- **Subtask 3**: Implement the basic code structure that contains the code for "main" with simple declarations, expressions, and assignments only (see Slides 6-7 in the notes on "10-Intermediate Code Generation").

Implementation Details

(1) For the very initial implementation, we can use an empty main function such as "void main(void) {}". This should lead to the following sequence of instructions:

```
/* code for prelude */
...
/* code for i/o routines */
...
12: ST 0, -1(5)       save return address
13: LD 7, -1(5)       return back to the caller
11: LDA 7, 2(7)       jump forward to finale
 /* code for finale */
 ...
```

- After that, we can gradually add code generation functions for expressions, assignments, control structures, functions and recursions, nested blocks, arrays, and runtime error checking, as suggested in the marking scheme for Checkpoint Three.

(2) Implement all the emit routines in CodeGenerator.java:

- Note that every time we emit an instruction, "emitLoc" is always incremented, and if it exceeds "highEmitLoc", the latter is also adjusted up.

- When generating an instruction, avoid using "fprintf" directly, but use the related "emit" routine since we can increment "emitLoc" and "highEmitLoc" as well.

```
void emitRO( char *op,
  int r, int s, int t, char *c ) {
  fprintf( code, "%3d: %5s %d, %d, %d",
    emitLoc, op, r, s, t );
  fprintf( code, "\t%s\n", c );
  ++emitLoc;
  if( highEmitLoc < emitLoc )
    highEmitLoc = emitLoc;
}
```

(3) Maintain three different offsets during code generation:

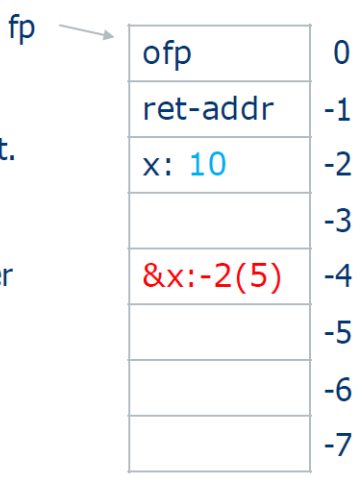
- For iMem, declare "emitLoc" and "highEmitLoc" as global variables. The former points to the current instruction we are generating (may go back to an earlier location for backpatching), while the latter always points to the next available space so that we can continue adding new instructions.

- The global stackframe at the top of dMem is pointed by the "gp" register, and its bottom is indicated by the global variable "globalOffset". If we have "int a" and "int b" declared in the global scope, we will have "globalOffset=-2". If we have "int x[10]" declared in the global scope, we will have "globalOffset=-11" (10 integers plus 1 more for size).

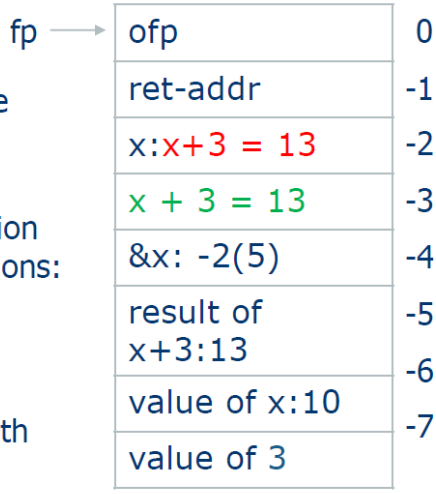
- The current stackframe in dMem is pointed by the "fp" register, and its bottom is indicated by a parameter "frameOffset", which is local in your recursive function for code generation. Since the first two locations are reserved for "ofp" and "return addr", the parameters and local variables will start from "-2" location (initFO) in the stackframe.

(4) Slides 6-7 in the notes on "10-CodeGeneration" show the key steps for generating intermediate code for expressions. For Checkpoint Three, however, we need to map them further to generate TM assembly code. This can be illustrated with the following example:

- Assuming the syntax tree for the expression "x = x + 3" and the stack frame on the right.
- The initial call is "visit(tree, -3, false)", where "tree" is an AssignExp and "-3" is the frameOffset.
- Inside the "visit" for AssignExp, we will call "visit(tree.lhs, -4, true)" and "visit(tree.rhs, -5, false) first in the post-order traversal. The former is for a SimpleVar and the latter is for an OpExp.
- Inside the "visit" for SimpleVar when used as the left-hand side of AssignExp, we will compute the address of "x" and save it in location "-4". This is done with these two instructions (assuming that we are starting with instruction 13):
  13: LDA 0, -2(5) and 14: ST 0, -4(5)



- Inside the "visit" for OpExp, we will call "visit(tree.left, -6, false)" and "visit(tree.right, -7, false)" in the post-order traversal. The former is for a SimpleVar and the latter is for an IntExp.
- Inside the "visit" for SimpleVar when not used as the left-hand side of AssignExp, we simply save the value of "x" to location "-6" with the following two instructions:
  15: LD 0, -2(5) and 16: ST 0, -6(5)
- Inside the "visit" for IntExp, we will save the value of "3" to location "-7" with these instructions:
  17: LDC 0, 3(0) and 18: ST 0, -7(5)
- Back to the "visit" for OpExp, we will do the addition save the result in location "-5" with these instructions:
  19: LD 0, -6(5) and 20: LD 1, -7(5)
  21: ADD 0, 0, 1 and 22: ST 0, -5(5)
- Back to the "visit" for AssignExp, we will do the assignment and save the result to location "-3" with the following instructions:
  23: LD 0, -4(5) and 24: LD 1, -5(5) and 25: ST 1, 0(0)
  26: ST 1, -3(5)



- Note that in the above illustration, we need to handle the "visit" for SimpleVar differently depending on whether we are computing the left-hand side of AssignExp or not. This is distinguished by "isAddr" parameter in the "visit(Absyn tree, int offset, boolean isAddr)".
- The value for "isAddr" is false for most cases except when calling "visit(tree.lhs, offset, true)" of AssignExp, since this is when we need to compute and save the address of a variable into a memory location.
- For the case of IndexVar, we naturally compute the address of an indexed variable, and that value can be saved directly into a memory location when used in the left-hand side of AssignExp.
- As a general principle, we use the given location to save the result of an OpExp, and the next two locations for its left and right children. In addition, register "0" is used heavily for the result, which needs to be saved to a memory location as soon as possible.