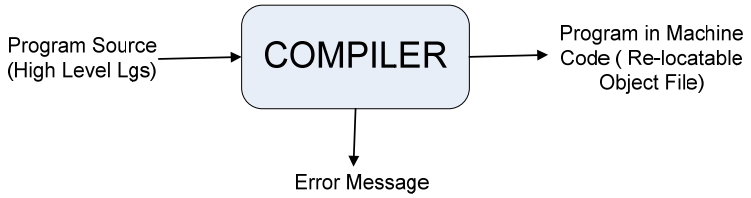
1.1 Definition

A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.



Some Applications

In addition to the development of a compiler, the techniques used in compiler design can be applicable to many problems in computer science.

- Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
- Techniques used in a parser can be used in a query processing system such as SQL.
- Much software having a complex front-end may need techniques used in compiler design.
  - A symbolic equation solver program which takes an equation as input and that program should parse the given input equation.
- Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems.

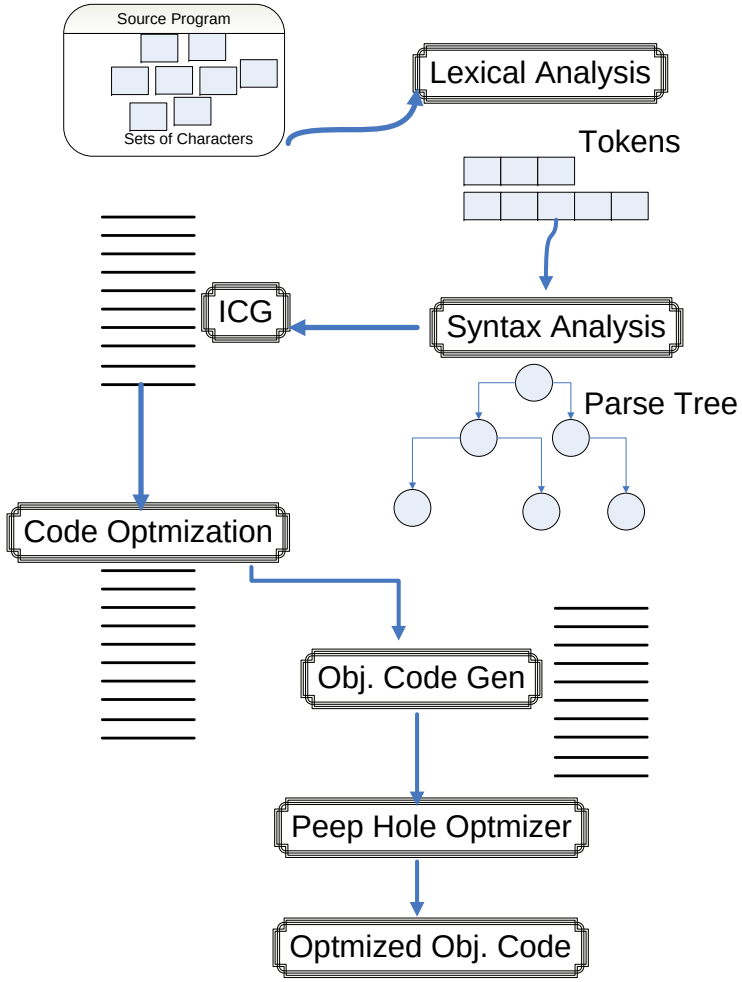
1.2 Major parts of Compiler

There are two major parts of a compiler: **Analysis** and **Synthesis.**

(a) Analysis: An intermediate representation is created from the given source program. Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.

