Phase 3 Design Document

Design

Scanning & Parsing

The scanner was written manually (no using ANTLR or similar). It consists of a loop that repeatedly looks at the first one or two characters of the input and then, based on that, decides what token to try to scan and calls the appropriate helper function, e.g. "scan_str_lit".

The parser was also written manually (no using ANTLR or similar). I defined a "parsing function" to be a function of type "token-iterator -> Option<T>". Then I defined functions parse_or, parse_concat, parse_star that take parsing functions as input and return another parsing function. Writing the recursive-descent parser (parse_expr, parse_lit, etc) in terms of these basically amounted to copying down the spec.

High Level Intermediate Representation

Our intermediate representation mimicked the structure of the abstract syntax tree created by the parser. It has only a few differences, which were made for the sake of simplicity. For example, the parser outputs AddExprs, MulExprs, etc, but our high-level IR has no notion of AddExprs or MulExprs—it just has an Expr type.

The structure of the high-level IR is conducive to semantic checking. For example, along with a list of statements, an IR Block contains a list of tuples of the form “(identifier, type)”, where type gives the relevant information that can be used for semantic checking.

Semantic Checking

We wrote mutually recursive semantic-checking functions check_program, check_block, check_stmt, etc. that took program components as input and semantically checked them.

To handle scoping, we created a simple Scope data type, where a Scope consists of a local-variables map and a reference to a parent scope. Each semantic-checking function (except for check_program) takes a Scope as an argument. Scope-generating program elements (namely, methods and blocks) create a new scope (consisting of a reference to the passed-in parent scope, along with the appropriate local-variables map) to pass to recursive calls.

We handled programs with multiple errors by passing along a list of strings that got mutated as the checker processes the program. This provided an advantage over explicitly throwing errors or panicking, which would prevent us from continuing to check the program.

Control Flow Graph

Our low-level intermediate representation models the control flow graph(CFG) of the program. The basic unit of the CFG is the instruction. The instructions allowed in the cfg can be placed into 3 categories: instructions that set some variable, method calls, and returns from methods.

The allowed set instructions are chosen so that they closely model what must happen at the assembly level. Set instructions include instructions of the form t1 <- t2 op t3, t1 <- op t2, and t1 <- t3. For simplicity, instructions operate only on variables, not immediates.

To convert complex expressions represented in the high level IR we must destructure them. To demonstrate how we do this in the cfg consider the example, a = 5 + 6 * 9. In the cfg, this would be converted into the following instructions:

```
t0 <- 5
t1 <- 6
t2<- 9
t3 <- t1 * t2
t4 <- t0 + t3
a <- t4
```

The next high level of organization in the cfg is the basic block. Each basic block contains a list of instructions and which block to jump to next. There are three types of jumps: unconditional jumps, conditional jumps, and jumps to nowhere. Unconditional jumps require the label of the next block, conditional jumps take in a source variable, and the labels of a true and false block. Jumps to nowhere represent returns from the method.

Finally, we represent methods in the cfg as the graph formed by the basic blocks.

Assembly Generation

Our method of converting methods in the cfg to assembly starts with allocating space on the stack to contain the method parameters, local variables, and all temporary variables introduced in the cfg. We retrieve the method parameters from registers and off of the stack and place them into their designated spots.

We translate the cfg basic block by basic block, instruction by instruction. Translating each instruction consisted of loading the relevant stack variables or immediates into registers, doing the desired operations on them, then returning the new value to the assigned position on the stack.

Converting the jumps between blocks is straightforward since both unconditional and conditional jumps exist in assembly.

At the end of methods, we push the return value into %rax if needed and restore the stack.

Testing Infrastructure

To test our code we wrote a bash script that automates the public tests and some test that we wrote on our own.

Extras

We did not do anything out of the ordinary; we attempted to do things in the most straightforward way possible. We debugged Rust code by using print statements. We debugged assembly code mainly by looking at it. We also used gdb (or, for the two of us that use macs, an online gdb thing).

To write test cases, we found it helpful to give an LLM the Decaf specification and ask it to write test cases for a Decaf compiler.

We used exactly one external dependency (other than "clap"): an "enum-iterator" library, which, given an enum, derives an iterator over all values of the enum. This was useful in the scanner: if you want to implement a from_string function for a type T, then (if T is finite, and you have an iterator over all values of T) it suffices to implement a to_string function for T.

Difficulties

There are no known bugs in our compiler. However, due to a bug in one of the test cases on the server (not a bug in our compiler), the compiler we submitted by the Friday-night deadline was not passing all the Phase 3 test cases on Gradescope.

Contribution Statement

We wrote the code to build the cfg collaboratively. The bulk of the code to convert the cfg to assembly was written by Marshall and the debugging of both the cfg and assembly was done collaboratively.