The syntax tree is a useful intermediate representation (IR) that is independent of both the source language and the target ISA. It contains information about the sequencing and grouping of operations that isn't apparent in individual machine language instructions.

And it allows frontends for different source languages to share a common backend targeting a specific ISA. As we'll see, the backend processing can be split into two sub-phases. The first performs machine-independent optimizations on the IR. The optimized IR is then translated by the code generation phase into sequences of instructions for the target ISA.

A common IR is to reorganize the syntax tree into what's called a control flow graph (CFG).

Each node in the graph is a sequence of assignment and expression evaluations that ends with a branch. The nodes are called "basic blocks" and represent sequences of operations that are executed as a unit.

Once the first operation in a basic block is performed, the remaining operations will also be performed without any other intervening operations.

This knowledge lets us consider many optimizations, e.g., temporarily storing variable values in registers, that would be complicated if there was the possibility that other operations outside the block might also need to access the variable values while we were in the middle of this block. The edges of the graph indicate the branches that take us to another basic block. For example, here's the CFG for GCD.

If a basic block ends with a conditional branch, there are two edges, labeled "T" and "F" leaving the block that indicate the next block to execute depending on the outcome of the test.

Other blocks have only a single departing arrow, indicating that the block always transfers control to the block indicated by the arrow. Note that if we can arrive at a block from only a single predecessor block, then any knowledge we have about operations and variables from the predecessor block can be carried over to the destination block.

For example, if the "if (x > y)" block has generated code to load the values of x and y into registers, both destination blocks can use that information and use the appropriate registers without having to generate their own LDs.

But if a block has multiple predecessors, such optimizations are more constrained.

We can only use knowledge that is common to *all* the predecessor blocks.

The CFG looks a lot like the state transition diagram for a high-level FSM!

We'll optimize the IR by performing multiple passes over the CFG.

Each pass performs a specific, simple optimization. We'll repeatedly apply the simple optimizations in multiple passes, until we can't find any further optimizations to perform.

Collectively, the simple optimizations can combine to achieve very complex optimizations.

Here are some example optimizations: We can eliminate assignments to variables that are never used and basic blocks that are never reached.

This is called "dead code elimination". In constant propagation, we identify variables that have a constant value and substitute that constant in place of references to the variable. We can compute the value of expressions that have constant operands. This is called "constant folding".

To illustrate how these optimizations work, consider this slightly silly source program and its CFG. Note that we've broken down complicated expressions into simple binary operations, using temporary variable names (e.g, "_t1") to name the intermediate results. Let's get started!

The dead code elimination pass can remove the assignment to Z in the first basic block since Z is reassigned in subsequent blocks and the intervening code makes no reference to Z. Next we look for variables with constant values.

Here we find that X is assigned the value of 3 and is never re-assigned, so we can replace all references to X with the constant 3. Now perform constant folding [CLICK], evaluating any constant expressions. Here's the updated CFG, ready for another round of optimizations. First dead code elimination.

Then constant propagation. And, finally, constant folding.

So after two rounds of these simple operations, we've thinned out a number of assignments.

On to round three! Dead code elimination.

And here we can determine the outcome of a conditional branch, eliminating entire basic blocks from the IR, either because they're now empty or because they can no longer be reached. Wow, the IR is now considerably smaller.

Next is another application of constant propagation. And then constant folding.

Followed by more dead code elimination. The passes continue until we discover there are no further optimizations to perform, so we're done!

Repeated applications of these simple transformations have transformed the original program into an equivalent program that computes the same final value for Z.

We can do more optimizations by adding passes: eliminating redundant computation of common subexpressions, moving loop-independent calculations out of loops, unrolling short loops to perform the effect of, say, two iterations in a single loop execution, saving some of the cost of increment and test instructions.

Optimizing compilers have a sophisticated set of optimizations they employ to make smaller and more efficient code. Okay, we're done with optimizations.

Now it's time to generate instructions for the target ISA.

First the code generator assigns each variable a dedicated register.

If we have more variables than registers, some variables are stored in memory and we'll use LD and ST to access them as needed. But frequently-used variables will almost certainly live as much as possible in registers. Use our templates from before to translate each assignment and operation into one or more instructions.

The emit the code for each block, adding the appropriate labels and branches.

Reorder the basic block code to eliminate unconditional branches wherever possible.

And finally perform any target-specific peephole optimizations.

Here's the original CFG for the GCD code, along with the slightly optimized CFG.

GCD isn't as trivial as the previous example, so we've only been able to do a bit of constant propagation and constant folding. Note that we can't propagate knowledge about variable values from the top basic block to the following "if" block since the "if" block has multiple predecessors. Here's how the code generator will process the optimized CFG. First, it dedicates registers to hold the values for x and y. Then, it emits the code for each of the basic blocks. Next, reorganize the order of the basic blocks to eliminate unconditional branches wherever possible.

The resulting code is pretty good. There no obvious changes that a human programmer might make to make the code faster or smaller. Good job, compiler!

Here are all the compilation steps shown in order, along with their input and output data structures. Collectively they transform the original source code into high-quality assembly code. The patient application of optimization passes often produces code that's more efficient than writing assembly language by hand.

Nowadays, programmers are able to focus on getting the source code to achieve the desired functionality and leave the details of translation to instructions in the hands of the compiler.