Introduction to Compilers and Translation Engineering

ECE 46800/57300/59500, Fall 2023

Step 4, Functions

last update: 31 August at 11:36pm

Table of contents

Step 4: Functions and Type checking

Suggested date of completion: Oct 20th 2023, 5PM ET

Due: Dec 1st 2023, 5 pm ET

Setting up your repository

Set up your GitHub classroom repository for step 4 using the link on Brightspace.

Set up your project development and testing environment as directed in the project environment documentations.

Then clone your repository as directed in the cloning and submitting instructions.

Background

Up until now, we have been working with a fairly simplified lanugage:

1. Only one function (main)
2. No local variables (all variables are global)
3. No type checking (we assume that all programs are correctly typed)

In this step, we will fix all of these shortcomings.

Functions

The primary background you need for this step is the notes on functions:

1. How to set up the program stack for functions.
2. How to save registers on the stack
3. How to access local variables and function arguments

We have provided new AST nodes (`FunctionNode` and `CallNode`) to facilitate building your function AST. We now focus on code generation.

Program Stack

Whie Risc-V provides a number of registers for passing arguments to functions and returning values from functions, we are going to simplify our task by passing all arguments and return values on the stack. (You may choose to use registers for better performance if you'd like).

To call a function, you must:

1. Push the arguments for the function onto the stack (it will be helpful to define a `pushRegister` helper function to generate this code)
2. Push space for the return value onto the stack.
3. Save the (old) return address register onto the stack (this is always stored in a register named `ra`)
4. Call the function

> It is important to make sure that you push arguments onto the stack in the right order!

When coming back from a function, you must:

1. Restore the old return address into `ra` (it will be helpful to define a `popRegister` helper function to generate this code)
2. Pop the return value into a register
3. Pop the arguments off the stack

In the callee, you must:

1. Save the old frame pointer on the stack (this will always be in a register named `fp`)
2. Update the frame pointer to point to where the old frame pointer is.
3. Save any registers on the stack that this function writes to
4. Allocate space for local variables on the stack

When the callee `return` s, you must:

1. Save the return value on the stack (In our calling convention, the return value always sits 8 above the frame pointer)
2. Restore the saved registers
3. De-allocate the frame (move the stack pointer to point to the frame pointer)
4. Return from the function

> Don't forget to pay attention to the types of arguments and return values to make sure that you're using the right kinds of load operations!

Saving registers

In a callee-saves calling convention, we need to save all the registers that the callee writes to in case the caller also needed to use them. The easiest way to do this is to count how many registers were allocated while generating the code for the function. Note that this has several implications.

First, this means that we do not know how many registers to save until *all* the code has been generated for the function. This means that the register save and restore code should be generated after the body of the function has been processed.

Second, because we cannot generate register restore code until the function body has been processed, this means that it is not possible to know how to restore registers when generating code for a `return` statement. One easy way to deal with this problem is for the `return` statement to include a jump to a single block at the end of the function that performs steps 2–4 above. The jump target can be generated in a `preprocess` method for the CodeGenerator while the actual register restore code can be generated after generating the code for the function body.

Accessing local variables

Local variables and arguments need to be accessed by offsets relative to the frame pointer. The way we have set up the symbol table for you, global variables have absolute addresses stored, while local variables and arguments have relative addresses stored. You merely need to take care that loads and stores address things correctly depending on whether a particular variable is local or global. Note that the `VarNode` code generation stores a pointer to a variable's symbol table entry in the `CodeObject` , so you can use that information to determine whether a variable is local or not, and hence whether the address information is relative or absolute.

Type checking [Optional for 468/595]

The main background you need for type checking is the lecture notes on type checking. Recall that just because a program is syntactically correct does not mean that it is *well-typed*: the grammar does not provide any facilty for ensuring that integers are stored into integer variables, or that functions are called with the right number and type of arguments.

Recursive type checking is easy to implement on an AST: you can write a new visitor class that will walk over the AST and recursively check that each appropriate AST node is typed correctly:

1. Binary expression nodes: the left and right operands should be the same type
2. Assignment nodes: the LHS and the RHS must be the same type
3. Conditional nodes: the left and right operands should be the same type
4. Function call nodes: the number and type of all the arguments correspond to the function declaration.
5. Return nodes: the type of the return expression should match the return type of the function.

> Note: to facilitate this last check, we have tweaked the definition of the `ReturnNode` AST node to include a pointer to the function it is in. You should modify `MicroC.g4` to make sure that the `ReturnNode` is constructed properly. Hint: `SymbolTable.getFunctionSymbol` will get a function symbol table entry given a function name; `SymbolTable.currentScope()` will return the current scope during the parse; and `Scope.getName` will return the name of a scope – which is the function name, if the scope is local.

What you need to do

468/595 students

Add support for functions, as outlined above

573 students

We want you to add support for functions and type checking to your compiler, as outlined above.

You should run your type checker before any code generation. If your compiler detects a type error, you should print:

```
TYPE ERROR
```

to `stderr` and exit with exit code 7. Make sure your `runme` script correctly passes along the exit code of your compiler.

Running your code

To run your code, you can use the RISC simulator we have provided for you. When using the container environment, this is present in ~/RiscSim. When using the ecegrid environment script, the path to the driver script for it is set in $RISCSIM.

Once your compiler has created an assembly file, you can run it like so:

- In container: `python3 ~/RiscSim/driver.py [assembly file]`
- On ecegrid: `python3 $RISCSIM [assembly file]`

Sample inputs and outputs

The inputs and outputs we will test your program on can be found here.

For 468/595 students, all inputs in base of `tests/` directory are correctly typed, and these are the only tests we will test in the grading process.

For 573 students, please check the `tests/type_error/` directory for incorrect inputs that should trigger your type checker. We will check **both** correct and incorrect inputs when grading.

As usual, there is a testall script that can be run with './testall'. To include the type-check tests, run it with './testall typecheck'.

What you need to submit

- All of the necessary code for building your compiler.
- A Makefile with the following targets:
  1. `compiler` : this target will build your compiler
  2. `clean` : this target will remove any intermediate files that were created to build the compiler
- A shell script (this *must* be written in bash, which is located at `/bin/bash` on the ecegrid machines) called `runme` that runs your scanner. This script should take in two arguments: first, the input file to the compiler and second, the filename where you want to put the compiler's output. You can assume that we will have run `make clean; make compiler` before running this script, and that we will invoke the script from the root directory of your compiler.

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.

See the submission instructions document for instructions on how to submit. You should tag your step 4 submission as `submission`

Most course announcements will be posted on Piazza and/or the corresponding webpage. Only a few critical announcements will be posted on main page.