

# 6.110 Design Document

---

This document outlines our design for the 6.110 Decaf compiler up to phase 5.

## Overview

---

Our compiler is written in Rust (~15k LoCs) and is capable of

- Tokenizing and parsing Decaf programs (phase 1);
- Performing semantic analysis, and emit an SSA-like IR (phase 2);
- Generating unoptimized x86\_64 assembly from IR (phase 3);
- Perform various dataflow analysis (phase 4 & 5);
- Perform register allocation (phase 5);
- Generate efficient assembly with peephole optimizations applied (phase 5);
- Reporting errors clearly in a format similar to the one offered by the Rust compiler;
- Being used as a Godbolt backend!

We will elaborate on the design and implementation of these features below.

## Choice of Language

---

We choose to write our compiler in Rust because Rust has a very powerful type system and a robust borrow checker that we believe will help us eliminate bugs early in the development process. Rust's support for pattern matching and functional-style programming has also proven helpful for writing compilers.

## Tokenization (Phase 1)

---

The tokenizer is implemented by hand and converts the input into a series of whitespace-separated tokens, each associated with a `Span` representing their location in the source code. Errors such as invalid character literals or invalid strings are raised here, and our tokenizer attempts to recover from these errors by ignoring the offending character.

Different types of tokens are represented as variants of the `Token` enum, defined in `src/scan/token.rs`. We adopt a fine-granularity design where every type of token corresponds to a different variant, e.g., we have `Token::Add` and `Token::Sub` as opposed to `Token::Op("+")` and `Token::Op("-")`. We believe that this design helps better leverage Rust's type system and is easier to pattern-match against in downstream processing.

Another noteworthy design of our tokenization step is the association of source location with the tokens. This enables helpful error messages downstream and is achieved with the `Span` data type. Because `Span` is so ubiquitous throughout our compiler, we carefully engineered its representation to take minimal space and to be easily clonable:

```
pub struct Span(Rc<SpanInner>);

struct SpanInner {
    start: Location,
    end: Location,
}
```

```
pub struct Location {
    pub source: Rc<Source>,
    pub offset: usize,
    pub line: usize,
    pub column: usize,
}

pub struct Source {
    pub filename: String,
    pub content: Rc<[char]>, // Guaranteed O(1) indexing and cloning.
}
```

`Span` is immutable and multiple `Span`s can share the same `SpanInner` object with `Rc`. Each `SpanInner` records start (inclusive) and end (exclusive) `Location`s of the span. `Location`s again share the same `Source` through `Rc`, which records both the filename and content of the source file. Content is included in `Source` to make it self-contained and to decouple the abstract notion of "source file" from an OS file, so we don't need to access the file system every time to read the source's content in later stages of our compiler. The source file's content is stored as `Rc<[char]>` as opposed to the more intuitive `String` to allow efficient random access (Rust's `String` uses UTF-8 and does not support random access of Unicode code points by index).

We define a generic type, `Spanned<T>`, to represent "something attached with a `Span`".

```
pub struct Spanned<T> {
    pub inner: T,
    pub span: Span,
}
```

The tokenizer returns a stream of `Spanned<Token>`s.

## Parsing (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; instead, we directly construct an AST.

We implemented this step by hand using a greedy look-ahead parser. Definitions of AST types can be found in `src/parse/ast.rs`. Locational information is kept in this process by attaching `Span`s to many AST nodes.

For bad inputs, our parser recovers from parsing errors by consuming tokens until hitting a certain set of synchronization tokens (such as semicolons). This simple yet effective strategy allows our parser to report multiple syntax errors for a bad source file.

## Semantics Check (Phase 2)

In this step, we type-check our AST and construct a low-level IR. Our low-level IR follows LLVM's design and uses static single-assignment (SSA). That said, our IR is more restrictive than LLVM's IR in that it only allows loads and stores from global variables or predeclared `StackSlots` (for local and arguments). This is fine in Decaf, as Decaf does not support pointers, and we believe that incorporating this restriction into our IR simplifies our design. Below is the skeleton of our IR (full source at `src/inter/ir.rs`):

```

/// An opaque reference to an instruction
pub struct InstRef(usize);
/// An opaque reference to a block
pub struct BlockRef(usize);
/// An opaque reference to a stack slot.
/// Stack slots are used to represent local variables and function parameters.
pub struct StackSlotRef(usize);
/// An address in memory. This is used for loads and stores.
/// We don't support pointer arithmetic, so we don't need to support arbitrary
/// addresses.
pub enum Address {
    Global(IdentStr),
    Local(StackSlotRef),
}
/// An IR instruction
pub enum Inst {
    Phi(HashMap<BlockRef, InstRef>),
    Add(InstRef, InstRef),
    Sub(InstRef, InstRef),
    // ... More arithmetic, comparison & logical instructions omitted.
    LoadConst(Const),
    Load(Address),
    Store { addr: Address, value: InstRef },
    // Loads and stores to arrays use separate instructions from scalar loads and
    // stores, because they need to take an index and need to be bounds-checked.
    LoadArray { addr: Address, index: InstRef },
    StoreArray { addr: Address, index: InstRef, value: InstRef },
    /// Initialize a stack slot with a value.
    /// Differs from Store in that Initialize works for array too.
    Initialize { stack_slot: StackSlotRef, value: Const },
    Call { method: IdentStr, args: Vec<InstRef> },
    /// Loads a string literal, only used in external calls.
    LoadStringLiteral(String),
}

pub struct Block {
    pub insts: Vec<InstRef>,
    pub term: Terminator,
}

pub struct StackSlot {
    pub ty: Type,
    pub name: Ident,
}

pub enum Terminator {
    Return(Option<InstRef>),
    Jump(BlockRef),
    CondJump { cond: InstRef, true_: BlockRef, false_: BlockRef },
}

pub struct Method {
    pub name: Ident,
    pub blocks: Vec<Block>,
    pub insts: Vec<Inst>,
}

```

```

    stack_slots: Vec<StackSlot>,
    pub entry: BlockRef,
    pub return_ty: Type,
    // ... field omitted
}

pub struct GlobalVar {
    pub name: Ident,
    pub ty: Type,
    pub init: Const,
}

pub struct Program {
    pub imports: HashMap<String, Ident>,
    pub methods: HashMap<String, Method>,
    pub globals: HashMap<String, GlobalVar>,
}

```

Our 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. It roughly corresponds to the `alloca` instruction in LLVM.

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.

It’s probably possible to convert our low-level IR data structure into LLVM IR, but we have not tested.

As we traverse the AST, we maintain a stack of scopes and symbol tables, and use this to do type-checking. Hierarchical symbol tables based on lexical scopes are transient, and can be represented as in a stack at any point in time. This eliminates the need to maintain parent pointers to symbol tables, which is challenging to do in Rust.

