6.110 Design Document

Design

Our compiler does the following steps:

- Tokenize.
- Parse into Abstract Syntax Tree (high-level IR)
- Validate AST and convert into a low-level IR
- (not implemented yet) optimize the low-level IR
- Convert the low-level IR into assembly

Tokenizer (Phase 1)

The tokenizer is implemented by hand and converts the input into a series of whitespace-separated tokens. Errors such as invalid character literals or invalid strings are raised here.

AST / High-level IR Generation (Phase 1)

The output of the tokenizer is converted into an AST following the grammar rules of Decaf. Type-checking is not done at this step; the only rules that are checked are the grammar rules. We do not explicitly construct a parse tree; we directly construct a high-level IR (ie. an AST).

We implemented this step by hand using a greedy look-ahead parser.

Semantic Checker and Low-level IR generation (Phase 2)

In this step, we do type-checking on our AST and construct a low-level IR. Our low-level IR follows LLVM's design and uses static single-assignment. It's probably possible to convert our low-level IR data structure into LLVM IR.

As we traverse the AST, we maintain a stack of scopes and symbol tables, and use this to do type-checking. This eliminates the need to maintain parent pointers to symbol tables, which is challenging to do in Rust.

Our low-level IR contains a list of methods in a Decaf file. Each method includes a list of Blocks, Instructions, and Stack Slots.

Each Instruction is one "SSA instruction"; for example, `Add(InstRef, InstRef)` is one possible instruction.

Local variables are stored on the stack in a Stack Slot. A Stack Slot represents a certain amount of space reserved in the stack, used to hold the contents of a variable.

A Block is a list of instruction references, as well as a Terminator. A block can be thought of as one node in a control-flow graph. An instruction reference points to one instruction in the method. Instructions are stored in the *method*, not the *block*, because instructions in one block may reference instructions in another block. The Terminator controls where the program should jump to after the block has finished executing; this can be thought of as the outgoing edge of a node in a control-flow graph.

Low-level IR optimization (Not Implemented Yet)
We will likely implement the bulk of our optimizations on the low-level IR we have constructed.

Assembly Generation (Phase 3)
Given our low-level IR, we output x86_64 assembly code. Our low-level IR is already in static single-assignment form, so this step mostly just has to map SSA instructions into assembly instructions. Runtime array-out-of-bounds checking, function calling, etc. is implemented in this step.

Our current assembly generation is very simple and quite inefficient: Every SSA instruction writes its result into a unique location in the stack.

Extras
Our assembly generation includes `.loc` debugging directives, and we set up a custom Godbolt instance so we can examine the assembly output of any Decaf program.

Difficulties
There are no known difficulties at the moment.

Contribution
Work was done equally by all team members.