CS316: Compilers Lab

Programming Assignment 5: For Loop and Functions, Due: 31/3/2021

1 Introduction

Your goal in this step is to generate executable code for loops and programs with multiple functions. You have already generated code for altering the flow of control in case of 'if' construct in the previous assignment. In this assignment, you will generate code for the 'for' loop. In addition, you will generate code for programs with multiple functions. Generating code for functions means that you will have to handle two aspects: (i) what should a *caller* function do to prepare for calling a subroutine; (ii) what should a *callee* function do to set up its local variables and environment?

2 Function calls

The primary mechanism for handling function calls is the *program stack*, which is where the local environment (*activation record* or *frame*) for each currently executing function (i.e., functions that have started executing but have not yet returned) is stored.

2.1 Activation records

An activation record, or frame, stores all of the data required to execute a function. In particular, this means that the activation record stores all of the local variables in a function.

We declare global variables with `var` *declarations in Tiny code, but that doesn't work for local variables. Why? Because a local variable is specific to* that *function invocation – it's not global. Consider what would happen if you wrote a recursive function: the two versions of that recursive function each need their own copy of their local variables.*

An activation record is delimited by two "pointers": the *stack* pointer (which is controlled with the instructions `push` and `pop`) and the *frame* pointer (which is controlled with the instructions `link` and `unlink`). The stack pointer points to the "top" of the stack, while the frame pointer points to the "base" of the activation record.

Local variables (as well as arguments to a function, and its return value) are assigned "slots" on the stack relative to the frame pointer. When you access a local variable `x`, you won't access a memory location named `x` (as if it were a global variable); instead, you'll access a memory location that is "3 slots below the frame pointer"

In our stack organization, the stack conceptually grows "down". Local variables thus have negative offsets from the frame pointer, while arguments and return values have positive offsets from the frame pointer

You will need to augment your symbol table to maintain a mapping between each local variable and its slot in an activation record. (Don't forget to reset the slot counter for each new function!)

We recommend that you draw out the program stack for a simple program to understand how to correctly generate code for it.

2.2 Implementing a function call

You can divide up the work done for a function call into two responsibilities: those of the *caller* and those of the *callee*. Here is what each one needs to do:

Caller before the call

1. Push any registers that you want to save on the stack (using `push`)
2. Push a space on the stack for the return value of the callee
3. Push any arguments onto the stack
4. Call the function (using `jsr`)

Note: in some of the outputs, step 1 is performed after 2 and 3; this is fine, as long as you are consistent and are able to correctly know where arguments/return values are

Callee

1. Allocate space on the stack for all the local variables (using `link`)
2. Generate code, accessing local variables and arguments to the function relative to the frame pointer (Use `$-n` to access slots below the frame pointer, with n replaced with the slot location, and `$n` to access slots above the frame pointer)
3. When returning from the function, save the return value (if any) in the appropriate slot "above" the frame pointer (remember how the caller set up its portion of the stack).
4. Deallocate the activation record (using `unlink`)
5. Return to the caller (using `ret`)

Caller after the call

1. Pop arguments off the stack
2. Pop the return value of the stack, remembering to store it in an appropriate place (local variable, global variable, register, etc., as needed by the source code)
3. Pop any saved registers off the stack.

In this step, your code generation strategy likely means that no registers actually need to be saved on the stack by the caller, because none are "live" across the function call. If you choose not to save registers in this step, remember to add that functionality back in for the next step (register allocation)

Testing your Tiny code You can test your Tiny code by using the same simulator as in the previous step. Your compiler will be tested against only the inputs that we provide. *However, in the next assignment, there will be hidden test cases .*

Sample inputs and outputs: inputs and outputs.

3 What you need to do

In this step, you will be generating assembly code for 'for' loops and function calls, as described above. You should correctly be able to handle functions with return values, functions where complex expressions are passed in as arguments (store the result in a temporary, then push that temporary onto the stack as the argument), and recursive functions.

Handling errors All the inputs we will give you in this step will be valid programs. We will also ensure that all expressions are type safe: a given expression will operate on either INTs or FLOATs, but not a mix, and all assignment statements will assign INT results to variables that are declared as INTs (and respectively for FLOATs).

Grading In this step, we will only grade your compiler on the correctness of the generated code. We will run your generated code through the Tiny simulator and check to make sure that you produce the same result as our code. When we say result, we mean the outputs of any WRITE statements in the program (not details such as how many cycles the code uses, how many registers, etc.)

We will not check to see if you generate exactly the same code that we do – no need to diff anything. We only care if your generated code *works correctly.* You may generate slightly different code than we did.

4 What you need to submit

- Merge the branch you created for pa4, `cs316pa4`, into the master branch (-1 point for not doing the merge).
- Create a branch called `cs316pa5` in the GitHub repository that you created for executing PA1-PA4 (-1 point for not creating a branch).
- All of the necessary code for your compiler that you wrote yourself. You do not need to include the ANTLR jar files if you are using ANTLR. *Make sure you name the ANTLR jar file as antlr.jar and work with this in your local development environment*
- A Makefile with the following targets:
  1. `compiler`: this target will build your compiler (-1 for warings)
  2. `clean`: this target will remove any intermediate files that were created to build the compiler (-1 for not doing the clean correctly)
  3. `dev`: this target will print the same information that you printed in previous PA.
- A shell script (this must be written in bash) called `runme` that runs your scanner. This script should take in two arguments: first, the input file to the scanner and second, the filename where you want to put the scanner's output. You can assume that we will have run `make compiler` before running this script (-1 for not providing the script).
- You should tag your programming assignment submission as pa5submission

While you may create as many other directories as you would like to organize your code or any intermediate products of the compilation process, both your `Makefile` and your `runme` script should be in the root directory of your repository.

Do not submit any binaries. Your git repo should only contain source files; no products of compilation. If you have a folder named `test` in your repo, it will be deleted as part of running our test script (though the deletion won't get pushed) – make sure no code necessary for building/running your compiler is in such a directory.