Introduction to Compiling

Mrs. Sunita M Dol

(sunitaaher@gmail.com)

Assistant Professor, Dept of Computer Science and Engineering

Walchand Institute of Technology, Solapur

(www.witsolapur.org)

Introduction to Compiling

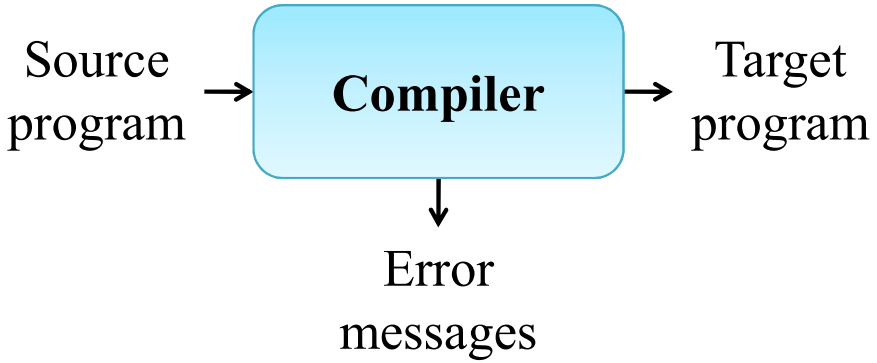
- Introduction
- Compilers
- Phases of a compiler
- Compiler construction tools
- A simple one pass compiler

Introduction

- All the software running on all the computers was written in some programming language.
- But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.
- The software systems that do this translation are called **compilers**.

Compilers

- A compiler is a program that can read a program in one language - the source language and translate it into an equivalent program in another language - the target language

