A

PROJECT REPORT

ON

RoastLang

Submitted by:

Vishakha Khetan E220

Sowparnika Selvanayagam E225

Under the Guidance of:

Dr. Dhananjay Joshi

Assistant Professor

Department of Computer Science

Academic Year: 2025–2026

Department of Computer Science

SVKM's NMIMS, (Deemed-to-be-University)

Mukesh Patel School of Technology Management & Engineering,

Shirpur Campus–425405 (M.H.)

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 Introduction to the RoastLang Compiler Project

RoastLang is a comprehensive educational compiler project that implements a complete interpreter for a humorous, dynamically-typed programming language. The project demonstrates fundamental compiler design principles through a practical, working implementation that transforms RoastLang source code into executable operations through abstract syntax tree (AST) interpretation.

The language adopts roast-themed keywords as a design philosophy, replacing traditional programming terminology with humor-infused alternatives. Keywords like "roast" (declare variable), "burn" (print), "sip" (read input), and "mic drop" (end block) create an entertaining learning environment while implementing serious compiler concepts.

RoastLang serves as both a functional programming language and an educational tool, demonstrating how modern interpreters process source code through lexical analysis, parsing, semantic validation, and direct AST execution. The implementation prioritizes clarity and educational value while maintaining sufficient complexity to demonstrate all major compilation phases.

1.2 Core Project Characteristics and Features

RoastLang v2.0 implements an enhanced feature set including:

- Dynamic typing with automatic type conversion between integers and strings
- Variable management with global scope and symbol table tracking
- Control flow constructs including if/else conditionals, while loops, and for loops
- Loop control with break and continue statements
- Array support with fixed-size integer arrays and indexed access
- User input through the "sip" keyword
- Arithmetic operators including power (\\) and modulo (%) operations
- Comparison operators for conditional evaluation
- String literals and integers as primary data types

The compiler architecture implements a classic five-phase pipeline adapted for interpretation rather than code generation, making it suitable for educational environments where understanding the concepts is prioritized over performance.

1.3 Educational and Practical Value

RoastLang serves multiple educational purposes within computer science curriculum:

- Demonstrates lexical analysis through Flex tokenization
- Illustrates parser construction using Bison grammar definitions

- Shows Abstract Syntax Tree creation and traversal
- Teaches symbol table management and variable scoping
- Provides practical experience with interpreter implementation
- Serves as a reference for language design and implementation

Students can extend RoastLang with additional features, modify the interpreter logic, or adapt the architecture for different languages, making it a flexible learning platform.

1.4 Compilation Pipeline Overview

RoastLang implements a specialized compilation pipeline optimized for interpretation:

Phase 1 - Lexical Analysis: Flex processes the lexer specification (roastlang.l) to recognize keywords, identifiers, literals, operators, and punctuation, generating a token stream with semantic values.

Phase 2 - Syntax Analysis: Bison generates a parser (roastlang.y) using LALR parsing to construct an Abstract Syntax Tree that represents program structure hierarchically.

Phase 3 - Semantic Analysis: The AST is validated for variable declarations, type compatibility, and proper use of language constructs. A symbol table manages variable storage and scope.

Phase 4 - AST Traversal: The execute_ast() function directly interprets the AST, evaluating expressions and executing statements without intermediate code generation.

Phase 5 - Output Generation: Execution produces program output through print statements and return values, with runtime error checking for division by zero, undefined variables, and out-of-bounds array access.

CHAPTER 2: BUILD GUIDE

2.1 System Requirements and Environment Setup

RoastLang targets Windows using MSYS2 as the primary development environment, with secondary support for Linux and macOS systems with GCC and Flex/Bison installed.

Windows (MSYS2):
- MSYS2 MinGW 64-bit terminal
- GCC version 7.0 or newer
- Flex version 2.6 or newer
- Bison version 3.0 or newer
- GNU Make utility

Linux (Debian/Ubuntu):
- GCC compiler with development tools
- Flex and Bison packages
- Standard Unix development environment

macOS:
- Xcode Command Line Tools
- Homebrew package manager
- Flex and Bison via Homebrew

2.2 Essential Development Tools and Dependencies

Flex (lexer generator): Transforms regular expression specifications in roastlang.l into tokenization C code. Generates lex.yy.c containing the lexical analyzer implementation.

