Milestone #2: CFG & LLVM

Overview

For this milestone, you will complete your first compiler, but will do so in a manner to set up the framework for later milestones. There are two major, interlinked aspects of this milestone: the control-flow graph (CFG) and the linear intermediate representation (IR). You will define your own representation of a control-flow graph based, in part, on discussions in lecture. You are required to use the LLVM assembly language for your linear intermediate representation.

This milestone requires that your program transform the abstract syntaxt tree, for each function, into a control-flow graph with LLVM instructions contained within each basic block. Your program must then support printing a valid LLVM representation of the input program to a file.

Though the control-flow graph and linear IR are linked, you can, and likely should, approach the implementation of this milestone in distinct steps. Some suggestions follow.

Part 1: CFG

The control-flow graph depends only on the control constructs in the language (since this language does not support short-circuit evaluation) and, thus, can be created without concern for the contents of the basic blocks (the nodes of the CFG).

Construct a control-flow graph for each function. You will likely want to add labels (as appropriate according the LLVM requirements) to the nodes at this point.

How can you verify that your graph is correct? You control your test input, so you need only find a way to visualize the constructed graph. This visualization can be a plain text dump of each block's label and the labels of those blocks to which it connects, a graphical display of the graph using the DOT language for GraphViz, or a three-dimensional rendering of the graph (ok, the last is clearly going too far).

Part 2: LLVM with Stack Allocation

Complete the implementation of this milestone by filling each basic block with the appropriate LLVM instructions. This is done by converting each portion of the high-level language into its corresponding LLVM instructions. You may take a simplistic approach to this conversion based on the code shape discussion in lecture. You are, of course, free to use more advanced patterns to generate improved LLVM, but are not required to do so.

Recall that we will use 32-bit integers for this project.

Note that your translation, at this point, should use stack allocation for all variables declared in the source program. Intermediate values can be placed in registers, but any variable for which a value might change will be on the stack. This simplification is removed in the next milestone where you will convert to static single assignment (SSA) form.

Command-line Options

Provide a `-stack` command-line option. If this option is present, and assuming the source program is valid, your program must output the LLVM representation of the program. It is common that the name of the output file match the name of the input file, but with an extension of ".ll".

Building an Executable with the `clang` Compiler

You should now be able to run `clang` on this file to generate an executable. You can do so on the department servers with (assuming a file named `output.ll`) `clang -m32 output.ll` (note that if you are using a utility library written in a C file, for print/read, then you will include this on the command-line for clang as well).