We type-check and build IR both in the same pass. For each statement, the checker & builder function takes the AST statement node and a current block reference, generates code, and returns a block reference to the block with the last IR instruction generated for that statement. Sequential control flow can be generated easily:

```

fn build_stmt(&mut self, stmt: &Stmt, cur_block: BlockRef) -> BlockRef;

fn build_block_no_new_scope(&mut self, block: &Block, mut cur_block: BlockRef)
    -> BlockRef {
    // ...
    for stmt in &block.stmts {
        cur_block = self.build_stmt(stmt, cur_block);
    }
    cur_block
}

fn build_expr(&mut self, expr: &Spanned<Expr>, cur_block: BlockRef)
    -> (BlockRef, InstRef, Type);

```

For an expression, the checker & builder function is similar except that it returns `(BlockRef, InstRef, Type)`, returning `InstRef` allows the value of the expression to be used down the line and the returned `Type` facilitates type checking. If the expression is semantically invalid, an error is emitted to the internal buffer and a special `InstRef, InstRef::invalid()`, is returned so the error propagates.

```

fn build_cond(&mut self, expr: &Spanned<Expr>, cur_block: BlockRef,
    true_block: BlockRef, false_block: BlockRef);

```

Things are again a bit different when it comes to conditional expressions due to short-circuit evaluation. We take the suggested approach covered in lecture by supplying the checker & builder function with block references to the true and false branch at the call site.

While our IR has the `phi` instruction, our semantic checker does not emit `phi`s, and the IR is not in strict SSA due to extensive use of stack slots. We expect to introduce `phi`s later using a pass similar to LLVM's `mem2reg`.

## Unoptimized Code Generation (Phase 3)

Given our low-level IR, we output x86\_64 assembly code. Our low-level IR is already linearized, so this step mostly just has to simply map IR instructions into assembly instructions. Runtime array-out-of-bounds checking, function calling, etc. is implemented in this step.

Booleans are considered to be just as big as ints (8 bytes). The size of `bool` and `int` are controlled by two constants in our program so we may change them later if necessary.

The skeleton for our assembler is shown below. We generate assembly function by function. When generating a function, our code can add assembly instructions to both `.data` and `.text` sections. At the end, all `.data` instructions will be put together and the same goes for `.text`. The merged string is then returned as the full generated assembly.

```

pub struct Assembler {
    program: Program,
    // corresponds to .data
    data: Vec<String>,
    // corresponds to .text
    code: Vec<String>,
}

```

```

impl Assembler {
    pub fn new(program: Program) -> Self { ... }

    // while generating assembly for a method, this method can
    // push new lines of assembly to either .data or .text sections.
    fn assemble_method(&mut self, method: &Method) { ... }

    // Adds some data to .data and label it with the name of the variable.
    fn assemble_global(&mut self, var: &GlobalVar) { ... }

    pub fn assemble(&mut self, file_name: &str) -> String {
        for global in self.program.globals.clone().values() {
            self.assemble_global(global);
        }
        for method in self.program.methods.clone().values() {
            self.assemble_method(method);
        }
        // ... join .data and .text sections
        // together and return the full assembly
    }
}

```

Our current assembly generation is very simple and quite inefficient:

- Every SSA instruction writes its result into a unique location in the stack,
- The function prologues first copies all arguments into their homes on stack,
- We only use `%rax` for most of our assembly code (except function calls). Typically, for an operation, we use `%rax` to store an operand we just read from the stack, and then the next instruction that does the job will read the other operand directly.
- We save all callee-saved registers blindly in function prologues despite us not using any of them.
- Comparison-based jumps are slow because in our IR it is represented in two steps: compare (as `Inst`) and jump (as `Terminator`). Their codes are generated separately. Consequently, the generated code will read the operands from memory, compare with `cmpq`, set result with `setxx`, write the result to memory, read the result back to register, and do a `cmpq` with `$0` and jump with `je` -- that's a lot of instructions and memory traffic and it could have been a simple `cmpq` and `jxx`.

We expect this to improve in phase 4 and 5 as we implement mem2reg and register allocation.

## Data flow optimizations (Phase 4)

For phase 4 we implemented three data flow optimizations:

- Global copy propagation
- Dead code elimination
- Global common subexpression elimination / Global value numbering

We will elaborate on these optimizations in this section.

## Conversion in and out of SSA form

All of our data flow optimizations operate on SSA forms. While our semantic checker already emits SSA-like IR, it cheated a little bit by generating stack loads and stores for every variable reads and writes, and hence did not insert any phi instructions. Our assembly generation module does not accept phi instructions either. Hence, as a prerequisite to our optimizations, we implemented routines that converts our IR into and out of SSA form.

We convert the IR generated by the semantic checker into SSA form following [the algorithm by Cytron et al.](#) In particular, for every variable,phis are inserted at the iterated dominance frontier of blocks containing stores to the variable. We compute the dominance relation of our CFG following ["A Simple, Fast Dominance Algorithm" by Cooper et al.](#) In addition, we augment the SSA conversion algorithm by first performing liveness analysis and insert phis of a variable only at blocks it is live-in. Hence, the SSA form emitted by our conversion routine is both **minimal and pruned**.

Just before assembly generation, we implement an SSA-destruction routine that eliminates all phi instructions and convert the IR out of SSA form. We use the standard technique: assign each phi instruction a stack slots, split critical edges, and convert each phi instruction as stores at the end of predecessor blocks and a load at the beginning of the successor block. Because each variable (which used to have one stack slot each) can have multiple phi instructions inserted, and we do not coalesce the stack slots of these phi instructions, the SSA-construction-destruction round trip can cause the function to use more stack space. We consider this de-optimization acceptable for now. This SSA-destruction routine is only temporary and we plan to replace it with a SSA-based register allocator (and rewrite the assembly generator) in phase 5.

An amazing property of SSA form is that for each SSA variable, all uses except those in phi instructions are dominated by a single definition. This effectively reduces a lot of the worklist-based dataflow analysis covered in lectures to DFS's on the dominator tree. As such, **we do not list the dataflow equations we used (as required by the handout)**, because our optimizations, when implemented in SSA form, are not based on dataflow equations and fixed-point worklist algorithm.

## Global copy propagation

Thanks to the single-definition-dominates-all-uses property of SSA form, global copy propagation is very easy to perform on SSA form. In pseudo code:

```
another_iteration = true
while another_iteration:
    another_iteration = false
    initialize map src: Map<Var, Var> with the identity map
    for block in dominator tree preorder:
        for inst in block.instructions:
            if inst is "y <- x":
                set src[y] = src[x] and remove inst
            else if inst is "y <= phi(...)":
                if all phi operands are the same, say x:
                    set src[y] = src[x] and remove inst
            else:
                replace each SSA variable x used by inst with src[x]
    for phi in all phi instructions:
        replace each operand x with src[x]
```

```
if all operands become the same:
    another_iteration = true
```

We note that copy propagation is still iterative and potentially requires more than one traversals of the CFG. This is because the use of a variable in a phi instruction is not dominated by its definition, so a single pre-order traversal does not guarantee that a phi node will be visited after all its predecessors.

## Dead code elimination

The single-assignment property of SSA form also greatly simplifies dead code elimination as an SSA variable can only either be entirely useful or entirely useless -- with no middle ground (e.g. interleaving of useful and useless assignments throughout the live range). This reduces the dead code elimination pass to two rules:

- `Call`, `Store`, `StoreArray`, and `Initialize` instructions are useful on their own.
- If an instruction is useful, all instructions / SSA variables it references are useful.

This can obviously be solved with a worklist-like algorithm or DFS. Note how the structure of CFG becomes entirely irrelevant here. Our full dead code elimination pass is implemented less than 40 lines of Rust code.

