Project 5: Code generation

Due: TBA

This is it: we finally get to actually dig in and generate assembly code. The input for this step are the decorated trees that have already been semantically analysed, so we know that what remains is a syntactically and semantically valid program, in AST form, with type information marked and type conversion steps baked right in to the tree. The task is thus a tree traversal, whose printed output will consitute an assembly-language file that can be passed to gcc or another assembler for the final conversion to machine code.

You should take as your model the existing `print_with_prefix` function, which also performs a tree traversal and prints things at each node. Indeed, your initial version of your code-generating function could well be an exact copy of `print_with_prefix` (one per node type), leaving you the ability to convert node types to generating code one-by-one. You'll need to carry state during the code generation process, so in much the same way the CLexer object got passed around in `print_with_prefix` (so that all the different versions of the method could use it), you'll need to create a generator-state object and pass it through your code-generating functions in much the same way.

There are no changes to the grammar for this project.

Phase 1: three-address code

The first pass you will make will be to generate three-address code for each different type of expr and stmt. You may want to refer back to pp. 364–65 in the book, where the "rules" of this code format are laid out, but the main additions to the code at this point are as follows:

- A class to represent generator state. As mentioned above in the overview, this object will be passed around by the code generator methods. For this phase, the only thing it needs to be able to do is generate temporary variable names that have never been used before, and jump labels that have never been used before. A reasonable choice for the temp variables would be names of the form "`t_1`", "`t_2`", etc

(these are for compiler-internal use anyway and won't get printed to the assembly file), and for the jump labels (which *do* get printed to the file), you can match what gcc does and use "`.L1`", "`.L2`", and so on.

- To the expr nodes, a variable indicating the (possibly temporary) variable name where its result is stored. This value is only set after the expr's code generation method has run (and is accessed by the expr's parent).
- The code generation functions themselves, which can start out as exact copies of the `print_with_prefix` methods but should evolve. If you want to continue printing the trees themselves, print them to `cerr`, and then just print the three-addres code to `cout`. (These two outputs can then later be separately redirected if you want to look at them.) All the stmt and expr nodes should have some code that they print out.

As I said in class today, the purpose of generating this code is primarily as a checkpoint; it helps me see where you're at, and it helps you separate the issues of making the code sequential and making temporaries from the issues of register allocation and management.

Phase 2: basic assembly

In this phase, you will generate assembly for any program (within the subset) that contains all its code inside `main`.

The largest single portion of what you're doing here is implementing the register management algorithms of section 8.6 in the book. Specifically:

- Your object that keeps track of generator state will need to have two additional maps in it: one that tracks, for every register, what location names(s) it corresponds to; and one that tracks, for every location name, what actual location or locations it corresponds to. See p 545.
- You will also need to write a method (presumably a method of the generator state object) that chooses registers for the three named locations used in a particular statement. See Section 8.6.3.
- Each expr and stmt will need to revisit what it prints (to actually print out assembly code). See p 544.

At this point, you should implement *calling* functions, or at least the subset of them that take a single int argument and don't make use of their return value. This lets you make calls to library functions like `putchar`, which means your programs can actually do something, which will make testing easier. A call to

```
putchar('A');
```

translates into

```
movq $65, %rdi
call putchar
```

where 65 is the ASCII value of `'A'`, and `rdi` is the register where the first argument to a function is passed. (More about this below.)

Phase 3: functions

The last phase of the project involves actually generating code for function definitions, as well as fleshing out the function calling code to work for any function call (not just the simplest ones). Every defined function should start with a label (with the function name), then a prologue that sets up the stack frame, making copies of whatever needs to be copied, making space for any local variables, and updating the stack pointer; then a function body, that translates the actual statements inside the function; and finally a function epilogue, that restores any register values that need to be restored and resets the stack pointer. See Section 7.2.3, and `http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/` (also linked from the course webpage).

Function calls mostly just need to make sure the arguments are put in the correct places, and the return value is found in the correct place.

In particular:

- On a function call, the first argument is placed in `%rdi`, and subsequent arguments are placed in the registers `rsi`, `rdx`, `rcx`, `r8`, and `r9` (in that order), with subsequent arguments (if any) placed on the stack. This convention is specific to 64-bit machines; older x86 architectures passed arguments primarily on the stack.

- Return values from function calls can be found in the register `%rax`.
- For a function definition, you can treat the prologue and epilogue as largely boilerplate (and note that you can get rid of all the `.cfi` lines that are there just to give debugging info). The main non-boilerplate thing is that the amount of space reserved for local variables will depend on how many local variables there are, and how large they are.
- After you print the prologue, you'll update your generator state object and its maps to include the initial location-name correspondences: Local variables are at their respective offsets from the stack pointer, and function parameters are in their respective registers (starting with `rdi`, as laid out above). Note that the first six parameters are not copied to the stack unless you do it—this is part of the function prologue in the gcc-generated assembly.
- A return statement should (after computing its expr's value) copy the return value into `%rax` (or eax, ax, or al), then jump to a label immediately before the function epilogue.

Schedule, handing in

Wednesday is an in-class work day.

By Friday, you should have phase 1 done. I'll talk a bit about phase 2 and/or the stack frames; not a laptop-based work day, but we'll spend the whole time discussing the project.

By Monday, you should have the maps more-or-less in place, with the general framework of Section 8.6 implemented, so that Monday can be spent debugging this (another work day). The hard part of phase 2 is the register management—once that part is done, the printing of the assembly code becomes essentially mechanical.

So by Wednesday the 25th you should have Phase 2 done. Again, we'll have a non-laptop day, where I'll talk about stack frames for sure and we'll generally discuss what happens next in the project.

Friday the 27th is the last day of class (!) and you'll have to submit *some portion of* phase 3 at this point. I will determine just how much, and how or whether this last phase folds into the final exam, after I see how the next week goes.

On torvalds, type

```
handin cmsc445 proj5  compilerdir/
```

to hand in your entire `compilerdir` directory. Be sure to include a readme with any notes I'll need, and your Makefile if you have one.

As with previous project assignments, this was inspired by and adapted from a project I did when I took a compilers class at Brown.