Phase of a Compiler

Course Name: Compiler Design
Course Code: CSE331
Level:3, Term:3
Department of Computer Science and Engineering
Daffodil International University

The Structure of compiler is divided into two parts:

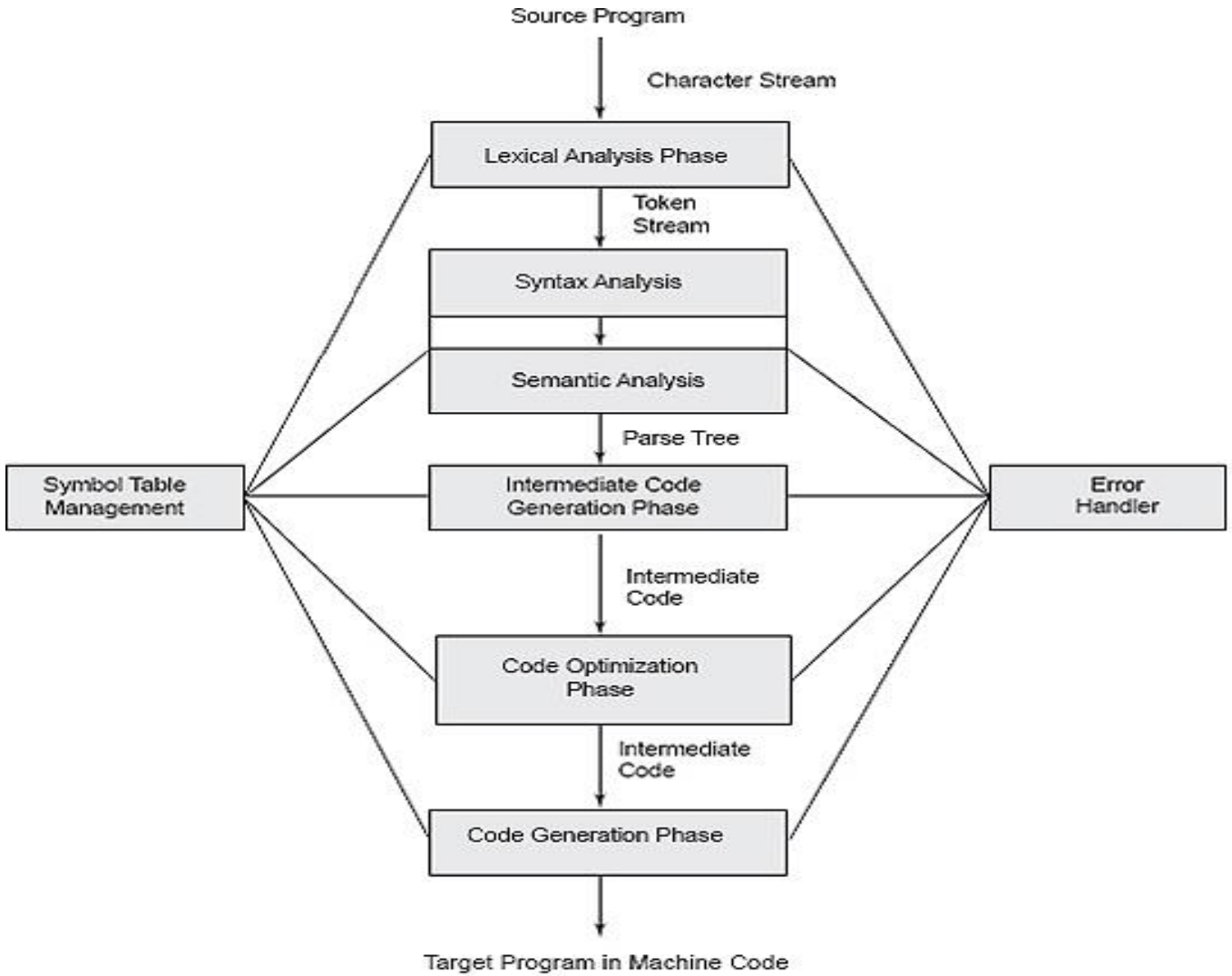
1. **Analysis**
2. **Synthesis**

The Structure of a Compiler

- The analysis part 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.
- If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.
- The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.
- The synthesis part 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.

Phases of a compiler

The compilation process operates as a sequence of phases, each of which transforms one representation of the source program to another. The symbol table, which stores information about entire source program, is used by all phases of the compiler.