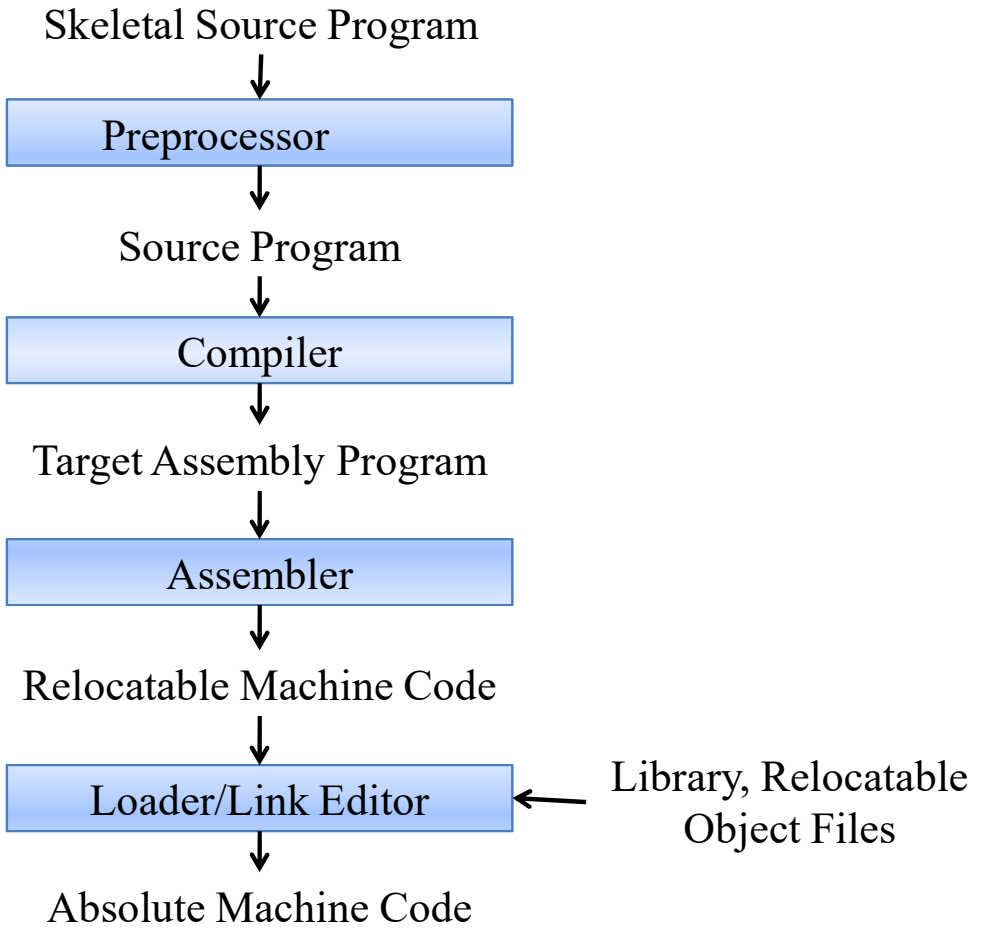


Compilers

- **Analysis and Synthesis model of compilation**
  - There are two parts of compilation
    - ✓ **Analysis part**: breaks up the source program into constituents pieces and creates the intermediate representation of the source program.
      - Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
    - ✓ **Synthesis part**: constructs the desired target program from the intermediate representation
      - Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

