COMSATS UNIVERSITY

ATTOCK CAMPUS

NAME: SARA BIBI

REG# (SP21-BCS-033) SUBMITTED

TO:

SIR BILAL HAIDER DATE:

03,JAN , 2025

TOPIC

MINI COMPILER

Q1: Briefly Explain Your Mini Compiler?

Ans: Phase 1: Source Code Input

Description: This is the initial phase where the user provides the C source code to be compiled. The source code is read by the compiler and prepared for the next phase. **Your Compiler**: The user writes the C code and inputs it into the compiler. This code can include definitions, declarations, and instructions written in C.

Phase 2: Lexical Analysis

Description: This phase uses Lex to perform lexical analysis. The source code is scanned to generate tokens, which are the smallest units of meaning, such as keywords, identifiers, literals, and operators. Whitespace and comments are ignored. **Your Compiler**: Lex is used to tokenize the source code. For example, the code `int main()` is converted into tokens: `int`, `main`, `(`, `)`.

Phase 3: Syntax Analysis

Description: In this phase, Yacc performs syntax analysis by parsing the tokens according to grammar rules to build an Abstract Syntax Tree (AST). This tree represents the syntactic structure of the source code. **Example**: Parsing `int main()` to create an AST with `int` as the return type and `main` as the function name. **Your Compiler**: Yacc takes the tokens from Lex and constructs the AST. For example, the `int main()` declaration is parsed to ensure it follows the C syntax rules and an AST is created.

Phase 4: Semantic Analysis

Description: This phase checks for semantic errors such as type mismatches, undeclared variables, and scope resolution. It ensures that the syntax is meaningful. **Example**: Ensuring that a variable is declared before being used. **Your Compiler**: The semantic analysis phase checks that all identifiers are declared before use, types match in assignments, function calls use the correct number and types of arguments, and so on. For example, it ensures variables are declared in the proper scope.

Phase 5: Intermediate Code Generation

Description: The compiler generates an intermediate code representation, which is easier to optimize and translate into machine code. This code is a low-level representation of the source code. **Example**: Generating three-address code for expressions like `a = b + c`. **Your Compiler**: Your compiler converts the AST into an intermediate representation. This might involve translating a statement like `a = b + c` into a series of three-address code instructions.

Phase 6: Optimization

Description: Optimizations are applied to the intermediate code to improve efficiency. This includes constant folding, dead code elimination, and loop optimizations. **Example**: Evaluating constant expressions at compile time (e.g., `2 + 3` becomes `5`). **Your Compiler**: The intermediate code is optimized to remove redundant instructions, evaluate constant expressions at compile-time, eliminate dead code (code that is never executed), and optimize loops.

Phase 7: Code Generation

Description: The optimized intermediate code is translated into machine code. This machine code is specific to the target architecture and is what the processor executes. **Example**: Converting intermediate code instructions into assembly language. **Your Compiler**: The optimized intermediate code is converted into the target

machine's assembly language or machine code. For example, an expression like `a = b + c` is translated into the appropriate assembly instructions.

Phase 8: Assembly and Linking

Description: The machine code is assembled into an object file, and the linker combines multiple object files into a single executable. This phase resolves function calls and references between files. **Example**: Linking `main.o` with `stdio.o` to create the final executable. **Your Compiler**: The assembly code generated in the previous phase is converted into object files. The linker then resolves external references and combines these object files to create the final executable binary.

Phase 9: Error Handling

Description: Throughout the compilation process, the compiler detects and reports errors. This includes lexical, syntax, and semantic errors. Errors are reported with line numbers and descriptions. **Example**: Reporting a syntax error for a missing semicolon. **Your Compiler**: The compiler continuously checks for errors at each phase. Lexical errors are caught by the lexer, syntax errors by the parser, and semantic errors during semantic analysis. Detailed error messages with line numbers help the programmer correct their code.

Phase 10: Executable Output

Description: The final phase produces the executable file. This file can be run on the target machine, and it performs the operations specified in the original source code. **Example**: Generating an executable `main.exe` from `main.c`. **Your Compiler**: The linked object files result in an executable file, like `main.exe`, which contains the machine code ready to be executed by the processor.

Q3: Explain 2 Code Functions of Your Mini Compiler

1. **Lexical Analyzer (`yylex`)**: This function, generated by Lex, scans the source code and breaks it into tokens. It removes whitespace and comments and classifies tokens into categories such as keywords, identifiers, literals, and operators.
2. **Parser (`yyparse`)**: This function, generated by Yacc, parses the sequence of tokens produced by the lexical analyzer. It constructs the abstract syntax tree (AST) and ensures the code conforms to the grammar rules defined in the Yacc file.

Q4: Optimization in Your Mini Compiler

Optimizations improve the intermediate code to enhance its efficiency:

- **Constant Folding**: Evaluates constant expressions at compile-time, reducing computation during runtime.
- **Dead Code Elimination**: Removes code that will never be executed, cleaning up the code and improving performance.