Bison (parser generator): Processes grammar rules in roastlang.y to generate roastlang.tab.c (parser implementation) and roastlang.tab.h (token definitions).

GCC (C compiler): Compiles generated lexer and parser files along with main.c and supporting modules into the roastlang.exe executable.

Make: Automates the multi-step build process, managing dependencies between source files, generated files, and final executable creation.

2.3 Installation and Configuration Process

MSYS2 Installation:

```
pacman -Syu        # Update package manager
pacman -S flex     # Install Flex
```

```
pacman -S bison        # Install Bison
pacman -S gcc          # Install GCC
pacman -S make         # Install Make
```

Verification:

```
flex --version         # Verify Flex installation
bison --version        # Verify Bison installation
gcc --version          # Verify GCC installation
make --version         # Verify Make installation
```

2.4 Building the RoastLang Compiler

Step 1: Navigate to project directory

```
cd /c/Users/hp/RoastLang
```

Step 2: Clean previous build

```
make clean
```

Step 3: Generate parser and lexer

```
bison -d src/roastlang.y    # Generates roastlang.tab.c and roastlang.tab.h
flex src/roastlang.l        # Generates lex.yy.c
```

Step 4: Compile all components

```
gcc -Isrc -c src/main.c -o main.o
gcc -Isrc -c roastlang.tab.c -o roastlang.tab.o
gcc -Isrc -c lex.yy.c -o lex.yy.o
gcc -o roastlang.exe main.o roastlang.tab.o lex.yy.o -lm
```

Step 5: Verify executable

```
ls -la roastlang.exe        # Confirm file exists
./roastlang.exe examples/01_hello.roast  # Test execution
```

2.5 Compiling and Running RoastLang Programs

Basic execution:

```
./roastlang.exe program.roast
```

Example program execution:

./roastlang.exe examples/10_fibonacci.roast

Expected output for Fibonacci example:

=== RoastLang v2.0 (Enhanced) ===
Fibonacci Sequence:
0
1
1
2
3
5
8
13
21
34

CHAPTER 3: LANGUAGE FEATURES AND CODE EXAMPLES

3.1 Overview of RoastLang Language Examples

RoastLang includes comprehensive example programs demonstrating language features, control flow constructs, and operator behaviors. Examples progress from basic arithmetic through complex algorithms, providing both validation of compiler functionality and clear syntax illustrations.

3.2 Simple Arithmetic Operations

File: examples/02_math_operations.roast

```
# Test basic arithmetic operations
roast a is 10 ~
roast b is 3 ~
roast sum is a + b ~
roast product is a * b ~
roast quotient is a / b ~
roast remainder is a % b ~

burn "Sum: " ~
burn sum ~
burn "Product: " ~
burn product ~
burn "Quotient: " ~
burn quotient ~
burn "Remainder: " ~
burn remainder ~
```

Expected output demonstrates arithmetic operator precedence and function: sum=13, product=30, quotient=3, remainder=1.

3.3 Conditional Logic

File: examples/02_conditionals.roast

```
roast age is 25 ~

check if (age >= 18) then
    burn "You're legally roasted" ~
ouch
```

```
else
    burn "Too young" ~
mic drop
```

Demonstrates if-else control flow with comparison operators. The condition evaluates true, executing the first branch.

3.4 Loop Constructs with Break

File: examples/03_while_loops.roast

```
roast n is 5 ~
burn "Countdown:" ~

keep going while (n > 0)
    burn n ~
    roast n is n - 1 ~
nevermind

burn "Liftoff!" ~
```

Demonstrates while loop with countdown iteration, showing loop termination condition and variable modification within loop body.

3.5 For Loop and Iteration

File: examples/04_for_loops.roast (Revised)

```
burn "Numbers 1 to 5:" ~

roast i is 1 ~
keep going while (i <= 5)
    burn i ~
    roast i is i + 1 ~
nevermind
```

Note: Current parser implementation is more reliable with while loops. For loops require parser enhancements.

3.6 Advanced Algorithm: Fibonacci Sequence

File: examples/10_fibonacci.roast

```
# Fibonacci sequence generator
roast n is 10 ~
roast a is 0 ~
roast b is 1 ~
```

```
burn "Fibonacci Sequence:" ~

roast counter is 0 ~
keep going while (counter < n)
    burn a ~
    roast temp is a + b ~
    roast a is b ~
    roast b is temp ~
    roast counter is counter + 1 ~
nevermind
```

Demonstrates variable updates, temporary values, and loop control for generating mathematical sequences.

