CS 301 Spring 2016
Project Phase 3
March 28 – April 14

IC **Compiler Back End**

Plan

By the end of this phase of the project, you will be able to run IC programs generated by your compiler! You will implement:

- a translator to convert your AST representation to a TAC intermediate representation; and
- a code generator to convert TAC programs to x86 assembly code.

Code generators would seem no more complex than other parts of a compiler, but they are notorious for subtle bugs. **Start *early* and design carefully before you start coding. Implement, test, and commit your code incrementally.** If you are debugging issues with the code generator, seek help with effective debugging techniques before burning valuable time on misinformed or haphazard bug hunts.

Implementation: Lowering ASTs to TAC

Three-Address Code. The translator should translate the code of each method to an appropriate sequence of TAC IR instructions. These will probably include the standard kinds of instructions: unary and binary operations, data movement instructions, labels and branch instructions, method calls, return instructions. You are responsible for choosing the specific instructions for your own TAC instruction set. Take the TAC description on the project web page as a starting point.

You must clearly describe your TAC instruction set in your documentation.

Generating Three-Address Code. After deciding on a TAC instruction set, you will implement a translation from your AST representation to TAC. Your translation phase must convert high-level constructs such as `if` and `while` statements, short-circuit conditional expressions, `break` and `continue` statements, etc. into low-level code using jump instructions. Optionally, you may try to generate "clean" TAC code with no consecutive labels, no unnecessary jumps, etc. These are optimizations — do not implement them unless you have completed the basic assignment with time to spare. Do not try to minimize the number of temporary names within a method, for the reasons we discussed in meetings.

Run-time Checks. Your three-address code must also include appropriate run-time check instructions.

- For 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 expression *`e`*`.length`, the compiler must insert a null check for *`e`*, followed by the instruction that retrieves the array length.
- To translate a dynamic array allocation `new T[`*`e`*`]`, the compiler must insert a check to verify that the array size *`e`* is non-negative.
- For each field access *`e`*`.f` or method call *`e`*`.m()`, the compiler generates a null check for *`e`* before the code that accesses the field or calls the method.
- For each division or mod operation, check for division by zero.

Method Calls and String Concatenation. 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`.

To lower the concatenation of strings *s* `+` *t* to TAC, the compiler should generate a call to the hidden library function `Library.stringCat(`*s*`,`*t*`)`.

Code Structure. Implement the TAC representation and TAC generator in an `ic.tac` package. Design suggestions:

- I strongly encourage implementing the TAC instructions as a collection of case classes extending an abstract `TACInstr` class.
- Create a `TACList` class that stores a list of TAC instructions. After translating to TAC, each method declaration in your AST would then have a `TACList` attached.
- Note that the operands to most TAC instructions can be (a) program variables, (b) temporary variables, or (c) constants. The design of your TAC classes should support this.
- You may wish to design your `TACList` and `TACInstr` classes to support generic mapping/folding operations to make it easy for clients to traverse and process TAC lists (for printing, generating x86 code, etc.).
- To help understand generated code and debug TAC and x86 code generation, I encourage you to design your TAC classes (and x86 code generator) to support String annotations on individual instructions that can be printed out as "comments" in your output later, as in:

```
label _if3:              # true branch for if on line 3
...
t1 = x + 1               # line 11
```

  You can even include a `TACComment` instruction form that does nothing but generate a comment to be stored at a specific point in the target program.

Command line invocation. Your compiler will be invoked with a single file name as argument, as before. With this command, the compiler will perform all of the tasks from the front end. Next, it will convert the AST into three-address code.

In addition to all of the options from the previous assignments, your compiler must support the command-line option `-printIR`: print a description of the three-address code for each method in the program. Indicate the class name and the method name for each method. For readability, separate the code for different methods by blank lines.

Finally, as described in the next section, the compiler will generate assembly code in a file with the same name as the input program, but with additional extension "`.s`".

Implementation: x86 Code Generation

Simple Code Generation. Next, you will translate 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. Do *not* attempt to optimize at this point. The generated assembly code may be (very) 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 procedure (prologue and epilogue). Registers `%rax`, `%rcx`, `%rdx`, `%rdi`, `%rsi`, `%r8 - %r11` are caller-save, and registers `%rbx`, `%rbp`, `%rsp`, and `%r12 - %r15` are callee-save. You must assume that the contents of caller-save registers might be destroyed at each call. On the other hand,

if a procedure (including one you generate) modifies callee-save registers, it must restore them to their original values before returning. Follow these rules in all generated code.

