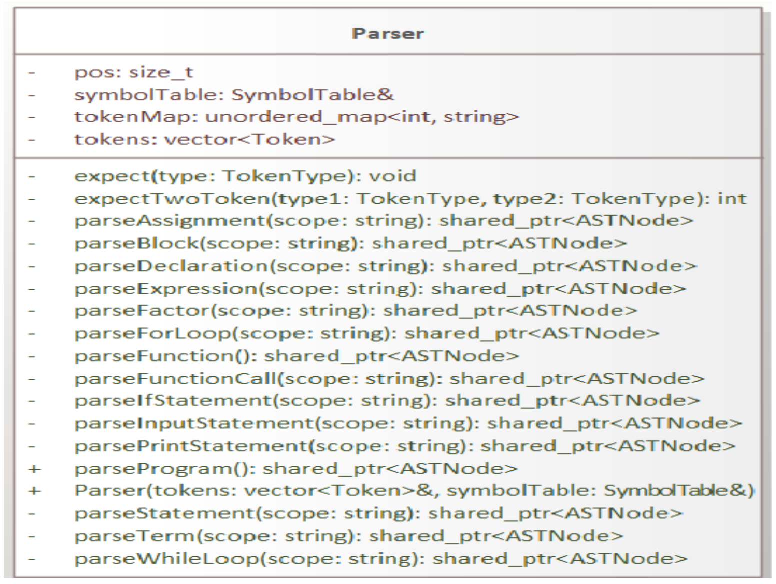
```
struct ASTNode
{
    string value;
    vector<std::shared_ptr<ASTNode>> children;

    ASTNode(const std::string &val) : value(val) {}
};
```

Listing 3.1: ASTnode Struct.

## 3.2 Explanation of Parser Code

The Lexer class is responsible for parsing. Following is detail explanation of lexer class.



```
                         Parser
-   pos: size_t
-   symbolTable: SymbolTable&
-   tokenMap: unordered_map<int, string>
-   tokens: vector<Token>
-   expect(type: TokenType): void
-   expectTwoToken(type1: TokenType, type2: TokenType): int
-   parseAssignment(scope: string): shared_ptr<ASTNode>
-   parseBlock(scope: string): shared_ptr<ASTNode>
-   parseDeclaration(scope: string): shared_ptr<ASTNode>
-   parseExpression(scope: string): shared_ptr<ASTNode>
-   parseFactor(scope: string): shared_ptr<ASTNode>
-   parseForLoop(scope: string): shared_ptr<ASTNode>
-   parseFunction(): shared_ptr<ASTNode>
-   parseFunctionCall(scope: string): shared_ptr<ASTNode>
-   parseIfStatement(scope: string): shared_ptr<ASTNode>
-   parseInputStatement(scope: string): shared_ptr<ASTNode>
-   parsePrintStatement(scope: string): shared_ptr<ASTNode>
+   parseProgram(): shared_ptr<ASTNode>
+   Parser(tokens: vector<Token>&, symbolTable: SymbolTable&)
-   parseStatement(scope: string): shared_ptr<ASTNode>
-   parseTerm(scope: string): shared_ptr<ASTNode>
-   parseWhileLoop(scope: string): shared_ptr<ASTNode>
```

FIGURE 3.1: CRC card for Parser.

### 3.2.1 Explanation

- Token Map: Maps token types (e.g., T_IF, T_ID) to their string representations for error messages.
- Symbol Table: Tracks declared variables, functions, and their scopes.
- Token List: Stores the list of tokens generated by a lexer.
- **parseProgram():** Entry point of the parser. Starts by parsing a block and ensures all tokens are processed without errors.
- **parseStatement(scope):** Handles various statements (if, while, for, declarations, assignments, etc.) based on the current token type.
- **parseBlock(scope):** Parses a sequence of statements enclosed in .
- **parsePrintStatement(scope):** Parses print statements and ensures the variable exists in the symbol table.
- **parseInputStatement(scope):** Parses input statements, similar to print, and checks variable existence.