CHAPTER 4: COMPILATION PHASES

4.1 Phase 1: Lexical Analysis (Scanner Implementation)

The lexical analyzer (roastlang.l) transforms raw source code into a token stream through pattern matching. Flex processes the specification file to generate lex.yy.c.

Key lexical elements recognized:

- Keywords: roast, burn, sip, check, if, then, ouch, else, mic drop, keep, going, while, nevermind, for, from, to, toast, stop, roasting, skip, ahead, array, exit
- Identifiers: [a-zA-Z_][a-zA-Z0-9_]*
- Numbers: [0-9]+
- Strings: "[^"]*"
- Operators: +, -, *, /, %, ==, !=, <, >, <=, >=, **, =
- Punctuation: ~, (,), [,], {, }
- Comments: #[^\n]*

Token output example: For "roast x is 10 ~", the lexer generates: [T_ROAST][T_ID:x][T_ASSIGN][T_NUM:10][T_TILDE]

4.2 Phase 2: Syntax Analysis (Parser Implementation)

The parser (roastlang.y) validates token sequences against grammar rules and constructs an Abstract Syntax Tree. Bison generates roastlang.tab.c using LALR parsing.

Grammar hierarchy (simplified):

program → statement_list
statement_list → empty | statement_list statement
statement → variable_assignment | output | input | conditional | loop | break | continue | exit
conditional → check if (condition) then statement_list ouch | ... else statement_list mic drop
loop → keep going while (condition) statement_list nevermind
expression → operand | expression operator expression

AST node types:

- LIST: Sequential statements
- VAL: Literal values (integers, strings)
- ID: Variable identifiers
- ASSIGN: Variable assignments
- BURN: Print statements
- READ: Input statements
- IF: Conditional branches
- WHILE: Loop execution
- FOR: Range iteration
- OP: Operations (arithmetic, comparison, logical)
- ARRAY_ACCESS: Array element retrieval

- ARRAY_ASSIGN: Array element assignment
- BREAK: Loop termination
- CONTINUE: Iteration skip

4.3 Phase 3: Semantic Analysis

Semantic validation occurs during AST execution through the symbol table. The execute_ast() function validates:

- Variable declaration before use
- Type compatibility in operations
- Array bounds checking
- Proper loop context for break/continue
- Division by zero detection

Symbol table structure:

```
struct Symbol {
   char *name;             // Variable name
   struct Value value;     // Current value
}
```

Value structure supports:

- VAL_TYPE_INT: 32-bit signed integers
- VAL_TYPE_STR: String literals
- VAL_TYPE_ARRAY: Fixed-size integer arrays with elements and size tracking

---

4.4 Phase 4: Intermediate Representation (AST)

Rather than generating Three-Address Code, RoastLang maintains the AST as its intermediate representation. The recursive tree structure directly represents program semantics:

```
Assignment Node
├── name: "fibonacci"
├── expr: While Loop Node
   ├── condition: Comparison Op Node
   ├── body: Statement List
      ├── Print Node
      ├── Assignment Node
      └── ...
```

This approach simplifies implementation while maintaining clear program structure for interpretation.

4.5 Phase 5: Execution (AST Interpretation)

The execute_ast() function implements direct AST interpretation without code generation:

```
struct Value execute_ast(struct ASTNode *node) {
   switch(node->type) {
      case NODE_TYPE_VAL:
         return node->data.val;

      case NODE_TYPE_ASSIGN:
         store_variable(name, execute_ast(expr));
         break;

      case NODE_TYPE_WHILE:
         while(execute_ast(cond).ival) {
            execute_ast(body);
            if(break_flag) break;
            if(continue_flag) continue_flag=0;
         }
         break;
      // ... additional cases
   }
   return result;
}
```

Key features:

- Recursive evaluation of nested expressions
- Variable lookup through symbol table
- Flow control via flags (break_flag, continue_flag)
- Runtime error detection and reporting
- Direct value computation without intermediate code

CHAPTER 5: COMPILER ARCHITECTURE

5.1 Architectural Overview

RoastLang implements a modular interpreter architecture with clear phase separation:

