Union College

Assignment #X: Generating Code

Course: *CSC-375 Compiler Design* – Professor: *Aaron Cass*
Due date: *DUE DATE*

Objectives

- Complete the last part of code compilation, generating LLVM assembly from the output of your semantic analyzer
- Pass all tests given in *ir_tests.rs*
- Work in groups of 3-4

An explanation of code generation with LLVM IR: At long last, this is it! This is the step of compilation that you've been waiting for, actually generating code runnable by a computer! All of the steps before this have been preparing you for this point, and it's finally time to take the ASTs that you have semantically analyzed and turn that into code.

If you're here from CSC-270, you might be a little nervous that you have to write assembly out by hand, but this won't be what we do here. Instead, we'll be using LLVM IR (Intermediate Representation) code. LLVM IR is a higher level assembly-like language which can itself be turned into proper assembly for your processor so that you can write the code here and have it work on many CPUs, much like the same Java code can be compiled and run on many CPUs.

LLVM IR Example

Listing 1: code for a while loop in C

```c
int testFunctionWithWhileLoop() {
        while (1) {
            return 42;
        }
    }
```

Listing 2: equivalent code for a while loop in LLVM IR

```llvm
; ModuleID = 'dummy_module'
source_filename = "dummy_module"

define i64 @testFunctionWithWhileLoop() {
entry:
  br label %while_cond

while_cond:
  br i1 true, label %while_body, label %while_end

while_body:
  ret i64 42
  br label %while_cond

while_end:
}
```

This step of code compilation will bridge the gap between code in C and code in LLVM IR, completing the transformation from a program string to usable code. This may seem like a daunting task, but you've been provided with a set of tools that will make writing this out much easier.

In *ir_codegen_core.rs*, the IRGenerator struct is what will be used to generate IR code from a given AST, and it uses another struct called ResourcePools. ResourcePools is the link between your code and generating LLVM code. For example, if you need to create an integer, the *create_integer()* function of the resource pool will handle this for you.

At the heart of this is the tag system. When you call *create_integer()* for example, what it will give back is a ValueTag. Think of tags as unique names that are linked to the object you've created, so when you call *create_integer()* it creates the value, stores it, and then gives you a tag that represents it, sort of like a ticket you can use to get your dry cleaning.

Another key idea is the basic block. A basic block is simply a section of code that you can write instructions into. The code above for the while loop has four basic blocks: entry, while_cond, while_body, and while_end. Each are separate basic blocks that code is written inside.

code walk: ir_codegen_core.rs

This is the file where the main router function that handles generating IR is. This time, a fair amount of this will be given to you completed, as the focus of this part is learning how to generate the IR and not how to write the router function. You can see that the main router match statement is all unimplemented though, and this is where you come in. You must write the functions that the router points to, and generate IR for these cases appropriately.

code walk: ir_primitive.rs

This file contains the functions that generate IR for data types and literal values like integers.

code walk: ir_statement.rs

This file contains the functions that generate IR for statements like returns and breaks, as well as variable assignment.

code walk: ir_block.rs

This file contains the functions that generate IR for complex code blocks like block statements and loops.

code walk: ir_top_level.rs

This file contains the functions that generate IR for function definitions and other top level statements. This is probably the most difficult to implement.

Follow these guidelines to build the IR Generator struct effectively

1. Do not change any of the function signatures, i.e. leave all parameters, parameter types, and return types as you found them.
2. Follow all instructions given in the comments where they exist, and take any potential hints into consideration when making your design.
3. Keep track of tags! Each tag is your ticket to something you created in resource pools.
4. Remember to create a basic block every time you need a new section of code.
5. Remember that at this stage you've already done semantic analysis, you may assume the AST you're recursing over is always correctly formatted at this point!
6. IMPORTANT: Due to the way we use recursion to go over our AST, we had to implement thread safety for the resource pools. What this means for you is that every time you use a function of resource pools, you must first get the resource pools, try to lock the resource pools (expecting a panic error is ok here because otherwise we couldn't continue anyway), work with the resource pools through the lock object, and drop the lock when you're done. A diagram of this process in code is shown below and you can see this for yourself in action in the *generate_literal_ir()* function.

Listing 3: Process for working with the resource pools after locking

```
// Step 1: get the resource pools in a variable.
let resource_pools: Arc<Mutex<ResourcePools>> = self.
    get_resource_pools();

// Step 2: try to lock the resource pools, expecting an error
    if it's already locked.
let resource_pools_lock: MutexGuard<ResourcePools> =
    resource_pools.try_lock().expect("Can't lock!");

// Step 3: do what you have to do with the resource pools
    through this lock.
resource_pools_lock.create_integer(FUNCTION ARGUMENTS);

// Step 4: drop the lock when you're done with modifying the
    resource pools so you don't cause locking issues
drop(resource_pools_lock);
```