Assignment Description

In this assignment, you will: a) translate the AST representation to a low-level 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:

- **Low IR**. This is the low-level intermediate representation of the computation. The Low IR uses three-address code and models the instruction set of an abstract machine. For each method, you will have a sequence of low-level instructions representing the body of that method.

  The low-level instructions may include: unary and binary instructions, copy instruction, array access instructions, length-of instruction, field access instructions, method calls, return instructions, labels, unconditional jumps, and conditional jumps branching on true or false conditions. You will also use a special instruction for library function calls, and another for run-time checks. Your are allowed to extend this instruction set with additional Low IR instructions if you consider so, but you must clearly describe them in the write-up.

  The operands of the Low IR instructions can be either constants, program variables, or temporary variables created by the compiler during the translation to low-level code. Whenever it creates temporary variables, the compiler must insert them in the symbol table. We recommend that you represent variables using references to their entries in the symbol table.

- **IR Lowering**. You must implement then a translation from the AST representation to Low IR instructions. Your lowering phase should translate high-level constructs such as if and while statements, short-circuit conditional expressions, break and continue statements into low-level code using jump instructions. There should be no consecutive labels in the generated Low IR code.

  Your low-level intermediate code must also include run-time check instructions. To lower an array access `a[i]`, you must generate 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. To lower the length-of expression `a.length`, you will generate a null check for `a`, followed by the length-of instruction. To lower a dynamic array allocation `new T[n]`, you will check if the array size `n` is positive, then invoke a library function `__allocateArray(n,s)`, which will return a reference to the newly allocated array. The allocation function takes two arguments, the number of elements `n` and the element size `s`. For simplicity, you can use a size of 4 bytes for all types: booleans, integers, or references.

  For each field access `o.f` or method call `o.m()`, the compiler generates a null check for `o`, followed by the field access or call instruction. The call instruction generated for a method call `o.m(...)` must have the receiver object `o` as its first argument, followed by the actual parameters. For method calls of the form `m(...)`, the first argument must be `this`. To lower an object allocation expression `new C()`, you will invoke a library function `__allocateObject(s)`, which will return 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`, you will generate a library call `__stringCat(s,t)`.

- **Simple Code Generation**. Next, you will translate the low-level intermediate representation into x86 assembly code. You will perform a straightforward, unoptimized code generation by translating each low-level IR instruction into a sequence of assembly instructions. The generated assembly code may be inefficient, but must be correct and 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 may 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. All of the method calls are virtual calls: for a call `o.m(...)` you must look up the dispatch vector of object `o` for method `m` and perform an indirect call. You must pass the object `o` as the first parameter, followed by the formal parameters of the method. For calls of the form `m()` you must pass `this` as the first parameter. 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.* Unlike methods, each of these functions must be invoked using direct (static) calls. The name of each library must be preceded by two underscores in the generated code. For instance the instruction `call __println` represents a call to `println`.
- *Main function.* If your main function belongs to class `A`, you must generate a wrapper function `_ic_main` which just creates an object `A` and invokes the method `main` of that object.

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 Low IR, 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-lir`", which will print at `System.out` a description of the Low IR for each method in the program. Be sure you indicate the class name and the method name for each method. The code for different methods must be separated 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 on paper:

- A brief, clear, and concise document describing the your code structure and testing strategy. Include a description of your Low IR design.
- Feedback. Please continue to provide one paragraph to tell us your overall thoughts about the assignment, especially how much time you spent on it and how you think it could be improved.

Turn in electronically:

- A file `pa3.tar.gz` containing all of your source code and test cases (in directories `/src` and `/test`).

Checkpoint submission:

As in the previous assignment, to encourage you to start working on the project early, there will be a checkpoint submission halfway through the assignment. You have to submit the current state of your work as a file `checkpt.tar.gz` by March 12, using CMS.