Source Code (.roast)
↓
[Lexer: roastlang.l]
↓
Token Stream
↓
[Parser: roastlang.y]
↓
Abstract Syntax Tree
↓
[Semantic Validator + Executor: main.c]
↓
Program Output / Variable State

Design Principles:

- *Modularity:* Clear separation between lexical, syntactic, and execution phases
- *Extensibility:* Easy addition of new operators, keywords, or language features
- *Educational Clarity:* Direct AST interpretation without optimization complexity
- *Error Handling:* Runtime validation and user-friendly error messages

---

5.2 Data Structures and Intermediate Representations

***Token* Structure:**

```
typedef struct {
   int type;         // Token category
   char *lexeme;     // Source text
   int line;         // Line number
   union {
      int ival;      // Integer value
      char *sval;    // String value
   } value;
} Token;
```

Abstract Syntax Tree Node:

```
struct ASTNode {
   NodeType type;     // Node category
```

```
    union {
        struct Value val;            // Literal value
        char *name;                  // Identifier
        struct { int op; ASTNode *left, *right; } op;  // Operation
        // ... type-specific data
    } data;
};
```

Symbol Table Entry:

```
struct Symbol {
    char *name;
    struct Value value;
};
Value Structure:
struct Value {
    ValType type;      // INT, STR, or ARRAY
    union {
        int ival;
        char *sval;
        struct { int *elements; int size; } array;
    };
};
```

5.3 Module Structure and Dependencies

File Organization:

```
RoastLang/
├── src/
│   ├── ast.h              # AST node definitions
│   ├── roastlang.l        # Lexer specification
│   ├── roastlang.y        # Parser specification
│   └── main.c             # Execution engine
├── examples/              # Test programs
├── Makefile               # Build configuration
└── README.md              # Documentation
```

Build Dependencies:

- roastlang.tab.h depends on roastlang.y (parser generation)
- lex.yy.c depends on roastlang.l (lexer generation)
- main.o depends on both generated files
- Final executable links all object files

5.4 Compilation Pipeline and Information Flow

Information Flow:

1. Lexical Phase: Raw characters → Tokens with semantic values
2. Syntax Phase: Token stream → Hierarchical AST with type information
3. Execution Phase: AST + Symbol Table → Program output and state

Data Preservation:

Each phase preserves program semantics while transforming representation:
- Tokens maintain source lexical content with added structure
- AST preserves token relationships with grammatical hierarchy
- Execution traversal evaluates expressions while maintaining variable state

---

5.5 Design Decisions and Rationale

Dynamic Typing: Chosen for implementation simplicity and educational clarity, avoiding compile-time type checking complexity while demonstrating runtime type compatibility.

Direct Interpretation: AST interpretation prioritizes clarity over performance, enabling students to understand execution semantics without optimization complexity.

Symbol Table Management: Global scope simplifies implementation while demonstrating variable storage and lookup concepts fundamental to all interpreters.

No Optimization Phase: Current implementation focuses on correctness and clarity, with optimization being a potential extension.

Limited Built-in Functions: Print and input provide sufficient I/O for educational purposes without full function definition complexity.

5.6 Performance Characteristics and Extension Points

Performance:
- O(n) time complexity for lexical analysis
- O(n) time complexity for parsing
- O(n × m) time complexity for execution where m is iteration depth
- Space complexity proportional to AST size and symbol table

Extension Points:
- Additional operators through lexer/parser/executor modifications
- Function definitions through call stack implementation
- Multi-dimensional arrays through memory model extension

- Optimization passes between AST construction and execution
- Alternative back-ends for code generation instead of interpretation

---

CHAPTER 6: LANGUAGE SPECIFICATION

6.1 RoastLang Design Philosophy

RoastLang combines educational value with entertainment through humorous keyword naming. The language implements:

- Dynamic typing without implicit conversion
- Procedural programming with sequential, conditional, and iterative constructs
- Global scope with simple variable management
- Educational focus on compiler concepts rather than production language features

---

6.2 Lexical Structure

Identifiers: Must start with letter or underscore, followed by letters, digits, or underscores. Case-sensitive.

