Lexer

We modified our lexer to accommodate our parser:

1) Included Headers: The lexer starts by including necessary header files, such as the header file generated by the parser and the standard input-output header.

2. Token Definitions:
   - *Keywords*: Recognizes keywords like "int", "char", "else", and others.
   - *Identifiers*: Identifies variable names and other identifiers, which may start with a letter or underscore followed by letters, underscores, or numbers.
   - *Constants*:
     - *Numerical Constants*: Recognizes numbers, both with leading zeros and without.
     - *Character Constants*: Recognizes character constants and supports a variety of escape sequences.
   - *Punctuation Symbols*: Recognizes a variety of operators and delimiters.
   - *Comments: Detects and ignores both single-line (`//`) and multi-line (`/ */`) comments.
   - *String Literals*: Recognizes string literals enclosed in double quotes.
   - *Whitespace*: Identifies and appropriately handles whitespace characters like spaces, tabs, and newlines.

3. Token Actions: After recognizing a token, the lexer has defined actions that can be anything from printing out the token, returning a token value to the parser, or ignoring the token (as in the case of comments).

Note: As opposed to the previous assignment, we have also split the tokens into many individual components to enter into the parser. Bison recognizes these tokens and parses them according to the grammar rules entered. Additionally, since a more elaborate explanation for the lexer has been provided in the previous assignment, this was brief.

Parser

The parser reads the tokens produced by the lexer and tries to understand the structure of the source code by matching sequences of tokens against the predefined grammar of the nano language.

1. Included Headers and Functions: The parser starts by including necessary headers and defining some external functions, including the `yylex()` function (provided by the lexer) and the `yyerror()` function (for error reporting).

2. Token Declarations: The parser declares the tokens it expects to receive from the lexer. These tokens represent various elements of the language, such as keywords, constants, operators, and more. (declared with %)

Grammar Rules

(recursively defined, with each rule potentially referencing other rules, eventually breaking down the structure of the entire program)

Translation Unit, External Declaration, and Function Definition:

These rules define the structure of the code, including functions and declarations.
translation_unit is the entry point.
function_definition represents function definitions.
external_declaration handles declarations and function definitions.
Statements:

These rules deal with different types of statements in the code, including control flow and expressions.
They print "statement" when reduced.
Declarations and Initializers:

Rules for variable declarations and initializers.
They print "declaration" and "init-declarator" when reduced.
Types and Pointers:

These rules deal with type specifiers and pointers.
They print "type-specifier" and "pointer" when reduced.
Direct Declarators and Parameters:

These rules define variable and function declarations.

They print "direct-declarator," "parameter-list," and "parameter-declaration" when reduced.
Expressions:

Rules for handling expressions, including primary expressions and various operators.
They print "expression," "primary-expression," and "unary-expression" when reduced.
Operators and Operations:

Rules for operators and their operations (e.g., arithmetic, logical, conditional).
They print the respective operator name when reduced.
Logical and Relational Expressions:

These rules handle logical and relational expressions.
They print "logical-AND-expression," "logical-OR-expression," "relational-expression," and "equality-expression" when reduced.
Assignment Expressions:

Handles assignment expressions.
It prints "assignment-expression" when reduced.
Optional Elements and Lists:

These rules represent optional elements (e.g., optional pointer, optional parameter list).
They are used to make certain parts of the grammar optional.
Constant Values:

These rules recognize constants like integer and character constants.
Conditional Expressions:

Rules for conditional expressions using the ? and : operators.
Control Statements:

Rules for control flow statements (e.g., if, for, return).
Errors:

An error message is printed in case of parsing errors.
This Bison specification provides a structured framework for parsing the nanoC language and generating parse trees. It defines the language's grammar rules and actions to be taken when each rule is reduced.

TAC Translator

3-Address Code as Intermediate Representation

This format includes:

- Binary Operations: Operations involving two operands and an operator (e.g., addition, subtraction).
- Unary Operations: Operations with a single operand (e.g., negation).
- Copy Assignment: Simple value assignment from one variable to another.
- Jump Statements: Control flow instructions like unconditional and conditional jumps.