Under the x86_64 GNU/Linux ABI, the first 6 arguments (from left to right, don't forget the receiver for method calls) are passed in registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`. Remaining arguments are pushed onto the stack, with the earliest closest to the top of the stack (*i.e.*, the lowest memory address). You must follow this convention when calling external (*i.e.*, `__LIB`) code and where your code is called by external code (*i.e.*, in `__ic_main`).

You may adopt any calling convention in generated code that calls (or is called by) only other generated code. For example, you may find it easiest to store/find all arguments/parameters on the stack (I actually encourage this at least to start), optionally using `%rbp` as a base/frame pointer. Any consistent internal calling convention is acceptable for this project as long as you follow the ABI when interfacing with external code.

Return values are always passed in the `%rax` register.

Keep stack frames 16-byte aligned. The stack pointer will be properly aligned when your code is entered. All you need to do is ensure each stack frame (including all parts, *e.g.*, return address) is a multiple of 16 bytes in size.

- *Variables.* Each local variable should be allocated on the current stack frame at the beginning of the enclosing method. Both local variables and method parameters will be accessed using offsets in the current stack frame.

- *Objects.* For an object allocation expression `new C()`, your assembly code should invoke the library function `__LIB_allocateObject(s)`, which returns a reference to the start of the newly allocated and zeroed object. The size `s` of the allocated object must be chosen to accommodate all of the fields of the allocated object, plus 8 bytes to store a reference to the dynamic dispatch table (vtable/method vector). After allocating space for an object, your code must set up this reference and the use virtual table to resolve method invocations. For virtual calls `o.m(...)` the code must look up the vtable of object `o` for method `m` and perform an indirect call to the appriopriate code. 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 the library function `__LIB_allocateArray(n)`, which returns a reference to the first element of a newly allocated and zeroed array large enough to store n elements. All array elements have a size of 8 bytes, the size for all types in the language (booleans, integers, and references). The array allocation function stores the array length in the memory word preceding the base address of the array returned (*i.e.*, at offset -8).

  String constants should be allocated statically in the data segment using the ASCII encoding, with one byte per character. Strings do not have null terminators; instead, each string is preceded by a word indicating the length of the string. As with array lengths, this length should be stored at an offset of -8 from the address of the string character array. The length and representation of a string should treat escaped characters such as `\n` as one single character.

- *Run-time checks.* You must implement the run-time check instructions present in your low-level representation as sequences of assembly instructions that perform those checks. Print a message and call `__LIB_exit(n)` to terminate the process with exit code n after reporting the error.

- *Library functions.* Calls to `Library` functions (including the `stringCat` function that implements string `+`) are translated into function call sequences using the naming convention given above. For example `Library.readi()` should be converted into a call to the function `__LIB_readi` in the assembly code. You must pass the arguments to the library functions in registers: use `%rdi` for the first parameter and `%rsi` for the second. The result will be in `%rax` as usual. The code for these functions will be available in the IC runtime library.

- *Main function.* The assembly code must contain a global function named `__ic_main`. When a program is executed, the run-time library will set up the command-line argument list as a valid `string[]` object and then call your `__ic_main` with that string array in register `%rdi`. Given a program whose `main` method resides in class `A`, the `__ic_main` function should create an `A` object and then invoke its `main` method, passing along the provided argument list.

Code Structure Implement all code for TAC-to-x86 code generator in the `ic.cg` package. *I will provide a skeleton for a code generator that will produce much of the boilerplate assembly you need to get things working. More on that soon.* I suggest splitting your code generate into a couple separate passes:

- Compute Offsets:
  - the object offset for each declared field
  - the vtable (method vector) order for each class and the vtable index for each method
  - the stack frame offset for each TAC variable, including parameters, local variables, and temporary variables created during TAC generation.