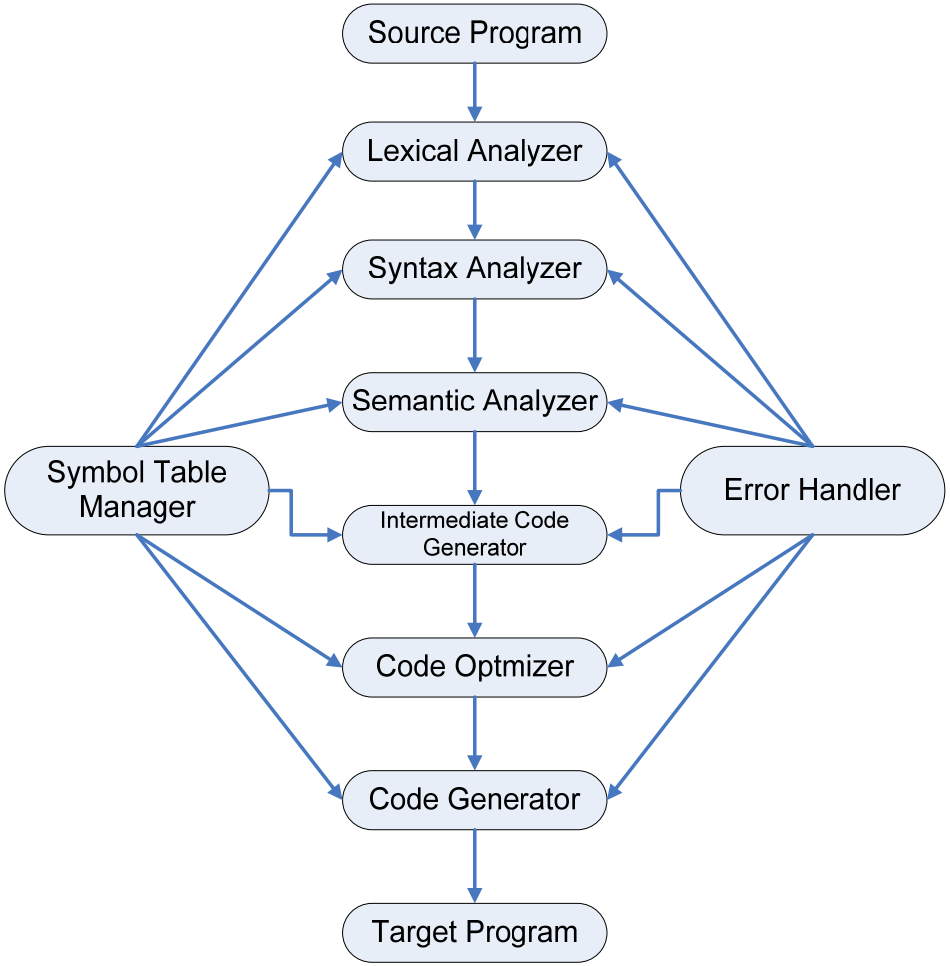
(b) Synthesis: The equivalent target program is created from this intermediate representation. Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

1.3 Process within Compiler [Load-Store-Execute]



1.4 Phases of Compiler

A compiler operates in phases, each of which transforms the source programs from one representation to another. A typical decomposition of compiler is shown below.



1.4.1 Lexical Analyzer

- *Lexical Analyzer* reads the source program character by character and returns the *tokens* of the source program.
- A *token* describes a pattern of characters having same meaning in the source program. (Such as identifiers, operators, keywords, numbers, delimiters and so on)

e.g.: *newval = oldval + 12*

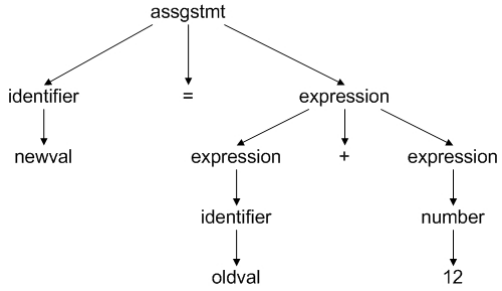
tokens:

newval	identifier
=	assignment operator
oldval	identifier
+	add operator
12	a number

- Keeps information of identifier into the symbol table.
- *Regular expressions* are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

1.4.2 Syntax Analyzer

- A *Syntax Analyzer* creates the syntactic structure (generally a parse tree) of the given program.
- A *Syntax Analyzer* is also called as a *Parser*.
- A *Parse Tree* describes a syntactic structure of a program statement.



