Khulna University of Engineering & Technology

Department of Computer Science & Engineering

Khulna-9203, Bangladesh

Report on CSE 3112: Compiler Design Laboratory Project

Date of Submission : January 13, 2025
Topic: CSE 3112 Laboratory Project

Submitted By:	***Submitted To:***
Name : Md. Sakibur Rahman **Roll:** 2007007 **Section:** A **Batch:** 2k20	Nazia Jahan Khan Chowdhury Assistant Professor Department of Computer Science & Engineering, KUET Dipannita Biswas Lecturer Department of Computer Science & Engineering, KUET

Objective :

i. To design and implement the front-end of a compiler using Flex and Bison.
ii. To perform lexical analysis for tokenizing input source code.
iii. To validate syntax using grammar rules and parsers.
iv. To integrate Flex and Bison for seamless compilation processes.
v. To implement error detection and recovery mechanisms.
vi. To create and manage a symbol table for semantic analysis.

Introduction

Compilers are the bridge between human-readable code and machine execution. In this project, we explore the basics of compiler design using **Flex** for lexical analysis and **Bison** for syntax analysis. The focus is on breaking code into tokens, validating its structure, and managing errors effectively. By integrating these tools, we'll create a functional front-end for a compiler. This hands-on approach provides insight into how programming languages work behind the scenes.

Lexical Analysis

Category	**Token**	**Pattern**	**Example**
Identifiers	ID	[_a-zA-Z][_a-zA-Z0-9]*	variable1, _function
Headers	HEADER	[u][s][i][n][g][ ][a-z]+.[h]	using myfile.h
Numbers	NUM	[-]?[0-9][0-9]*[.]?[0-9]*	123, -456.78
Characters	CH	@[a-zA-Z]@	@c@
Strings	STRING_LITERAL	\\"[^\\"]*\\"	"Hello, World!"
Basic Keywords	MAIN, RETURN	"main()", "return"	main(), return
	STRING, INT	"string", "int"	string, int
	CHAR, FLOAT	"char", "float"	char, float
Control Flow	IF, ELIF	"if", "elif"	if, elif
	ELSE	"otherwise"	otherwise
	SWITCH, CASE	"switch", "state"	switch, state
	DEFAULT	"complementary"	complementary
Loops	FROM, TO	"from", "to"	from, to
	INC, DEC	"inc", "dec"	inc, dec
	MAX, MIN	"max", "min"	max, min
Functions	SQRT, ABS	"sqrt", "abs"	sqrt, abs
	LOG, SIN	"log", "sin"	log, sin
	COS, TAN	"cos", "tan"	cos, tan
	POWER, FACTO	"power", "facto"	power, facto
	PRIME	"checkprime"	checkprime
Data Structures	DICT, GET	"dict", "get"	dict, get
	SET, CONCAT	"set", "concat"	set, concat
	COPY, SIZE	"copy", "size"	copy, size
	COMPARE	"compare"	compare

	STACK, QUEUE	"stack", "queue"	stack, queue
Operators	STRICT_EQUAL	===	===
	STRICT_NEQ	!==	!==
	EQUAL, NOTEQUAL	==, !=	==, !=
	GT, LT	>, <	>, <
	AND, OR	&&, `	
Arithmetic	PLUS, MINUS	+, -	+, -
	MUL, DIV	*, /	*, /
	MOD, POW	%, ^	%, ^
Special Symbols	(,)	(,)	(,)
	{, }	{, }	{, }
	COMMA, SEMICOLON	,, ;	,, ;
Comments	-	#.*	# This is a comment

Context Free Grammar

Non-Terminals:

- program
- main
- function_list
- function
- return_statement
- code
- function_call
- power_code
- factorial_code
- prime_code
- max_code
- min_code
- print_code
- read_code
- switch_code
- case_code
- casenum_code
- default_code
- for_code
- while_code
- condition
- if_statement
- elif_list
- expression
- e
- f
- t
- bool_expression
- declaration
- init_list
- init_item
- assignment
- TYPE
- dict_operation
- stack_operation
- queue_operati

Terminals:

- MAIN
- INT
- CHAR
- FLOAT
- POWER
- FACTO
- PRIME
- READ
- PRINT
- IF
- ELIF
- ELSE
- SWITCH
- CASE
- DEFAULT
- FROM
- TO
- INC
- DEC
- MAX
- MIN
- ID
- NUM
- PLUS
- MINUS
- MUL
- DIV
- EQUAL
- NOTEQUAL
- GT
- GOE
- LT
- LOE
- STRING
- STRING_LITERAL
- FUNCTION

- RETURN
- MOD
- POW
- SQRT
- ABS
- LOG
- SIN
- COS
- TAN
- INCREMENT
- DECREMENT
- AND
- OR
- NOT
- NEQ
- STRICT_EQUAL
- STRICT_NEQ
- WHILE
- DICT
- GET
- SET
- CONCAT
- COPY
- SIZE
- COMPARE
- STACK
- PUSH
- POP
- TOP
- ISEMPTY
- STACKSIZE
- QUEUE
- ENQUEUE
- DEQUEUE
- FRONT
- REAR
- QSIZE
- QEMPTY

CFG Rules:

Program	program -> function_list main | main
Main	main -> MAIN '{' code '}' | MAIN '(' ')' '{' code '}'
Function List	function_list -> function_list function | function
Function	function -> FUNCTION ID '(' ')' '{' code return_statement '}'
Return Statemen	return_statement -> RETURN expression ';'
Code	code -> declaration code | assignment code | dict_operation code | condition code | for_code code | while_code code | switch_code code | print_code code | read_code code | power_code code | factorial_code code | prime_code code | min_code code | max_code code | function_call code | stack_operation code

	| queue_operation code | ε
Function Call	function_call -> ID '(' ')' ';'
Power Function	power_code -> POWER '(' NUM ',' NUM ')' ';'
Factorial Function	factorial_code -> FACTO '(' NUM ')' ';'
Prime Function	prime_code -> PRIME '(' NUM ')' ';'
Max Function	max_code -> MAX '(' ID ',' ID ')' ';'
Min Function	min_code -> MIN '(' ID ',' ID ')' ';'
Print Function	print_code -> PRINT '(' ID ')' ';' | PRINT '(' STRING_LITERAL ')' ';'
Read Function	read_code -> READ '(' ID ')' ';'
Switch Statement	switch_code -> SWITCH '(' ID ')' '{' case_code '}' case_code -> casenum_code default_code casenum_code -> CASE NUM '{' code '}' casenum_code | ε default_code -> DEFAULT '{' code '}'
For Loop	for_code -> FROM ID TO NUM INC NUM '{' code '}' | FROM ID TO NUM DEC NUM '{' code '}'
While Loop	while_code -> WHILE '(' ID LOE NUM ')' '{' code '}'
Conditionals	condition -> if_statement if_statement -> IF '(' bool_expression ')' '{' code '}' | IF '(' bool_expression ')' '{' code '}' ELSE '{' code '}' | IF '(' bool_expression ')' '{' code '}' elif_list ELSE '{' code '}' | IF '(' bool_expression ')' '{' code '}' elif_list elif_list -> elif_list ELIF '(' bool_expression ')' '{' code '}' | ELIF '(' bool_expression ')' '{' code '}'

Expressions	expression -> e e -> e PLUS f | e MINUS f | f f -> f MUL t | f DIV t | f MOD t | t POW f | t t -> '(' e ')' | NUM | ID | SQRT '(' e ')' | ABS '(' e ')' | LOG '(' e ')' | SIN '(' e ')' | COS '(' e ')' | TAN '(' e ')'
Boolean Expressions	bool_expression -> bool_expression AND bool_expression | bool_expression OR bool_expression | NOT bool_expression | '(' bool_expression ')' | expression GT expression | expression LT expression | expression GOE expression | expression LOE expression | expression EQUAL expression | expression STRICT_EQUAL expression | expression STRICT_NEQ expression | expression NOTEQUAL expression
Declarations	declaration -> TYPE init_list ';' init_list -> init_list ',' init_item | init_item init_item -> ID | ID '=' expression | ID '=' STRING_LITERAL
Assignments	assignment -> ID '=' expression ';' | ID INCREMENT ';' | ID DECREMENT ';'

	| ID '=' STRING_LITERAL ';'
Type	TYPE -> INT | FLOAT | CHAR | STRING | DICT | STACK | QUEUE
Dictionary Operations	dict_operation -> SET '(' ID ',' NUM ',' expression ')' ';' | GET '(' ID ',' NUM ')' ';' | CONCAT '(' ID ',' ID ')' ';' | COPY '(' ID ',' ID ')' ';' | SIZE '(' ID ')' ';' | COMPARE '(' ID ',' ID ')' ';'
Stack Operations	stack_operation -> PUSH '(' ID ',' expression ')' ';' | POP '(' ID ')' ';' | TOP '(' ID ')' ';' | ISEMPTY '(' ID ')' ';' | STACKSIZE '(' ID ')' ';'
Queue Operations	queue_operation -> ENQUEUE '(' ID ',' expression ')' ';' | DEQUEUE '(' ID ')' ';' | FRONT '(' ID ')' ';' | REAR '(' ID ')' ';' | QEMPTY '(' ID ')' ';' | QSIZE '(' ID ')' ';'

Discussion:

This project represents a robust implementation of a custom programming language parser and interpreter, showcasing an in-depth understanding of compiler design principles. By utilizing **Lex** for tokenization and **Yacc** for grammar parsing, the system handles variable declarations, mathematical operations, data structure manipulations, and control structures such as loops and conditionals. The integration of advanced features like **stacks**, **queues**, and **dictionaries** demonstrates versatility and real-world applicability, enabling the simulation of various programming paradigms. Furthermore, the project ensures compatibility with different data types while offering custom operations like mathematical functions, string manipulation, and comparisons. Type checking and error handling mechanisms are well-integrated to maintain language integrity and provide informative feedback to users.

Conclusion:

This parser and interpreter project successfully encapsulates the complexities of building a custom programming language, offering a comprehensive set of features for users to explore programming concepts. It highlights the effectiveness of modular design through its structured handling of variables, functions, and control logic. By implementing real-world data structures and operations, the project achieves practical utility and serves as an excellent educational tool for understanding language parsing. Future expansions could include enhanced syntax support, optimizations for runtime efficiency, and integration with graphical tools for visualizing code execution. Overall, this work stands as a testament to the synergy between theoretical knowledge and practical implementation in computer science.

Github Link: https://github.com/SakiburRahman07/CSE-3212-Compiler-Project-