- **parseFunction():** Handles function declarations. Validates parameter uniqueness and records function metadata in the symbol table.
- **parseFunctionCall(scope):** Handles function call syntax and validates argument matching.
- **parseForLoop(scope) & parseWhileLoop(scope):** Parse loop constructs, including initialization, condition, and increment/decrement.
- **parseDeclaration(scope):** Parses variable declarations and ensures no redeclaration within the same scope.
- **parseAssignment(scope):**
- Displays descriptive error messages for undeclared variables, duplicate declarations, and invalid syntax.
- Returns an AST representing the structure of the input code.

# Chapter 4

# Three Address Code Generation Phase

## 4.1 TAC Struct

Each TAC is represented by a struct with the following properties.

```
struct TAC
{
    std::string result;
    std::string op;
    std::string arg1;
    std::string arg2;
    vector<string> extras;
};
```

LISTING 4.1: TAC Struct.

## 4.2 Explanation of TAC generator Code

The TACGenerator class generates Three Address Code (TAC) from an Abstract Syntax Tree (AST).

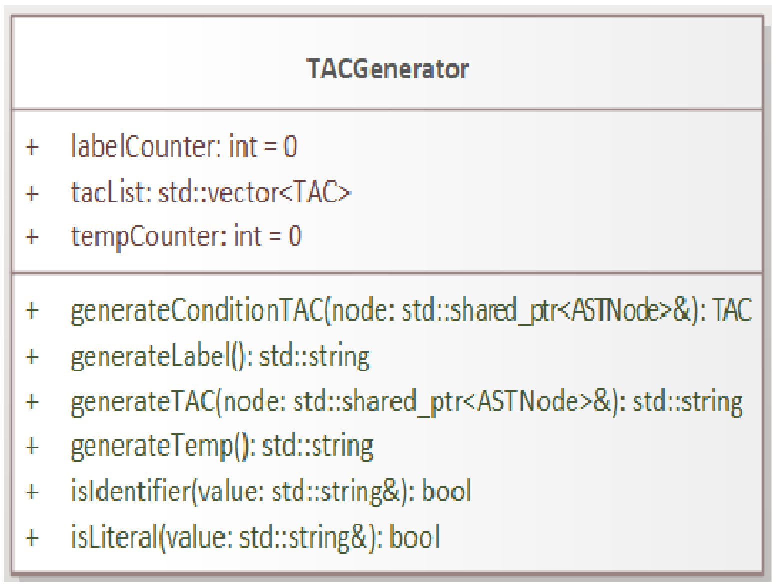

TACGenerator
+ labelCounter: int = 0
+ tacList: std::vector<TAC>
+ tempCounter: int = 0
+ generateConditionTAC(node: std::shared_ptr<ASTNode>&): TAC
+ generateLabel(): std::string
+ generateTAC(node: std::shared_ptr<ASTNode>&): std::string
+ generateTemp(): std::string
+ isIdentifier(value: std::string&): bool
+ isLiteral(value: std::string&): bool

FIGURE 4.1: CRC card for TACgenerator.

### 4.2.1 Explanation

- **generateTemp():** Produces unique temporary variables (t1, t2, etc.).
- **generateLabel():** Creates unique labels (L1, L2, etc.) for loops, conditions, and function blocks.
- **generateTAC(node):**

  Recursively processes an AST node and its children. Handles various constructs: print, input, functions, loops, if statements, arithmetic, etc. Returns a temporary variable or directly generates TAC for the operation.
- **generateConditionTAC(node):**

  Generates TAC for conditional expressions (e.g., x ¡ y) and returns a TAC instruction.
- Stores all generated TAC instructions in **tacList**.

# Chapter 5

# Assembly Code Generation Phase

Following are helper structs for assembly generation.

## 5.1 RegisterInfo Struct

Each Register Meta Data is represented by a struct with the following properties.

```
struct RegisterInfo
{
    bool isFree;
    string variable; // the variable whose value is inside the register
    RegisterInfo(string var = "", bool isFree = true)
        : variable(var), isFree(isFree) {}
};
```

Listing 5.1: RegisterInfo Struct.