- The syntax of a language is specified by a *Context Free Grammar (CFG).*
- The rules in a CFG are mostly recursive.

– A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not. If it satisfies, the syntax analyzer creates a parse tree for the given program.

e.g.: We use BNF (Backus Naur Form) to specify a CFG

assgstmt → *identifier* = *expression*

expression → *identifier*

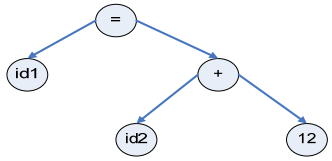
expression → *number*

expression → *expression* + *expression*

1.4.3 Syntax Analyzer vs. Lexical Analyzer

- Which constructs of a program does recognize by the lexical analyzer, and which ones by the syntax analyzer?
  - Both of them do similar things; but the lexical analyzer deals with simple non-recursive constructs of the language.
  - The syntax analyzer deals with recursive constructs of the language.
  - The lexical analyzer simplifies the job of the syntax analyzer.
  - The lexical analyzer recognizes the smallest meaningful units (tokens) in a source program.
  - The syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognize meaningful structures in our programming language.

  e.g.: *newval = oldval + 12*



1.4.4 Parsing Techniques

Depending on how the parse tree is created, there are different parsing techniques. These parsing techniques are categorized into two groups: *Top-Down Parsing and Bottom-Up Parsing.*

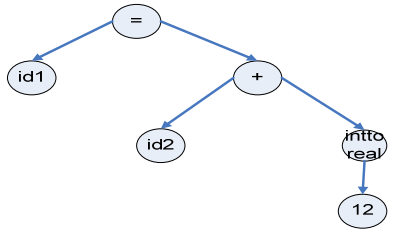
- Top-Down Parsing:
  - Construction of the parse tree starts at the root, and proceeds towards the leaves.
  - Efficient top-down parsers can be constructed easily by hand.
  - Recursive Predictive Parsing, Non-Recursive Predictive Parsing (LL Parsing) examples of this type.
- Bottom-Up Parsing:
  - Construction of the parse tree starts at the leaves, and proceeds towards the root.
  - Normally efficient bottom-up parsers are created with the help of some software tools.
  - Bottom-up parsing is also known as shift-reduce parsing.
  - Operator-precedence parsing is simple, restrictive and easy to implement.
  - LR Parsing are general form of shift-reduce parsing. e.g. CLR, SLR and LALR.

1.4.5 Semantic Analyzer

- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- Type-checking is an important part of semantic analyzer.
- Normally semantic information cannot be represented by a context-free grammar used in syntax analyzers.
  - Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules) and the results are syntax-directed translation, Attribute grammars.

- e.g.:

  newval = oldval + 12



The type of the identifier *newval* must match with type of the expression *(oldval+12)*

1.4.6 Intermediate Code Generation

- A compiler may produce an explicit intermediate codes representing the source program.
- These intermediate codes are generally machine (architecture independent). But the level of intermediate codes is close to the level of machine codes.
- e.g.:

  *newval = oldval * fact + 1*
  *id1 = id2 * id3 + 1*

  Intermediates Codes (Quadruple representation)

*	*id2*	*id3*	*t1*
+	*t1*	*#1*	*t2*
=	*t2*		*id1*

1.4.7 Code Optimizer

- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.
- e.g.:

*	*id2*	*id3*	*t1*
+	*t1*	*#1*	*t2*
=	*t2*		*id1*

→

*	*id2*	*id3*	*t1*
+	*t1*	*#1*	*id1*

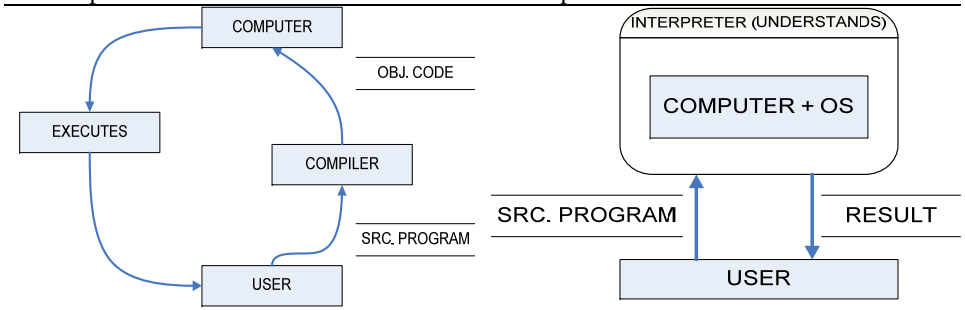
1.4.8 Code Generator

- Produces the target language for a specific architecture.
- The target program is normally is a re-locatable object file containing the machine codes.
- e.g.: Assume that we have an architecture with instructions whose at least one of its operands is a machine register.

  MOVE id_2, R_1
  MULT id_3, R_1
  ADD *#1*, R_1
  MOVE R_1, id_1

1.5 Compiler Vs Interpreter

Compiler	***Interpreter***
It translates the source program and creates the object code and executes only after a execution command is given	It translates one instruction at a time and executes immediately. *It is software which runs in a computer system to simulate a hypothetical system for which the machine level lgs becomes the source lgs.*
It is efficient (Faster).	It is not efficient in terms of program execution (Slower).
Detects all errors at time.	Detects one error at a time.
It is complex to implement compared to interpreter.	It is simple and easy.
It is not good for interactive program development.	It is good for interactive program development.



1.6 Symbol Table Management

An essential function of a compiler is to record the identifier used in the source program and collect the transformation about various attributes of each identifier.

A symbol table is a data structure containing a record for each identifier with fields for the attributes of the identifier. This data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. When an identifier in the source program is detected by the lexical analyzer, the identifier is extended into the symbol table, also the remaining phase enter information about the identifier.

This is the process responsible for maintaining the various symbols (tokens) used in the source code or generated by compiler.

1.7 Error Management (Detection & Reporting)

Each phase can encounter errors, however after detecting an error, a phase must deal somehow with that error, so that computation can proceed, allowing further errors in the source program to be detected. This is the user interface of the compiler which manages the errors found in the different phases and displays appropriate error message.

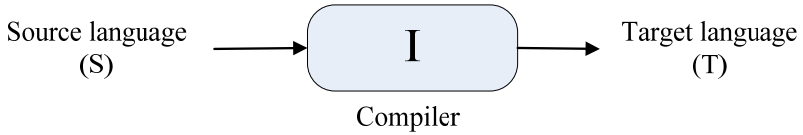
The syntax and semantic analysis usually handle a large fraction of errors detectable by the computer. The lexical phase can detect error where the characters remaining in the input do not form any token of the language and are determined in the syntax analysis phase. Most of the general programming errors are positioned on the below categories:

- Syntax Errors
- Run-Time Errors
- Logic Errors

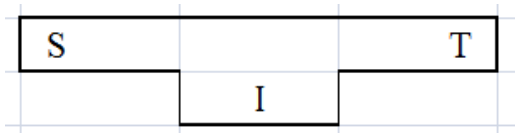
1.8 Bootstrapping

It is an important concept in building new compilers. There are three languages involves for a compiler construction. They are:

1. Source language (S), which we are creating.
2. Target language (T)
3. The language (I), in which the compiler is written.



This can be written in T-diagram shown below.



The above T-diagram is denoted by C_I^{ST}. Using this concept we can create a compiler that may run on one machine and produce target code for some other machines. Such a compiler is called *Cross compiler*.