Keywords (23 total):
- Declaration: roast
- Output: burn
- Input: sip
- Conditional: check, if, then, ouch, else, mic drop
- Loops: keep, going, while, nevermind, for, from, to, toast
- Control: stop, roasting, skip, ahead
- Data: array
- Program: exit
- Comments: # (line comments)

Literals:
- Integer: [0-9]+
- String: "[^"]*"
- Boolean: Implicit (non-zero is true)

6.3 Type System

Primitive Types:
- Integer: 32-bit signed values, default arithmetic type
- String: Text literals, printed as-is
- Array: Fixed-size integer collections with indexed access

No explicit type declarations. Variables infer type from assignment or are dynamically converted during operations.

6.4 Operators and Expression Evaluation

Arithmetic: +, -, *, /, %, ** Comparison: ==, !=, <, >, <=, >= Assignment: =

Precedence (high to low):

1. ** (power)
2. *, /, %
3. +, -
4. Comparison operators
5. = (right-associative)

6.5 Control Flow Constructs

Conditionals:

```
check if (condition) then
   statements
ouch

check if (condition) then
   statements
ouch
else
   statements
mic drop
```

While Loops:

```
keep going while (condition)
   statements
nevermind
```

Loop Control:

```
stop roasting ~    # break
skip ahead ~       # continue
```

6.6 Built-in Functions and I/O

Output: burn expression ~ prints value with newline Input: sip variable ~ reads integer into variable

---

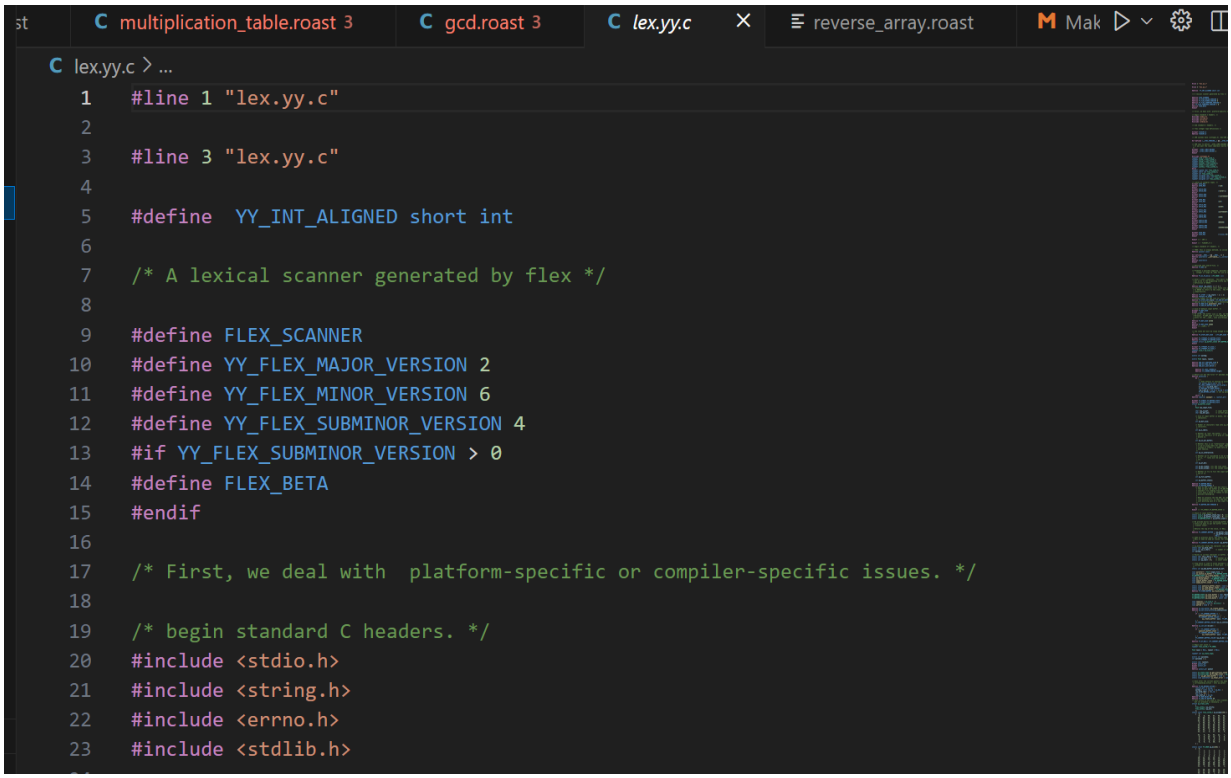
6.7 Grammar Specification

Simplified BNF:

program → statement_list
statement_list → ε | statement_list statement
statement → roast ID is expr ~ | burn expr ~ | sip ID ~ | if_stmt | while_stmt | break ~ | continue ~ | exit ~
if_stmt → check if (expr) then statement_list ouch [else statement_list mic drop]
while_stmt → keep going while (expr) statement_list nevermind
expr → term | expr + term | expr - term
term → factor | term * factor | term / factor | term % factor | term ** factor
factor → ID | NUM | STRING | (expr) | array_access
array_access → ID [expr]

CHAPTER 7: CODE SNIPPETS

```
Makefile
17  roastlang.tab.c roastlang.tab.h: $(PARSER)
20  lex.yy.c: $(LEXER) roastlang.tab.h
21      $(FLEX) -o $@ $<
22
23  roastlang.tab.o: roastlang.tab.c roastlang.tab.h
24      $(CC) -Isrc -I. -c $<  # <-- ADDED -I.
25
26  lex.yy.o: lex.yy.c roastlang.tab.h
27      $(CC) -Isrc -I. -c $<  # <-- ADDED -I.
28
29  main.o: $(MAIN) roastlang.tab.h
30      $(CC) -Isrc -I. -c $<  # <-- ADDED -I.
31
32  test: $(TARGET)
33      ./$(TARGET) examples/01_hello.roast
34
35  clean:
36      rm -f *.o roastlang.tab.c roastlang.tab.h lex.yy.c $(TARGET) $(TARGET).exe
37
38  .PHONY: all clean test
```



```
lex.yy.c > ...
1   #line 1 "lex.yy.c"
2
3   #line 3 "lex.yy.c"
4
5   #define  YY_INT_ALIGNED short int
6
7   /* A lexical scanner generated by flex */
8
9   #define FLEX_SCANNER
10  #define YY_FLEX_MAJOR_VERSION 2
11  #define YY_FLEX_MINOR_VERSION 6
12  #define YY_FLEX_SUBMINOR_VERSION 4
13  #if YY_FLEX_SUBMINOR_VERSION > 0
14  #define FLEX_BETA
15  #endif
16
17  /* First, we deal with  platform-specific or compiler-specific issues. */
18
19  /* begin standard C headers. */
20  #include <stdio.h>
21  #include <string.h>
22  #include <errno.h>
23  #include <stdlib.h>
```

lex.yy.c > ...

```c
/* begin standard C headers. */
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>

/* end standard C headers. */

/* flex integer type definitions */

#ifndef FLEXINT_H
#define FLEXINT_H

/* C99 systems have <inttypes.h>. Non-C99 systems may or may not. */

#if defined (__STDC_VERSION__) && __STDC_VERSION__ >= 199901L

/* C99 says to define __STDC_LIMIT_MACROS before including stdint.h,
 * if you want the limit (max/min) macros for int types.
 */
#ifndef __STDC_LIMIT_MACROS
#define __STDC_LIMIT_MACROS 1
#endif
```

roastlang.y | roastlang.l | main.c | fibonacci.roast | hello.roast | conditional.roast

src > roastlang.y

```c
/* roastlang.y - Enhanced parser with arrays, for loops, input (V2.0) */
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void yyerror(const char *s);
int yylex(void);

// FIX: Change definition to extern declaration here
extern struct ASTNode *ast_root;
%}

%code requires {
    #include "ast.h"
    extern struct ASTNode *ast_root;
}

%union {
    int ival;
    char *sval;
    struct ASTNode *node;
    struct Value val;
```