## 5.2 DataSegment Struct

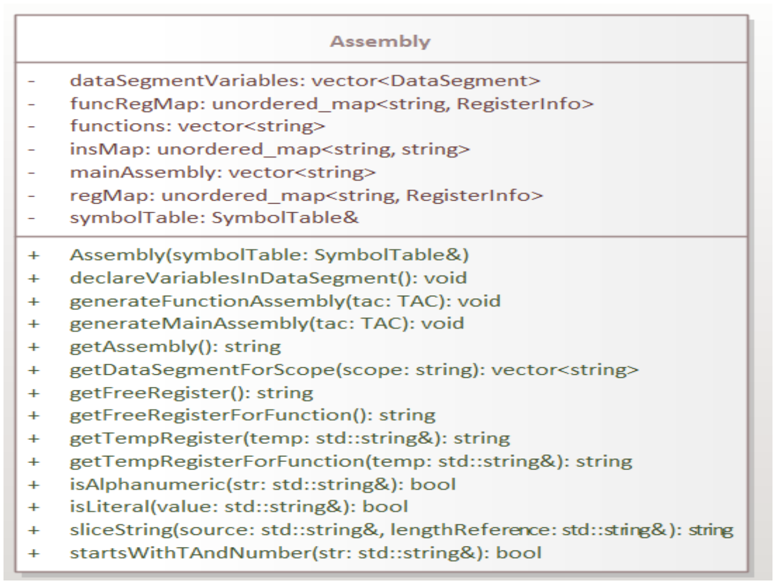
Each Memory reference is represented by a struct with the following properties.

```
struct DataSegment
{
    string var;
    string type;
    string val;
    string scope;
    DataSegment(string var, string type, string val, string scope = "main") : var(var), type(typ
};
```

Listing 5.2: DataSegment Struct.

## 5.3 Explanation of Assembly Generator Code

This code is part of a compiler backend that translates three-address code (TAC) into x86 assembly.



```
Assembly
-   dataSegmentVariables: vector<DataSegment>
-   funcRegMap: unordered_map<string, RegisterInfo>
-   functions: vector<string>
-   insMap: unordered_map<string, string>
-   mainAssembly: vector<string>
-   regMap: unordered_map<string, RegisterInfo>
-   symbolTable: SymbolTable&
+   Assembly(symbolTable: SymbolTable&)
+   declareVariablesInDataSegment(): void
+   generateFunctionAssembly(tac: TAC): void
+   generateMainAssembly(tac: TAC): void
+   getAssembly(): string
+   getDataSegmentForScope(scope: string): vector<string>
+   getFreeRegister(): string
+   getFreeRegisterForFunction(): string
+   getTempRegister(temp: std::string&): string
+   getTempRegisterForFunction(temp: std::string&): string
+   isAlphanumeric(str: std::string&): bool
+   isLiteral(value: std::string&): bool
+   sliceString(source: std::string&, lengthReference: std::string&): string
+   startsWithTAndNumber(str: std::string&): bool
```

FIGURE 5.1: CRC card for Assembly.

### 5.3.1 Explanation

- Translates TAC to x86 assembly code using predefined instruction mappings.
- **Initialization:** Maps operations (+, -, etc.) and control flow statements (if, goto) to corresponding x86 instructions.
- **Main Assembly:** Manages the program's main procedure assembly, including control flow, arithmetic operations, input/output, and function calls.
- **Function Assembly:** Handles generating assembly for defined functions.
- **Instruction Mappings** A mapping (insMap) ties TAC operations to x86 equivalents: Arithmetic: $+ - >$ Add, $* - >$ Imul. Relational: $< - >$ Cmp, $== - >$ Cmp. Control Flow: if $- >$ conditional jumps like JC or JZ.
- **getDataSegmentForScope()** generates assembly declarations for variables scoped within a specified function or main.
- **Register Allocation** Registers (regMap and funcRegMap): Tracks whether a register is free or occupied and which variable it holds. Manages spills when registers are full or data needs to be stored temporarily.

