CSE 440/598 Fall 2010
PROJECT 3/4
Assigned 11/2/2010

PLEASE READ EVERYTHING AND ASK QUSETIONS EARLY AND OFTEN.

This is the third and last installment of the project. In this project, you are asked to write a compiler to transform programs written in simplified Pascal into C programs in a restricted form.

- For part 3 of the project, you will only support global (explained below) and heap variables.
- For part 4 of the project, you will support methods and arrays.

I will first explain the restricted form of the output of your compiler. Then I will explain what features you need to support.

# OUTPUT RESTICTIONS

The C programs you will generate have no function calls (I will explain below how to generate code for function calls without using C functions) and have only one global variable called memory (or another name if you prefer). The `memory[]` variable is an array of integers:

```
int memory[MEM_MAX];
```

where `MEM_MAX` is the maximum memory size. YOU ARE NOT ALLOWED TO DECLARE ANY VARIABLES OTHER THAN `memory[]` in the generated C program.

All global, heap, and stack variables will be in `memory[]`. In addition, any temporary "registers" you might need will also be in `memory[]` (more about that later).

You only need to support the integer and boolean basic types in your input program.

## STORING, READING and WRITING VARIABLES

### GLOBAL VS. HEAP VARIABLES

The Pascal grammar you are using allows the declaration of variables only inside classes. The "main" class (the class whose name is the same as the program) is a special class that need not be instantiated to execute its "main" (constructor) function.

**[Assumptions about the main class]** You can assume that the "main" function has no parameters. Also, you can assume that the "main" class has no fields and has no methods other than the "main" function. All variables declared in the "main" method need not be allocated on the stack. Instead, they can be allocated in a global area of `memory[]` (technically, these are not global variables because they cannot be accessed directly by methods of other classes (for part 3 of the project we do not support methods for other classes).

**OFFSETS**

Each global variable will be assigned an offset in `memory[]`. This offset is calculated at compile time. Since you are only supporting the integer basic type, a variables that is not an instance of a class or an array can fit in one entry in `memory[]`. The size of any user-defined type should be calculated by your compiler.

**Example**

Assume your input Pascal program has 4 integer variables a, b, c, and d. Assuming you are not using any of the low memory area for registers (more about that below), these variables will get offsets 0,1,2, and 3 respectively.

```
a := c+d
```

will be translated to

```
memory[0] = memory[2] + memory[3]
```

In the code, the array index determines the location where a variable is stored. Boolean values of boolean expression can be stored as integers.

This approach works well for integer (or boolean) variables, but does not work well for variables whose offset is not known until runtime. For example,

- local variables (in functions, which need not be supported in this part of the project, but the discussion is relevant to dynamically allocated variables also) have an offset (known at compile time) within the frame (activation record), but to that offset we need to add the value of the frame pointer (or added to "stack pointer-frame size").
- The addresses of attributes (fields) of an object (class instance) whose address is not known at compile time are calculated as an offset from the address of the object.

In both cases, we need to be able to calculate something like `memory["value of frame pointer"+offset)`. The frame pointer is a register on many architectures. The frame pointer can be located at a well defined address (index) in memory (calculated at compile time). So, the code will look like

```
memory[memory[stackpointerlocation]+offset)
```

In a way, this is relative addressing using the `stackpointer` register. You are allowed to have a constant (does not depend on the number of variables in the program or the number of times `new()` is called) number of registers to execute calculations like the one above (Indirect reference using a register will look like

```
memory[memory[register_index]+offset]
```

You should be able to handle all variable types with the above scheme. The key is to calculate the offsets correctly.

For objects, offsets of attributed are calculated relative to the start address of the object. Since objects are created dynamically, accessing attributes requires relative addressing as discussed for local variable, with the difference being that the base address is the start address of the object and not the frame or stack pointer. Here, you should not use a different register for every object. Rather, you should use one register to which you copy the base address of the object.

Your implementation should support objects whose attributes (fields) can themselves be objects and hence should support dynamically allocated structures such as linked lists.

Calculating offsets is covered in details in the textbook and I will be discussing it in details in class.

To handle control flow statements, you are allowed to use

1. goto L, where L is a label
2. if .. then .. else .., where the condition is a boolean variable that is stored in a given location. So, the only expressions allowed in the condition in your generated code is an identifier.

All assignments generated by your compiler should be in a 3-address code format, so a = b+c+d is not allowed. No statement your compiler generates should refer to more than three addresses (with the exception of relative addressing discussed above).

For this project, you can assume that there is only one function which is the constructor of the "main" object (the object whose name is the same as that of the program). You can also assume that all variables in that constructor are global variables (the only code is the code of that constructor) and that the constructor has no parameters.

You should provide an implementation of a memory allocator, but your implementation need not support de-allocation.

Finally, you should make sure that you implement early on a function to print variables in your generated code. You are allowed to support printing multiple variables as well as a string to make the output readable, but that has to be generated, so you have to support a form of writeln that you find useful for your work.

What the output should look like:

To make your code more readable, you are allowed to use #define as long as the preprocessed code follows the requirements. In particular, you can #define offsets that are known at compile time. You can #define locations to be used as registers or temporary variables. You can also #define memory access so that it is more compact.

**PART 3 REQUIREMENTS**

You input is a "object Pascal" program and your output is a C program as described above and with support for the functionality described above.

**PART 4 REQUIREMENTS**

For this part of the project, you will support functions (methods) and arrays. If you are not able to implement everything, you should clearly state in your submission any restrictions your implementation has.

1. Supporting methods
   - passing parameters: scalar, objects, arrays
   - pass by value
   - pass by reference
2. Supporting arrays
3. You should make sure that you support inheritance in your solution.

**PART 4 ALTERNATIVE REQUIREMENTS**

```
Send me a proposal by email of what you would like to do for project 4
```

**GRADING**

I will grade your project based on the correctness of the output that the generated programs produce. **Programs that use an unbounded number of registers or more C functionality than is allowed by the specs will get no credit.**

**I will be providing you with sample programs and the expected output of the programs your will generate for these sample programs.**