Phase of a Compiler

Phases of a compiler

- **1. Lexical Analysis Phase:** 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, attribute-value}

- That it passes on to the subsequent phase, syntax analysis . In the token, the first component token- name is an abstract symbol that is used during syntax analysis , and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol- table entry 'is needed for semantic analysis and code generation.

For example, suppose a source program contains the assignment statement

position = initial + rate * 60…… (1.1)

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. 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.
2. 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
3. initial is a lexeme that is mapped into the token (id, 2) , where 2 points to the symbol-table entry for initial .
4. + is a lexeme that is mapped into the token (+).
5. rate is a lexeme that is mapped into the token (id, 3) , where 3 points to the symbol-table entry for rate.
6. * is a lexeme that is mapped into the token (*) .
7. 60 is a lexeme that is mapped into the token (60) .

Blanks separating the lexemes would be discarded by the lexical analyzer. Figure 1.7 shows the representation of the assignment statement (1.1) after lexical analysis as the sequence of tokens

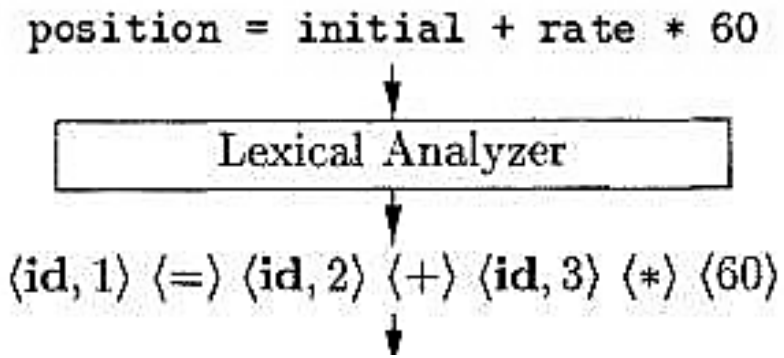
(id, l) (=) (id, 2) (+) (id, 3) (*) (60) ……..(1.2)

In this representation, the token names =, +, and * are abstract symbols for the assignment, addition, and multiplication By Jaoydpeeepr aPattoilrs, respectively.

Phases of a compiler

1	position	···
2	initial	···
3	rate	···

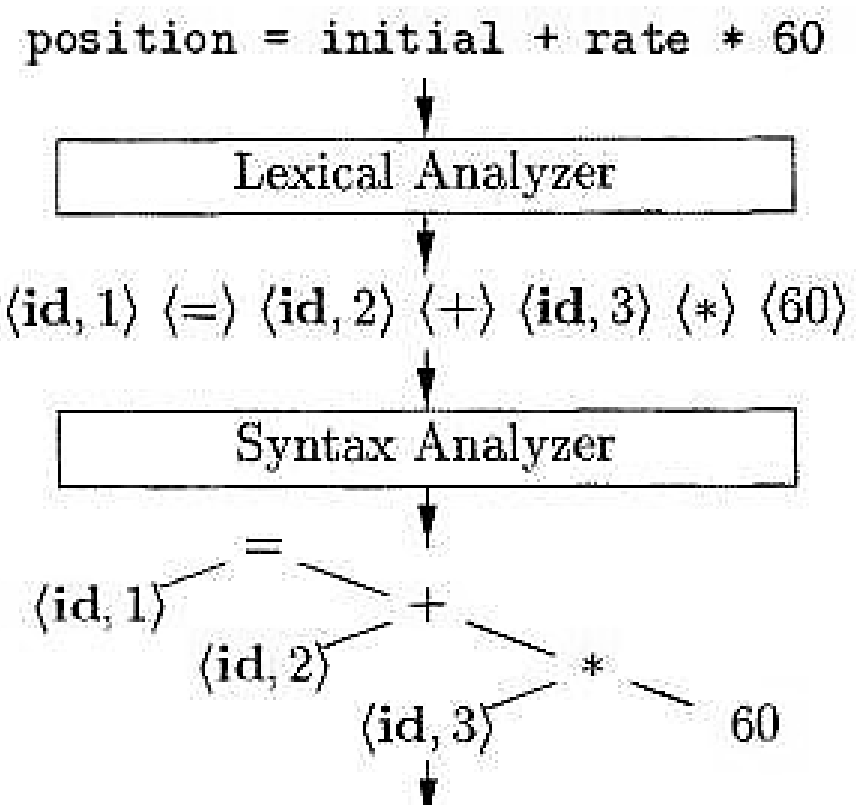
SYMBOL TABLE



Phases of a compiler

Syntax Analysis: 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 that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.

