Phase 3 Design Document

Nitya Babbar, Sean Huckleberry, Evan Andrews

I. Design Choices & Implementation

A. Scanner and parser

The scanner was implemented in `scan.rs` simply by composing two main types of functions: `gobble_whitespace` and `lex_literal`. The first type of function was used for consuming any whitespace or comments found within the source Decaf code, which were considered unimportant to the compiler. The second type of function was used for building up the token string that would be passed into the parser. We defined a variety of token types within `token.rs`, corresponding to the types of tokens present in legal decaf. The scanner loops through the characters in the source file, consuming text with functions like `gobble_whitespace` or `lex_literal` until it reaches *eof*.

The parser was implemented in `parse.rs` with `ast.rs` providing the main data structures that the parser used to convert the stream of tokens into an AST. The structure of both of these files mirrors very closely the grammar provided in the Decaf spec. The parser was built with the *nom parser combinator*, which let us mitigate complexity by building many small parsers corresponding to the simpler productions of the grammar, and then composing them to parse the entire program. An advantage of this parser combinator is that we didn't need to implement backtracking or lookahead parsing manually – this was taken care of by the parser combinator. On the other hand, we gained greater control over our parser by implementing it manually instead of using a generator.

B. High-level IR

During Phase II, we converted our AST representation into another high-level IR to perform semantic checking. This IR, known as the *symtable IR* ("symbol table IR"), is defined in `symtable.rs` and constructed in `semcheck.rs`. In comparison with the AST representation, this IR prioritizes information central to semantics, primarily by augmenting the AST with scopes. Each `SymProgram`, `SymMethod`, or `SymBlock` has an accompanying scope, which supports the insert and recursive lookup of local variables. The scopes have parent pointers to enable this recursive lookup.

C. Semantic checker

Our semantic checking takes place over two stages: (1) *during* construction of the symtable IR (`semcheck.rs`), and (2) *after* the construction of the IR (`traverse.rs`). Originally, in order to reduce code complexity, we opted to completely modularize the construction of the semantic IR from the semantic checks. However, we quickly realized that some semantic checks were much more cleanly implemented during construction of the IR, which led to the design decision to spill semantic checking across both of these two stages. Both files have the same code structure: they

recursively match on increasingly-specific elements of the abstract syntax tree to traverse it cleanly.

D. CFG / low-level IR

Our CFG IR was generated directly from the semantic IR discussed above. The main supporting structures to our CFG are `BasicBlock` and `CFGScope` implemented in `cfg.rs` as well as `Instruction` implemented in `tac.rs`. `Instruction`s were designed to map cleanly to one or more x86 instructions, `BasicBlock`s were vital for organizing labels and code blocks within our methods, and `CFGScope`s were used to keep track of the temporary variables that we assigned to each local variable in the program. Originally, we were simply using the original names of local variables, but we realized that we could have problems with shadowing in the generated code.

Our CFG is created in `buildcfg.rs`. The bulk of the CFG construction is driven from `destruct_method`, which recursively calls `destruct_statement` and `destruct_expression`. Each of these functions adds instructions to the given `BasicBlock`, creates new ones as necessary, and returns the final `BasicBlock` at the end of its execution. Global *temp* and *block* counters are used to give each temporary variable and `BasicBlock` a unique ID. Each time we add a new temp variable, we assign a stack offset for it. For now, we allocate 8 bytes for everything no matter its type (we do not differentiate ints and longs yet). We also allocate an additional 8 bytes at the beginning of each array to store its length.

As we build our CFG, if we encounter any string literals, we store them in a data structure so that we can add them to the data section during code generation. Similarly, we have a data structure for all the global variables in the program.

Short-circuiting is implemented on all conditionals (not only in control flow statements). We originally considered splitting the CFG generation and linearization of our IR into two stages, but found that this wasn't necessary since our CFG was robust enough to use directly for code generation.

E. Assembly code generator

After generating the CFG IR, assembly code generation primarily consisted of matching on instructions within our CFG and mapping them to concrete x86 instructions. Important data structures defined for code generation involved `X86Insn`, `X86Operand`, and `Register`. Defined in `x86.rs`, these structures represent the final state of the code, which can be directly emitted by the display formatting that we defined for each of them.

Of course, some additional key considerations in code generation included adhering to calling convention and maintaining stack alignment for imported function calls. For now, most of our instructions use `%rax` as a working register (and `%r10` to store indices for array accesses) with all

intermediate values stored on the stack, so we didn't have to consider caller or callee-saved registers, though we did need to consider where to place our function call arguments and return values. A key design choice in CFG construction made stack alignment much more manageable. Instead of pushing and popping from the stack constantly as temps were allocated, we stored the total stack size needed for a given method during CFG construction, and allocated that entire stack frame at the start of the method call. This meant that we could simply 16-byte align that single allocation instead of worrying about aligning every variable allocation!

F. Testing infrastructure

Our local testing infrastructure consists of cargo testing and shell script testing. For earlier phases, the infrastructure provided by cargo was incredibly useful and easily augmentable. The code generation tests were a little more difficult to run since most of our teammates were on arm architecture, so we designed a shell script to be executed from a virtual machine (OrbStack or Athena) that generated x86 code using our compiler, compiled it with gcc, and then executed it and compared the results to the expected results. Both testing strategies proved effective, efficient to run, and easily expandable for regression testing.

Due to the addition of private test cases, our test coverage became much stronger during Phases II and III (many of our test cases can be seen in *tests/semantics* and *tests/phase3*). For Phase II, we created test cases corresponding to each semantic rule to ensure basic coverage of each rule. Then, we also created tests for larger programs that violated many semantic rules to ensure that our compiler was able to recognize and note them all. For Phase III, in particular, we recognized that our initial test cases weren't thorough enough, as even after passing all of our tests locally we were missing quite a few private test cases on Gradescope. To this end, we used the names of the test cases we were failing on Gradescope as a guide to create additional test cases, some of which caught additional errors, while others didn't.

II. Extras

As mentioned in the previous section, *nom* was a very interesting Rust library used to build the parser that provided a good balance of control and support. We recommend this to future groups building a parser in Rust. One clarification we were interested in was understanding whether we are correctly writing messages to `stdout` and `stderr`. We've been using the writer provided in the skeleton code to interface with the outside world, which seems to be functioning properly from our side, but aren't sure if this is being used exactly as intended.

III. Difficulties

1. *Ints and longs:* Our compiler currently treats everything as 64-bit longs. This is something we hope to update early into Phase IV as we recognize that this will likely be a key differentiation required for some optimizations.

2. *Failed test cases:* There are four failed test cases for Phase III that we were unable to recreate with our own testing efforts. We intend to debug these as soon as the private test cases are released for this phase.

IV. Contribution statement

Work was primarily completed asynchronously, but in an efficient manner and with good communication between teammates. The primary responsibilities for the project have included designing and defining data structures, developing based on these data structures, and testing. The great thing about Rust has been that once we have met and agreed on a shared set of data structures, there are often huge match arms requiring code contributions that can be completed asynchronously. This has enabled us to divide up work generally equitably by assigning match arms or function implementations to different members of the team.