## Global common subexpression elimination

Although phase 4 requires global CSE in the hand out, we actually implemented global value numbering (GVN) instead because they are close enough in SSA world and GVN is more powerful. Our value numbering scheme computes a "canonical form" as a string key for every SSA definition. For commutative operators, our value numbering scheme handles commutativity by always having the lexically smaller operand as the first operand. SSA helps us again here by allowing us to perform the entire optimization as a single DFS pass down the dominator tree, as follows:

```
canonical: Map<Var, String>
stack: Stack<Map<String, Var>>

def dfs_gvn(block):
    for inst in block.instructions:
        c = compute canonical form of variable defined by inst
        canonical[inst] = c
        eliminated = false
        for map in stack from top to bottom:
            if map[c] exists:
                replace inst with a copy from map[c]
                eliminated = true
                break
        if not eliminated:
            stack.top()[c] = inst
    for child with block as immediate dominator:
        stack.push({})
        dfs_gvn(child)
        stack.pop()

dfs_gvn(entry block)
do copy propagation
```



Our GVN replaces common subexpressions with copies, and we immediately perform a copy propagation pass to propagate them.

We note that this approach differs greatly from the available expression analysis covered in lecture. In particular, our GVN scheme will not eliminate common expressions of the following forms:

```
if ...:
    c = a + b
else:
    d = a + b
e = a + b
```

This is because neither the definition of `c` nor `d` dominates that of `e`, although they jointly do. An available expression analysis will show that "a + b" is available immediately before `e`, so a CSE backed by available expressions will be able to optimize the redundant `a + b` away. However, it's hard to do available expression analysis *of the original program* in its transformed SSA form because equivalence in lexical form in the original program can be broken in SSA.

## Order of optimizations

We apply the optimizations in the following order:

- Dead code elimination
- Copy propagation
- Common subexpression elimination

All optimizations are implemented in a way that does not assume any particular optimization has been applied. For example, our common subexpression elimination pass can be performed before copy propagation because it does not rely on lexical form but value numbering. We choose to make it the last pass because it does the most work and having a copy propagation pass that reduces the number of instructions up front will help. Similarly, we choose to run dead code elimination first because it's the simplest and runs in guaranteed linear time.

## Register allocation + more optimizations (Phase 5)

In Phase 5 we implemented many more optimizations. We present all of them in this section. We note that many optimizations we implement do not appear to have standard names to our limited knowledge, so we name them somewhat arbitrarily.

### Overview: all optimization flags

Our compiler features a handful of optimizations that can all be turned on or off individually:

- `cp`: Copy propagation
- `dce`: Dead code elimination
- `cse`: Common subexpression elimination
- `cf`: Constant propagation / constant folding
- `inline`: Function inlining
- `dfe`: Dead function elimination
- `gvnpre`: GVN-PRE (Global value numbering - Partial redundancy elimination)

- `psr`: Polynomial strength reduction
- `as`: Array splitting / scalar replacement of array
- `rgae`: Redundant global and array access elimination
- `dase`: Dead array store elimination
- `licm`: Loop invariant code motion
- `unroll`: Loop unrolling
- `indvar`: Induction variable strength reduction
- `omit-frame-pointer`: Omit frame pointer
- `align-loops`: Align loops
- `align-bc`: Align bounds check
- `fuse-bc`: Bounds check fusion
- `coalesce`: Heuristically coalesce registers to minimize copys on phi edges
- `coalesce-ilp`: Optimal register coalescing with integer linear programming
- `nm-imm`: Constant nonmaterialization as instruction immediates
- `nm-array`: Constant array offset nonmaterialization as displacements
- `nm-cond`: Jump condition nonmaterialization / fusion
- `const-divisor`: Constant divisor strength reduction
- `const-modulo`: Constant modulo strength reduction
- `peephole`: Various peephole optimizations on generated assembly

As register allocation is such an integral part of our phase 5 compiler. *We choose not to expose options that turn it off.* Otherwise we would have to essentially maintain two versions of the code generator, one from phase 3 and one from phase 5. We think this is a reasonable choice: production compilers like LLVM and GCC always do register allocation as well.

## Changes to optimization ordering

In Phase 4, we were able to carefully order optimizations and run them once in a row. As we implemented more and more optimizations in phase 5, and it quickly become unwieldy to manually decide which optimization should run first. Instead, we choose to run all optimizations in some arbitrary sequence 10 times. In this way every optimization gets a chance to run after every other optimization and 10 iterations is mostly sufficient for the algorithm to converge for moderately-sized functions.

## Register allocation

We implemented register allocation following ["Register Allocation for Programs in SSA form"](#) by Sebastian Hack (2007). Register allocation on SSA-form programs is theoretically superior because the interference graph for SSA-form programs are chordal and there exists polynomial-time algorithms for finding a minimum coloring for chordal graphs. In addition, the maximum register pressure for an SSA program can be computed exactly as the max live set size at any point in the program.

These properties enable the separation of spilling and register allocation, which are intertwined in a loop in traditional register allocators. In our case, the spiller inserts spills to reduce the max live set size to at most the number of available registers. The register allocator then simply traverses the blocks in dominator tree order and allocates registers to live variables, which is now guaranteed to succeed. The allocate--fail--spill--reallocate--try again cycle is avoided.

We note that while Hack's thesis is very helpful, it handwaved over many details. In these cases, we referred to [libFirm](#), which contains a C implementation of Hack's algorithm by himself. Still, our implementation deviates from Hack's implementation in the following aspects:

- Hack's spiller uses Belady's MIN heuristic. That is, the spill weight of a variable at a program point is the distance to its next use, taking *minimum* of all successor blocks. We replaced minimum with *average weighed by execution frequency*. Taking minimum is theoretically better as it implies a lattice on which a worklist algorithm is guaranteed to converge. But we find taking weighted average a more practical choice as it discourages spilling inside loop body. However, averaging breaks the lattice property. We handle that by running the worklist algorithm until the live sets converge, and then running the entire analysis 10 more times over the entire CFG to produce a reasonably good steady-state solution. A theoretically better approach also exists: since averaging is a linear operation the exact spill weights can be solved as a linear system. We did not implement this approach, however, since "running 10 more times" works well enough.
- Hack's spiller reconstructs SSA form for both regular variables and memory variables (spill slots). Our spiller only does the former. Enforcing SSA form on spill slots does not yield much practical benefit but will likely introduce much complexity into our codebase.
- Our spiller recognizes that for a large call (i.e., with 16 arguments) it's impossible to place all arguments in registers. Hence some arguments must be loaded directly from their spill slots. We call these arguments "non-materialized arguments" and modify the spiller algorithm to handle them correctly.

Hack's algorithm also requires a coalescing step after register allocation. While not strictly necessary, the coalescer attempts to locally re-color the interference graph to assign the same register to variables used in phi instructions to the best extent possible -- then no `mov`s need to be generated for phi instructions. We implemented a coalescing heuristic and an optimal ILP formulation, both proposed in Hack's paper. For ILP, we call [HiGHS](#), an open source ILP solver. We pass the heuristic solution as an initial solution to the solver and set a 60-second time limit. In practice, all most all programs have their ILP instances solved under a second (except `12-large`, which typically takes 5-7 seconds).

We also extend the heuristic and the ILP solver to do some basic register targeting, in particular,

- We try to assign argument variables the register from which it's passed in (e.g., `%rdi` for the first argument), to save a `mov` in the function prologue.
- For a call, we try to assign every materialized argument the register used to pass that argument, to save a `mov` in the call sequence.
- For non-argument variables that live through a call, we try to put them in callee-saved register, to save a `push` + a `pop` around the calls.