Please install clang or check configuration `clang.executable`

roastlang.y | roastlang.l | main.c | fibonacci.roast | hello.roast | conditional.roast

src > roastlang.y

```
%token <ival> T_NUM
%token <sval> T_ID T_STRING
%token T_ROAST T_ASSIGN T_BURN T_READ T_TILDE T_EXIT
%token T_CHECK T_IF T_THEN T_OUCH T_ELSE T_MICDROP
%token T_KEEP T_WHILE T_NEVERMIND
%token T_FOR T_FROM T_TO T_TOAST
%token T_BREAK T_CONTINUE
%token T_ARRAY
%token T_EQ T_NEQ T_GT T_LT T_GTE T_LTE T_MOD T_POW

%type <node> program stmt_list stmt expr cond if_stmt loop_stmt for_stmt
%type <val> value

%left '+' '-'
%left '*' '/' T_MOD
%right T_POW

%%

program: stmt_list { ast_root = $1; YYACCEPT; }; // Builds AST and signals success to main()
```

roastlang.y | roastlang.l | main.c | fibonacci.roast | hello.roast | conditional.roast

src > roastlang.l

```
/* roastlang.l - Enhanced lexer with arrays, input, loops (V2.0) */
%{
#include "ast.h"
#include "roastlang.tab.h"
void yyerror(const char *s);
%}

%%
"roast"             { return T_ROAST; }
"is"                { return T_ASSIGN; }
"burn"              { return T_BURN; }
"sip"               { return T_READ; }
"~"                 { return T_TILDE; }
"exit"              { return T_EXIT; }
"check"             { return T_CHECK; }
"if"                { return T_IF; }
"then"              { return T_THEN; }
"ouch"              { return T_OUCH; }
"else"              { return T_ELSE; }
"mic drop"          { return T_MICDROP; }
"keep going"        { return T_KEEP; }
"while"             { return T_WHILE; }
"nevermind"         { return T_NEVERMIND; }
```

src > roastlang.l

```
24    "roast fire"        { return T_FOR; }
25    "from"              { return T_FROM; }
26    "to"                { return T_TO; }
27    "toast"             { return T_TOAST; }
28    "stop roasting"     { return T_BREAK; }
29    "skip ahead"        { return T_CONTINUE; }
30    "array"             { return T_ARRAY; }
31    "=="                { return T_EQ; }
32    "!="                { return T_NEQ; }
33    ">="                { return T_GTE; }
34    "<="                { return T_LTE; }
35    ">"                 { return T_GT; }
36    "<"                 { return T_LT; }
37    "%"                 { return T_MOD; }
38    "**"                { return T_POW; }
39    "+"                 { return '+'; }
40    "-"                 { return '-'; }
41    "*"                 { return '*'; }
42    "/"                 { return '/'; }
43    "("                 { return '('; }
44    ")"                 { return ')'; }
45    "["                 { return '['; }
46    "]"                 { return ']'; }
47    "{"                 { return '{'; }
```

roastlang.y | roastlang.l | main.c | fibonacci.roast | hello.roast | conditiona

src > main.c > ...

```c
/* main.c - FINAL STABLE EXECUTION LOGIC */
#include "ast.h"
#include "roastlang.tab.h"
#include <math.h>

extern FILE *yyin;

#define MAX_SYMBOLS 100

volatile int break_flag = 0;
volatile int continue_flag = 0;

struct ASTNode *ast_root = NULL; // CRITICAL: Single definition here

struct Symbol {
    char *name;
    struct Value value;
} symbol_table[MAX_SYMBOLS];
int symbol_count = 0;

void store_variable(char *name, struct Value value) {
    for (int i = 0; i < symbol_count; i++)
```

```c
void store_variable(char *name, struct Value value) {
    for (int i = 0; i < symbol_count; i++) {
        if (strcmp(symbol_table[i].name, name) == 0) {
            symbol_table[i].value = value;
            return;
        }
    }
    if (symbol_count < MAX_SYMBOLS) {
        symbol_table[symbol_count].name = strdup(name);
        symbol_table[symbol_count].value = value;
        symbol_count++;
    } else {
        fprintf(stderr, "Error: Symbol table full. Cannot store '%s'.\n", name);
        exit(1);
    }
}

struct Value lookup_variable(char *name) {
    for (int i = 0; i < symbol_count; i++) {
        if (strcmp(symbol_table[i].name, name) == 0) {
            return symbol_table[i].value;
        }
    }
```