Address Types

In 3-address code, addresses can be:
- Names: Representing variables and function names in the source code.
- Constants: Literal values directly used in the code.
- Compiler-Generated Temporaries: Temporary variables created by the compiler during translation for intermediate steps.

Instruction Types

These are specific forms of TAC, including:
- Binary Assignment Instructions: For operations like x = y op z, where op is a binary operator.
- Unary Assignment Instructions: For operations like x = op y, where op is a unary operator.
- Copy Assignments: Direct assignments like x = y.
- Conditional/Unconditional Jumps: Instructions for flow control, like goto statements.

Symbol table, TAC functions

Defines the attributes of symbols in the symbol table, including name, type, value, size, offset, and any nested symbol table. This structure is crucial for semantic analysis, symbol management, and memory allocation during the compilation process.

symlook(char s):

- Line 1 (symboltable *symlook(char *s)): This function signature indicates that symlook takes a string s (representing a symbol name) and returns a pointer to a symboltable entry.
- Line 2 (symboltable *sp;): Declaration of a pointer to symboltable, used for iterating over the symbol table.
- Line 3-5 (for (sp = symtab; sp < &symtab[NSYMS]; sp++)): The loop iterates through the symbol table array symtab. It stops when it reaches the end of the array, which is indicated by &symtab[NSYMS].
- Line 7-9 (if (sp->name && !strcmp(sp->name, s)) return sp;): Checks if the current symbol table entry (sp->name) matches the input symbol s. If a match is found, it returns a pointer to this entry, thus avoiding duplicate entries for the same symbol.
- Line 10-14 (if (!sp->name) { sp->name = strdup(s); return sp; }): If an empty entry is encountered (indicated by !sp->name), it allocates memory and copies the symbol name s into this entry using strdup(s), then returns a pointer to this new entry.
- Line 15-17 (yyerror("Too many symbols"); exit(1);): If the function iterates over the entire symbol table without finding an empty slot or a matching entry, it calls yyerror indicating the symbol table is full, and then exits the program.

gentemp():

- Line 1 (symboltable *gentemp()): Function signature, indicating gentemp returns a pointer to a symboltable entry and requires no parameters.
- Line 2 (static int c = 0;): Static integer c is initialized. It retains its value between function calls, effectively counting the number of temporary variables generated.
- Line 3 (char str[10];): Declares an array str to hold the name of the temporary variable.
- Line 5 (sprintf(str, "t%02d", c++);): Generates a unique name for the temporary variable (like t01, t02, etc.) and increments c.
- Line 7 (return symlook(str);): Calls symlook with the generated temporary variable name, thus adding it to the symbol table and returning a pointer to its entry.

emit_binary(char *s1, char s2, char c, char s3):

- Line 1-5 (void emit_binary(char *s1, char *s2, char c, char *s3)): Function signature for emit_binary, which takes four arguments: three strings (s1, s2, s3) representing the result variable, first operand, and second operand, and a character c representing the binary operator.

- Line 6 (printf("\t%s = %s %c %s\n", s1, s2, c, s3);): Outputs the formatted binary operation to the console or file. This is part of the TAC generation, where an operation like a = b + c is translated to the corresponding 3-address code.

emit_unary(char s1, char s2, char c):

- Line 1-4 (void emit_unary(char *s1, char *s2, char c)): Function signature for emit_unary, similar to emit_binary but with one less operand, as unary operations only require one operand.
- Line 5 (printf("\t%s = %c %s\n", s1, c, s2);): Outputs the formatted unary operation, such as a = -b.

emit_copy(char s1, char s2):

- Line 1-3 (void emit_copy(char *s1, char *s2)): Function signature for emit_copy, which takes two string arguments representing the source and destination of the copy operation.
- Line 4 (printf("\t%s = %s\n", s1, s2);): Outputs a simple assignment operation, like a = b, to the console or file.