These register preferences are represented as reward terms (weighed by execution frequency) in the ILP objective and extra affinity edges to fix-colored nodes in the heuristic.

Our register allocator uses 12 registers. Among the 4 we don't use, `%rsp` and `%rbp` are stack registers and we use `%rax` and `%rdx` as scratch registers. When `omit-frame-pointer` is enabled, `%rbp` is added to the list of available registers. It is theoretically possible to use `%rdx` as a general purpose register too, but `imul` and `idiv` modifies it and writing code to avoid `%rdx` through `imul` and `idiv` complicates our codebase.

## Non-materializers

### Constant non-materializer

We made an early design decision in our IR that it does not take immediates -- all consts must be explicitly inserted as a `LoadConst` instruction. While this design helped to simplify some of our other optimizations, it creates unnecessary pressure for the register allocator by forcing it to first move all constant into a register before it can be used. In x86, 32-bit constants can usually be encoded as immediates, and it would be a pity if we did not exploit this convenience. Hence, we run a *constant non-materialization pass* before running the register allocator. This pass identifies constant instruction arguments that can be encoded as immediates and mark them as "non-materialized." This mark is then picked up by the code generator which emits an immediate rather than the register assigned (if any).

The liveness analysis internally used by the spiller and the register allocator takes non-materialization marks into consideration. If a `LoadConst` instruction is marked non-materialized in all its use sites, then it's considered dead. Our spiller, the register allocator, and the generator all skip dead instructions.

### Jump condition non-materializer

In our IR, a conditional jump always takes in a boolean value. In x86, however, a jump can be conditioned on many condition codes. This mismatch has caused our code generator to produce inefficient code, i.e., to compare two values, our code generator would generate

```
cmp %rax, %rbx
setl %rcx # The result of the comparison is "materialized" in a register
cmp %0, %rcx
jnz label
```

where the following code suffices:

```
cmp %rax, %rbx
jl label
```

The jump-condition non-materializer pass, which runs between constant non-materializer and spiller, is designed to solve this issue. It identifies comparisons close to a conditional jumps, copies it, and attaches it to the conditional jump terminator. The code generator, upon seeing this attachment, generates the fused compare-and-jump two liner shown above.

We arrange this pass to happen before spilling (and hence register allocation) instead of afterwards (as a peephole optimization) because if the non-materialization succeeds, and the conditional jump was the only user of the result of that comparison, then the register allocator should know that it has effectively one more register to use. Practically, this is realized by having the liveness analysis recognize copied comparisons attached to a conditional jump terminator. If the original comparison instruction has no other users, the liveness analyzer declares it dead.

## Array index offset non-materializer

We found that a single iteration of a loop can access many neighboring elements in an array. Consider the innermost loop in `segovia_blur.dcf`:

```
for (c = 3; c < cols; c += 1) {
    int dot;
    dot = (p1 * kernel[0]);
    dot += (p2 * kernel[1]);
    dot += (p3 * kernel[2]);
    dot += (image[r*768 + c] * kernel[3]);
    dot += (image[r*768 + c + 1] * kernel[4]);
    dot += (image[r*768 + c + 2] * kernel[5]);
    dot += (image[r*768 + c + 3] * kernel[6]);
    p1 = p2; p2 = p3; p3 = image[r*768 + c];
    image[r*768 + c] = dot / kernel_sum;
}
```

With CSE, the four array loads are transformed into the following:

```
for (c = 3; c < cols; c += 1) {
    int dot, t;
    // ...
    t = r*768 + c;
    dot += (image[t] * kernel[3]);
    dot += (image[t + 1] * kernel[4]);
    dot += (image[t + 2] * kernel[5]);
    dot += (image[t + 3] * kernel[6]);
    // ...
}
```

Naively, the register allocator would allocate registers for `t + 1`, `t + 2`, and `t + 3`, but they are not necessary because these constant offsets can be absorbed into the displacement field of the address. Hence, all four loads can use the same register that holds `t`. The array index offset non-materializer pass identifies array accesses shaped like `a[t + c]` where `c` is a small constant. It replaces, in the instruction itself, `t + c` with `t` and attaches a marker on the instruction which holds the offset. The code generator picks up these markers and emit array loads and stores with these offsets absorbed. For the code snippet above, the generated assembly is

```
cmpq $799996, %r10 # r10 holds t, array is 800000 elements long
ja main.bound_check_fail.r10.0.800000.0
movq image(, %r10, 8), %r14
# ...
.set image.1, image + 8
movq image.1(, %r10, 8), %r13
# ...
.set image.2, image + 16
movq image.2(, %r10, 8), %rbx
# ...
.set image.3, image + 24
movq image.3(, %r10, 8), %rbx
```

We use GAS's `.set` directives to compute offset labels at compile time. As we see all array loads use the `%r10` register as the index.

## More assembly-level optimizations

### Fused bounds checks

Note that in the code snippets above only one bounds check is emitted. It checks if `%r10` is in the range `[0, 799996]`. If the check passes, all four loads will succeed. This is the result of bounds check fusion, implemented as follows:

- For each block, we maintain a map from every SSA variable to its *checked range* -- that is, the value is guaranteed to be within the range given previous bounds checks. Checked ranges are initialized to  $[-2^{63}, 2^{63})$ , i.e., the variable can hold any value.
- For every access to array `a`, indexed with variable `i` and with potentially an offset `c`. Say its required range is `[-c, len(a) - c]`.
- For every array access, we scan forward through all subsequent array accesses indexed with the same SSA variable up to block boundary or a function call. We take the intersections of the required ranges of all these accesses. This will be the range checked by the bounds check we are about to emit.
- If the range to check is a superset of the checked range of the variable, don't emit the bounds check.
- After the bounds check has been emitted, update the checked range map by intersection with the range just checked.

We do not fuse bounds checks across function calls because this might alter the observable behavior of the program. Should we do so, a bounds check that would fail after the call may be fused into one that fails before. If the call is side effective (e.g., `printf`), then the user would not be able to observe the side effect after the fusion.

We use unsigned comparison `ja` to check both endpoints of a range at the same time. In particular, `ja` can check, with one comparison, if a value lies in `[0, x]`. For bounds `[l, r]` where `l` is not 0, we offset the checked value by `l`, i.e.,

```
leaq $<-l>(%reg), %rax
cmpq $<r - l>, %rax
ja <label>
```

Up to phase 4, the error-and-exit block for failed bounds checks are inserted at bounds check sites. For phase 5, we deduplicate such blocks and move them all to the end of the method so "useful instructions" can be packed more compactly.

### Constant divisor and constant modulo strength reductions

Integer division is slow. On Skylake, an `idivq` can take 42-95 cycles to execute while an `imulq` takes only up to 4 cycles. The well-known paper ["Division by Invariant Integers Using Multiplication"](#) by Granlund and Montgomery showed that divisions with constant divisors can be strength-reduced to not require `idiv` instruction at all. We implemented this optimization for division and modulo (using `a % d = a - (a / d) * d`). For example, the following code computes `%rbx = %rbx / 1000000` (taken from `segovia_blur.dcf`):

```
movabsq $-8775337516792518218, %rax
imulq %rbx
addq %rbx, %rdx
sarq $19, %rdx
shrq $63, %rbx
addq %rdx, %rbx
```

These 6 instructions only take 9 cycles to execute, much faster than using `idivq`.