- **getFreeRegister():** Finds an available register.
- **getTempRegister(temp):** Finds a register holding a specific temporary variable.
- **I/O:** Handles TAC print and input using Irvine32 library calls.
- **Function Calls:** Manages parameter passing via stack and cleans up after the call.

# Chapter 6

# Assembly Language Program Compilation and Execution

In this chapter, we delve into the process of transforming your assembly language source code into different formats, including the listing file, object file, and executable file. This process is essential for compiling and executing programs written in assembly language. First you have to ensure MASM and its linker is installed on system.

## 6.1 Source Code Preparation

Before generating the required files, ensure your source code is well-written and follows the syntax of the assembler being used (MASM). The code typically includes:

- **Data Segment:** For defining variables and constants.
- **Code Segment:** For program instructions.
- **Directives:** Like INCLUDE or INCLUDELIB for linking libraries.

## 6.2 Generating Object and Listing file

The listing file contains the source code along with additional information such as memory addresses, machine code, and symbolic labels. This file helps in debugging and analyzing the program. The object file is an intermediate binary file containing machine code for the program. It is created after the assembler processes the source code.

```
int assembleResult = system(("ml /I C:\\Irvine /c /Fl /Fo" + string(argv[2]) +
".obj "+ string(argv[2]) + ".asm").c_str());
    if (assembleResult != 0)
    {
        std::cerr << "Error during assembly.
        Check your source file or MASM setup." << std::endl;
        return 1;
    }
```

Listing 6.1: Code for object and listing file

## 6.3 Linking and Generating Executable file

The linker links the Irvine32 library with the object file and then generates the executable file. The linking typically include following configurations:

- **Kernal32.lib** The kernel32.lib library is a core part of the Windows API, providing fundamental system-level functions for managing processes, threads, memory, files, and system resources. that's why kernel32.lib is included
- **User32,lib** The user32.lib library is part of the Windows API and provides functions for creating and managing graphical user interfaces (GUIs) and interacting with the operating system's user interface components, such as windows, buttons, and messages. Since Irvine32.lib use this that's why we also need to link it.

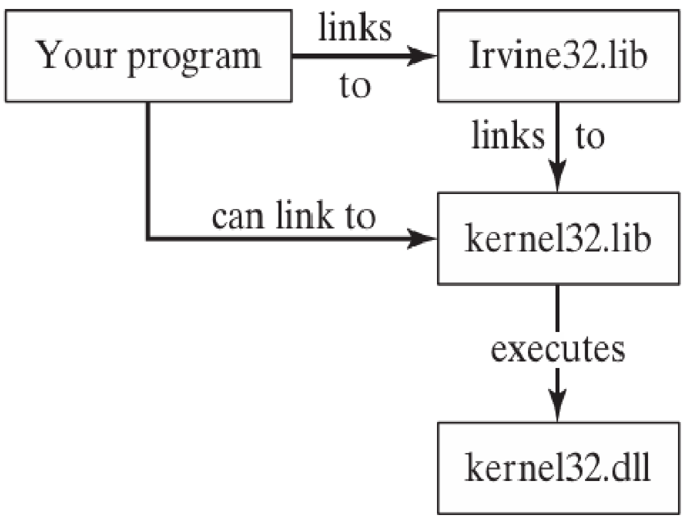


Figure 6.1: Irvine32.lib linking.

```
int linkResult = system(("link /SUBSYSTEM:CONSOLE /MACHINE:X86 /OUT:"
+ string(argv[2]) + ".exe "+ string(argv[2]) + ".obj Irvine32.lib
/LIBPATH:C:\\Irvine kernel32.lib user32.lib").c_str());
    if (linkResult != 0)
    {
        std::cerr << "Error during linking. Check library paths and dependencies."
        << std::endl;
        return 1;
    }
```

