Compilers Project

Phase II

Name	ID
Abdallah Ahmed Ali Abouelabbas	9210652
Omar Mahmoud Mohammed	9210758
Aliaa Ghaith	9210694
Ahmed Osama Helmy	9213061

Technologies Used:

- Bison
- Flex
- Pure C
- ReactJs GUI
- TypeScript

Project Overview:

In this project, we designed and implemented a Compiler for a simplified language inspired by C++, supporting variables, control structures, functions, and data types like **int**, **float**, **bool**, and **string**.

One major highlight of our language and compiler design is:

- Internal String Handling:

Strings are managed internally without relying on external libraries, making the language self-contained and lightweight.

- Dynamic Constant Management:

Constants such as **integers**, **floats**, **booleans**, and **strings** are handled dynamically.

Project Pipeline:

A code written in our language is passed through multiple stages until getting converted to instruction (quadruples in our case). These stages are:

- Lexical Analysis
- Syntax Analysis (Parsing)
- Semantic Analysis
- Code Generation

Lexical Analysis

The first phase of the compiler is the Lexical Analysis, where the input source code is read character-by-character and divided into meaningful sequences called tokens. The lexer is implemented using Flex (Fast Lexical Analyzer), and it transforms the raw text into a structured stream of tokens for the parser to consume.

Tokens we use:

Type	Lexeme	Token name
Data Types	int	KW_INT
	bool	KW_BOOL
	string	KW_STRING
	float	KW_FLOAT
	void	KW_VOID
Keywords	if	IF
	else	ELSE
	while	WHILE
	repeat-until	REPEATUNTIL
	for	FOR
	switch	SWITCH
	case	CASE
	default	DEFAULT
	break	BREAK
	continue	CONTINUE
	return	RETURN
Constants	true	BOOLEAN
	false	BOOLEAN
	{IDENT_START}{IDENT_CHAR}*	IDENTIFIER
	-?{DIGIT}+	INTEGER
	-?{DIGIT}+.{DIGIT}+	FLOAT
	\"([^\\\"]|\\.)*\"	STRING
Expression Operators	"+"	PLUS
	"-"	MINUS
	"*"	STAR
	"/"	SLASH
	"%"	PERCENT
	"=="	EQ
	"!="	NEQ
	"<"	LT
	">"	GT
	"<="	LE
	">="	GE
	"&&"	AND

	"||"	OR
	"&"	BIT_AND
	"|"	BIT_OR
	"^"	BIT_XOR
	"~"	BIT_NOT
	"<<"	SHIFT_LEFT
	">>"	SHIFT_RIGHT
	"++"	INCREMENT
	"--"	DECREMENT
	"!"	NOT
	":"	COLON
	";"	SEMICOLON
	","	COMMA
	"("	LPAREN
	")"	RPAREN
	"{"	LBRACE
	"}"	RBRACE
Assignment Operators	"+="	PLUS_ASSIGN
	"-="	MINUS_ASSIGN
	"*="	STAR_ASSIGN
	"/="	SLASH_ASSIGN
	"="	ASSIGN
Comments	"/*", <COMMENT>\n, <COMMENT>"*/", <COMMENT>.	-
Ignore	[\t]+, \n	-

The above list of tokens is those we are going to create our grammar from in the next stage.

Syntax Analysis (Parsing)

After the lexical analysis produces a stream of tokens, the next step in the compiler pipeline is Syntax Analysis — also called Parsing. The parser verifies that the sequence of tokens follows the grammatical structure of the language and builds a parse tree. This parse tree is that we will use to perform semantic analysis and code generation.

We used custom syntax error handler:

```c
void yyerror(const char *msg) {
    FILE *file = fopen("errors_and_warnings.txt", "w");

    printf("Error: at line %d: %s\n", yylineno, msg);

    fprintf(file, "Error: at line %d: %s\n", yylineno, msg);

    syntax_error = 1;
}
```

Semantic Analysis

We here are using the parse tree got from the parsing stage to check on semantics. We validate generally on.

Type	Validation
Variable Validations	Variable Redefinition Check: Error if a variable name is redefined inside the same scope.
	Variable Used Before Initialization: Error if a variable is used but not declared yet.
	Variable Used Before Being Defined: Error if a variable is declared but not assigned a value yet (for non-parameters).
	Variable Type Match on Assignment: Error if the assigned value has a different type than the variable type (with allowed type "elevations" like int → float).
	Division by Zero Detection: Error if division (/=) assigns by zero (either integer 0 or float 0.0).
Constant Validations	Constant Redefinition
	Constant Reassignment
Function Validations	Function Name Conflict Error if the function name conflicts with an existing variable name.
	Function Redefinition Error if a function is defined after it was already fully defined.
	Function Prototype Type Mismatch Error if a function re-declared prototype doesn't match the original prototype's type.
	Function Parameter Count Mismatch Error if the parameter count does not match between the prototype and implementation.
	Function Call Existence Check Error if calling a non-declared or non-function identifier.