GCC and LLVM also perform this optimization. While the overall algorithm is the same, they choose slightly different magic numbers and instruction orders than our compiler. We are unsure why that's the case.

The algorithm above applies for non-power-of-2 divisors. For power-of-2 constant divisors, shifts and masking are used to perform division and modulo.

We did not implement strength reduction for multiplication -- it had lower priority on our todo list because multiplications on modern CPUs are reasonably fast already.

## Omit frame pointer

As was discussed in lecture, the base pointer is only needed if a function allocates variable-size memory on the stack. For Decaf, the stack allocation size for each function can always be determined at compile time. This means we can omit pushing and popping `%rbp` in function prologues and epilogues and use `%rbp` as a general purpose register. Empirically, this seems to improve the performance of our program by 3-5%.

## Exploiting `leaq`

The x86 `add` instruction takes in two registers. If we want to add two values from two different registers into another register, we'll need to emit two instructions: one that moves one operand into the destination register, and the other that accumulates the other operand. On the other hand, we can achieve the same with `leaq` in one instruction: `leaq (%rax, %rbx), %rcx`. `leaq` has the same 1-cycle latency as `add` and only slightly lower throughput. Our compiler emits `leaq`s for additions where the destination register differ from both source registers (or a register plus a constant).

## Peephole optimizations

We also perform a series of peephole optimizations on the generated assembly.

- **Removing redundant jumps:** if a `jmp` to a label is immediately followed by the label itself, that jump can be removed.
- **Removing empty blocks:** if a label is immediately followed by a `jmp`, we can remove the label and replace all its occurrences with the jump target. Empty blocks result from preemptively splitting critical edges in our spiller and register allocator -- if a phi argument is assigned a different register than the phi variable, it is at these empty blocks that `mov` instructions will be inserted. However, our coalescer is often so good that `mov`s forphis are not needed at all. This optimization removes these empty blocks.
- **Removing unreachable code:** all code after a `jmp` up to the next label is unreachable and can be removed.
- **Adjusting comparison order:** rewrite conditional jumps such as

```

    cmpq ...
    jl label_a
    jmp label_b
label_a:

```

to

```

    cmpq ...
    jge label_b
    jmp label_a
label_a:

```

to save one instruction. `jmp label_a` will be removed by the "removing redundant" jumps pass.

- **Replace `movq $0, %reg` with `xorq %reg, %reg`:** the latter can be encoded with less bytes and is more idiomatic.
- **Remove `mov` ing registers into itself:** These instructions do nothing.

## Alignment of loops and bounds checks

During our testing, we found a huge variance in running time for different assembly outputs generated by our compilers from the same input. Our compiler is mostly deterministic and outputs almost identical code every time, except with potentially a different permutation of registers and with functions emitted in different orders. This variance is also architecture specific: we could not replicate this variance on our laptops, but instead only on the grading servers and an AWS c5 instance we set up later which also uses a Skylake CPU. Through extensive testing, we eventually narrowed down the list of suspects to instruction alignment.

Intel suggests that all loop labels should be 16-byte-aligned so the instruction fetcher and decoder can read them quickly. We implemented this optimization using GAS's `.p2align` directive. In particular, our compiler inserts the following directives before the header label of a loop:

```

.p2align 4,,10
.p2align 3
loop_header:

```

The two directives express the wish that `loop_header` should be at least 8-byte-aligned, or 16-byte-aligned if doing so requires filling no more than 10 bytes. These two directives are copied from GCC outputs with `-falign-loops`. We believe that the 16-byte alignment directive includes a 10-byte threshold because there are multi-byte NOPs up to 10 bytes. We found that this partially reduces the variance in performance we observed.

Empirically, we found that aligning each bounds check jump instruction to 8-byte boundaries also improve performance (by ~50-100ms). The effect is especially visible if multiple bounds checks are clustered close to each other in a small loop. However, this is highly microarchitecture specific: while we observed this on our AWS test server, we were unable to reliably replicate this phenomenon on the grading server. This is why we disabled the `align-bc` flag by default even with `-o a11`. We had a hypothesis that could potentially explain this phenomenon. It assumes



- The CPU's BTB has very limited associativity for jumps close to each other (say from the same 64-byte block), and / or
- A branching instruction takes up an entry in the BTB even if it's not taken.

In that case, there might be a conflict if a loop backedge jump has the same alignment (say modulo 16 bytes) as a bounds check close to it. The loop backedge is strongly taken, but when the control flow reaches the bounds check jump, the CPU looks up the same BTB set and reuses the saturation counter used by the loop backedge or replaces the entry with a new one for the jump. In the former case, the CPU mispredicts a branch prediction failure as strongly taken; in the latter case, the CPU mispredicts a loop backedge as not taken. Both incur a high penalty to performance. Aligning all bounds check jumps to 8-bytes mitigate the issue as now all bounds check jumps maps to the same set (say set 0) in the BTB, so they are less likely to clash with other regular jumps with different alignments.

## Constant propagation

The constant propagation pass of our compiler is very simple and runs in linear time. In particular, it does not propagate constant through phi by itself (i.e. replace a phi instruction with a constant if all its arguments are the same constant). However, the same effect can be achieved in combination with GVN-PRE and copy propagation: GVN-PRE will replace all instructions loading the same constant to one `LoadConst` instructions and our copy propagation pass is phi-aware.

## GVN-PRE

To address the limitation of our current GVN algorithm, we plan to implement [the GVN-PRE algorithm by VanDrunen and Hosking](#). PRE stands for Partial Redundancy Elimination and is more powerful than CSE or GVN, which only captures "total redundancy". However, GVN-PRE is more complicated than CSE, so we plan to defer its implementation to phase 5.

Look at the following example

```
if(v1 < v2) {
    v3 = v1 + v2;
} else {
    v3 = v1 - v2;
}
v4 = v1 + v2;
```

The computation of v4 is partially redundant because if the program enters the `(v1 < v2)` branch, then the value is already computed. Since the `(v1 < v2)` branch does not dominate the computation `v4=v1+v2`, ordinary GVN does not take advantage of this redundancy. GVN-PRE "hoists" the computation of `v4` to both branches of the if statement, effectively creating the following piece of code:

```
if(v1 < v2) {
    v3 = v1 + v2;
    v4 = v3;
} else {
    v3 = v1 - v2;
    v4 = v1 + v2;
}
```

In the event that  $v1 < v2$ , the number of computations is reduced by 1. In the other case, no additional computations are incurred. In fact, GVN-PRE guarantees that in all computation paths, the number of computations does not increase.

GVN-PRE is a three-phase algorithm that operates on a SSA control flow graph with no critical edges; that is, if a node has multiple predecessors, then all of its predecessors have only this node as their successor. All critical edges can be removed from a control flow graph in a straightforward way by inserting extra empty blocks.

In the first phase, we build the "leaders" and "antileaders" of each block. The leaders of a block are all values that are already computed during the execution of the block. If ever an expression is equal to some known value, we can replace that expression with a copy from the leader corresponding to that value. The antileaders are the anticipated values in a block; that is, all values that are computed in every computation path following this block. The computation of antileaders is done using a worklist algorithm, similar to the one taught in class.

The second phase is insertion; In this phase, we detect partial redundancy. If this were skipped, then GVN-PRE would behave exactly as ordinary GVN would. Insertion "hoists" antileaders (anticipated computations) of a block up to its predecessors. In particular, if any antileader is an expression that is also present in a predecessor's leader, then we insert this expression to all predecessors. Like in the example above, the redundant computation is eliminated from computation paths coming from all predecessors that have already computed the value.