Quad Structure, functions

Establishes the format for quads used in TAC. It includes the operation type and operands. The quad structure is instrumental in converting complex expressions and operations into a simpler, more manageable form for further compilation stages.

new_quad_binary(opcodeType op1, char *s1, char s2, char s3):

- Line 1-5 (quad *new_quad_binary(opcodeType op1, char *s1, char *s2, char *s3)): Function to create a new quad for binary operations. It takes an opcodeType and three strings as arguments.
- Line 6 (quad *q = (quad *)malloc(sizeof(quad));): Allocates memory for a new quad.
- Line 7-9 (q->op = op1; q->result = strdup(s1); q->arg1 = strdup(s2); q->arg2 = strdup(s3);): Initializes the fields of the quad: operation type, result, and two arguments.
- Line 10 (return q;): Returns a pointer to the newly created quad.

new_quad_unary(opcodeType op1, char s1, char s2):

- Similar to new_quad_binary, but tailored for unary operations. The key difference is in the initialization of the quad's fields, where q->arg2 is set to 0, indicating no second argument for unary operations.

print_quad(quad q):

- The function print_quad translates the abstract quad representation into a readable format. It checks the operation type and formats the output accordingly.
- For binary operations, it outputs the operation (like +, -, *, /) between arg1 and arg2.
- For unary operations, it displays the operator (like -, &, *, !) before arg1.

Structures and Functions for Machine Code Generation (A5)

Note: I was not able to separate the outputs for TAC and assembly code in different files. It would be great if you could execute them separately. I apologize for the inconvenience caused. I hope I am given some marks for the implementation I have attempted, even though it is not ideal. We both had to study for the final.

AssemblyLine:

- This defines a new structure type named AssemblyLine. The typedef keyword allows this structure to be referenced simply as AssemblyLine.
- char code[100]: This declares an array code within the structure, intended to hold a line of assembly code. Assuming each line of assembly code will be less than 100 characters in length.

assemblyLines[MAX_ASSEMBLY_LINES]:

- This line declares a global array assemblyLines of AssemblyLine structures. The array is sized to MAX_ASSEMBLY_LINES (1024), can hold up to 1024 lines of assembly code.
- int assemblyLineCount = 1;: tracks the number of lines of assembly code currently stored in assemblyLines. It's initialized to 1, likely indicating that the assembly code generation starts from the index 1 in the array (considering prologue is stored at 0)
- Store individual lines of assembly code, easily iterate through and access these lines for further processing or output generation. Also maintain a count of how many lines of assembly have been generated,

void handleMemoryBinding(symboltable):

- Accept the symbol table as input, add offsets to each entry to store the variable in the required memory location.
- Iterates over each entry in the symbol table to do this.

convertTACtoAssembly(quad)):

- This is the primary function for my machine code generation
- It is not ideal or exactly what was expected, but I have done my best to yield a close enough result with the given time constraints
- When the quad array is being generated, this is called to convert each quad into the corresponding x86 instruction according to the parameters
  Arithmetic Operations:

- PLUS, MINUS, MULT, DIV: These are converted to their respective assembly instructions (add, sub, imul, idiv). For example, the PLUS TAC operation results in an add assembly instruction. These instructions assume the use of the eax register for arithmetic operations. The sprintf function is used to format the assembly instruction as a string. strcpy copies the generated assembly code into the assemblyLines array at the current count index
- Unary Minus: This TAC operation is converted into the neg assembly instruction, which negates the value in eax.
- COPY: Translates the TAC copy operation into a mov assembly instruction. It checks if the first character of the source operand is a temporary variable and accordingly constructs the assembly instruction.
- Logical: These are converted to their corresponding assembly instructions (and, mov, not). star is used for dereferencing a pointer.

implementIOfunctions():

- I was not able to implement this
- It is responsible for defining a set of input/output (I/O) operations in assembly language. These I/O operations are designed to work with the Linux operating system using system calls or syscalls. System calls are a way for programs to interact with the operating system kernel to perform tasks,

printMachineCode():

- First, the standard prologue commands are printed
- After calculating the number of variables that need to be stored in the stack, memory is allocated accordingly
- The standard epilogue commands are added to the end
- The commands for the program body are generated using the three address codes in the quadarray

Test Cases

- Test case 7 is not working because some errors related to floating point handling. I didn’t understand
- Test case 1,2,3 seem to be yielding correct or close to correct outputs
- Test case 4,5,6 outputs are different from the expected outputs a little more.
- A lot of the discrepancies are due to incorrect implementation of backpatching and handling if conditions, along with incomplete implementation of the symbol table.