Translator Design - Lab 2

1. Consider the exponentiation function from `examples/func.imp`. Unfortunately, variable declarations are not yet supported. Rewrite `exp` without the use of variables, ensuring the time complexity is unchanged. Hint: define a helper, `func exp_helper(a: int, n: int, b: int): int`.

2. Implement the missing arithmetic operators, including `==`, `*`, `/`, `%` and `-`.

   (a) Add the required tokens to the lexer.
   (b) Define the appropriate binary operators.
   (c) Produce the AST nodes in the parser, taking care of precedence and associativity.
   (d) Define the instructions to which the operators map to.
   (e) Translate the AST nodes to instructions.

   Note that operator precedence, associativity and ambiguity is relevant. In the BNF form used by parser generators such as *yacc*, one would be tempted to define the grammar using the following rules:

   ```
   expr := INT
         | REF
         | ( expr )
         | expr PLUS expr
         | expr MUL expr
         | ...
   ```

   However, this grammar presents multiple issues. Primarily, it is ambiguous, with no clear specification for the associativity of the operators, since an expression such as $1 + 1 + 1$ can be parsed as either $(1+1)+1$ or $1+(1+1)$. Additionally, the precedence of multiplication over addition is not preserved, as $1 + 2 * 3$ can be interpreted incorrectly as $(1 + 2) * 3$. The grammar cannot be parsed using a recursive-descent parser due to the presence of left-recursion. To correct these problems, the grammar can be modified, we first split off terms which are not involved in decisions about precedence:

   ```
   term := INT | REF | ( expr)
   ```

   To address the issue of left-recursion, it might be tempting to redefine expressions as follows:

   ```
   expr := term PLUS expr
         | term MUL expr
   ```

   However, in this form, $1+1+1$ is parsed as $1+(1+1)$, breaking the rules of associativity. The solution to this problem comes in the form of EBNF, which allows grammars to be defined using regex-like repetitive structures:

   ```
   expr := term [( MUL | PLUS) term]*
   ```

   In this form, an expression is defined as a series of terms with operators in-between. In the parser, the terms can be grouped left-to-right or right-to-left to respect the associativity of the operators, although the problem of precedence is still not addressed, as multiplication is parsed without regard for precedence, consuming operators in the order they appear in the stream. The problem can be solved by ensuring that expressions containing operators of higher precedence are parsed before those containing other operators:

```
add-expr := mul-expr [PLUS mul-expr]*
mul-expr := term [MUL term]*
```

Now all higher-priority terms are considered before moving on to the next lower-precedence operator, parsing $1 + 2 * 3 + 4$ correctly as $(1 + (2 * 3)) + 4$. The `while` loop from the `ParseAddSubExpr` function of the recursive-descent parser originates from these definitions: in EBNF form, the `*` combinator expresses repetition zero or more times, with the `while` loop ensuring the same number of iterations. When faced with a term of the form `T := A (+ B)*`, the following parser should be defined:

```
void ParseT()
{
    ParseA();
    while (Current() == '+') {
        Next();
        ParseB();
    }
}
```

3. The function now relies on `if` statements - add the keywords used in if statements to the lexer.

4. Define an AST node, `IfStmt`, deriving from `Stmt`, to represent `if` statements, ensuring the presence of an optional alternative branch for any statement after an `else` keyword. The class should have a method `std::shared_ptr<Stmt> GetElseStmt() const` to return the alternative if present or `nullptr` otherwise.

5. Compile `if` statements to bytecode, distinguishing two cases based on the presence of an alternative. The lowering of the statement should follow the following pattern, ensuring that in the absence of an else branch, no redundant jump is emitted.

```
    ... condition ...
    JUMP_FALSE else
    ... true branch ...
    JUMP end
else:
    ... alternative ...
end:
```

```
    ... condition ...
    JUMP_FALSE else
    ... true branch ...
else:
```

   (a) Identify the number of labels required and allocate them.

   (b) Lower the condition check and emit a conditional jump.

   (c) Recursively compile the bodies of the branches.