Topic

- **Code generation and "optimization"**
  - **(notes from lecture)**
    - optimize mult: arg signs, arithmetricks, in-lined shifts & adds, shift and sub for runs of 1's
    - distributive rule and mult-elimination: not likely usable, but maybe in array indexing
    - also, note that "responsible programmer wouldn't write a*b + a*c" also reflects back on our progr
    - aggressive optimization comes at a cost: compiler call flags a good idea
    - do one extreme example (63 * x^2 +7*(x=READ) + (y=12))
  - **Context**
    - we have seen how to construct a ***parse tree*** using a stack (or the more concise ***abstract syntax*** use this representation of program structure to implement its "meaning" as a computation
    - specifically, we have seen how we can ***interpret*** the tree ***dynamically*** (e.g., reading input intera corresponding expression and ***static translation*** (or compilation) to generate code
    - in general, interpretation and code generation involve walking the tree in some pre-determined ord respect the meaning of the language) and generating a sequence of instructions (or several sequen prologue/main body/epilogue)
      - (in more realistic compilers a ***code*** **graph** is often used, in order to reflect more complex lang
    - finally, we have seen how we might generate code for different machines, either directly for the "re ultimate target or indirectly through an ***abstract machine*** such as the ***stack-environment m***
  - **Problems and strategies**
    - we are now faced with some decisions about what techniques to use to implement various features the target machine (or some intermediate machine)
    - as discussed in lecture, we may choose **simpler** strategies (which generally have less coverage but more **complex** ones, or we may choose to **vary** our techniques on a case-by-case basis
    - in addition to general issues of "aggressiveness" (how much work to do to try and optimize), we m specific issues and interaction between features (e.g., competition for scarce resources such as reg
    - ***optimizations*** (a misnomer: they might not actually be optimal) can either be done on the tree b during the generation process or on the resulting instruction sequence (perhaps even different mod stages of intermediate code)
    - in many cases a ***static analysis*** of the program (computing some kinds of measures, statistics o source code itself) will be necessary or useful for determining what optimizations to try
  - **Modifications to the tree**
    - ***static evaluation:*** we can analyze the tree to find sub-trees which have no variable references or them with their actual values
      - (we must take care to ensure that our compile-time arithmetic accurately models the run-time e
    - ***variable value propagation:*** we can extend the idea above to include the values of variables, a to evaluation order and the changes in a variables value
    - ***sub-tree reordering:*** we have seen that a stack-based approach is biased in that right-leaning t leaning ones; we may want to try and re-order the tree using, e.g., associativity and commutativity
      - (again, we must be careful about changing the order of evaluation of variables, inputs, etc.)
    - ***algebraic rewriting:*** in addition to the use of associativity and commutativity, we may want to u distributivity laws in order to consolidate results or to (e.g.) reduce expensive operations like multi
  - **Handling multiplication**
    - generally speaking, we might either choose to write multiplications ***in-line*** or to call a ***sub-routi***
    - we have code already to handle multiplcation (from lab), but note that, for example, handling nega expensive (in run time, code size and register usage)

·amming

: ***tree***) and how we can

:tively) to evaluate the

ler (usually in order to
:es, in the case of

uage features)

al" machine which is our
achine used in class

of the source language on

are more efficient) or

ust consider feature-
isters)
efore we generate code,
lifications on different

or other data on the

· inputs and then replace

environment/machine)
s long as we are sensitive

rees are preferred to left-
properties

use identity, zero and
plication

ne

tive numbers was