Introduction

- **Language Processing System**



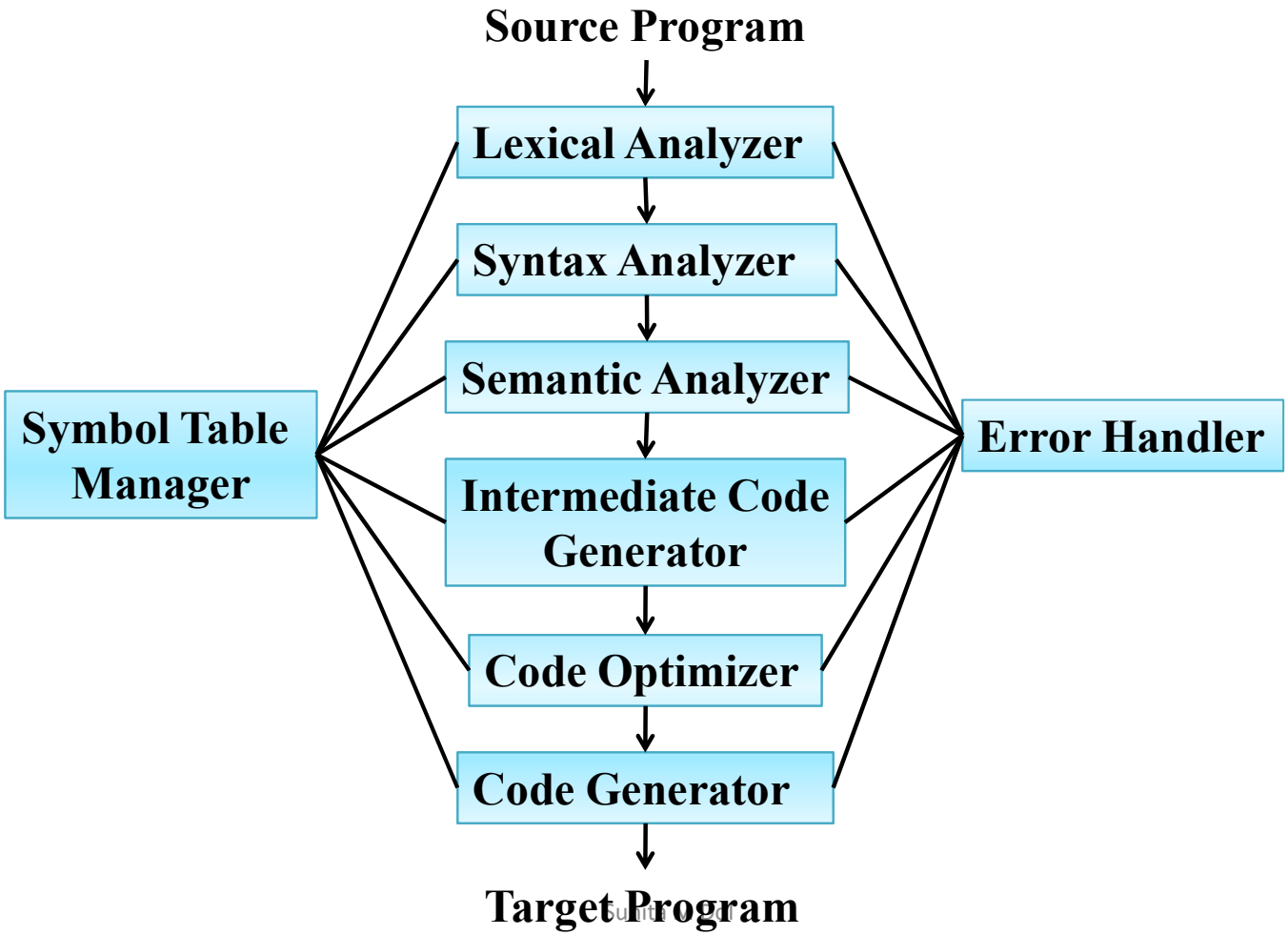
Compilers

- **Language Processing System**
  - In addition to a compiler, several other programs may be required to create an executable target program.
  - A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a distinct program, called a **preprocessor**.
  - The preprocessor may also expand shorthands, called macros, into source language statements.
  - The target program created by the **compiler** may require further processing before it can be run.
  - The compiler creates assembly code that is translated by an **assembler** into machine code and then linked together with some library routines into the code that actually runs on the machine.

Phases of Compiler

- Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another.
- The first three phases form the bulk of the analysis portion of a compiler.
- Two other activities, symbol-table management and error handling, are shown interacting with the six phases of lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and code generation.
- Informally, we shall also call the symbol-table manager and the error handler "phases."

Phases of Compiler



Phases of Compiler

- **Lexical Analyzer**
  - It 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, delimeters and so on)

    Example: The assignment statement position = initial + rate * 60

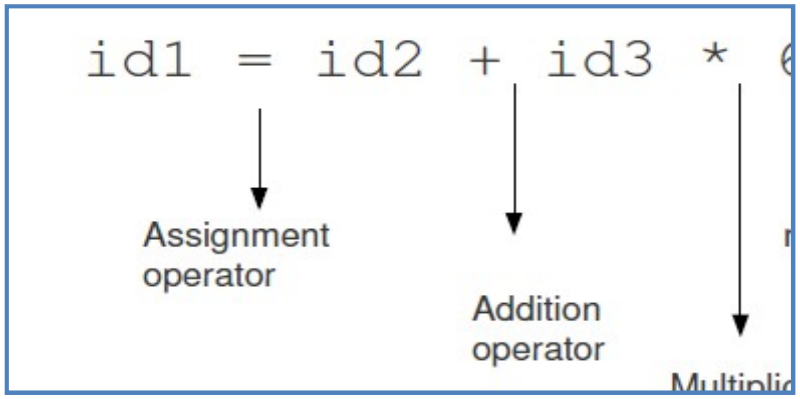
    The scanner throws:

    position : identifier
    = : assignment operator
    initial : identifier
    + : addition operator
    rate : identifier
    * : multiplication operator
    60 : number

  - It puts information about identifiers into the symbol table.

Phases of Compiler

- **Lexical Analyzer**
  - After Lexical Analysis



  - 7 tokens
  - 3 identifiers are put into the symbol table
  - If there are errors, they are reported.

Phases of Compiler

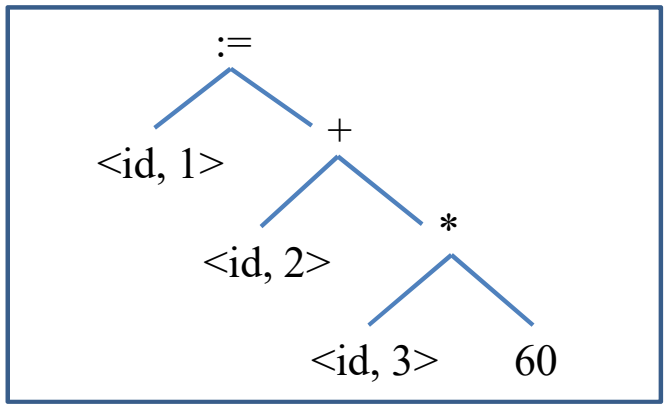
- **Lexical Analyzer**
  - position is a lexeme that would be mapped into a token <id,1>, where id is an abstract symbol standing for identifier and 1 points to the symbol table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.
  - The assignment symbol = is a lexeme that is mapped into the token <=>. Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as assign for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