	Function Call Parameters Type Mismatch Error if the passed argument types mismatch with expected parameter types.
	Function Call Parameters Count Mismatch Error if the number of arguments passed differs from the expected number.
	Assigning from Void Functions Error if trying to assign the result of a function that returns void to a variable.
Type Checking in Expressions	Arithmetic Type Checking: Errors if types between operands in +, -, *, / don't match or cannot be implicitly converted safely.
	Logical Operations Type Checking (&&, ||): Errors if types are not booleans (or convertible from int/float).
	Comparison Operations Type Checking (==, <, etc.): Errors if types are incompatible for comparison.
	Bitwise Operations Type Checking (&, |, ^, ~, <<, >>): Errors if operands are not integers.
Switch Statement Validations	Switch Variable Type Validation: Error if switch variable is not int or string.
	Case Labels Type Validation: Error if a case label type does not match the switch variable type.
	Constant Used in Case Declaration: Error if constant used in case is not declared.
	Case Label Validity: Error if invalid case types (non-constant, non-int, non-string)
General Semantic Validations	Implicit Conversions ("Elevations") Warnings: Warnings if safe automatic conversions happen: • int → float • bool → int • int → bool • bool → float • float → bool
	Invalid Expression Types: Error when evaluating an expression and the type cannot be resolved.
	Block Scoping Management: Create new scopes for BLOCK, FOR, IF bodies (even if non-block in syntax).
	Scope Exit Management: Correctly revert back to parent scopes after a block ends.

Code Generation

Finally, we got to the final stage to produce our instructions (quadruples). This is our list of rules we used to build our code generator:

Constant Values and Literals:

Rule	Quadruple	Meaning
Integer constant 5	(= , 5 , _ , temp)	Load integer value into temp variable
Float constant 5.5	(= , 5.5 , _ , temp)	Load float value into temp variable
Boolean constant true/false	(= , true/false , _ , temp)	Load bool value into temp
String constant "hello"	(= , hello , _ , temp)	Load string into temp

Variable and Constant Declarations

Rule	Quadruple	Meaning
Variable declaration	(DECL , _ , _ , var_name)	Declare a variable
Constant declaration	(DCON , constant_name , _ , temp)	Define a constant

Assignments

Rule	Quadruple	Meaning
Simple assignment a = expr	(= , expr_result , _ , a)	Assign expression result to variable
Compound assignment a += expr	1. (+ , a , expr_result , temp) 2. (= , temp , _ , a)	Perform addition and assign back

(Also for -= , *= , /= exactly same logic but different operator.)

Arithmetic Operations

Rule	Quadruple	Meaning
Addition a + b	(+ , a , b , temp)	Sum two values into a temp
Subtraction a - b	(- , a , b , temp)	Subtract two values into a temp
Multiplication a * b	(* , a , b , temp)	Multiply two values into a temp
Division a / b	(/ , a , b , temp)	Divide two values into a temp

Logical Operations

Rule	Quadruple	Meaning
Logical AND a && b	(&& , a , b , temp)	Logical AND
Logical OR	(|| , a , _ , temp)	Logical OR

Logical NOT !a	(! , a , _ , temp)	Logical NOT

Comparison Operations

Rule	Quadruple	Meaning
Equal a == b	(== , a , b , temp)	
Not Equal a != b	(!= , a , b , temp)	
Greater Than a > b	(> , a , b , temp)	
Less Than a < b	(< , a , b , temp)	
Greater or Equal a >= b	(>= , a , b , temp)	
Less or Equal a <= b	(<= , a , b , temp)	

Bitwise Operations

Rule	Quadruple	Meaning
Bitwise AND a & b	(& , a , b , temp)	
Bitwise OR a | b	(, a , b , temp)	
Bitwise XOR a ^ b	(^ , a , b , temp)	
Bitwise NOT ~a	(~ , a , _ , temp)	
Shift Left a << b	(<< , a , b , temp)	
Shift Right a >> b	(>> , a , b , temp)	

Increment / Decrement

Rule	Quadruple	Meaning
Increment a++ or ++a	1. (+ , a , 1 , temp) 2. (= , temp , _ , a)	
Decrement a-- or --a	1. (- , a , 1 , temp) 2. (= , temp , _ , a)	

Function Calls

Rule	Quadruple	Meaning
Push Argument	(ARG , _ , _ , argument_name)	Push an argument for a call
Function Call	(CALL , function_name , n_args , temp)	Call function, store result in temp
Return from function	(RETURN , return_value , _ , _)	Return from a function

Control Flow Statements

IF / ELSE	Quadruple	Meaning
IF condition	(BEGIN IF , _ , _ , _)	Start of IF
Jump if false	(JZ , placeholder , condition_result , _)	Jump to ELSE or END if false
Jump	(JMP , placeholder , _ , _)	Jump over ELSE block
ELSE block	(BEGIN ELSE , _ , _ , _)	Start ELSE
End IF	(END IF , _ , _ , _)	End IF

FOR Loop	Quadruple	Meaning
Start FOR	(BEGIN FOR , _ , _ , _)	
Condition check	(JZ , placeholder , condition_result , _)	
After step	(JMP , loop_back , _ , _)	
End FOR	(END FOR , _ , _ , _)	

WHILE / REPEAT-UNTIL Loop	Quadruple	Meaning
Start loop	(BEGIN WHILE / BEGIN REPEAT-UNTIL , _ , _ , _)	
Jump if false	(JZ , loop_start , condition_result , _)	

BREAK/CONTINUE	Quadruple	Meaning
Break	(BREAK , _ , _ , _)	
Continue	(CONTINUE , _ , _ , _)	

Switch-Case Statements

Rule	Quadruple	Meaning
Start switch	(BEGIN SWITCH , varname , _ , _)	
Start case	(BEGIN CASE , _ , _ , _)	
Switch match multiple cases	(SWITCH-OR , case1_result , case2_result , temp_result)	
Jump if no match	(JNE , placeholder , switch_var , case_result)	
End case	(END CASE , _ , _ , _)	
End switch	(END SWITCH , _ , _ , _)	