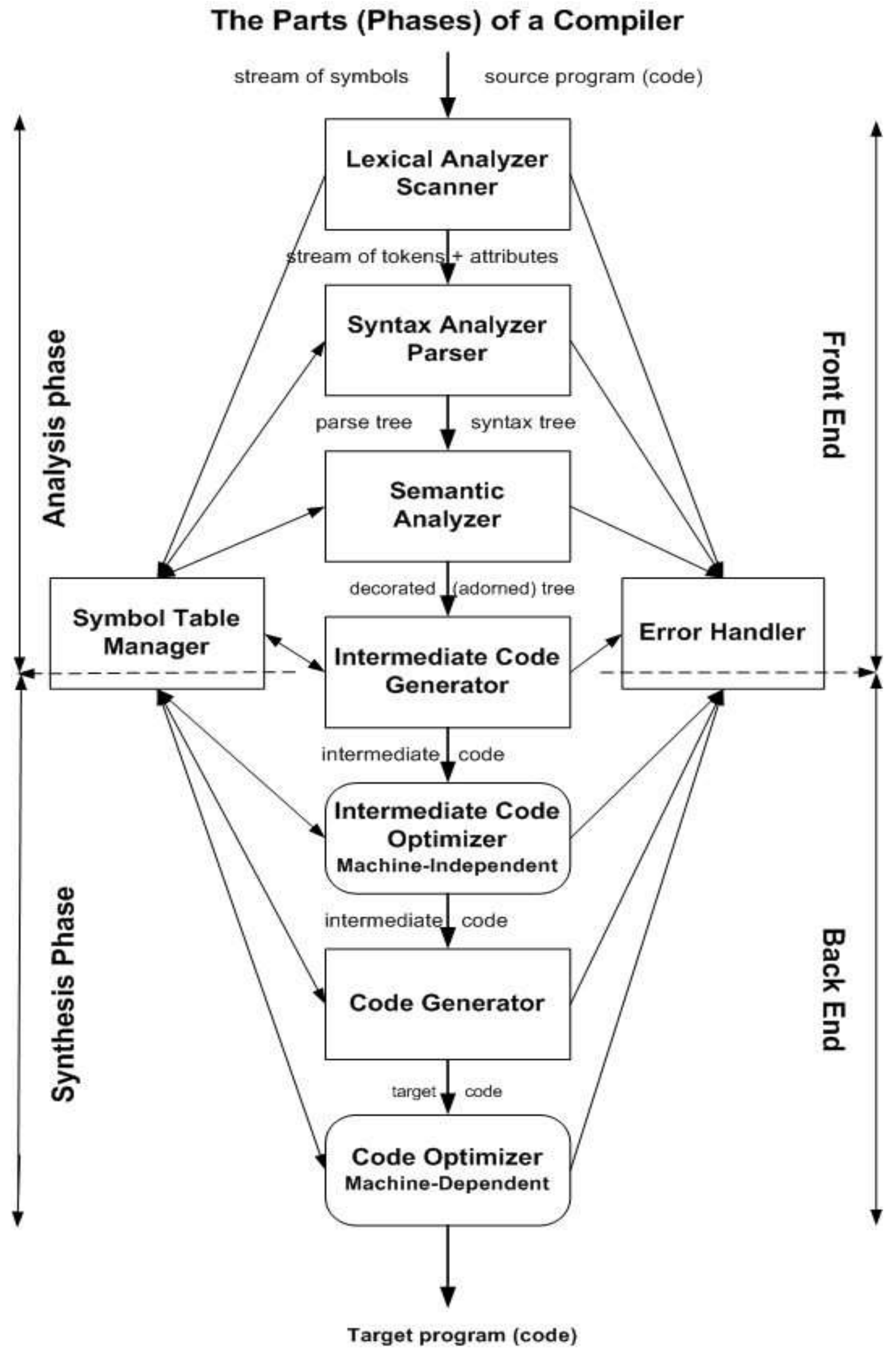
CST8152 – Compilers
Article #4
The Inner Workings of a Compiler



The ***analysis part*** (***phase***) breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program.

The ***synthesis part*** (***phase***) constructs the desired target program from the intermediate representation and the information in the symbol table.

The analysis part is often called the ***front end*** of the compiler; the synthesis part is the ***back end***.

The first phase of a compiler is called ***lexical analysis*** or ***scanning****.* The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form: (token-name (code), attribute-value).

The second phase of the compiler is ***syntax analysis*** or ***parsing***. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation called *parse tree* that depicts the grammatical structure of the token stream. Typically the parse tree is reduced to another form of representation called *syntax tree* in which each interior node represents an operation and the children of the node represent the arguments of the operation.

The ***semantic analyzer*** uses the syntax tree and the information in the ***symbol table*** to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

In the process of translating a source program into target code, a compilers construct one or more intermediate representations, which can have a variety of forms. For example, syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis. After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or ***machine-like intermediate representation***, which we can think of as a program for an *abstract machine*. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine. Typically the intermediate representation separates the front end from the back end.

The **machine-independent code-optimization** phase attempts to improve the intermediate code so that better target code will result.

There is a great variation in the amount of ***code optimization*** different compilers perform. In those that do the most, the so-called "optimizing compilers," a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.

The ***code generator*** takes as input an intermediate representation of the source program and maps it into the target language

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned. The ***symbol table*** is a data structure containing a record for each variable name, with fields for the attributes of the name.

The ***error handler*** is responsible to report to the programmer the lexical, syntactical, and the semantic error discovered during the compilation process. It is also responsible to prevent the compiler from producing a target code is an error has been detected.

The discussion of ***phases*** deals with the logical organization of a compiler. In an implementation, activities from several phases may be grouped together into a ***pass*** that reads an input file and writes an output file. For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass. Code optimization might be an optional pass. Then there could be a back-end pass consisting of code generation for a particular target machine.

"*Why is that when a programmer writes code it is democracy but when a programmer runs a compiler it is a dictatorship?*" Confused Philosopher