Listing 6.2: Code linking and exe file

## 6.4 Flow Diagram

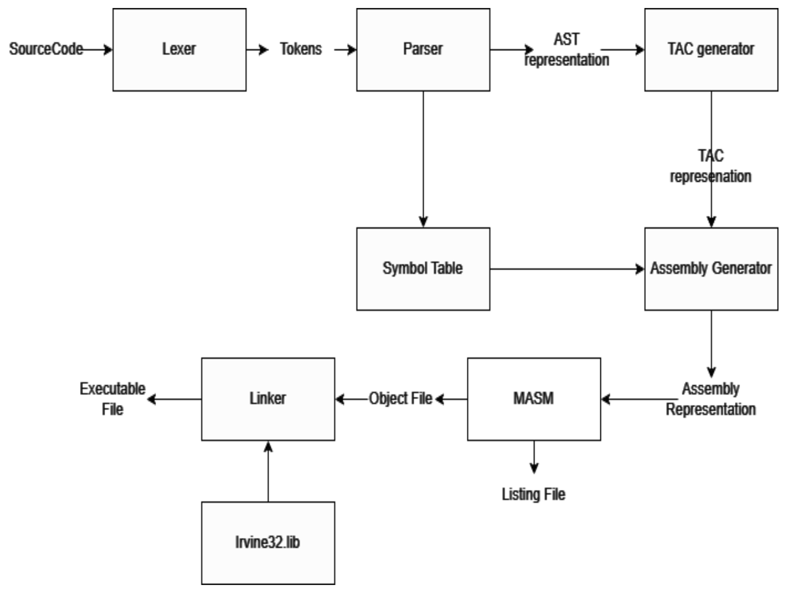


Figure 6.2: Compier Flow Diagram

# Chapter 7

# Extra Features

This chapter highlights the advanced and unique features of the compiler and language, Pookie-lang.From efficient resource allocation to enhanced functionality and code generation.

## 7.1 All Relational Operators

Our compiler supports all standard relational operators to make comparisons and logical expressions straightforward: These are pivotal in conditional statements (if, while, for) and loops, allowing complex logical evaluations.

## 7.2 Abstract Syntax Tree (AST)

The compiler generates an Abstract Syntax Tree (AST) for the source code during the parsing phase: **Purpose:** Represents the hierarchical structure of the program for optimized processing and error checking. **Advantage:** Allows the compiler to implement advanced features and scalable.

## 7.3 Input and Output Statements

The language supports: **Input:** Use input() for user-provided data during runtime. **Output:** Use print() to display results.

## 7.4 Efficient Register Allocation with Limited Resources and System Stack usage

With only four registers, EAX,EBX,ECX,EDX our compiler effeciently use them and use stack in cases where registers are not available.

## 7.5 Loop Constructs

The language provides robust support for:

**For Loops:** Ideal for counter-based iterations. **While Loops:** Suitable for condition-based iterations.

## 7.6 Functions

The language supports: **Function Definition and Calls:** Functions can be defined with parameters and can be called for code reusablity.

## 7.7 Variable Scope

The language define scope for each function and the main code. They are treated separately.

## 7.8 Efficient Use of Symbol Table

**Symbol Table:** Tracks variables, functions, and their scopes efficiently. **Optimization:** Reduces lookup time during code generation and type checking.

## 7.9 Integration with Irvine32 Library

To enhance functionality, the language links to the Irvine32 Library, providing:

- System-level I/O operations.
- Easy memory access.

## 7.10 Listing File

The compiler generates Listing file includes the source code, memory layout, and assembly instructions.

## 7.11 Object file and Executable file

The compiler generates:

- **Object File (.obj):** Intermediate binary containing machine instructions.
- **Executable File (.exe):** Final runnable file, linked to necessary libraries for execution.

## 7.12 Pookie-Lang Syntax

The languae use sarcastic syntax, the syntax mapping is as follows

- **int − > mera_poo**
- **if − > kya_yeh_pookie**
- **else − > wrna_yeh_poo**
- **for − > pookie_ghumega**
- **while − > pookie_fas_gaya**
- **print − > dikhao_pookie**
- **input − > pochu_pookie**
- **function defination − > mera_batman**
- **function call − > aja_batman**