TDT 4205
Problem Set 4

The deadline for this problem set is friday, October 26th. Submissions will only be accepted through *It's learning*.

1 Memory layout

1.1 10%

Does a VSL program require any heap memory at run time? Explain.

1.2 10%

Describe the layout for a stack frame of the following function:

```
FUNC sumsquares ( a, b )
{
VAR c,d
c := a * a
d := b * b
return c + d
}
```

2 Scoping

A hypothetical language which treats functions as values may admit the following code:

```
function differentiate ( a, n )  // Differentiate a x^n
{
    return function d_dx ( x ) {
        return a * n * (x^(n-1));
    };
}

df_dx = differentiate ( 2, 3 );  // Differentiate f( x ) = 2x^3
dg_dx = differentiate ( 3, 4 );  // Differentiate g( x ) = 3x^4
x = 5;
print df_dx( x ), dg_dx( x );    // Evaluate at x=5
```

When the language binds the free variables of *d_dx* (that is, a, n) to their values in the enclosing scope ($differentiate$) at the time of invocation, this construct is called a *closure*. (Two notable languages which feature closures are Scheme and Perl.)

2.1 10%

Suppose a compiler for this language handles the evaluation of *df_dx* and *dg_dx* by allocating the activation environment for $differentiate$ statically, and looking up a, n in this environment when one of the returned *d_dx* is called. Which values would you expect the above code to print under these assumptions?

2.2 15%

Show how you would structure the symbol table entries for *df_dx* and *dg_dx* in order for the language to handle closures properly.

3 Practical work

The skeleton code for these tasks expands upon last week's work by adding the files *scope.c/.h*, and *symtab.c/.h* in the *src* and *include* directories. The ultimate purpose of these files is to link identifiers in the syntax tree with a table of data structures, but the structures for this symbol table are not yet in place.

The following tasks concentrate on implementing auxiliary functions (in *scope.c*) which will come into use when the table structures are implemented. Their purpose is to isolate the text representation of identifiers, as C can be somewhat clumsy with respect to string manipulation.

3.1 10%

Implement the functions

```
void init_scopes ( uint32_t size );
void destroy_scopes ( void );
```

in *scope.c*, which allocate and free a stack of strings.

3.2 10%

Implement the functions

```
void push_scope ( char *text );
void push_nameless_scope ( void );
void pop_scope ( void );
```

in *scope.c*. `push_scope` should add a given string (which you may assume is dynamically allocated), `push_nameless_scope` should add a programatically generated unique string (a simple counter is fine), and `pop_scope` removes the last string added, freeing its memory.

3.3 20%

Implement the functions

```
char *generate_scope_string ( uint32_t depth );
char *augmented_scope_string ( uint32_t depth, char *text );
```

in *scope.c*. `generate_scope_string` should return a dynamically allocated copy of a concatenation of *depth* items from the present stack (separated by a constant string given in the code), while `augmented_scope_string` should return a similar string with an extra item (which is not on the stack) added at the end.

3.4 15%

Implement the function

```
void find_symbols ( node_t *root );
```

in *symtab.c*, which should traverse a syntax tree recursively, tracking scope and printing the full path of identifiers (functions and variables) as they are encountered. (Generate the strings such that functions and variables add their name to the path, while blocks add an unnamed scope.)