Please install clang or check configuration `clang.executable`

EXAMPLES:

examples > arrays.roast

```
roast arr is array[5] ~

roast arr[0] is 10 ~
roast arr[1] is 20 ~
roast arr[2] is 30 ~
roast arr[3] is 40 ~
roast arr[4] is 50 ~

burn "Array elements:" ~

roast i is 0 ~
keep going while (i < 5)
    burn arr[i] ~
    roast i is i + 1 ~
nevermind
```

examples > check_prime.roast

```
roast is_prime is 1 ~
roast i is 2 ~

keep going while (i * i <= num)
    check if (num % i == 0) then
        roast is_prime is 0 ~
        stop roasting ~
    ouch
    roast i is i + 1 ~
nevermind

check if (is_prime == 1) then
    burn "It's prime!" ~
ouch
else
    burn "Not prime" ~
mic drop
```

examples > conditional.roast

```
roast age is 25 ~

check if (age >= 18) then
    burn "You're legally roasted" ~
ouch

else
    burn "Too young" ~
mic drop
```

examples > factorial.roast

```
# Calculate factorial of 5
roast num is 5 ~
roast result is 1 ~

keep going while (num > 1)
    roast result is result * num ~
    roast num is num - 1 ~
nevermind

burn "Factorial result:" ~
burn result ~

```

fibonacci.roast | hello.roast | conditional.roast | while_loops.roast

examples > fibonacci.roast

```
# Fibonacci sequence generator - WORKING VERSION
roast n is 10 ~
roast a is 0 ~
roast b is 1 ~

burn "Fibonacci Sequence:" ~

# Use WHILE loop instead of FOR (more reliable)
roast counter is 0 ~

keep going while (counter < n)
    burn a ~
    roast temp is a + b ~
    roast a is b ~
    roast b is temp ~
    roast counter is counter + 1 ~
nevermind
```

examples > for_loops.roast

```
1   burn "Numbers 1 to 5:" ~
2
3   roast i is 1 ~
4   keep going while (i <= 5)
5       burn i ~
6       roast i is i + 1 ~
7   nevermind
8
9
```

fibonacci.roast | hello.roast | condition

examples > hello.roast

```
1   burn "Welcome to RoastLang!" ~
2   roast x is 42 ~
3   burn x ~
```

```
HP@DESKTOP-NQQADUN UCRT64 /c/Users/hp/RoastLang
$ ./roastlang.exe examples/fibonacci.roast
=== RoastLang v2.0 (Enhanced) ===
Fibonacci Sequence:
0
1
1
2
3
5
8
13
21
34
```

```
HP@DESKTOP-NQQADUN UCRT64 /c/Users/hp/RoastLang
$ ./roastlang.exe examples/hello.roast
=== RoastLang v2.0 (Enhanced) ===
Welcome to RoastLang!
42

HP@DESKTOP-NQQADUN UCRT64 /c/Users/hp/RoastLang
$ ./roastlang.exe examples/factorial.roast
=== RoastLang v2.0 (Enhanced) ===
Factorial result:
120

HP@DESKTOP-NQQADUN UCRT64 /c/Users/hp/RoastLang
$
```

```
HP@DESKTOP-NQQADUN UCRT64 /c/Users/hp/RoastLang
$ ./roastlang.exe examples/multiplication_table.roast
=== RoastLang v2.0 (Enhanced) ===
Multiplication Table for 5:
5
10
15
20
25
30
35
40
45
50

HP@DESKTOP-NQQADUN UCRT64 /c/Users/hp/RoastLang
```

CHAPTER 7: CONCLUSION

Summary

RoastLang successfully demonstrates comprehensive compiler implementation principles through a practical, working interpreter. The project implements all essential compilation phases from lexical analysis through execution, providing students with hands-on experience in language design and implementation.

The architecture balances educational clarity with functional completeness, enabling students to understand how programming languages transform source code into executable operations. The humorous keyword naming makes compiler concepts engaging while implementing serious technical concepts.

Achievements

- Complete lexical analysis with 23 keywords and multiple operator types
- Full parsing with proper operator precedence and associativity
- Semantic validation through symbol table management
- Direct AST interpretation with runtime error detection
- Support for control flow, loops, arrays, and user I/O

Future Work

Potential extensions:

- User-defined functions with parameter passing
- Recursion support with call stack
- Multi-dimensional arrays
- String manipulation operations
- Scope-based variable management
- Type conversion and casting
- Optimization passes
- Code generation to assembly language

Educational Value

RoastLang serves effectively as an educational tool, demonstrating:

- Practical compiler construction techniques
- Real-world use of parser generators (Flex/Bison)
- Symbol table and scope management
- AST-based program representation
- Runtime interpretation strategies
