Translator Design - Lab 3

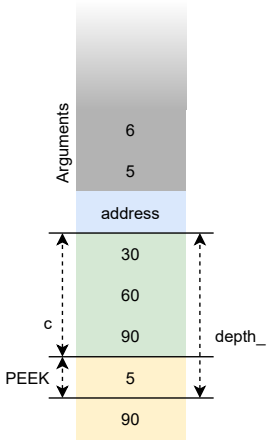
This set of exercises is concerned with local variable declarations respecting the following syntax:

```
let a:int = b + 1
```

1. Identify the new keyword and add it to the lexer.
2. Define a new AST node, `LetStmt`, to represent declarations, capturing the name, type and the expression computing the initial value.
3. Generate code for the declarations, modifying the scoping mechanism.

```
func f(a: int, b: int): int {
    let c : int = a * b;
    let d : int = c + c;
    if (a == 0) {
        let c : int = c + d;
        if (b == 1)
            return a + c;   // HERE
        else
            return d;
    } else {
        let c : int = a + b;
        if (b == 2)
            return c + 2;
        else
            return c;
    }
}
```

(a) A sample program with multiple declarations



(b) The stack at the indicated point, calling `f(5, 6)`

Figure 1: Stack layout with local declarations

Observe the stack layout of the function in Figure 1. On the call to `f`, the arguments highlighted in grey are placed on the stack by the caller and the call instruction pushes the address to which the function should return to. The stack frame belonging to the function itself is considered to start below the address - for the purposes of identifying the location of local declarations, the address is assumed to be at offset 0. The local variables are indicated in green: they are simply placed on the stack, at clearly identifiable offsets. Yellow identifies the temporaries pushed and popped during the evaluation of expressions. The reference to `a` in the expression pushes 5 onto the stack, while the reference to `c` requires 90 to be added as well.

In order to copy a value from the stack to the bottom-most location, the `PEEK` instruction is used, copying a value from a given offset from the bottom of the stack to the end. For example, assuming the stack layout from Figure 1b, `PEEK 1` would push 5, while PEEK 5 would copy the value of the return address. Note that the location of items is usually saved relative to the start of the frame (the address), although peek always references them from the bottom, requiring the offset to be computed.

Local declarations are involved with scoping, as their lifetime and visibility is limited to the scope (pair of curly braces) they are defined in. In the code generator, the `Scope` and its derived classes

implement this functionality, containing bindings mapping the names of variables to the locations they are available at or the values they are mapped to. An object in the chain contains a mapping for all locals defined in the relevant block scope, also pointing to the parent, encompassing, scope where variables not defined in the scope should be located. As an example, at the point indicated in Figure 1a, multiple scopes are active. The innermost most is between the braces of the true branch of the if statement, declaring `c`. This scope is nested inside the body of the function, which is also a `BlockStmt`, declaring `c` and `d`. The parent of this scope is the function scope containing the bindings for the arguments to the function, `a` and `b`. Finally, bindings for the names of the functions are located in the root global scope. When searching for `c` at that point, only the first scope is inspected, as it contains a binding for `c`. However, the read from `a` must traverse multiple parents, until reaching the function scope which binds `a`.

(a) Adjust the binding in the code generator to allow the location of locals to be recorded. A binding maps the name of some variable to the location it can be loaded from - for example, in `f`, `a` is already bound to argument index 0. Upon declaring other variables, they should be bound to a local location, saved as an index from the start of the frame. For example, the first declaration of `c` should be bound to location 0, while the second declaration of `c` to location 2.

(b) Extend the code generator to lower *let* statements. The initialiser expression should simply be lowered, after which the current scope object should be altered to bind the name of the declaration to the index the value was placed at. In the code generator, `depth_` always records the number of elements on the stack at the point of the execution of the last emitted instruction.

(c) Variables are limited to their own scopes, reason why the scopes delimited by braces form a chain, with all block statements adding a local scope to it. Once a block statement finishes execution, the local variables declared inside must be de-allocated. Find a way to record the number of variables declared in a local scope and adjust the stack after executing all statements in a scope. Either re-use an existing instruction or define a new, faster one.