Now let's turn our attention to compile statement.

The first two statement types are pretty easy to handle.

Unconditional statements are usually assignment expressions or procedure calls.

We'll simply ask compile_expr to generate the appropriate code.

Compound statements are equally easy.

We'll recursively call compile_statement to generate code for each statement in turn.

The code for statement_2 will immediately follow the code generated for statement_1.

Execution will proceed sequentially through the code for each statement.

Here we see the simplest form the conditional statement, where we need to generate code to evaluate the test expression and then, if the value in the register is FALSE, skip over the code that executes the statement in the THEN clause.

The simple assembly-language template uses recursive calls to compile_expr and compile_statement to generate code for the various parts of the IF statement.

The full-blown conditional statement includes an ELSE clause, which should be executed if the value of the test expression is FALSE.

The template uses some branches and labels to ensure the course of execution is as intended.

You can see that the compilation process is really just the application of many small templates that break the code generation task down step-by-step into smaller and smaller tasks, generating the necessary code to glue all the pieces together in the appropriate fashion.

And here's the template for the WHILE statement, which looks a lot like the template for the IF statement with a branch at the end that causes the generated code to be re-executed until the value of the test expression is FALSE.

With a bit of thought, we can improve on this template slightly.

We've reorganized the code so that only a single branch instruction (BT) is executed each iteration, instead of the two branches (BF, BR) per iteration in the original template.

Not a big deal, but little optimizations to code inside a loop can add up to big savings in a long-running program.

Just a quick comment about another common iteration statement, the FOR loop.

The FOR loop is a shorthand way of expressing iterations where the loop index ("i" in the example shown) is run through a sequence of values and the body of the FOR loop is executed once for each value of the loop index.

The FOR loop can be transformed into the WHILE statement shown here, which can then be compiled using the templates shown above.

In this example, we've applied our templates to generate code for the iterative implementation of the factorial function that we've seen before.

Look through the generated code and you'll be able to match the code fragments with the templates from last couple of slides.

It's not the most efficient code, but not bad given the simplicity of the recursive-descent approach for compiling high-level programs.

It's a simple matter to modify the recursive-descent process to accommodate variable values that are stored in dedicated registers rather than in main memory.

Optimizing compilers are quite good at identifying opportunities to keep values in registers and hence avoid the LD and ST operations needed to access values in main memory.

Using this simple optimization, the number of instructions in the loop has gone from 10 down to 4.

Now the generated code is looking pretty good!

But rather than keep tweaking the recursive-descent approach, let's stop here.

In the next segment, we'll see how modern compilers take a more general approach to generating code.

Still though, the first time I learned about recursive descent, I ran home to write a simple implementation and marveled at having authored my own compiler in an afternoon!