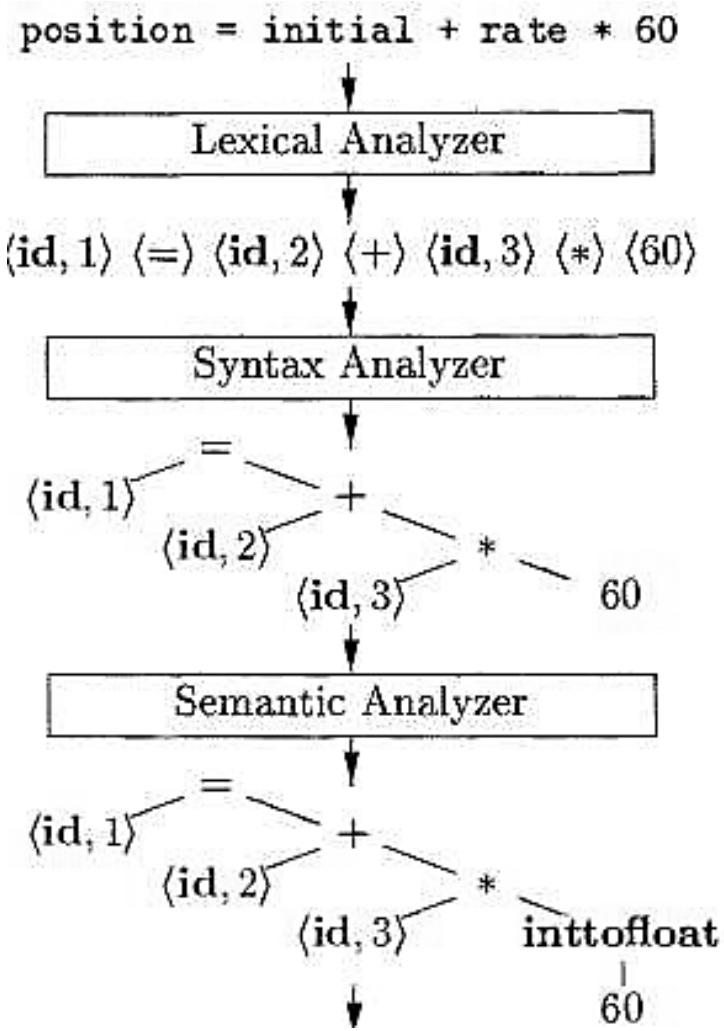
Phases of a compiler



Phases of a compiler

- **Semantic Analysis**: 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.
- An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.
- The language specification may permit some type conversions called **coercions**.
- For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

