Random Compilers

Design (new changes; see below for old)

The main change to the design after the codegen submission is a partial implementation of DCE. In implementing this, there were two main visitors added to the IR. First, there was a USEVisitor, which, given an IRNode, will return the set of variables used by that expression. Similarly, there is a ASSIGNVisitor, which will do a lookup for variables which have been assigned values by the IRNode. Both of these are implemented with simple recursive calls, as the logic in determining both values is straightforward. Right now, these are being used in DCEVisitor, found in the 'cfg' folder. This visitor is responsible for modifying the set of live variables at each CFGLine and returning whether something has been modified. This logic then feeds into a method in CFG itself, which is responsible for the global implementation of DCE. It moves up from the bottom of the CFG updating each CFGLine, then when that finishes, goes through again and replaces any unneeded assignment statements with NoOps. We chose to do this directly as the structure of the call does not depend on specifics of the nodes beyond how they transform in data into out data. So, it makes sense to have that grouped with the rest of the code. As many of the compiler improvements have a similar structure, this code will be abstracted for use in more situations, making adding compiler features much easier. The DCE feature is currently not working (see Difficulties), and so we are unable to have any examples of what it does, as this may significantly change before the final version.

We also refactored the old code as planned, particularly shifting the IR to use the visitor pattern.

Design (copied from P3 documentation as a reminder)

The "cfg" folder of our project contains all the classes needed to make a CFG representation of the code. Each basic element of the program has an associated CFGLine, and these are clumped together into CFGBlocks which have lists of CFGLines within them. (CFGBlock is technically a subclass of CFGLine with the extra field for a list of lines, so that they can take advantage of common functionality.) CFGLines know their true and false children CFGLines (which are the same for non-branching statements) and how many parents they have (to determine whether they are a merge of control from multiple previous statements). The CFGCreator file goes through the IR in a single pass to make the CFGLines for each object and add pointers connecting them to each other, then makes another pass to condense these into blocks, starting a new block whenever it sees a branch or merge. Meanwhile each CFGLine is assigned a block, so it can point to the block it is in.

There are various types of CFGLines: CFGExpression, CFGDecl, CFGMethodDecl, CFGStatement, and CFGNoOp. Each stores the relevant IR object. As CFGCreator runs through the program, it keeps a stack of loops that it is currently in (one for each for/while statement it has entered) and where they start and end, so that when it encounters break or continue statements it can cause that node of the graph to point to the correct next line. Expressions are short-circuited as described in lecture notes.

The actual code is built in two files. There's Assembler, which takes in methods and an outputstream, and is in charge of making sure the entire assembly is generated and output to

the stream. It does this by adding a few global things and then creating a different (misnamed) BlockAssembler for each individual method's code generation. The BlockAssembler stores a number of important things about the method: how many variables it has allocated, a table for them, how many blocks or string literals it's written so far (so it knows how to number the next one), what type it's supposed to return, etc. It starts with the first block of the method, generates code for it, then does a DFS on the block's children generating code for anything that's not already in the map from labels to code, and placing jumps to the label if it is already in the map. Within blocks it does an ugly mess of switch statements and helper functions to figure out what sort of thing it is supposed to be handling and handle it.

The code currently only uses registers %r10 and %r11; it's assumed that calls to helper functions will generate code that puts the desired temporary value into %r10. Other registers are basically only used for passing arguments. Unary expressions currently handle "not" by subtracting the boolean 0/1 value from 1 and "negate" by subtracting from 0.

Another design choice we used was having many of the same types present in the IR, symbol tables, and control flow graphs. While this cut down on the code we needed to write, it has introduced a lot of interdependence between the different parts of the code. While this code eliminated some storage overhead, it has definitely made the logic more complicated, possibly adding more work than the overhead would have been. This is also another likely candidate for refactoring.

Extras

None

Difficulties

Dead Code Elimination seems not to be working and creates a weird bug where temporary variable names directly end up in the file. We solved many previous bugs including "it eliminates all code because it iterates improperly and ends up terminating with all lines' liveness equal to the empty set" and "much of our line file needs to be refactored to get this working." We're still not sure where is best to call dead code elimination and have it awkwardly stuffed into the CFG and CFGCreator files, so that could use refactoring. Overall comments on our design of it would be appreciated, with the understanding that we still need to work on debugging it. (Since it's not working, we commented out the line 40 in CFGCreator that enables it; feel free to uncomment.)

Contribution

Arkadiy - worked on refactoring some of the old code to improve the code and make future work easier to do. In particular, redoing the IR to use visitor pattern and figuring out how to make everything there work. Also working on some bug fixes from the codegen portion, especially with ternary expressions.

Jackie - implemented variable initialization, tracked down and made a list of bugs to fix, fixed a couple. Then wrote the DeadCodeElimination and associated visitors entirely in one day (after getting lots of design input from the other two) and debugged it a bit.

Maya - worked on fixing the variable renaming system so that variables shadowing outer scopes would work properly. Fixed our exit statuses so we could pass dataflow tests. Worked on debugging and refactoring DCE code to hopefully get it to work, but no success yet.