ECE 468: Intro to Compilers and Translation Systems Engineering

Fall 2017

MWF, 1:30–2:20, WALC 3087

Project Step 5 — Control Structures

Due: November 1st

This step builds on Step 4. Now that we are able to generate code for lists of statements, what happens if those lists of statements are embedded in control structures? (IF statements and FOR loops)?

(Note: as in step 4, we will only have one function in our program, `main`.)

ASTs for Control Structures

ASTs for control structures are, intuitively, simple: each control structure will have several children (3 in the case of an IF statement, 4 in the case of a FOR loop) that are themselves ASTs (ASTs for statement lists in the case of the bodies of IF statements and FOR loops, ASTs for conditional expressions in the case of the conditions in the IF statements and FOR loops, etc.). Because you already have working code for building an AST for statement lists (and can readily adapt your code for binary expressions to build ASTs for conditional expressions), all you have to do is create semantic actions for the control structures that "stitch together" the existing ASTs.

Generating 3AC for Control Structures

Generating 3AC for control structures builds directly on your code for step 4, which is able to generate code for lists of statements. This means that when you are generating code for an IF AST node, you know that the 3AC for the three children already exists. All that is left is to put them together in the correct order (see the Lecture 5 notes for details on this) and insert any necessary labels and jumps.

There are two things that you need to pay attention to when putting together 3AC:

Generating Labels

At various points in your code, you will need to insert labels and jumps to allow control to transfer from one part of your code to another. You will need to make sure that you can generate *unique* labels every time (since your code will not work properly if there are multiple labels with the same name).

The 3AC you will generate for labels looks like:

```
LABEL STRING
```

Where STRING is whatever name you decide to give to your label

Generating Jumps

Unconditional jumps (like you might use to jump over an ELSE block) are easy:

```
JUMP STRING
```

Where STRING is the label you want to jump to.

Conditional jumps are a little bit tricky in our 3AC (and in Tiny): you need to generate the right kind of jump:

```
GT  OP1 OP2 LABEL (Jump to Label if OP1 > OP2)
GE  OP1 OP2 LABEL (Jump to Label if OP1 >= OP2)
LT  OP1 OP2 LABEL (Jump to Label if OP1 < OP2)
LE  OP1 OP2 LABEL (Jump to Label if OP1 <= OP2)
```


Quick Links

Home
Syllabus (PDF)
Piazza
Blackboard
Calendar

Course details

Instructor
Milind Kulkarni
milind 'at' purdue 'dot' edu
EE 324A

Office hours:
- Mondays, 2:30–4:00 PM
- Thursdays, 10:00–11:30 AM

Teaching Assistanta
Chris Wright
wrigh338 'at' purdue 'dot' edu

Instructional Lab
Location: EE 207
Hours:
- Wednesdays, 5:00–7:00 PM
- Fridays, 3:00–5:00 PM

Assignments

Submission instructions
Step 0: Test submission. Due 8/25
Step 1: Scanner. Due 9/6
Step 2: Parser. Due 9/15
Step 3: Symbol table. Due 9/29
Step 4: Expressions. Due 10/18
Step 5: Control Structures. Due 11/1
Step 6: Functions. Due 11/17
Step 7: Register Allocation. Due 12/4

```
NE  OP1 OP2 LABEL (Jump to Label if OP1 != OP2)
EQ  OP1 OP2 LABEL (Jump to Label if OP1 == OP2)
```

To generate the right kind of jump for a conditional expression, you will need to inspect the AST node for the comparison operation, and use that information to select the right 3AC instruction.

> Note: the 3AC does not preserve type information about what kind of comparison you are doing, but the Tiny code for jumps does; you may want to extend either your 3AC or your data structure to keep track of this information.

Testing your Tiny code
You can test your Tiny code by running it on our Tiny simulator. This simulator can be built by running:

```
> g++ -o tiny tinyNew.C
```

You can then run a program with tiny commands using:

```
> ./tiny <code file>
```

What you need to do

In this step, you will be generating assembly code for IF statements and FOR loops. Use the steps outlined above to generate Tiny code. Your compiler should output a list of tiny code that we will then run through the Tiny simulator to make sure you generated the right result.

For debugging purposes, it may also be helpful to emit your list of IR code. You can precede a statement with a `;` to turn it into a comment that our simulator will not interpret.

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).

Sample inputs and outputs
The inputs and outputs we will test your program can be found here. Note that in the `inputs` subdirectory, files with extension `.micro` are the test programs, and files with extension `.input` are sample inputs that work with the test programs (i.e., provide inputs for READ commands).

A sample compiler (a .jar file that you can invoke with `-jar`) is available here.

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.

Extra credit
For full credit on this assignment, your generated code merely needs to work properly. We will not consider how fast your code runs. *However*, we will also evaluate how fast your Tiny code runs (the "Total Cycles" reported by the Tiny simulator).

The groups whose generated Tiny code runs fastest (averaging across all the inputs) will receive bonus points for this step: 15% for the fastest code, 10% for second, and 5% for third.

What you need to submit

- 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.
- 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
    3. `team`: this target will print the same team information that you printed in step 0.
- 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 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.

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 or test cases. 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.

You should tag your step 5 submission as `step5-submission`