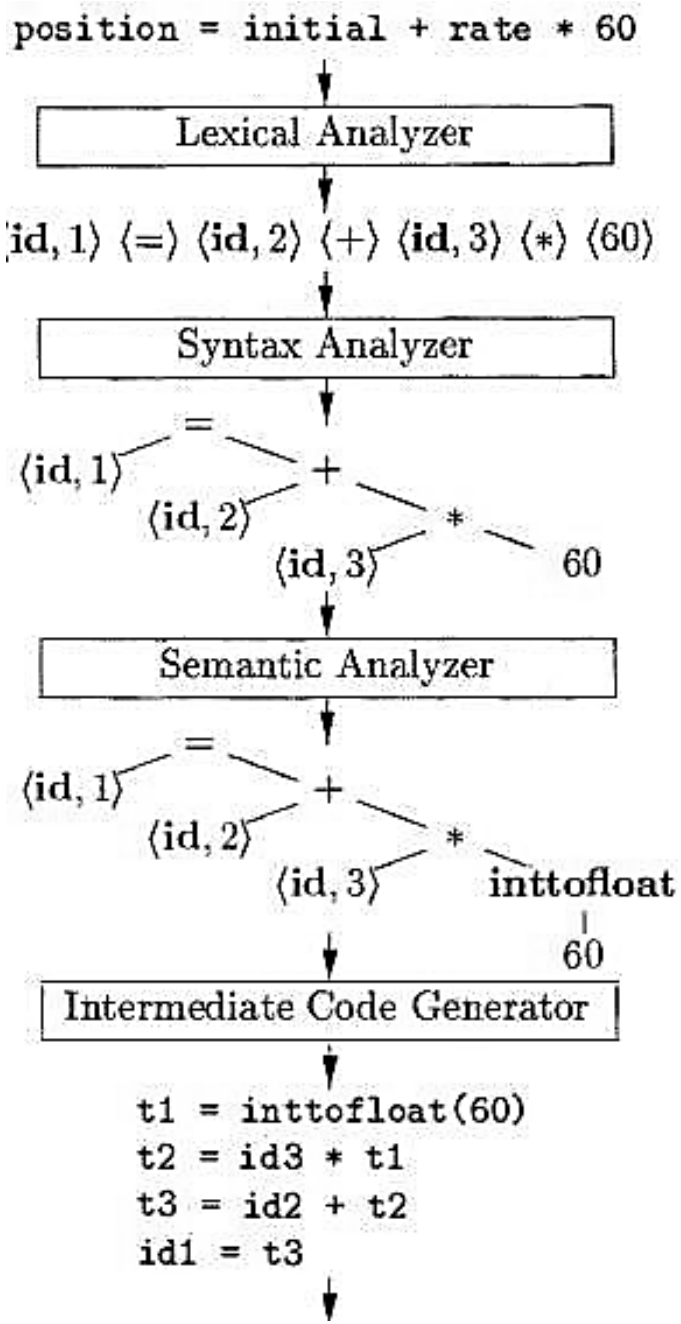
Phases of a compiler



Phases of a compiler

- **Intermediate Code Generation:** In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. 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.

Phases of a compiler



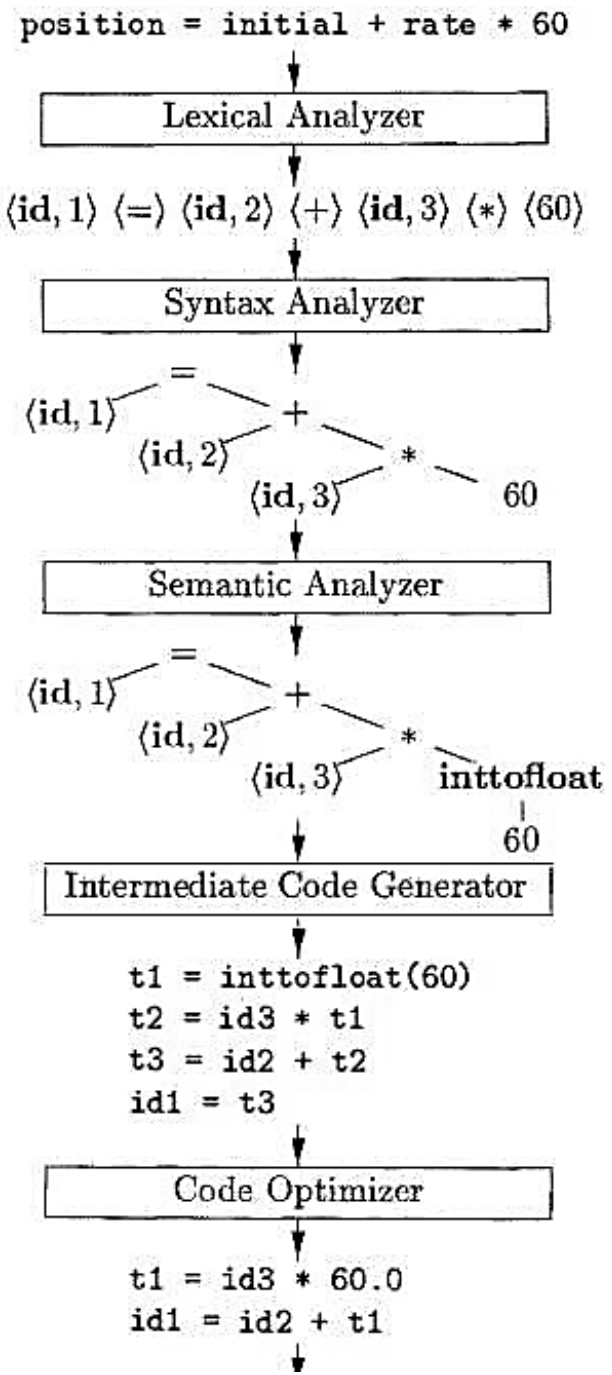
Phases of a compiler

- **Code Optimization:** The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power. For example, a straightforward algorithm generates the intermediate code, using an instruction for each operator in the tree representation that comes from the semantic analyzer.
- A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. 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 inttofloat operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, t3 is used only once to transmit its value to id1 so the optimizer can transform into the shorter sequence
  - t 1 = id3 * 60 . 0
  - id1 = id2 + t 1

Phases of a compiler

- 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.