Phases of Compiler

- **Lexical Analyzer**
  - initial is a lexeme that is mapped into the token <id, 2> where 2 points to the symbol-table entry for initial.
  - + is a lexeme that is mapped into the token <+>.
  - rate is a lexeme that is mapped into the token <id, 3>, where 3 points to
  - the symbol-table entry for rate.
  - * is a lexeme that is mapped into the token <*>.
  - 60 is a lexeme that is mapped into the token <60>

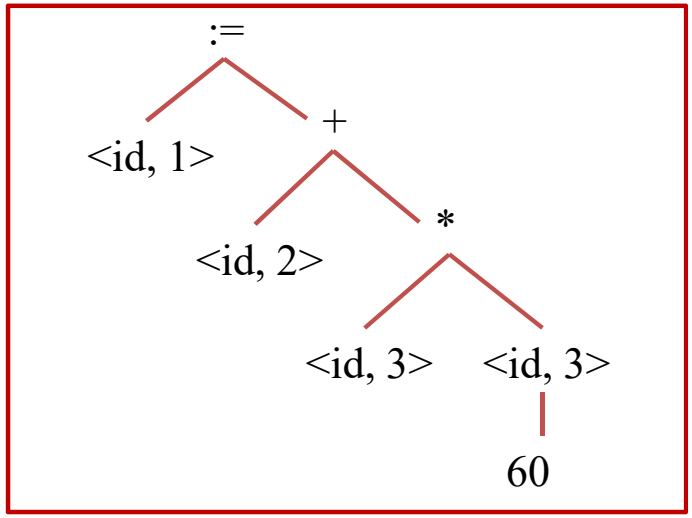
Phases of Compiler

- **Syntax Analysis**
  - A Syntax Analyzer creates the syntactic structure (generally a parse tree) of the given program.
  - A syntax analyzer is also called as a **parser or hierarchical analysis**.
  - A parse tree describes a syntactic structure.
  - In a parse tree, all terminals are at leaves.
  - All inner nodes are non-terminals in a context free grammar.



Phases of Compiler

- **Semantic Analysis**
  - 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 language used in syntax analyzers.
  - Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules)



Phases of Compiler

- **Intermediate Code Generator**
  - 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.
  - The output of the intermediate code generator consists of the three-address code sequence.

```
t1 = inttoreal(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

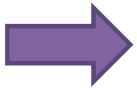
Phases of Compiler

- **Intermediate Code Generator**
  - There are several points worth noting about three-address instructions.
    - ✓ First, each three-address assignment instruction has at most one operator on the right side. Thus, these instructions x the order in which operations are to be done; the multiplication precedes the addition in the source program.
    - ✓ Second, the compiler must generate a temporary name to hold the value computed by a three-address instruction.
    - ✓ Third, some three-address instructions have fewer than three operands

Phases of Compiler

- **Code Optimization**
  - The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
  - The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the inttoreal operation can be eliminated by replacing the integer 60 by the floating-point number 60.0.

```
t1 = inttoreal(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```



```
t1 = id3 * 60.0
id1 = id2 + t1
```

Phases of Compiler

- **Code Generation**
  - The code generator takes as input an intermediate representation of the source program and maps it into the target language.
  - If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

  - The first operand of each instruction specifies a destination.
  - The F in each instruction tells us that it deals with floating-point numbers.

