Assignment Description

In this assignment, you will: a) translate the AST representation to a three-address intermediate representation; and b) generate unoptimized assembly code. We expect you to build upon the code that you wrote for the previous assignments. You are required to implement the following:

- **Three-address code**. You will translate the code of each method to an appropriate sequence of IR instructions. These will probably include the standard classes of instructions: unary and binary operations, data movement instructions, labels and branch instructions, method calls, return instructions. However, it is your duty to choose the particular instructions in each class. *You must clearly describe your instruction set in the write-up.*

- **Generating three-address code**. You must implement then a translation from the AST representation to three-address code. Your translation phase must convert high-level constructs such as if and while statements, short-circuit conditional expressions, break and continue statements into low-level code using jump instructions. We expect no consecutive labels in the generated Low IR code.

  Your three-address code must also include appropriate run-time check instructions. For each each array access `a[i]` (read or write), the compiler must insert two checks before the actual instruction that accesses the array: one check that tests if `a` is not null, and one that tests if the access is within bounds. When translating the length-of expression `a.length`, the compiler will insert a null check for `a`, followed by the instruction that retrieves the array length. To translate a dynamic array allocation `new T[n]`, the code must check if the array size `n` is positive, then invoke a function `__allocateArray(n)` that returns a reference to the first element of the newly allocated array. The allocation function is provided as part of the IC library. The size of each array element has a fixed value of 4 bytes and need not be passed to `__allocateArray(n)`. This is the size for all types in the language (booleans, integers, and references).

  For each field access `o.f` or method call `o.m()`, the compiler generates a null check for `o` before the code that accesses the field or calls the method. The instruction generated for a method call `o.m(...)` must have the receiver object `o` as its first argument, followed by the other explicit arguments. For non-qualified calls `m(...)` where the invoked method `m` is virtual, the first argument of the call is `this`. For static method calls (both qualified `C.m(...)`, or not qualified `m(...)`), just the explicit arguments are passed.

  To translate an object allocation expression `new C()`, the code will invoke the library function `__allocateObject(s)` that returns a reference to the newly allocated object. The size `s` of the allocated object must accommodate all of the fields of the allocated object, plus 4 bytes (to store a reference to the dynamic dispatch table).

Finally, to lower the concatenation of strings `s + t`, the compiler will generate a call to the library function `__stringCat(s,t)`.

- **Simple Code Generation**. Next, you will translate the your three-address code into x86 assembly code. You will perform a straightforward, unoptimized code generation by translating each IR instruction into a sequence of assembly instructions. The generated assembly code may be inefficient, but it must be correct and it must match the semantics of the input program. Your translation must correctly handle all of the following:

  - *Stack Frames*. Generate the calling sequences before and after invoking functions, and at the beginning and the end of each function (prologue and epilogue), as discussed in class. Registers eax, ecx, and edx are caller-save, and registers ebx, esi, edi, ebp, esp are callee-save. You must assume that the contents of caller-save registers might be destroyed at each method call. On the other hand, if a function modifies these registers, it must restore them to their original values before returning. The function parameters are being pushed on the stack by the caller, in reverse order. That is, the first parameter is pushed last on the stack. The return values are always passed in the eax register.
  - *Variables:* Each local variable will be allocated on the current stack frame at the beginning of their enclosing method. Both local variables and method parameters will be accesses using offsets in the current stack frame.
  - *Objects*. You must provide support for objects, as discussed in class. Your code must set up and use dispatch vectors (DV). For virtual calls `o.m(...)` the code must look up the dispatch vector of object `o` for method `m` and perform an indirect call. However, static calls will be translated into direct calls in the assembly program. For method names in the generated assembly, use a naming scheme where a method `m` of a class `A` is named `_A_m`. For field accesses `o.f` you must access the memory location at address `o` plus the constant offset for field `f`. Field accesses of the form `f` are equivalent to `this.f`.
  - *Arrays and Strings*. Arrays and strings will be stored in the heap. To create new arrays, use `__allocateArray`; to concatenate strings, use `__stringCat`. String constants will be allocated statically in the data segment. Strings don't have null terminators; instead, each string is preceded by a word indicating the length of the string. The length of a string should treat escaped characters such as `\n` as one single character. For arrays, the array length is stored in the memory word preceding the base address of the array (i.e., the location at offset -4).
  - *Run-time checks*. You must implement the run-time check instructions in your low-level representation as a sequence of assembly instructions that perform those checks.
  - *Library functions*. Call to these functions are translated as any other static method calls. The code for these functions will be available in the IC library.

– *Main function.* The main method must be translated into static call to `__ic_main`. The library contains a wrapper function that sets up the argument list as a valid `string[]` object before calling `__ic_main`.

Command line invocation. Your compiler will be invoked with a single file name as argument, as in the previous assignment. With this command, the compiler will perform all of the tasks from the previous assignments. Next, it will convert the AST into three-address code, and then will generate assembly code into a file with the same name as the input program, but with extension "`.s`".

In addition to all of the options from the previous assignments, your compiler must support an additional command-line options "`-print-ir`", which will print at `System.out` a description of the three-address code for each method in the program. Be sure you indicate the class name and the method name for each method. For readability, please separate the code for different methods by blank lines.

Invoking the Assembler and the Linker

Given an input file `file.ic`, your compiler will produce an assembly files `file.s`. You can then use the assembler to convert this assembly code into an object file `file.o`, and use the linker to convert this object file into an executable file. To run the gnu assembler `as` on `file.s`, use the command:

```
as -o file.o file.s
```

To run the gnu linker `ld` on `file.o` and link this object file with the library `libic.a`, use the command:

```
ld -o file.exe file.o /lib/crt0.o libic.a -lcygwin -lkernel32
```

The library file `libic.a` is a collection of `.o` files bundled together, containing the code for the library functions that are defined in the language specification, along with run-time support for garbage collection. The library uses a freely available conservative collector (`http://www.hpl.hp.com/personal/Hans_Boehm/gc`).

Supporting Material. You can find the library file `libic.a` along with supporting material for this assignment on the web site for this course. The supporting web page includes documentation for the `as` assembler; documentation for the x86 instruction set; and several example IC programs along with the corresponding x86 assembly code.

What to turn in

Turn in electronically:

- A file `pa3.zip` containing all of your source code and test cases (in directories `/src` and `/test`).
- A brief, clear, and concise document describing the your code structure and testing strategy. Include a description of your three-address instruction set. Document challenges that you encountered in the implementation and your solutions to these problem.

Finally, tell us about how you divided the project into smaller components, what where the interfaces between these components, and who worked on each component. Include this writeup in the directory `writeup`.