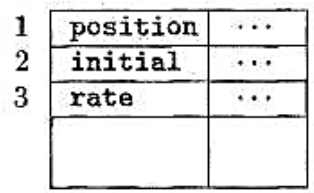
Phases of a compiler



Phases of a compiler

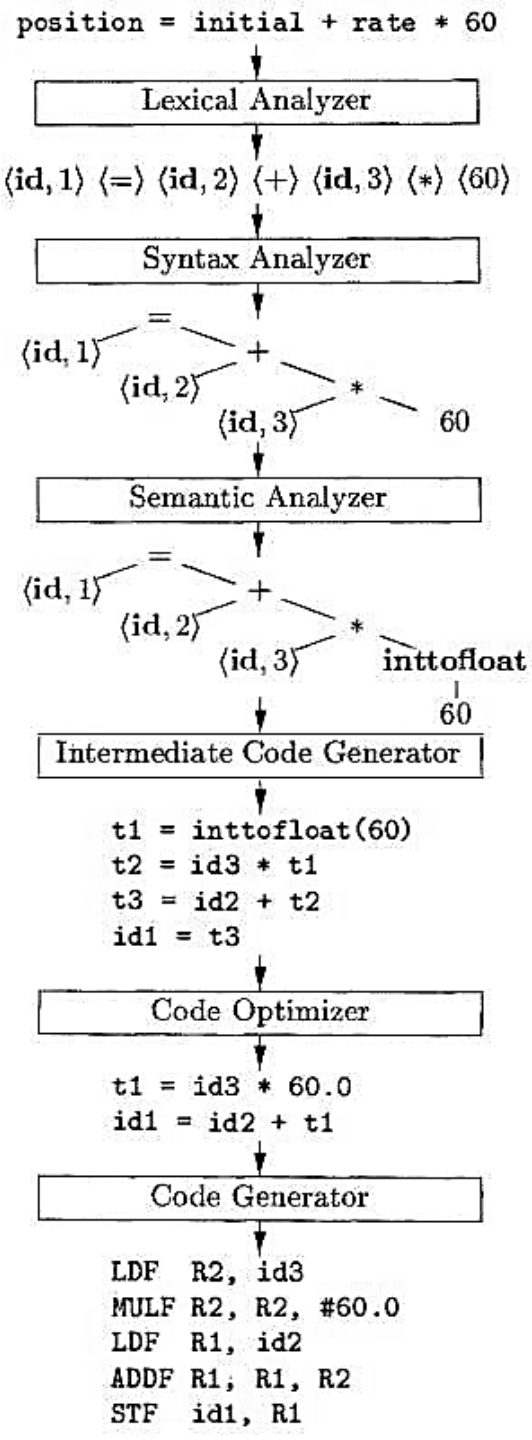
- **Code Generation:** The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers Or memory locations are selected for each of the variables used by the program. Then, the intermediate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

Phases of a compiler



1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE



Phases of a compiler

- **Symbol-Table Management:** 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.

Compiler- Construction Tools

1. Parser generators that automatically produce syntax analyzers from a grammatical description of a programming language.
2. Scanner generators that produce lexical analyzers from a regular expression description of the tokens of a language.
3. Syntax-directed translation engines that produce collections of routines for walking a parse tree and generating intermediate code.
4. Code-generator generators 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.
5. Data-flow analysis engines that facilitate the gathering of information about how values are transmitted from one part of a program to each 1other part. Data-flow analysis is a key part of code optimization.
6. Compiler- construction toolkits that provide an integrated set of routines for constructing various phases of a compiler.

Runtime Environments

A compiler must accurately implement the abstractions embodied in the source language definition.

– such as names, scopes, bindings, data types, operators, procedures, parameters, and flow-of-control-constructs

• The compiler creates and manages a run-time environment in which it assumes its target programs are being executed.

Runtime Environments

Program vs program execution:

- A program consists of several procedures (functions)
- An execution of a program is called as a process
- An execution of a program would cause the activation of the related procedures
- A name (e.g. a variable name)in a procedure would be related to different data objects

Notes: An activation may manipulate data objects allocated for its use.

THANK YOU