  Store this information in declaration nodes in the AST (classes, fields, methods, declared parameters/variables) and in TAC variables (temporaries). Extend your symbol table printer and TAC printer to show this information.
- Generation x86 Assembly Code:
  - Generate all necessary data literals (string literals, vtables/method vectors).
  - Generate code for each method, code for error handlers, and code for the `__ic_main` wrapper.

Tools

When you get to code generation, you will want to use an x86-64 GNU/Linux environment. The wx appliance should work well. Things can also work on Mac OS X, but there are a couple small workarounds necessary. Windows differs significantly, but you might have luck with Cygwin. I will support only GNU/Linux.

If you are working on lab machines, you can run the compiler directly, but since these poor machines are still stuck with a 32-bit version of GNU/Linux, you will need to `ssh` to `tempest` to run the generated code. Since you have the same home directory on `tempest` and all lab machines, this is actually pretty transparent. Just keep an `ssh` session open.

Assembling and Linking. *I have provided a simple* `Link.scala` *file that will allow you to run this step directly from your compiler if you wish. Run the compiler using the* `icc` *wrapper script for* `Link.scala` *to work without extra configuration.*

Given an input file `file.ic`, your compiler will produce an assembly file `file.ic.s`. You can then use an assembler to convert this assembly code into an object file `file.o` and the linker to convert this object file into an executable file. While there are separate assembler and linker tools, it is simplest to use `gcc`, the GNU C compiler, to do both steps for us:

```
gcc -m64 -g -o file file.ic.s runtime/libic64.a
```

The library file `libic64.a` (in the runtime directory) 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 via a freely available conservative collector: `http://hboehm.info/gc/`

The `-g` flag to `gcc` will create executables that can be run inside `gdb`, the GNU debugger. The `-m64` flag ensures the compiler uses the `x86_64` instruction set.

You may also find it useful to look at the assembly code generated by a C compiler. You can do this in gcc with the `-S` option. For example, `gcc -m64 -S a.c` generates the assembly code file `a.s`.

Mac OS X. *Ignore this unless you really want to run your* IC *programs directly on your Mac.* Use the `libic64.osx.a` library instead of `libic64.a`. Also, unlike GCC, the assembler in the LLVM toolchain (installed by `xcode-select --install` on Mac OS X and aliased to `gcc`) does not accept absolute addresses as values. This means that if your code generator produces something like this, which uses the absolute address corresponding to the label `__string_literal_13`:

```
movq $__string_literal_13, 8(%rsp)
```

It will need to produce something like this instead, which uses a relative address:

```
leaq __string_literal_13(%rip), %rax
movq %rax, 8(%rsp)
```

GDB. Use GDB to help understand the execution of your generated code. See tools page for more. To run executable `foo` under GDB: `gdb foo`. Typically, set a break point at `__ic_main` or other later point of interest, type `run`, then step through and inspect your program's data from that point on.

Command	Meaning
`b __ic_main` or `break __ic_main`	Set a breakpoint at the start of `__ic_main`.
`r` or `run`	Run the program.
`c` or `continue`	Continue running the program until the next breakxpoint.
`s` or `step`	Execute one line of assembly.
`n` or `next`	Execute one line of assembly. If it is a call instruction, run until the function call returns.
`si` or `stepi`	Execute one instruction (even if compiled from C).
`ni` or `nexti`	Execute one instruction (even if compiled from C). If it is a call instruction, run until the function call returns.
`list`	Print out the assembly code around the current program counter.
`disas`	Print out the disassembled machine code of the current procedure, including a cursor pointing to the next instruction to execute.
`info registers`	Print out the contents of the registers.
`help`	Prints help info.
`x` *address*	Print the contents of the given memory address. Type `help x` for more details.
`bt`	Show the call stack.
`quit`	Quit the debugger.

Schedule

There is one intermediate checkpoint for this phase. Be sure to add comments to your code to document any special cases, describe tricky parts, and provide an overview of how any non-trivial class is designed.

Thursday, April 7: Your compiler must support the `-printIR` option to print out the TAC instructions for each method. Document the design of your TAC instruction set. Your compiler should also complete the first pass in code generation: computing offsets.

Thursday, April 14: Full back end due.

Extras

If your back end is done early, I suggest getting a head start on the next project phase: the optimizer. If you are unhappy with the inefficient *machine code* generated, you could begin work on a variety of improvements to instruction selection or register allocation. Talk to me.