Team Phase 3 Documentation Notes

Design

Compilation is broken into four passes:

1) Parsing
    a) We used ANTLR to scan and collect tokens (package `grammar`)
    b) We implemented a custom top-down lookahead parser, that builds an abstract syntax tree (package `parser`)
2) Semantic Checks and Symbol Tables (package `inter`)
    a) Our symbol tables (generated in the .inter package) are extensions of hashmaps from strings to descriptors, and are implemented close to the design shown in class. Our program descriptor contains references to a method table, import table, and field table. We also have local tables attached to our AST Blocks, to represent the variable scope which the block is in. Each local table contains references to its "parent table," or the enclosing scope.
        i) We had to go back and add StringLitCollector —- TODO
    b) Our semantic checks are implemented using a Visitor pattern. Each visitor uses global variables to keep track of its progress traversing the AST created during the Parsing phase. One weakness of the choice to only have symbol tables point to their parent table (and not the other way around) was that we had to create a "local table stack" to keep track of the current scope when traversing the AST. Otherwise, each semantic check was relatively simple to implement over the AST abstraction, with the necessary type and
        i) In a future pass, we will move the semantic check visitors into the inter package, and keep only the ASTVisitor interface in the visitor package.
            (1) The interface was originally named Visitor, before we decided to build the similar CFVisitor interface for our control flow graph.
3) Control Flow Graph Generation (package `assembly`)
    a) Our overall strategy for assembly code generation is to build a control flow graph for each method from the AST (`MethodCFGFactory`) Control flow graphs are represented by objects (CFNodes). A control flow graph looks like a sequence of CFNodes hooked up by forward and parent pointers which represent the flow of control. We distinguish CFBlocks from CFConditionals, but have all nodes implement a DFS interface (dfsTraverse) for easy generic traversal. Nodes in the top level control flow graph can contain a mini CFG with no CFReturn, CFBreak, or CFConditional statements, which evaluates nested expressions.
        i) Our control flow graphs are built in two passes. In the first pass, we hook up statements (CFBlock, CFReturn) with control flow structures

(CFConditional, CFContinue, CFBreak). In the second pass, we populate nodes that could contain short-circuitable subexpressions (CFBlock, CFConditional, CFReturn) with "mini CFGs" that correctly implement short circuiting on those subexpressions. The second pass is done in TempifySubExpressions. In both passes, we run MergeBasicBlocksAndRemoveNops to put nodes together using the peephole removal strategy shown in lecture. To do so, we must maintain parent pointers.

ii) The CFStatement classes represent atomic control flow operations that operate on a small number of temps or variables, which are stored on a CFNode.

b) The design choice of implementing TempifySubExpressions leads to some extraneous operations but guarantees correctness in short circuiting.

i) Also, our miniCFG scheme has the desired attribute that all temps can be allocated just for that one miniCFG and we don't have to maintain their memory across CFNodes in the outer CFG.

4) Assembly Code Generation (package `assembly`)

a) Program: AssemblyFactory handles code generation for an entire program. In the global scope, we allocate space for global fields, as well as any strings that are used throughout the program. Additionally we provide code for prematurely exiting due to runtime errors.

b) Method Calls: Here, we use a visitor (MethodStackOffsetsPopulator) to set stack offsets for local variables as well as temps. Each unique local variable in the method (shadowed variables are considered unique) has its own space on the stack. We also use a visitor (TempOffsetAssigner) to allocate enough space for temps within each outer CFNode. We follow standard method call conventions for the prologue and epilogue for method calls. Then, assembly for the body of each method is produced from the control flow graphs. Each CFNode mentioned above has a unique toAssembly method that handles and returns the corresponding assembly code. (The CFStatements within nodes also implement the toAssembly function.)

c) To implement runtime checks, we maintain a global flag which we set and check after every `call`. If the flag is set, we immediately return and leave the %rax intact. This way, in an arbitrary method call stack, we always leave the return value intact so that the error code of the function that threw an error can be preserved. We also considered waiting until return time to check the flag — however, in an infinitely looping program that had a runtime error, we would incorrectly loop instead of throw.

Extras

To make our process more iterative, we implemented an extra target: target=cfg, which prints out our CFG representation for the graph. It prints out each node on its own line, printing miniCFGs, and each CFNode has a UID.

Our UIDObject class provides UIDs to nodes, statements, and scopes, which makes our graphs more readable. Unique IDs are assigned using a static counter.

We also print error messages for runtime errors in addition to returning exit codes.

Difficulties

Our CFG implementation went through multiple iterations. We were thinking of making an extra MiniCFNode class to better distinguish between inner CFGs and outer CFGs.

We also realize that our label names could theoretically collide with imports. For example, we could loop through all imports, determine the one with the longest length, and append that number of underscores to all of our internal labels. We were thinking this was mostly unnecessary because nobody names their imports "_block_30."

We generate a lot of extra assembly code, and look forward to removing it during the optimization phase.

Contribution

We each contributed code whenever we were available to do so. Much of the work was done in the last week as we had to redo our CFG structure to accommodate TempifySubExpressions.