The final phase is elimination. This is the simplest phase, and eliminates all expressions that are now partially redundant. For each block (in arbitrary order), remove all expressions that are in the block's leader set and replace them with a copy to the already computed value. Running copy propagation will subsequently remove all of the extra copies.

GVN-PRE is more powerful than CSE so we retired CSE after GVN-PRE has been implemented. Technically, GVN-PRE runs slower than CSE so CSE may still be desirable if compile time is critical, but for our case it's less of an issue.

## Function inlining

Inlining small functions can speed up the program by reducing the overhead of function calls. For our compiler, we inlined functions that contained 25 or fewer instructions, or were only called a single time in the program. Additionally, we did not inline functions that called themselves, or functions with a non-void return type that did not return a value in certain branches (since the expected behavior is for these programs to crash).

We inlined functions before constructing the SSA form and applying all the other optimizations. To inline function A into function B, we did the following steps:

1. Copy over all blocks and instructions from A to B.
2. Copy over all stack slots from A to B.
3. Update all copied instruction references, stack slot references, and block references (which changed when we copied them from A to B).
4. If the function had a return value, create a new stack slot to store the return value. When we construct the SSA form, this stack slot will likely be optimized to become a register.
5. Update all return block terminators to write the return value to the return stack slot.
6. Replace the call instruction with a load instruction that loads the return value from the return stack slot.

7. Move function A's arguments into the corresponding stack slots.

## Dead function elimination

After function inlining, some functions may no longer be reachable from the main function (because they have been inlined). The dead function elimination pass removes these dead functions from the program and prevents the compiler from wasting time optimizing functions that will never be called.

## Redundant global and array access elimination

Our common subexpression elimination pass (and later GVN-PRE) is conservative. It does not consider two loads to the same global as common subexpression. The same applies to two loads to the same array with the same index. However, there are cases where this is possible and desired, if we can prove that there's no store in between two loads that could modify the value being read. The motivating example is from `noise_median.dcf`:

```
t[0] = p4 - p1;
if(t[0] < 0){
    p1 = p1 + t[0];
    p4 = p4 - t[0];
}
```

In this case it is clear that the last three `t[0]`s can be replaced with a copy from the SSA instruction computing `p4 - p1`. We implemented this optimization can called it *Redundant Global and Array access Elimination (RGAE)*. RGAE is essentially available expression analysis ran over arrays and globals.

RGAE is performed as a dataflow analysis. The case for global scalars is very similar to the standard available expression analysis except that any call invalidates all available globals (we pessimistically assume that the callee could modify any globals). We now expand on how the analysis is performed on a local array. For every array `t[i]`, its abstract state at any program point consists of two maps:

```
struct AvailArrayElems {
    /// Instructions that compute t[c] for constant c's.
    consts: HashMap<i64, InstRef>,
    /// Instructions that compute t[v] for variable v's.
    vars: HashMap<InstRef, InstRef>,
}
```

When two states are joined via "intersection:" for each map, only key-value pairs present in both sides are kept. All such states, plus a separate  $\perp$ , define a lattice with the empty state is the top element. The transfer function is as follows.

```

for inst x:
    if x loads t[c] where c is constant:
        consts[c] = x
    if x loads t[v] where v is a variable:
        vars[v] = x
    if x stores y to t[c] where c is a constant:
        consts[c] = y
        clear vars since they may alias with c
    if x stores y to t[v] where v is a variable:
        clear consts and vars since v may alias with them
        vars[v] = y

```

After the worklist algorithm reaches a fixed point, we perform the elimination pass

```

for each inst x:
    if x loads t[c] where c is constant:
        if consts contains key c:
            replace x with copy(consts[c])
    if x loads t[v] where v is a variable:
        if consts contains key v:
            replace x with copy(vars[v])

```

Subsequent copy propagation will further simplify the IR.

We note that while this is a standard dataflow analysis, initial values for each block can be tricky to implement because the bottom element is special. Practically, our code starts by running the update step for all blocks in reverse postorder, ignoring all backedges during joins. That solution is then used to initiate the fixed-point algorithm. This is equivalent to initializing every block with  $\perp$  while being a lot easier to code.

We also note that there are rooms for improvement for this optimization. For example, the values of the map can be augmented from a single instruction value to a phi-tree. The join operations can be modified accordingly, and the elimination pass can insert phi nodes. Careful value range analysis of indices could also help by invalidating less entries upon stores. We did not implement these optimizations as they make the code more complicated.

While the optimization looks effective on paper, we found that the effect of these optimizations on the actual running time is very small (~1% to ~5%). Redundant array loads typically hit L1 cache anyway. While L1 cache is slower than register, the slightly larger latency is well hidden by the long pipelines of modern CPUs.

## Dead array store elimination

Once RGAE eliminates redundant array loads, some array stores become redundant. This optimization removes such accesses by performing a liveness analysis on arrays. In particular, we perform a backward data analysis. For each array, the abstract state is

```

enum Liveness {
    // Only these indices are live.
    Indices(HashSet<i64>),
    // All indices are live.
    All,
}

```

States are joined by union. Loads with constant indices extend the `Indices` set while loads with variable indices promote the state to `All`. Stores with constant indices removes the index from the `Indices` set (or do nothing for `All`). Stores with variable indices do not change the state. An elimination pass is performed after the worklist algorithm reaches a fixed point. If all stores to an array is eliminated the array itself is eliminated as well.

This optimization, together with RGAE, eliminates all array operations in `noise_median.dcf`.

## Loop unrolling

We implement a basic loop unroller for small loops that run for a small ( $\leq 8$ ) constant number of iterations. Other than the size limit, for a loop to qualify for unrolling,

- The loop must only have one back/continue edge and one break edge.
- The break edge must originate from the header.
- The break condition must be a comparison ( $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ) between a variable and a constant.
- That loop variable must be initialized with a constant outside the loop and be incremented / decremented by a constant.

We acknowledge that this kind of loop unrolling is very basic and a lot can be improved (even given our existing infrastructure). Nevertheless, we were motivated to unroll a loop not because it can make the loop run faster but because it enables other optimizations downstream. Consider the example in `segovia_blur.dcf`:

```
void gaussian_blur () {
    int kernel_sum;
    int kernel[7];
    int i,r,c;
    kernel[0] = 4433;   kernel[1] = 54006; kernel[2] = 242036; kernel[3] =
399050;
    kernel[4] = 242036; kernel[5] = 54006; kernel[6] = 4433;
    for (i = 0; i < 7; i += 1) { kernel_sum = kernel_sum + kernel[i]; }
    for (r = 0; r < rows; r += 1) {
        int p1, p2, p3;
        p1 = image[r*768]; p2 = image[r*768 + 1]; p3 = image[r*768 + 2];
        for (c = 3; c < cols; c += 1) {
            int dot;
            dot = (p1 * kernel[0]);
            dot += (p2 * kernel[1]);
            dot += (p3 * kernel[2]);
            dot += (image[r*768 + c] * kernel[3]);
            dot += (image[r*768 + c + 1] * kernel[4]);
            dot += (image[r*768 + c + 2] * kernel[5]);
            dot += (image[r*768 + c + 3] * kernel[6]);
            p1 = p2; p2 = p3; p3 = image[r*768 + c];
            image[r*768 + c] = dot / kernel_sum;
        }
    }
}
```

Unrolling the `kernel_sum` loop enables efficient constant propagation (with the help of RGAE) that turns `kernel_sum` into a compile time constant; `kernel_sum` being a constant divisor enables strength reduction at assembly generation phase; In addition, dead array store elimination works best when all indices are constant (i.e. loop is unrolled) -- in this case the `kernel` array can be completely eliminated.