Q5: Input/output

- **Input**: The mini compiler accepts C source code.
- **Output**: It produces an executable machine code file.

Q6: Error Production Strategies

Error handling in the mini compiler includes:

- **Lexical Errors**: Detected by the lexical analyzer and reported with line numbers for easy debugging.
- **Syntax Errors**: Detected by the parser using error productions and synchronization tokens to recover and continue parsing. Errors are reported with line numbers.
- **Semantic Errors**: Detected during semantic analysis, such as type mismatches or undefined variables, and reported to the user with line numbers and error descriptions.

Q7: Test Cases

Four test cases to test your mini compiler might include:

1. **Simple Arithmetic**: Testing basic arithmetic operations (e.g., addition, subtraction).
2. **Control Structures**: Testing if-else statements, while loops, and for loops.

3. **Functions**: Testing function definitions, calls, and return statements.
4. **Error Handling**: Testing the detection and reporting of lexical, syntax, and semantic errors (e.g., missing semicolons, type mismatches).

Q8: RES and CFGS Regular Expressions (REs)

Regular expressions (REs) are patterns used in lexical analysis to match sequences of characters in the source code. In your mini C compiler, REs are defined in the Lex file to identify various tokens such as keywords, identifiers, literals, operators, and special characters. Here are the REs used:

- **Keywords**: These are reserved words in the C language, such as `int`, `float`, `if`, `else`, etc. The RE for keywords is: regex

  `int|float|if|else|while|for|return|void|char|printf|scanf` This pattern matches any of the specified keywords.

- **Identifiers**: Identifiers are names for variables, functions, and other entities. The RE for identifiers is:

  regex

  `[a-zA-Z_][a-zA-Z0-9_]*`

This pattern matches any sequence that starts with a letter or underscore, followed by any combination of letters, digits, or underscores.

- **Integer Literals**: Integer literals are sequences of digits. The RE for integer literals is:

  regex

  ```
  [0-   9]+
  ```

  This pattern matches any sequence of one or more digits.

- **Floating-Point Literals**: Floating-point literals represent decimal numbers. The RE for floating-point literals is:

  regex

  ```
  [0-   9]*\.[0-9]+
  ```

  This pattern matches any sequence with an optional integer part, followed by a decimal point and one or more digits.

- **Operators**: Operators are symbols that represent operations like addition, subtraction, etc. The RE for operators is:

  regex

  ```
  \+|\-|\*|\/|\%|\=\=|\!\=|\<|\>|\<\=|\>\=|\|\||\&\&
  ```

  This pattern matches arithmetic, relational, and logical operators.

- **Special Characters**: Special characters include punctuation and delimiters like semicolons, commas, and parentheses. The RE for special characters is: regex

  ```
  ;|,|\(|\)|\{|\}|
  ```

CFGS

Production Rules	Grammar		
Begin	begin : external_declaration \	begin external_declaration \	Define begin
Primary Expression	primary_expression : IDENTIFIER \	CONSTANT \	STRING_LITERA \
Define	Define : DEFINE		
Postfix Expression	postfix_expression : primary_expression \	postfix_expression '[' expression ']' \	...
Argument Expression List	argument_expression_list : assignment_expression \	argument_expression_list '' ...	
Unary Expression	unary_expression : postfix_expression \	INC_OP unary_expression \	...
Unary Operator	unary_operator : '&' \	'*' \	'+' \

Cast Expression	cast_expression : unary_expression |	'(' type_name ')' cast_expression
Multiplicative Expression	multiplicative_expression : cast_expression |	multiplicative_expression '*' ...
Additive Expression	additive_expression : multiplicative_expression |	additive_expression '+' ...
Shift Expression	shift_expression : additive_expression |	shift_expression LEFT_OP ...
Relational Expression	relational_expression : shift_expression |	relational_expression '<' ...
Equality Expression	equality_expression : relational_expression |	equality_expression EQ_OP ...
And Expression	and_expression : equality_expression |	and_expression '&' ...

Regular Expressions (REs)	Context-Free Grammars (CFGs)							
Keywords: `int	float	if	else	while	for	return	void	c
Identifiers: `[a-zA-Z_][a-zA-Z0-9_]*`	**Declaration List:** `decl_list → decl_list decl \	decl`						
Integer Literals: `[0-9]+`	**Declaration:** `decl → var_decl \	fun_decl`						
Floating-Point Literals: `[0-9]*\.`	**Variable Declaration:** `var_decl → type_spec	type_spec id '[' INT_NUM						
Operators: `\+	\-	*	\/	\%	\=\=	\!\=	\<	\
Special Characters: `;	,	\(	\)	\{	\}			

Q9: Symbol table of your project?

Symbol Table

SNo	Identifier	Scope	Value	Type	Parameter Type (for functions)
1	main	0	0	FUNCTION	INT, FLOAT
2	printf	0	0	FUNCTION	CHAR
3	a	0	10	INT	
4	b	0	20.5	FLOAT	
5	c	0		ARRAY	
6	i	0	5	INT	
7	sum	0	15.2	FLOAT	