The Anatomy of a Compiler

The work of a compiler can be split into 5 major activities. The last of these is code optimization and time won't allow us to do much with that. The other 4 activities result in a working compiler that generates correct but rather inefficient code. An actual working compiler can break these activities up in many different ways, but in principle there are just a few primary modules:

The Scanner.

This does lexical analysis -- it takes as input a program in the source language and produces a stream of tokens representing the individual words -- identifiers, operators, punctuation symbols, and so forth -- of the program.

The Parser

This does syntatic analysis. Almost all languages since Algol-60 have been defined in terms of grammars. The parser takes as input the token stream from the Scanner and produces an intermediate representation for the program based on the language's grammar. This representation can take various forms; the simplest and most canonical is a parse tree. An important sideline for the Parser is to generate error messages when it finds constructs in the input program that do not match the grammar rules.

The Type Checker

This does semantic analysis. The input is the intermediate structure generated by the Parser; in our case this will be a parse tree. There are two stages to semantic analysis. One is resolving references -- connecting each use of an identifier to its declaration. The other stage is actual type-checking -- determining the type of every expression in the language and ensuring that it is type-correct. Of course, error messages are generated for any semantic errors (errors not caught by the Parser) that are detected. The Type Checker should modify the intermediate structure in ways to make it easy to generate code.

The Code Generator

This walks along the intermediate representation, as modified by the Type Checker, and generates assembly language code for the program. This code can be fed into an assembler, producing executable machine code equivalent to the source program.

It is not difficult to generate code from a good intermediate representation. Unfortunately, the code that is generated is incredibly inefficient, far worse than any human programmer would ever design. Production compilers use a further stage for optimization. There are a variety of optimization tasks that can take place at various times during the compilation process:

- Register allocation algorithms make optimal use of the available registers; this needs to happen during code generation.
- Source code optimizers replace portions of the input program with equivalent code that can be implemented more efficiently.
- Assembly code optimizers use a window of 6, or 7, or more statements in the assembly code and eliminate redundant or inefficient statements. This is a post-processing step.