As we only see loop unrolling as a way to "catalyze" other optimizations, we choose not to implement more advanced loop unrolling features (e.g., relax "constants" into "loop invariants", or do partial unrolling) because no other optimizations in our compilers can take advantage of these advanced features. To our knowledge, modern CPUs effectively unroll loops in hardware anyways, so the performance gain from unrolling alone is minimal.

## Array Splitting

Another optimization we implemented is to split small arrays that are only indexed by constants into multiple scalars (and then reconstruct SSA form). This is a restricted form of Scalars Replacement of Aggregates (SRA) optimization (unfortunately we did not know SRA when we named this optimization). While useful in its own right, RGAE + dead store elimination are more general and does what we originally expected it to do. Hence this optimization is disabled in our final submission.

## Polynomial strength reduction

Polynomial strength reduction (PSR) is a very powerful and versatile optimization pass that eliminates many hidden redundancies in a program. We conceived of this optimization independently, though we later found that similar (and more complicated) optimizations exist in literature (e.g. OSR by Cooper et al.).

PSR is similar to GVN, but instead of assigning each instruction a value, PSR assigns each instruction a *multivariate polynomial*. In particular,

```
for each inst x:
  if x = const c:
    poly[x] = c
  if x = copy(y):
    poly[x] = poly[y]
  if x = -y:
    poly[x] = -poly[y]
  if x = y op z where op is +, - or /:
    poly[x] = poly[y] op poly[z]
  else:
    poly[x] = x // x is an irreducible term
```

As we see, the variables in the polynomials are results of non-arithmetic operations (or arithmetic operations which polynomials are not closed under, such as division). Polynomials are stored under a canonical form (sum of terms ordered lexicographically), so they can be easily hashed and tested for equivalence. After all instructions have been assigned a polynomial, the strength reduction pass goes as follows:

```
fn psr(
  b: BlockRef,
  poly: HashMap<InstRef, Poly>, // instruction to polynomial
  avail: HashMap<Poly, InstRef>, // available polynomials to instructions
```

```

// available polys w/o consts to (inst, const)
avail_noconst: HashMap<Poly, (InstRef, int)>
):
  for each inst x in b:
    p = poly[x]
    (p_noconst, p_c) = (p without const term, p's const term)
    if p in avail:
      // If possible, replace x with a copy (to be eliminated by copy
prop.)
      update x = copy(avail[p])
    else if p is const c:
      // Replace x with a const
      update x = LoadConst(c)
    else if -p in avail:
      // Replace x with a cheap negation
      update x = -avail[p])
    else if p_noconst in avail_noconst:
      // Replace x with an addition with constant
      (y, c) = avail_noconst[p_noconst]
      update x = y + (p_c - c)
    else if x is mul:
      // For multiplication only, see if we can replace x with
      // additions or subtractions
      if exists p1, p2 s.t.  $p1 \pm p2 = p$ ,  $avail[p1] = y$ ,  $avail[p2] = z$ :
        update x =  $y \pm z$ 
    if p not in avail:
      avail[p] = x
    if p_noconst in avail_noconst:
      avail_noconst[p_noconst] = (x, p_c)
  for each child in b's children in dominator tree:
    psr(child, poly, avail.clone(), avail_noconst.clone())

```

Essentially, the strength reduction pass assumes the following hierarchy:

```
copy < const < neg < add w/ const < general add/sub < mul
```

Given that copies are eliminated, this is consistent with the latency of the respective instructions on modern CPUs. The strength reduction pass essentially looks at every instruction and see if it can be moved down the hierarchy given the available polynomials. This optimization is particularly effective in deeply nested loops. E.g., consider the following snippet in `noise_median.dcf`:

```

for (i = 1; i < rows-1; i += 1){
  for (j = 1; j < cols-1; j += 1){
    int p1, p2, p3, p4, p5, p6, p7, p8, p9;
    p1 = imageIn[(i-1)*303 + j-1];
    p2 = imageIn[(i-1)*303 + j];
    p3 = imageIn[(i-1)*303 + j+1];
    p4 = imageIn[i*303 + j-1];
    p5 = imageIn[i*303 + j];
    p6 = imageIn[i*303 + j+1];
    p7 = imageIn[(i+1)*303 + j-1];
    p8 = imageIn[(i+1)*303 + j];
    p9 = imageIn[(i+1)*303 + j+1];
  }
}

```

```
}
```

We do not want to generate 9 multiplications in the loop. Naive CSE can reduce the number of multiplications down to 3, by identifying  $(i-1)*303 + j$ ,  $i*303 + j$  and  $(i+1)*303 + j$  as common subexpressions. But CSE cannot relate the three because it does not know multiplication is distributive. PSR does and hence is able to further reduce the number of multiplications down to 1:

```
for (i = 1; i < rows-1; i += 1){
  for (j = 1; j < cols-1; j += 1){
    int p1, p2, p3, p4, p5, p6, p7, p8, p9;
    int t = (i-1)*303 + j - 1;
    p1 = imageIn[t];
    p2 = imageIn[t+1];
    p3 = imageIn[t+2];
    p4 = imageIn[t+303];
    p5 = imageIn[t+304];
    p6 = imageIn[t+305];
    p7 = imageIn[t+606];
    p8 = imageIn[t+607];
    p9 = imageIn[t+608];
  }
}
```

This form is perfect for the downstream assembly generation thanks to other two optimizations we implemented:

- The constant offset can be absorbed into the displacement field of the array load instruction. Hence we don't need to allocate registers or emit `add` instructions for `t+const`.
- The bound checks for all nine array accesses can be fused into one at the top.

We also note that PSR entails the following algebraic simplifications without resorting to special-cased pattern matching:

- Replace `2 * a` with `a + a`.
- Replace `a - a`, `0 * a` with `0`.
- Replace `1 * a` with `a`.
- Replace `-1 * a` with `-a`.

The implementation of PSR can be tricky. PSR is slow because it performs a lot of polynomial operations and hashes polynomials. We optimize by using immutable data structures to enable structural sharing across polynomials and by caching hashes. In addition, we must guard against rare but technically valid instruction sequences that blow up the size of the polynomial exponentially. We therefore limit the max number of terms to 64 and replace a polynomial with a single irreducible term if it exceeds the size limit. Finally, as we are using hashmaps, some care has to be taken to ensure that the pass is deterministic and idempotent.



# Induction Variable Strength Reduction

We also implemented induction variable strength reduction, building on top of our polynomial infrastructure.

