

## CS350 Lab 6: Code Generation for Simple Expressions

Change the definition of `which_cs350_lab` in `tiger.cc` so that it has the value 6, and do the work below in the `generate_HERA.cc` file or other file you create and call from `generate_HERA`.

Implement tree-based expression register allocation and simple code generation, going directly into HERA assembly language. You should generate code for constants; calls to the `printint` procedure from my extended version of the Tiger standard library; integer expressions involving `+`, `-`, `*`, `/` (using `div`), `mod`, `<`, `<=`, `=`, `>=`, `>`, and `<>`; sequences; `if` expressions; and string manipulations using the standard library functions. Use the “Sethi-Ullman” optimal tree register allocation algorithm. Your compiler should print a sequence of HERA instructions that correspond to the input program; the sequence should be correct but it does not have to be efficient. Remember to include one `SETCB()` instruction at the start of the HERA program, so that you won’t have to worry about setting or clearing the “carry flag” before each arithmetic operation.

As always, you should take this in small steps, testing at each point. You can test the output of your compiler by saving it in a file (e.g. `test.hera`) and using the `HERA-C-Run` command from the directory `/home/courses/bin`, e.g. `/home/courses/bin/HERA-C-Run test.hera`. You can debug HERA programs by cutting and pasting them into your HERA project from CS245 or by inserting calls to the HERA-C simulator instruction `dump()` into your HERA program.

You may assume that the AST is already in “canonical form”, which means *at least* that there is no place where the result of one function call appears as an argument to another (we will deal with this by re-writing our programs rather than writing a tree canonicalization pass, so don’t fret if your program can’t handle things like  $(16/4)/(14/7)$ ). If you require some other property of the tree, you *must* document it in your design document. If your code generator does not require the “no calls in parameters” invariant, please note that in your documentation too.

Record important design decisions in the file `Design_Documents/Lab6-Expressions`.

For this lab, do not address variables, records, arrays, function definitions, `while`, or `for` (you will work on these in later labs). For 70% credit, handle integer expressions but not strings.

*OPTIONAL:* If you wish (and for a to-be-determined amount of bonus credit), you may improve on this basic system by either

- a) implementing a variant on the register allocation algorithm that sends information about result registers back *down* the tree, to avoid the copy (“move”) instruction that occurs when the Sethi-Ullman algorithm has to generate code for two subtrees that need the same number of registers (but remember that this improvement is like “being able to ride a tricycle much faster”), or
- b) implement the “constant folding” optimization (e.g., generate the same code for the expression `if 3 = 2+1 then 1+2*3+4*(5+6) else (17 - 2)/2` that you would have generated for the expression 51), *but only if* you also do all the work to handle non-constants (you should not have to rewrite your code generation for `+` when we introduce variables in the next lab).

I strongly recommend that you do not attempt either optional step until you have the basic lab working, and that you do a “Team->Commit” before starting on optional work (as well as when you’re done with the lab).