PA 3: IC TAC and x86 Code Generation

CSCI 434T
Spring, 2019

Overview

In this assignment, you will implement:

- a translator to convert your AST intermediate representation to a TAC intermediate representation; and
- a back-end to generate x86 assembly code.

Code generators, while intuitively no more complex than other parts of a compiler, are notorious for subtle bugs. **Start early, design your system before you start coding, and implement and test your code incrementally. You will not get this working if you wait until the day before...**

Implementation Details

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, you are responsible for choosing the particular instructions in each case. The TAC description on the web page (from HW 4) should serve as a reasonable starting point.

You must clearly describe your instruction set in your `README.md`.

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. (Bonus points for having no consecutive labels, no unnecessary jumps, etc. These are optimizations — do not implement them until you have completed the basic assignment). Similarly, you need not worry about reusing temporary names within a method.

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-of expression `a.length`, the compiler must 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 compiler must insert a check to verify that the array size `n` is non-negative.

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`.

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

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. 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 methods, and at the beginning and the end of each function (prologue and epilogue).

  Your simple code generator may not need to worry about callee/caller-save registers, but for completeness the rules are as follows: 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 method call. On the other hand, if a function modifies callee-save registers, it must restore them to their original values before returning.

  The function parameters are pushed onto 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 `%rax` 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 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 newly allocated object. The size `s` of the allocated object must accommodate all of the fields of the allocated object, plus 8 bytes to store a reference to the dynamic dispatch table (vtable). 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 array large enough to store n elements. The size of each array element has a fixed value of 8 bytes and need not be passed to `__LIB_allocateArray(n)`. This is the size for all types in the language (booleans, integers, and references). The array allocation function also stores the array length in the memory word preceding the base address of the array return (i.e., the location at offset -8).

  String constants should 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.

- *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.

- *Library functions.* Calls to these functions are translated into function call sequences using the naming convention illustrated 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 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.

[*]The official x86_64 ABI requires the first six parameters to a function to be passed in registers — you are welcome to implement passing through registers, but you may find it easier to get the compiler working with only stack-based parameter passing first.

Package Structure. You should implement the new components of the compiler in the following sub-packages of the `ic` package:

- the `tac` package for your TAC intermediate representation; and
- the `cg` package for your x86 code generation classes.

For the `tac` package, you are strongly encouraged to implement the TAC instructions as a collection of case classes that all extend an abstract `TACInstr` class, and to build a `TACList` class that stores a list of TAC instructions. After translating to TAC, each method declaration in your AST would then be decorated with a `TACList`. Several other TAC details:

- Note that the operands to most TAC instructions can be 1) program variables, 2) temporary variables, or 3) constants. The design of your TAC classes should reflect this.
- To facilitate deciphering the compiler's output and debugging your back-end, I encourage you to design your TAC classes (and x86 code generator) to support String annotations on individual instructions that can be subsequently printed out as "comments" in your output, 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 in the target program.

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 it 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 "`-printIR`", 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.

Auxiliary Tools

Assembling and Linking. Given an input file `file.ic`, your compiler will produce an assembly files `file.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.exe file.s libic64.a
```

The library file `libic64.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
```

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 can find the library file `libic64.a` along with supporting material on code generation for this assignment on the web site. The additional material includes documentation for the gdb; documentation for the x86 instruction set; and several example IC programs along with the corresponding x86 assembly code.

(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`.)

GDB. GDB is a very powerful, but sometimes cryptic, debugger. You are strongly encouraged to use it to help debug your code generator. The online documentation available on the links page provides many details, but the following are the most important commands for us. To run GDB on a file `t.exe`, simply run `gdb t.exe` on the command line after generating `t.exe`, as described above.

Command	Meaning
`br __ic_main`	Set a breakpoint at the start of `__ic_main`.
`r` or `run`	Run the program.
`s` or `step`	Execute one instruction.
`n` or `next`	Execute one instruction. If it is a function call, run until the function call returns.
`list`	Print out the code around the current program counter.
`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.

Typically, you will start `gdb`, set a break point at main, type `run`, and then step through and inspect your program's data from that point on.

Testing

Test early. Test often. The web site includes a collection of small test files you may use, as well as their expected output and a script to compile and run each of them. You may design your own testing workflow if you prefer, but I expect all groups to have an automated, systematic testing workflow.

Schedule

There is one intermediate milestone for PA3. 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 11: Your compiler must support the `-printIR` option to print out the TAC instructions for each method. Include a status update in the writeup directory, as well as your TAC instruction set.

By this checkpoint, your compiler should also compute the object offset for each declared field; the vtable index for each declared method; and the offset from the frame pointer for each declared parameter and local variable (including the temporaries created during TAC generation). This information can simply be stored in the parse tree on declaration nodes and printed by an extension of your pretty printer.

Thursday, April 18: PA 3 due. Be sure to include in your writeup any important details about the TAC and Code Generation passes, a summary of your testing methodology, and any known bugs.