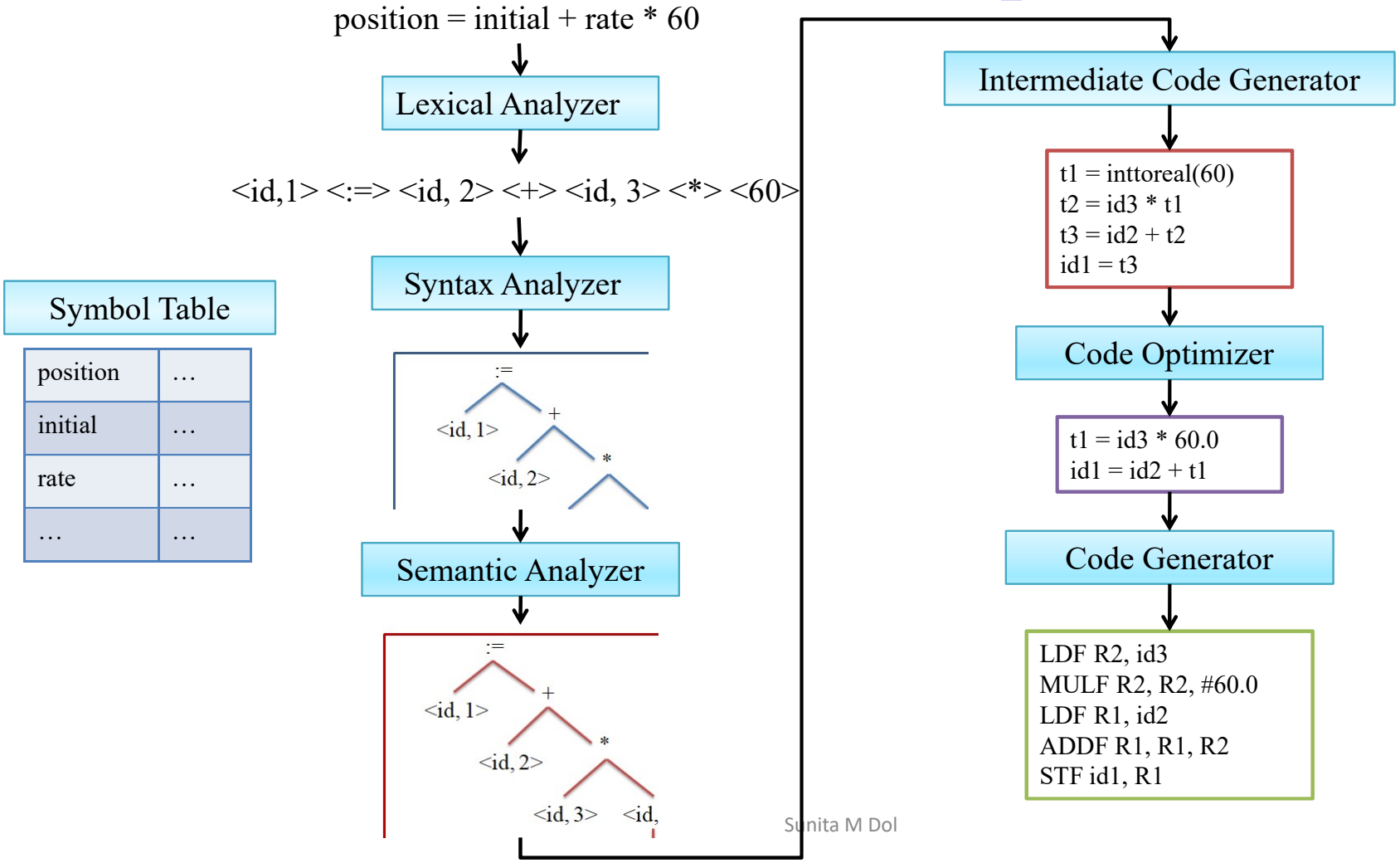
Phases of Compiler

- **Symbol Table Manager**
  - 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 data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly

Phases of Compiler

- **Error Handler**
  - Each phase can encounter errors.
  - After detecting an error, a phase must somehow deal with that error so that compilation can proceed allowing further error in the source program to be detected.
    - ✓ **Lexical error**: e.g. misspelling an identifier
    - ✓ **Syntactic error**: e.g. arithmetic expression with unbalanced parentheses
    - ✓ **Semantic error**: e.g. operator applied to incompatible operand
    - ✓ **Logical error**: e.g. infinitely recursive call

Phases of Compiler



Compiler Construction Tools

- The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on. In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler.
- These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms. The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler.

Compiler Construction Tools

- Some commonly used compiler-construction tools include
  - **Parser generators** that automatically produce syntax analyzers from a grammatical description of a programming language.
  - **Scanner generators** that produce lexical analyzers from a regular-expression description of the tokens of a language.
  - **Syntax-directed translation** engines that produce collections of routines for walking a parse tree and generating intermediate code.

Compiler Construction Tools

- Some commonly used compiler-construction tools include
  - **Code-generator** that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
  - **Data-flow analysis** engines that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.
  - **Compiler-construction** toolkits that provide an integrated set of routines for constructing various phases of a compiler.

References

- Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullmann “Compilers- Principles, Techniques and Tools”, Pearson Education.

Thank You !!!