In particular, a base induction variable is a phi instruction `i <- phi(...)` defined in a loop header, where the polynomial associated with the phi argument from the backedge differs from the polynomial for `i` by a loop invariant. This broad definition allows our compiler to spot induction variable in arbitrarily long chains and expressed in non-obvious way. For example,

```
int i, j, k, l;  
k = //...  
for (i = 0; i < 10000; i = 1) {  
    j = i + k;  
    l = 2 * j - i + 2;  
}
```

Our compiler can spot the base induction variable `i` with increment  $2*k + 2$ , even if the expression  $2*k + 2$  never appears literally in the program. Simple induction variable analysis with pattern matching can't have spot this. Likewise, the multipliers and offsets of derived induction variable are allowed to be polynomial invariants. Once we found all derived induction variables, we group them by their base induction variables and their multipliers. For each (base induction variable, multiplier), a representative is chosen to be initialized out of the loop and incremented on the backedge. Other derived induction variables within the same groups are replaced in the loop by the representative plus the difference in offset. The representative is chosen heuristically, favoring those used in non-arithmetic instructions (such as array accesses and calls), over those used in additions or multiplications -- because these are likely only intermediate results used to compute the values in the former group.

Because we rely on polynomials, in general, the multiplier or the offset of a derived induction variable may not be readily available in the original program, just like the  $2*k + 2$  in the example above. Thus, this pass must also be able to emit instructions that efficiently computes any polynomial. Evaluating a polynomial with minimum number of arithmetic operations is an open question, and we approximate such a solution with a heuristically constructed multivariate Horner scheme from ["Greedy algorithms for optimizing multivariate Horner schemes"](#) by Cerebio and Kreinovich. The instructions emitted are likely to be redundant if parts of the evaluated polynomial is indeed available. In this case, a subsequent PSR pass + DCE can remove these instructions.

We note that this pass does not eliminate induction variables, nor does it perform linear function test replacement. It would be desirable to have such functionality but we did not have the time to implement it.

## Loop invariant code motion

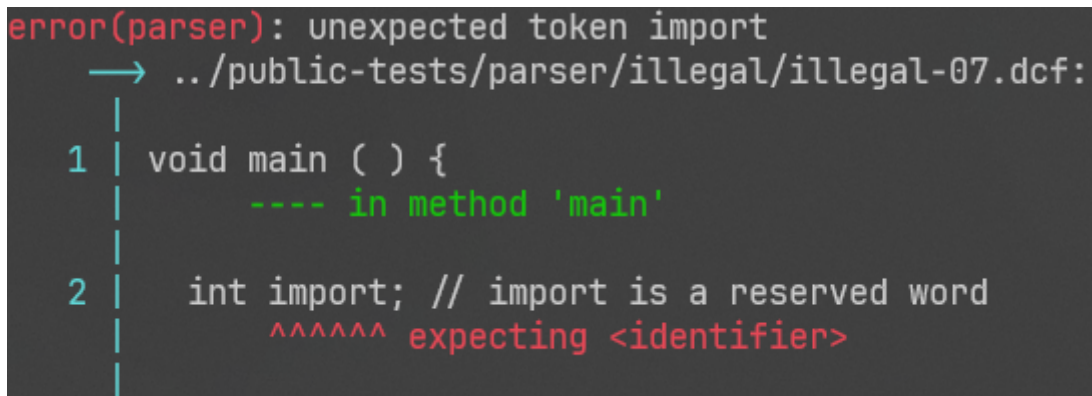
We implemented loop invariant code motion but disabled it under `-o opt` because our implementation is too aggressive -- it always move loop invariants as up as possible. This can increase the register pressure significantly and in many cases the moved loop invariant gets spilled and reloaded inside the loop. As almost all of the loop invariants in the public derby code are loads to global variables, replacing it with a reload does not help in terms of latency. Hence, we do not enable LICM unless explicitly requested with `-o licm`. Implementing LICM in a lazier way is possible (e.g., only move an invariant up by at most one level), but it requires more care to ensure that the optimization is idempotent. In addition, we remark that our induction variable analysis

pass also does some degree of loop invariant code motion and so does GVN-PRE, so in a sense a separate LICM pass does not help much.

## Error Reporting and Diagnosis

---

Our compiler reports error in a user-friendly manner similar to how the Rust compiler does it:



```
error(parser): unexpected token import
→ ../public-tests/parser/illegal/illegal-07.dcf:
1 | void main ( ) {
  |         ---- in method 'main'
2 |   int import; // import is a reserved word
  |         ^^^^^ expecting <identifier>
```

Internally, this is achieved with our `Diagnostic` API in `src/utils/diagnostics.rs`. A `Diagnostic` struct represents a diagnostic message, and includes

- A pre-text, e.g., "error(parser): unexpected token import."
- One or more `DiagnosticItem`s. Each `DiagnosticItem` underlines a span in the source file with some color and supplies a short piece of text to the side.
- A post-text, e.g., "(hint:) consider renaming the variable."

A `Diagnostic` struct can be instantiated with a builder-like API and all of our compiler error types implements `To<Diagnostic>`.

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

---

Implementing so many optimizations is no easy task. We spent a lot of time debugging both the algorithm and the programming language.

Many algorithms maintain intricate invariants that are not obvious by looking at the code, and inadvertent changes could break them easily. Luckily, we implemented utility code that dumps the IR as in dot/graphviz format back in phase 4, and the ability to easily inspect the CFG really helped us debug the algorithm. The extra effort we spent in previous phases to associate emitted assemblies with source code locations also proved invaluable. We are also grateful for the >100 test cases provided by the course staff. We have a test script that runs our compiler on all test cases and we run it before every push -- and seeing all test cases still passing gives us basic confidence that we did not break anything. As Professor Rinard said, implementing optimization in compiler is truly an art of brinkmanship. In addition, we found it to be a practice that rewards very defensive programming.

The decision to use Rust turned out to be a double edged sword. While its strong type system and borrow checker helped us immensely from time to time, we had to fight the borrow checker just as often because it's "too strict." Rust's ownership system does not allow clean expression of graph-like data structures, so we put `Block`s and `Inst`s in vectors and use `BlockRef` and `InstRef`, both essentially `usize`s, to refer to them. This made our code more verbose than it needs to be.

One thing we find it particularly hard given our IR design is to signal to the borrow checker that we only want to modify instructions within blocks but not the CFG. This happens in many cases, for example, consider the following code:

```
for block_ref in method.iter_block_refs() {  
    for inst_ref in method.block(block_ref).insts.iter() {  
        *method.inst_mut(inst_ref) = // updates...  
    }  
}
```

This code does not compile because it's mutating `method` while iterating over it. Although we clearly know that mutating instructions in-place do not invalidate any of the iterators in the loops, the borrow checker does not. This would not have been a problem if Rust allows partial self borrowing. Admittedly, we could have worked around this problem by redesigning our IR representation, i.e., the `Method` struct, but this becomes a harder and harder decision as more and more optimizations depend on its current representation. We were able to work around the issues locally by cloning the iterators or the `vec`s behind them before we iterate through them, but they are slower and ugly. Every codebase has within it a certain degree of ugliness. Our fights against the borrow checker contribute a significant portion of ugliness in our code. We are happy that by the time we finish phase 5 such ugliness has not yet spiraled out of control, but given more time and another chance we would have made more deliberated decisions on how our IR is represented in terms of Rust language constructs.

## Contributions

---

It would not have been possible to implement so many optimizations in time without contributions from every team members. All team members meet to discuss all optimizations implemented. Nathan implemented constant folding, the phase 3 assembler, function inlining, and Godbolt/docker debugging infrastructure; Siyong single handedly implemented GVN-PRE, the most complex optimization in our compiler; Chengyuan implemented the register allocation pipeline, relevant optimizations, PSR, RGAE, and induction variable; Jacob implemented more assembly-level optimizations such as constant divisor/modulo strength reduction.